

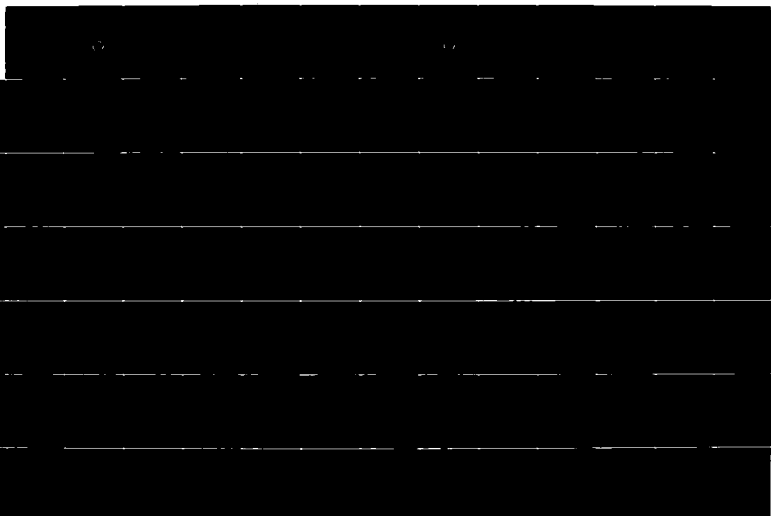
AD-A157-589

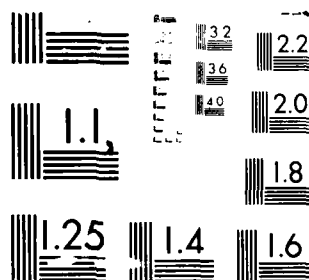
MILITARY STANDARD COMMON APSE (ADA PROGRAMMING SUPPORT
ENVIRONMENT) INTERFACE SET (CAIS) (U) ADA JOINT PROGRAM
OFFICE ARLINGTON VA JAN 89

1/4

UNCLASSIFIED

F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NOTE: This draft, dated 31 January 1985, prepared by the KAPSE Interface Team and KAPSE Interface Team from Industry and Academia CAIS Working Group for the Ada* Joint Program Office, has not been approved and is subject to modification.

DO NOT USE PRIOR TO APPROVAL.

(Project IPSC/ECRS 0208)

PROPOSED
MIL-STD-CAIS
31 JANUARY 1985

MILITARY STANDARD COMMON APSE INTERFACE SET (CAIS)



This document has been approved
for public release and its
distribution is unlimited.

AREA ECRS

NO DELIVERABLE DATA REQUIRED BY THIS DOCUMENT

*Ada is a Registered Trademark of the U.S. Government (Ada Joint Program Office)

N O T I C E

THIS DOCUMENT HAS BEEN REPRODUCED FROM THE
BEST COPY FURNISHED US BY THE SPONSORING
AGENCY. ALTHOUGH IT IS RECOGNIZED THAT CER-
TAIN PORTIONS ARE ILLEGIBLE, IT IS BEING RE-
LEASED IN THE INTEREST OF MAKING AVAILABLE
AS MUCH INFORMATION AS POSSIBLE.

AD-A157 589

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	12. GOVT ACCESSION NO.	13. RECIPIENT'S CATALOG NUMBER
AD-A157589		
4. TITLE (and Subtitle) Military Standard Common Apse Interface Set (CAIS)		5. TYPE OF REPORT & PERIOD COVERED January 1985 to January 1986
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) KAPSE Interface Team		8. CONTRACT OR GRANT NUMBER(s) MDA903-83-C-0306
9. PERFORMING ORGANIZATION NAME AND ADDRESS IIT Research Institute 1211 S. Fern St. Rm. C-107 Arlington, VA 22202		10. PROGRAM ELEMENT PROJECT TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office 3D139 (400 Army Navy) The Pentagon Washington, D.C. 20301		12. REPORT DATE January 1985
		13. NUMBER OF PAGES 319
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) Facilitate interoperability and transportability between APSEs. Proposed policy statement regarding appropriate application of CAIS version 1.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) 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 Environment (APSEs) to promote Ada tool transportability and interoperability. The initial interfaces for the CAIS were derived from the Ada Integrated Environment (AIE) and the Ada Language System (ALS). Since		

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)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

then the CAIS has been expanded to be implementable as part of a wide variety of APSEs. It is anticipated that the CAIS will evolve to meet new needs. Through the acceptance of this standard, it is anticipated that the source level portability of Ada software tools will be enhanced for both DoD and non-DoD users.

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

The initial effort for definition of the CAIS was begun in September, 1982.

This report should be processed. The controlling office is seeking comments from interested parties who use it. It is a proposed standard.

Per Mr. Burton Newlin, Ada Joint Program Office.

(Aug 83 telecon) /L

S/N 0102- LP- 014- 6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)



OFFICE OF THE UNDER SECRETARY OF DEFENSE

WASHINGTON D C 20301

31 JANUARY 1985

RESEARCH AND
ENGINEERING
(R&AT)

Dear CAIS Reviewer:

The Common APSE Interface Set (CAIS) has been developed to facilitate interoperability and transportability between APSEs. Its development was directed by a January 3, 1982 Memorandum of Agreement between the three Services and the Deputy Under Secretary of Defense for Research and Engineering (Acquisition Management). In that memorandum, the Services agreed to establish a set of interfaces upon which formal coordination as a military standard could begin. However, the CAIS is new and there is very little experience with it as yet. Therefore, its establishment as a military standard must be accompanied by a clear statement of the policy regarding its application to projects and contracts.

The attached is a proposed policy statement regarding appropriate application of CAIS Version 1 once it is approved as a military standard. The community at large has expressed a great deal of concern over the potential for misapplication of this interface set when it becomes a military standard. Therefore, we are submitting this draft policy statement for your review in addition to the MIL-STD-CAIS document itself. It is requested that you use the same comment procedures for returning feedback on this draft policy statement as for the CAIS document itself.

Robert F. Mathis
Robert F. Mathis
Director
STARS Joint Program Office

Attachment

PROPOSED CAIS POLICIES

1. Objective: The objective behind the creation of the Common APSE Interface Set (CAIS) is to promote the portability of tools and data between APSEs. The CAIS has been formulated to provide those interfaces most commonly required by tools in the course of their normal operations. When the CAIS has matured to the point of wide acceptance by industry, the DoD will move to apply this standard to the DoD-funded environments.
2. Purpose: This set of interfaces is being issued as a military standard in order to allow its application to government contracts. The principal purpose of such application is to allow contracts to specify the use of the CAIS in experimental implementations whose objective is to learn about the viability, feasibility, implementability and usability of the interface set as a component of a programming support environment. Implementations of this proposed interface set should provide knowledge about implementation of its features and feedback to the CAIS designers relevant to the development of Version 2 of the CAIS.
3. Proper Uses: Proper applications of this standard to contracts include: (1) prototype implementations of the interface set, either wholly or in part; (2) prototype implementations of tools written to run on top of a CAIS implementation; (3) implementation/comparison studies designed for such purposes as determining the probable ease of implementing the CAIS on a new operating system or bare machine or comparing the features available in the CAIS with those considered essential in another operating system; and (4) experimental studies designed to utilize a prototype CAIS and/or tool implementation in order to gather information regarding performance, usability, viability, etc.
4. Improper Uses: It is not intended that the CAIS Version 1 military standard be imposed on any development or maintenance project whose primary purpose is not explicitly to experiment with its implementation or that would be unnecessarily risking total project success on the (unproven) viability of the current CAIS. The CAIS should not be imposed nonchalantly or arbitrarily or without a clear understanding of the potential costs and risks involved.
5. Feedback: All uses made of the CAIS should require at least one report intended to provide feedback to the CAIS designers regarding the pros and cons of its implementation and use, ease or difficulty encountered with particular features, and suggestions for improvements to either the form or technical content of the military standard document.

NOTE: This draft, dated 31 January 1985, prepared by the KAPSE Interface Team and KAPSE Interface Team from Industry and Academia CAIS Working Group for the Ada* Joint Program Office, has not been approved and is subject to modification.

DO NOT USE PRIOR TO APPROVAL.

(Project IPSC/ECRS 0208)

PROPOSED
MIL-STD-CAIS
31 JANUARY 1985

MILITARY STANDARD COMMON APSE INTERFACE SET (CAIS)



AREA ECRS

NO DELIVERABLE DATA REQUIRED BY THIS DOCUMENT

*Ada is a Registered Trademark of the U.S. Government (Ada Joint Program Office)

PROPOSED MIL-STD-CAIS
31 JANUARY 1985

DEPARTMENT of DEFENSE
Washington, DC 20302

Common APSE Interface Set

MIL-STD-

1. This Military Standard is approved for use by all Departments and Agencies of the Department of Defense.
2. Beneficial comments (recommendations, additions, deletions) and any pertinent data which may be of use in improving this document should be addressed to KIT/KITIA CAIS Working Group and sent to Patricia Oberndorf, Naval Ocean Systems Center, Code 423, San Diego, CA, 92152-5000 by using the self addressed Standardization Document Improvement Proposal (DD Form 1428) appearing at the end of this document or by letter.

FOREWORD

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 Environment (APSEs) to promote Ada tool transportability and interoperability. The initial interfaces for the CAIS were derived from the Ada Integrated Environment (AIE) and the Ada Language System (ALS). Since then the CAIS has been expanded to be implementable as part of a wide variety of APSEs. It is anticipated that the CAIS will evolve to meet new needs. Through the acceptance of this standard, it is anticipated that the source level portability of Ada software tools will be enhanced for both DoD and non-DoD users.

The authors of this document include technical representatives from 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 from 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 (Institute for Defense Analyses), P. Oberndorf (Naval Ocean Systems Center), T. Taft (Intermetrics), R. Thall (SoftTech) and W. Wilder (NAVSEA PMS-408). In February 1983 the design team was expanded to include LCDR B. Schaar (Ada Joint Program Office), T. Harrison (Texas Instruments) and KAPSE Interface Team from Industry and Academia (KITIA) members: H. Fischer (Litton Data Systems), E. Lamb (Bell Labs), T. Lyons (Software Sciences Ltd., U.K.), D. McGonagle (General Electric), H. Morse (Frey Federal Systems), E. Ploedereder (Tartan Laboratories), H. Willman (Raytheon), and L. Yelowitz (Ford Aerospace). During 1984, the following people assisted in preparation of this document: F. Belz (TRW) and the TRW prototype team, K. Connolly (TRW), S. Ferdman (Data General), G. Fitch (Intermetrics), R. Gouw (TRW), B. Grant (Intermetrics), N. Lee (Institute for Defense Analyses), J. Long (TRW), and R. Robinson (Institute for Defense Analyses). Additional constructive criticism and direction was provided by G. Myers (Naval Ocean Systems Center), O. Roubine (Informatique Internationale), and the general memberships of the KIT and KITIA, as well as many independent reviewers. The Ada Joint Program Office is particularly grateful to these KITIA members and their companies for providing the time and resources that significantly contributed to this document.

This document was prepared with the Unilogic Ltd. SCRIBE typeset tool on the TRW Software Productivity Project development environment.

31 JANUARY 1985

FOREWORD

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 Environment (APSEs) to promote Ada tool transportability and interoperability. The initial interfaces for the CAIS were derived from the Ada Integrated Environment (AIE) and the Ada Language System (ALS). Since then the CAIS has been expanded to be implementable as part of a wide variety of APSEs. It is anticipated that the CAIS will evolve to meet new needs. Through the acceptance of this standard, it is anticipated that the source level portability of Ada software tools will be enhanced for both DoD and non-DoD users.

The authors of this document include technical representatives from 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 from 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 (Institute for Defense Analyses), P. Oberndorf (Naval Ocean Systems Center), T. Taft (Intermetrics), R. Thall (SoftTech) and W. Wilder (NAVSEA PMS-408). In February 1983 the design team was expanded to include LCDR B. Schaar (Ada Joint Program Office), T. Harrison (Texas Instruments) and KAPSE Interface Team from Industry and Academia (KITIA) members: H. Fischer (Litton Data Systems), E. Lamb (Bell Labs), T. Lyons (Software Sciences Ltd., U.K.), D. McGonagle (General Electric), H. Morse (Frey Federal Systems), E. Ploedereder (Tartan Laboratories), H. Willman (Raytheon), and L. Yelowitz (Ford Aerospace). During 1984, the following people assisted in preparation of this document: F. Belz (TRW) and the TRW prototype team, K. Connolly (TRW), S. Ferdman (Data General), G. Fitch (Intermetrics), R. Gouw (TRW), B. Grant (Intermetrics), N. Lee (Institute for Defense Analyses), J. Long (TRW), and R. Robinson (Institute for Defense Analyses). Additional constructive criticism and direction was provided by G. Myers (Naval Ocean Systems Center), O. Roubine (Informatique Internationale), and the general memberships of the KIT and KITIA, as well as many independent reviewers. The Ada Joint Program Office is particularly grateful to these KITIA members and their companies for providing the time and resources that significantly contributed to this document.

This document was prepared with the Unilogic Ltd. SCRIBE typeset tool on the TRW Software Productivity Project development environment.

Contents

Paragraph	Page
1. SCOPE	1
1.1. Purpose	1
1.2. Content	1
1.3. Excluded and deferred topics	2
2. REFERENCED DOCUMENTS	4
2.1. Issues of documents	4
2.2. Other publications	4
3. DEFINITIONS	5
4. GENERAL REQUIREMENTS	11
4.1. Introduction	11
4.2. Method of description	11
4.2.1. Allowable differences	11
4.2.2. Semantic descriptions	12
4.2.3. Typographical conventions	12
4.3. CAIS node model	13
4.3.1. Nodes	13
4.3.2. Processes	13
4.3.3. Input and output	14
4.3.4. Relationships and relations	14
4.3.4.1. Kinds of relationships	15
4.3.4.2. Basic predefined relations	15
4.3.5. Paths and pathnames	17
4.3.6. Attributes	19
4.4. Discretionary and mandatory access control	20
4.4.1. Node access	20
4.4.2. Discretionary access control	21
4.4.2.1. Establishing grantable access rights	21
4.4.2.2. Adopting a role	22
4.4.2.3. Evaluating access rights	23
4.4.2.4. Discretionary access checking	26
4.4.3. Mandatory access control	26
4.4.3.1. Labeling of CAIS nodes	27
4.4.3.2. Labeling of process nodes	28
4.4.3.3. Labeling of non-process nodes	28
4.4.3.4. Labeling of nodes for devices	28
4.4.3.5. Mandatory access checking	29
4.5. Pragmatics	29
4.5.1. Pragmatics for CAIS node model	29
4.5.2. Pragmatics for SEQUENTIAL_IO	30
4.5.3. Pragmatics for DIRECT_IO	30
4.5.4. Pragmatics for TEXT_IO	30
5. DETAILED REQUIREMENTS	31
5.1. General node management	31
5.1.1. Package NODE_DEFINITIONS	31
5.1.2. Package NODE_MANAGEMENT	33

Contents

Paragraph	Page
5.1.2.1. Opening a node handle	38
5.1.2.2. Closing a node handle	39
5.1.2.3. Changing the intention regarding node handle usage	40
5.1.2.4. Examining the open status of a node handle	41
5.1.2.5. Querying the intention of a node handle	41
5.1.2.6. Querying the kind of a node	41
5.1.2.7. Obtaining the unique primary pathname	42
5.1.2.8. Obtaining the relationship key of a primary relationship	42
5.1.2.9. Obtaining the relation name of a primary relationship	43
5.1.2.10. Obtaining the relationship key of the last relationship traversed	44
5.1.2.11. Obtaining the relation name of the last relationship traversed	44
5.1.2.12. Obtaining a partial pathname	44
5.1.2.13. Obtaining the name of the last relationship in a pathname	45
5.1.2.14. Obtaining the key of the last relationship in a pathname	45
5.1.2.15. Querying the existence of a node	45
5.1.2.16. Querying sameness	46
5.1.2.17. Obtaining an open node handle to the parent node	47
5.1.2.18. Copying a node	48
5.1.2.19. Copying trees	50
5.1.2.20. Renaming the primary relationship of a node	51
5.1.2.21. Deleting the primary relationship to a node	53
5.1.2.22. Deleting the primary relationships of a tree	54
5.1.2.23. Creating secondary relationships	55
5.1.2.24. Deleting secondary relationships	56
5.1.2.25. Node iteration types and subtypes	57
5.1.2.26. Creating an iterator over nodes	58
5.1.2.27. Determining iteration status	59
5.1.2.28. Getting the next node in an iteration	59
5.1.2.29. Setting the current node relationship	60
5.1.2.30. Opening a node handle to the current node.	61
5.1.3. Package ATTRIBUTES	62
5.1.3.1. Creating node attributes	62
5.1.3.2. Creating path attributes	63
5.1.3.3. Deleting node attributes	64
5.1.3.4. Deleting path attributes	65
5.1.3.5. Setting node attributes	66
5.1.3.6. Setting path attributes	67
5.1.3.7. Getting node attributes	69
5.1.3.8. Getting path attributes	69
5.1.3.9. Attribute iteration types and subtypes	71
5.1.3.10. Creating an iterator over node attributes	71
5.1.3.11. Creating an iterator over relationship attributes	72
5.1.3.12. Determining iteration status	73
5.1.3.13. Getting the next attribute	73
5.1.4. Package ACCESS_CONTROL	74
5.1.4.1. Subtypes	74
5.1.4.2. Setting access control	74

descendant (of a node) - Any node which is reachable from other nodes via primary relationships.

pragmatics - Constraints imposed by an implementation that are not defined by the syntax or semantics of the CAIS.

primary relationship - The initial relationship established from an existing node to a newly created node during its creation. The existence of a node is determined by the existence of the primary relationship of which it is the target.

process - The execution of an Ada program, including all its tasks.

process node - A node whose contents represent a CAIS process.

program - [LRM] A program is composed of a number of compilation units, one of which is a subprogram called the main program.

qualified area - A contiguous group of positions in a form that share a common set of characteristics.

queue - [IEEE] A list that is accessed in a first-in, first-out manner.

relation - In the node model, a class of relationships sharing the same name.

relation name - The string that identifies a relation.

relationship - In the node model, an edge of the directed graph which emanates from a source node and terminates at a target node. A relationship is an instance of a relation.

relationship key - The string that distinguishes a relationship from other relationships having the same relation name and emanating from the same node.

relevant grant items - The items in values of GRANT attributes of relationships of the relation ACCESS emanating from the object and pointing at any node representing a role which is an adopted role of the subject or representing a group, one of whose permanent members is an adopted role of the subject.

role - A set of access rights that a subject can acquire.

root process node - The initial process node created when a user logs on to an APSE or when a new job is created via the CREATE_JOB interface.

secondary relationship - An arbitrary connection which is established between two existing nodes.

security level - [TCSEC] The combination of a hierarchical classification and a set of non-hierarchical categories that represents the sensitivity of information.

source node - The node from which a relationship emanates.

start position (of a form terminal) - The position of a form identified by row one, column one.

latest key - The final part of a key that is automatically assigned lexicographically following all previous keys for the same relation names and initial relationship key character sequence for a given node.

list - [IEEE] An ordered set of items of data; in the CAIS, an entity of type LIST_TYPE whose value is a linearly ordered set of data elements.

list item - A data element in a list.

mandatory access control - See access control.

named item - a list item which has name associated with it.

named list - a list whose items are all named.

node - A representation within the CAIS of an entity relevant to the APSE.

node handle - An Ada object of type NODE_TYPE which is used to identify a CAIS node; it is internal to a process.

non-existing node - A node which has never been created.

object - [TCSEC] A passive entity that contains or receives information. In the CAIS, any node may be an object.

obtainable - A node is obtainable if it is created and not deleted.

open node handle - A node handle that has been assigned to a particular node.

parent - The source node of a primary relationship; also the target of a relationship of the predefined relation PARENT.

path - A sequence of relationships connecting one node to another. Starting from a given node, a path is followed by traversing a sequence of relationships until the desired node is reached.

path element - A portion of a pathname representing the traversal of a single relationship; a single relation name and relationship key pair.

pathname - A name for a path consisting of the concatenation of the names of the traversed relationships in the path in the same order in which they are traversed.

permanent member - A group member which is intrinsically related to the group via primary relationships of the predefined relation PERMANENT_MEMBER.

position (of a terminal) - A place in an output device in which a single, printable ASCII character may be graphically displayed.

potential member - A group member that may dynamically acquire membership in the group; represented by a node that is the target of a secondary relationship of the predefined relation POTENTIAL_MEMBER emanating from that group node or from any of that group nodes' descendants.

element (of a file) - A value of the generic data type with which the input and output package was instantiated; see [LRM] for additional information.

end position - The position of a form identified by the highest row and column indices of the form.

external file - [LRM 14.1.1 - Ada external file] Values input from the external environment of the program, or output to the environment, are considered to occupy external files. An external file can be anything external to the program that can produce a value to be read or receive a value to be written.

file - See **external file**.

file handle - An object of type FILE_TYPE which is used to identify an internal file.

file node - A node whose contents are an Ada external file, e.g., a host system file, a device, or a queue.

form - A form is a two-dimensional matrix of character positions.

group - A collection of nodes representing roles and identified by a structural node with emanating relationships of the predefined relations POTENTIAL_MEMBER and PERMANENT_MEMBER identifying each of the group's members. A member may be a user top-level node, a node representing the executable image of a program, or a node representing a group.

illegal identification - A node identification in which the pathname or the relationship key or relation name is syntactically illegal with respect to the syntax defined in Table 1.

inaccessible - The subject has not (adopted a role which has) been granted the access right of EXISTENCE to the object.

initiate - To place a program into execution; in the CAIS, this means a process node is created, a process is created as its contents, required resources are allocated, and execution is started.

initiated process - The process whose program has been placed into execution.

initiating process - The process placing a program into execution.

interface - [DACS] A shared boundary.

internal file - A file which is internal to a CAIS process. Such a file is identified by a file handle.

iterator - A variable which provides the bookkeeping information necessary for iteration over nodes (a node iterator) or attributes (an attribute iterator).

job - A process node tree, spanned by primary relationships, which develops under a root process node as other (dependent) processes are initiated for the user.

key - See **relationship key**. The key of a node is the relationship key of the last element of the node's pathname.

label group (of a magnetic tape) - One of the following: (I) a volume header and a file header label, (II) a file header label, or (III) an end-of-file label.

granted by an object to adopters of that role; in the CAIS this is accomplished by establishing a secondary relationship of the predefined relation **ADOPTED_ROLE** from the process node to the node representing the role.

adopted role of a process - The access rights associated with the node that is the target of a relationship of the predefined relation **ADOPTED_ROLE** emanating from the process node or with any group node one of whose permanent members is the target of such a relationship.

advance (of an active position) - (1) Scroll or page terminal: Occurs whenever (i) the row number of a new position is greater than the row number of the old or (ii) the row number of the new position is the same and the column number of the new position is greater than that of the old. (2) Form terminal: Occurs whenever the indices of its position are incremented.

approved access rights - Access rights whose names appear in resulting rights lists of relevant grant items for which either (i) the necessary right is null or (ii) the necessary right is an approved access right.

area qualifier - A designator for the beginning of a qualified area.

attribute - A named value associated with a node or relationship which provides information about that node or relationship.

closed node handle - A node handle which is not associated with any node.

contents - A file or process associated with a CAIS node.

couple - To establish a correlation between a queue file and a secondary storage file. If the queue file is a copy queue file, its initial contents is a copy of the secondary storage file to which it is coupled; if the queue file is a mimic queue file, its initial contents is a copy of the secondary storage file to which it is coupled, and elements that are written to the mimic queue file are appended to its coupled file.

current job - The root process node of the tree containing the current process node; represented by the predefined relation **CURRENT_JOB**.

current node - The node that is currently the focus or context for the activities of the current process; represented by the predefined relation **CURRENT_NODE**.

current process - The currently executing process making the call to a CAIS operation. Pathnames are interpreted in the context of the current process.

current user - The user's top-level node; represented by the secondary relationship of the predefined relation **CURRENT_USER**.

dependent process - A process other than a root process.

device [WEBS] - A piece of equipment or a mechanism designed to serve a special purpose or perform a special function.

device name - The keys of a primary relationship of the predefined relation **DEVICE**.

discretionary access control - See **access control**.

3. DEFINITIONS

The following is an alphabetical listing of terms which are used in the description of the CAIS. Where a document named in Section 2 was used to obtain the definition, the definition is preceded by a bracketed reference to that document.

abort - [IEEE] To terminate a process prior to completion.

access - [TCSEC] A specific type of interaction between a subject and an object that results in the flow of information from one to the other.

access checking - The operation of checking access rights against those rights required for the intended operation, according to the access control rules, and either permitting or denying the intended operation.

access control - [TCSEC] (1) discretionary access control: A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject. (2) mandatory access control: A means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity. In the CAIS, this includes specification of access rights, access control rules and checking of access rights in accordance with these rules.

access control constraints - The resulting restrictions placed on certain kinds of operations by access control.

access control information - All the information required to perform access checking.

access control rules - The rules describing the correlations between access rights and those rights required for an intended operation.

access relationship - A relationship of the predefined relation ACCESS.

access rights - Descriptions of the kinds of operations which can be performed.

access to a node - Reading or writing of the contents of the node, reading or writing of attributes of the node, reading or writing of relationships emanating from a node or of their attributes, and traversing a node as implied by a pathname.

accessible - The subject has (adopted a role which has) been granted the access right EXISTENCE to the object.

active position - The position at which a terminal operation is to be performed.

Ada Programming Support Environment (APSE) - [UK Ada Study, STONEMAN] A set of hardware and software facilities whose purpose is to support the development and maintenance of Ada applications software throughout its life-cycle with particular emphasis on software for embedded computer applications. The principal features are the database, the interfaces and the tool set.

adopt a role - The action of a process to acquire the access rights which have been or will be

2. REFERENCED DOCUMENTS

2.1. Issues of documents

The following documents of the issue in effect on date of invitation for bids or request for proposal form a part of this standard to the extent specified herein.

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

(Copies of specifications, standards, drawings, and publications required by contractors in connection with specific procurement functions should be obtained from the procuring activity or as directed by the contracting officer.)

2.2. Other publications

The following documents form a part of the standard to the extent specified herein. Unless otherwise indicated, the issue in effect on date of invitation for bids or request for proposal shall apply.

[ANSI 78]: American National Standards Institute, *Magnetic Tape Labels and File Structure for Information Interchange (ANSI Standard z3.27-1978)*. (Application for copies should be addressed to American National Standards Institute, Inc., 1430 Broadway, New York, NY 10018)

[DACS]: DACS Glossary, a Bibliography of Software Engineering Terms, GLOS-1, October 1979, Data and Analysis Center for Software. (Application for copies should be addressed to Data and Analysis Center for Software, RADC/ISISI, Griffiss AFB, NY 13441)

[IEEE]: IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std. 729-1983. (Application for copies should be addressed to Sales Department, American National Standards Institute, 1430 Broadway, New York, NY 10018)

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

[TCSEC]: Department of Defense Trusted Computer System Evaluation Criteria, Department of Defense Computer Security Center, CSC-STD-001-83, 15 August 1983. (Application for copies should be addressed to Department of Defense, Computer Security Center, Office of Standards and Products, Attention: Chief, Computer Security Standards, Fort George G. Meade, Maryland 20755.)

[UK Ada Study]: United Kingdom Ada Study Final Technical Report, Volume I, London, Department of Industry, 1981. (Application for copies should be addressed to Scientific Information Office, British Defence Staff, British Embassy, 3100 Massachusetts Avenue, NW, Washington, D.C. 20008.)

[WEBS]: Webster's New Collegiate Dictionary, G.&C. Merriam Company, Springfield, Massachusetts, 1979.

used by tool sets to constrain nodes, attributes, and relations, but it does not enforce a particular methodology. Currently deferred is a decision whether or not the CAIS should enforce a particular, more complete typing methodology and what kind of CAIS interfaces should be made available to support it.

- g. Archiving. The current CAIS does not define facilities for archiving data. Currently deferred is a decision regarding the form that archiving interfaces should take.

- b. Processes. This area covers program invocation and control.
- c. Input and Output. This area covers file input and output, basic device input and output support, special device control facilities, and interprocess communication.
- d. Utilities. This area covers list operations useful for manipulation of parameters and attribute values.

1.3. Excluded and deferred topics

During the design of the CAIS it was determined that interfaces for environments which are not software development environments (for example, interfaces on target systems) and interfaces for multilingual environments should be explicitly excluded. It has been decided that backup facilities will be supported transparently by the CAIS implementation. While the interface issues of most aspects of environments were considered, the complete resolution of several areas has been deferred until later revisions of the CAIS. These areas are:

- a. Configuration management. The current CAIS supports facilities for configuration control including keeping versions, referencing the latest revision, identifying the state of an object, etc.; but it does not implement a particular methodology. Currently deferred is the decision whether or not the CAIS should enforce a particular configuration management approach and, if so, what particular methodology should be chosen.
- b. Device control and resource management. The current CAIS provides control facilities for scroll, page and form terminals and magnetic tape drives. Currently deferred is the decision as to what additional devices or resources must be supported by the CAIS. Such resources and devices might include printers, disk drives, color terminals, vector- and bit-addressable graphics devices, processor memory, processor time, communication paths, etc. Also deferred is a decision regarding which other American National Standards Institute or International Standards Organization interfaces to adopt, such as the ISO/DIS 7942 Graphical Kernel System (GKS).
- c. Distributed environments. The existing CAIS packages are intended to be implementable on a distributed set of processors, but in a manner that is transparent to a tool. Currently deferred is the decision whether or not to provide to the user explicit CAIS interfaces to control the distribution of the environment, including designation of where nodes exist and where execution takes place. Note that a set of distributed processors could include one or more target machines.
- d. Inter-tool interfaces. The current CAIS does not define inter-tool calling sequences or data formats such as the data format within the compilation/program library system, the text format within editing systems, the command processor language syntax, the message formats of a mail system, or the interaction between the run-time system and debugger tools. Currently deferred are decisions regarding what inter-tool data should become part of the standard, what form such interfaces should take, and whether or not to place constraints on the run-time system to provide process execution information.
- e. Interoperability. The current CAIS provides only a very primitive, text-oriented interface for transferring files between a CAIS implementation and the operating system on which it may reside. It does not define external representations of data for transfer between environments or between a host and target.
- f. Typing methodology. The current CAIS provides attributes and relations which can be

1. SCOPE

1.1. Purpose

This document provides specifications for a set of Ada packages, with their intended semantics, which together form the set of common interfaces for Ada Programming Support Environments (APSEs). This set of interfaces is known as the Common APSE Interface Set (CAIS). This interface set is designed to promote the source-level portability of Ada programs, particularly Ada software development tools.

The CAIS applies to Ada Programming Support Environments which are to become the basic software life-cycle environments for Department of Defense (DoD) mission critical computer systems (MCCS). 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. Groups of tools that are composed of a number of independent but interrelated programs (such as a debugger which is related to a specific compiler) are classed as *tool sets*¹.

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 synonyms.²

The CAIS is intended to evolve 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 between CAIS implementations. Where tools function as a set, the CAIS facilitates transportability of the tool set as a whole; tools might not be individually transportable because they depend on inputs from other tools in the set.

1.2. Content

The CAIS establishes interface requirements for the transportability of Ada tool sets to be used in Department of Defense (DoD) APSEs. Strict adherence to this interface set will ensure that Ada tool sets will possess the highest degree of transportability across conforming 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. The CAIS is intended to provide the transportability interfaces most often required by common software development tools and includes four interface areas:

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

¹Requirements for Ada Programming Support Environments, STONEMAN; Department of Defense; February 1980.

²KAPSE Interface Team; Public Report, Volume I, 1 April 1982; p. C1.

Contents

Paragraph	Page
5.4.1.22.2 Extracting a floating point item from a list	210
5.4.1.22.3 Replacing a floating point item into a list	211
5.4.1.22.4 Inserting a floating point item into a list	212
5.4.1.22.5 Identifying a floating point item by value within a list	212
5.4.1.23. Package STRING_ITEM	213
5.4.1.23.1 Extracting a string item from a list	213
5.4.1.23.2 Replacing a string item in a list	214
5.4.1.23.3 Inserting a string item into a list	214
5.4.1.23.4 Identifying a string item by value within a list	215
6. NOTES	217
6.1. Keywords	217
Tables	
Table I. Pathname BNF	19
Table II. GRANT attribute value BNF	23
Table III. Predefined access rights	24
Table IV. Classification attribute value BNF	28
Table V. Intents	33
Table VI. Process status transition table	81
Table VII. Created and inherited relationships	82
Table VIII. Input and output packages for file kinds	101
Table IX. File node predefined attributes, attribute values and relation	103
Table X. Modes and Intents	103
Table XI. Volume header label	184
Table XII. File header label	186
Table XIII. End of file label	187
Table XIV. List external representation BNF	192
Figures	
Figure 1. Access relationships	24
Figure 2. Matrix of access synchronization constraints	37
Appendix A. PREDEFINED RELATIONS, ATTRIBUTES AND ATTRIBUTE VALUES	218
Appendix B. CAIS Specification	222
Appendix C. CAIS Body	249
Appendix D. PACKAGE LISTING OF CAIS PROCEDURES AND FUNCTIONS	298

Contents

Paragraph	Page
5.3.9.12. Rewinding the tape	182
5.3.9.13. Skipping tape marks	182
5.3.9.14. Writing a tape mark	183
5.3.9.15. Writing a volume header label	183
5.3.9.16. Writing a file header label	185
5.3.9.17. Writing an end of file label	187
5.3.9.18. Reading a label on a labeled tape	188
5.3.10. Package FILE_IMPORT_EXPORT	189
5.3.10.1. Importing a file	189
5.3.10.2. Exporting a file	190
5.4. CAIS Utilities	191
5.4.1. Package LIST_UTILITIES	193
5.4.1.1. Types and subtypes	193
5.4.1.2. Copying a list	194
5.4.1.3. Converting to an internal list representation	194
5.4.1.4. Converting to an external list representation	195
5.4.1.5. Determining the equality of two lists	195
5.4.1.6. Deleting an item from a list	195
5.4.1.7. Determining the kind of list	196
5.4.1.8. Determining the kind of list item	196
5.4.1.9. Inserting a sublist of items into a list	197
5.4.1.10. Merging two lists	197
5.4.1.11. Extracting a sublist of items from a list	198
5.4.1.12. Determining the length of a list	198
5.4.1.13. Determining the length of a string representing a list or a list item	199
5.4.1.14. Determining the name of a named item	199
5.4.1.15. Determining the position of a named item	200
5.4.1.16. Extracting a list-type item from a list	200
5.4.1.17. Replacing a list-type item in a list	201
5.4.1.18. Inserting a list-type item into a list	202
5.4.1.19. Identifying a list-type item by value within a list	202
5.4.1.20. Package IDENTIFIER_ITEM	203
5.4.1.20.1 Converting an identifier to a token	203
5.4.1.20.2 Converting a token to an identifier	204
5.4.1.20.3 Determining the equality of two tokens	204
5.4.1.20.4 Extracting an identifier item from a list	204
5.4.1.20.5 Replacing an identifier item in a list	205
5.4.1.20.6 Inserting an identifier item into a list	206
5.4.1.20.7 Identifying an identifier item by value within a list	206
5.4.1.21. Generic package INTEGER_ITEM	207
5.4.1.21.1 Converting an integer item to its canonical external representation	207
5.4.1.21.2 Extracting an integer item from a list	207
5.4.1.21.3 Replacing an integer item in a list	208
5.4.1.21.4 Inserting an integer item into a list	209
5.4.1.21.5 Identifying an integer item by value within a list	209
5.4.1.22. Generic package FLOAT_ITEM	210
5.4.1.22.1 Converting a floating point item to its canonical external representation	210

Contents

Paragraph	Page
5.3.7.10. Enabling echo on a terminal	152
5.3.7.11. Querying echo on a terminal	153
5.3.7.12. Determining the number of function keys	153
5.3.7.13. Reading a character from a terminal	154
5.3.7.14. Reading all available characters from a terminal	155
5.3.7.15. Determining the number of function keys that were read	156
5.3.7.16. Determining function key usage	158
5.3.7.17. Determining the name of a function key	157
5.3.7.18. Deleting characters	158
5.3.7.19. Deleting lines	159
5.3.7.20. Erasing characters in a line	160
5.3.7.21. Erasing characters in a display	160
5.3.7.22. Erasing characters in a line	161
5.3.7.23. Inserting space characters in a line	162
5.3.7.24. Inserting blank lines in the output terminal file	163
5.3.7.25. Determining graphic rendition support	164
5.3.7.26. Selecting the graphic rendition	164
5.3.8. Package FORM_TERMINAL	165
5.3.8.1. Types and subtypes	166
5.3.8.2. Determining the number of function keys	167
5.3.8.3. Defining a qualified area	167
5.3.8.4. Removing an area qualifier	168
5.3.8.5. Changing the active position	168
5.3.8.6. Moving to the next qualified area	169
5.3.8.7. Writing to a form	169
5.3.8.8. Erasing a qualified area	170
5.3.8.9. Erasing a form	170
5.3.8.10. Activating a form on a terminal	170
5.3.8.11. Reading from a form	171
5.3.8.12. Determining changes to a form	171
5.3.8.13. Determining the termination key	172
5.3.8.14. Determining the size of a form	172
5.3.8.15. Determining the size of a terminal	172
5.3.8.16. Determining if the area qualifier requires space in the form	173
5.3.8.17. Determining if the area qualifier requires space on a terminal	173
5.3.9. Package MAGNETIC_TAPE	174
5.3.9.1. Types and subtypes	175
5.3.9.2. Mounting a tape	175
5.3.9.3. Loading an unlabeled tape	176
5.3.9.4. Initializing an unlabeled tape	177
5.3.9.5. Loading a labeled tape	177
5.3.9.6. Initializing a labeled tape	178
5.3.9.7. Unloading a tape	179
5.3.9.8. Dismounting a tape	180
5.3.9.9. Determining if the tape drive is loaded	180
5.3.9.10. Determining if a tape is mounted	181
5.3.9.11. Determining the position of the tape	181

Contents

Paragraph	<u>Page</u>
5.3.4.10. Setting the error file	119
5.3.4.11. Determining the standard error file	120
5.3.4.12. Determining the current error file	120
5.3.5. Package IO _CONTROL	120
5.3.5.1. Obtaining an open node handle from a file handle	121
5.3.5.2. Synchronizing program files with system files	122
5.3.5.3. Establishing a log file	122
5.3.5.4. Removing a log file	123
5.3.5.5. Determining whether logging is specified	123
5.3.5.6. Determining the log file	123
5.3.5.7. Determining the file size	124
5.3.5.8. Setting the prompt string	124
5.3.5.9. Determining the prompt string	125
5.3.5.10. Determining intercepted characters	125
5.3.5.11. Enabling and disabling function key usage	126
5.3.5.12. Determining function key usage	126
5.3.5.13. Creating a queue file node	127
5.3.6. Package SCROLL _TERMINAL	130
5.3.6.1. Subtypes	130
5.3.6.2. Setting the active position	130
5.3.6.3. Determining the active position	131
5.3.6.4. Determining the size of the terminal	132
5.3.6.5. Setting a tab stop	133
5.3.6.6. Clearing a tab stop	133
5.3.6.7. Advancing to the next tab position	134
5.3.6.8. Sounding a terminal bell	135
5.3.6.9. Writing to the terminal	136
5.3.6.10. Enabling echo on a terminal	137
5.3.6.11. Querying echo on a terminal	138
5.3.6.12. Determining the number of function keys	138
5.3.6.13. Reading a character from a terminal	139
5.3.6.14. Reading all available characters from a terminal	140
5.3.6.15. Determining the number of function keys that were read	141
5.3.6.16. Determining function key usage	141
5.3.6.17. Determining the name of a function key	142
5.3.6.18. Advancing the active position to the next line	143
5.3.6.19. Advancing the active position to the next page	144
5.3.7. Package PAGE _TERMINAL	144
5.3.7.1. Types, subtypes and constants	145
5.3.7.2. Setting the active position	145
5.3.7.3. Determining the active position	146
5.3.7.4. Determining the size of the terminal	147
5.3.7.5. Setting a tab stop	148
5.3.7.6. Clearing a tab stop	148
5.3.7.7. Advancing to the next tab position	149
5.3.7.8. Sounding a terminal bell	150
5.3.7.9. Writing to the terminal	151

Contents

Paragraph	Page
5.1.4.3. Examining access rights	75
5.1.4.4. Adopting a role	76
5.1.4.5. Unlinking an adopted role	77
5.1.5. Package STRUCTURAL_NODES	77
5.1.5.1. Creating structural nodes	77
5.2. CAIS process nodes	79
5.2.1. Package PROCESS_DEFINITIONS	81
5.2.2. Package PROCESS_CONTROL	82
5.2.2.1. Spawning a process	83
5.2.2.2. Awaiting termination or abortion of another process	85
5.2.2.3. Invoking a new process	85
5.2.2.4. Creating a new job	88
5.2.2.5. Appending results	90
5.2.2.6. Overwriting results	90
5.2.2.7. Getting results from a process	90
5.2.2.8. Determining the status of a process	92
5.2.2.9. Getting the parameter list	92
5.2.2.10. Aborting a process	93
5.2.2.11. Suspending a process	94
5.2.2.12. Resuming a process	95
5.2.2.13. Determining the number of open node handles	96
5.2.2.14. Determining the number of input and output units used	97
5.2.2.15. Determining the time of activation	98
5.2.2.16. Determining the time of termination or abortion	99
5.2.2.17. Determining the time a process has been active	100
5.3. CAIS input and output	100
5.3.1. Package IO_DEFINITIONS	104
5.3.2. Package DIRECT_IO	104
5.3.2.1. Subtypes and constants	104
5.3.2.2. Creating a direct input or output file	105
5.3.2.3. Opening a direct input or output file	107
5.3.2.4. Deleting a direct input or output file	108
5.3.3. Package SEQUENTIAL_IO	109
5.3.3.1. Subtypes and constants	109
5.3.3.2. Creating a sequential input or output file	109
5.3.3.3. Opening a sequential input or output file	111
5.3.3.4. Deleting a sequential input or output file	112
5.3.4. Package TEXT_IO	113
5.3.4.1. Subtypes and constants	113
5.3.4.2. Creating a text input or output file	113
5.3.4.3. Opening a text input or output file	115
5.3.4.4. Deleting a text input or output file	116
5.3.4.5. Resetting a text file	117
5.3.4.6. Reading from a text file	118
5.3.4.7. Writing to a text file	118
5.3.4.8. Setting the input file	118
5.3.4.9. Setting the output file	119

structural node - A node without contents. Structural nodes are used strictly as holders of relationships and attributes.

subject - [TCSEC] An active entity, generally in the form of a person, process, or device, that causes information to flow among objects or changes the system state. In the CAIS, a subject is always a process.

system-level node - The root of the CAIS primary relationship tree which spans the entire node structure.

target node - The node at which a relationship terminates.

task - [LRM] A task operates in parallel with other parts of the program.

termination of a process - Termination (see [LRM] 9.4) of the execution of the subprogram which is the main program (see [LRM] 10.1) of the process.

token - An internal representation of an identifier which can be manipulated as a list item.

tool - [IEEE - software tool] A computer program used to help develop, test, analyze, or maintain another computer program or its documentation; for example, an automated design tool, compiler, test tool, or maintenance tool.

top-level node - A structural node representing the user. Each user has a top-level node.

track - (1) An open node handle is guaranteed always to refer to the same node, regardless of any changes to relationships that could cause pathnames to become invalid or to refer to different nodes. An open node handle is said to **track** the node to which it refers. (2) Secondary relationships.

traversal of a node - Traversal of a relationship emanating from the node.

traversal of a relationship - The act of following a relationship from its source node to its target node.

unique primary path - The path from the system-level node to a given node traversing only primary relationships. Every node that is not unobtainable has a unique primary path.

unique primary pathname - The pathname associated with the unique primary path.

unnamed item - No name is associated with a list item.

unnamed list - A list whose items are all unnamed.

unobtainable - A node is unobtainable if it is not the target of any primary relationship.

user - An individual, project, or other organizational entity. In the CAIS it is associated with a top-level node.

user name - The key of a primary relationship of the predefined relation USER.

4. GENERAL REQUIREMENTS

4.1. Introduction

The CAIS provides interfaces for data storage and retrieval, data transmission to and from external devices, and activation of programs and control of their execution. In order to achieve uniformity in the interfaces, a single model is used to consistently describe general data storage, devices and executing programs. This approach provides a single model for understanding the CAIS concepts; it provides a uniform understanding of and emphasis on data storage and program control; and it provides a consistent way of expressing interrelations both within and between data and executing programs. This unified model is referred to as the node model.

Section 4.2 discusses how the interfaces are described in the remainder of Section 4 and in Section 5. Section 4.3 describes the node model. Section 4.4 describes the mandatory and discretionary access control model incorporated in the CAIS. Section 4.5 describes limits and constraints not defined by the interfaces. Section 5 provides detailed descriptions of the interfaces. Section 6 provides information on the intended use of this document and relevant keywords for use by automated document retrieval systems.

Appendix A provides descriptions of the entities predefined in the CAIS. This appendix constitutes a mandatory part of this standard.

Appendix B provides a set of the Ada package specifications which have been organized for compilation of the CAIS interfaces. Appendix C provides a set of the corresponding Ada package bodies. Appendix D provides a list of all CAIS procedures and functions organized by the packages in which they appear.

4.2. Method of description

The specifications of the CAIS interfaces are divided into two parts:

- a. the syntax as defined by a canonical Ada package specification, and
- b. the semantics as defined by the descriptions both of the general node model and of the particular packages and procedures.

The Ada package specifications given in this document are termed canonical because they are representative of the form of the allowable actual Ada package specifications in any particular CAIS implementation. The packages which together provide an implementation of these specifications must have indistinguishable syntax and semantics from those stated herein.

4.2.1. Allowable differences

The packages which together provide a particular implementation of the CAIS must have the following properties:

- a. Any Ada program that is legal and not erroneous in the presence of the canonical package specifications as library units must be legal and not erroneous if the canonical packages are replaced by the packages of a particular CAIS implementation and the names of additional library units required for the implementation of this particular CAIS are not in conflict with the names of library units required by the Ada program. [Note: It is recommended,

although not required, that any Ada program that is illegal in the presence of the canonical package specifications as library units is also illegal if the canonical packages are replaced by the packages of a particular CAIS implementation.]

- b. The CAIS interfaces provided by the subprograms declared in the packages of a particular CAIS implementation must have the semantics described in this document for the corresponding subprograms in the canonical package specifications.

The actual Ada package specifications of a particular implementation may differ from the canonical specifications as long as properties (a) and (b) are preserved.

4.2.2. Semantic descriptions

The interface semantics are described in most cases through narrative. These narratives are divided into as many as five paragraphs. The *Purpose* paragraph describes the function of the interface. The *Parameters* paragraph briefly describes each of the parameters, and the *Exceptions* paragraph briefly describes the conditions under which each exception is raised. Any relevant information that does not fall under one of these three headings is included in a *Notes* paragraph. In cases where an interface is overloaded and the additional versions can be described in terms of the basic form of the interface and other CAIS interfaces, these versions are described in a paragraph, called Additional Interfaces, using Ada. This method of presenting the semantics of the Additional Interfaces is a conceptual model. It does not imply that the Additional Interfaces must be implemented in terms of the existing ones exactly as specified, merely that their behavior is equivalent to such an implementation. The semantics described in the Purpose, Parameters and Exceptions apply only to the principal interface; the Additional Interfaces may have additional semantics as implied by the given package bodies.

4.2.3. Typographical conventions

This document follows the typographical conventions of [LRM] where these are not in conflict with those of a MIL-STD. In particular:

- a. **boldface** type is used for Ada language reserved words.
- b. UPPER CASE is used for Ada language identifiers which are not reserved words.
- c. in the text, syntactic category names are written in normal typeface with any embedded underscores removed.
- d. in the text, where reference is made to the actual value of an Ada variable (for example, a procedure parameter), the Ada name is used in normal typeface. However, where reference is made to the Ada object itself (see [LRM] 3.2 for this use of the word object), then the Ada name is given in upper case, including any embedded underscores. For example, from [LRM] 14.2.1 paragraphs 17, 18 and 19

function MODE(FILE: in FILE_TYPE) **return** FILE_MODE;

Returns the current mode of the given file.

but

The exception STATUS_ERROR is raised if the file
is not open.

- e. at the place where a technical term is first introduced and defined in the text, the term is given in an *italic* typeface.

4.3. CAIS node model

The CAIS provides interfaces for administering entities relevant during the software life-cycle such as files, directories, processes and devices. These entities have various properties and may have a variety of interrelations. The CAIS model uses the concept of a *node* as the carrier of information about an entity. It uses the concept of a *relationship* for representing an interrelation between two entities and the concept of an *attribute* for representing a property of an entity or of an interrelation.

The model of the structure underlying the CAIS and reflecting the interrelations of entities is a directed graph of nodes, which form the vertices of the graph, and relationships, which form the edges of the graph. This model is a conceptual model. It does not imply that an implementation of the CAIS must use a directed graph to represent nodes and their relationships.

Both nodes and relationships possess attributes describing properties of the entities represented by nodes and of interrelations represented by relationships.

4.3.1. Nodes

The CAIS identifies three different kinds of nodes: structural nodes, file nodes and process nodes. A node may have contents, relationships and attributes. The *contents* vary with the kind of node. If a node is a *file node*, the contents is an Ada external file. There are four types of CAIS supported Ada external files: secondary storage, queue, terminal, and magnetic tape. The Ada external file may represent a host file, a device (such as a terminal or tape drive) or a queue (as used for process intercommunication). If a node is a *process node*, the contents is a representation of the execution of an Ada program. If a node is a *structural node*, there is no contents and the node is used strictly as a holder of relationships and attributes. The kind of a node is a predefined and implicitly established attribute on every relationship which points to the node.

Nodes can be created, renamed, accessed (as part of other operations), and deleted.

4.3.2. Processes

A *process* is the CAIS mechanism used to represent the execution of an Ada program. A process is represented as the contents of a process node. The process node and its attributes and relationships are also used to bind to an execution the resources (such as files and devices) required by the process. Taken together, the process node, its attributes, relationships and contents are used in the CAIS to manage the dynamics of the execution of a program. Each time execution of a program is initiated, a process node is created, the process is created, the necessary resources to support the execution of the program are allocated to the process, and execution is started. The newly created process is called the *initiated process*, while the process which caused the creation of that process is called the *initiating process*.

A single CAIS process represents the execution of a single Ada program, even when that program includes multiple tasks. Within the process, Ada tasks execute in parallel (proceed independently) and synchronize in accordance with the rules in [LRM] 9, paragraph 5:

Parallel tasks may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can

detect that the same effect can be guaranteed if parts of the actions of a given [Ada] task are executed by different physical processors acting in parallel, it may choose to execute them in this way; in such a case several physical processors implement a single logical processor.

When a task makes a CAIS call, execution of that task is blocked until the CAIS call returns control to the task. Other tasks in the same process may continue to execute in parallel, subject to the Ada tasking rules. If calls on CAIS interfaces are enacted concurrently, the CAIS does not specify their order of execution.

Processes are analogous to Ada tasks in that they execute logically in parallel, have mechanisms for interprocess synchronization, and can exchange data with other processes. However, processes and Ada tasks are dissimilar in certain critical ways. Data, procedures or tasks in one process cannot be directly referenced from another process. Also, while tasks in a program are bound together prior to execution time (at compile or link time), processes are not bound together except by cooperation using CAIS facilities at run time.

4.3.3. Input and output

Ada input and output in [LRM] 14 involves the transfer of data to and from Ada external files. CAIS input and output uses the same model and involves the transfer of data to and from the contents of CAIS file nodes. These file nodes may represent disk or other secondary storage files, magnetic tape drives, terminals, or queues.

CAIS file nodes represent information about and contain Ada external files. The underlying model for the contents of such a node is that of a file of data items, accessible either sequentially or directly by some index. The packages specified in Section 5.3 provide facilities that operate on CAIS external files.

Implementations of the standard Ada packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO specified in the [LRM] that operate upon CAIS files are to be constructed such that they meet the Ada standard and for CREATE and OPEN procedures:

1. The semantics of the use of the default value of the FORM parameter FORM : IN string := "" is specified within the context of the node model.
2. The syntax and semantics of the non empty FORM parameter is specified within the context of the NODE model.
3. Nothing in the implementation can violate the consistency of the CAIS NODE model.

The interfaces in the package MAGNETIC_TAPE have been modeled on the American National Standards Institute standards in [ANSI 78].

4.3.4. 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 5.1.5 STRUCTURAL_NODES and Section 5.2.2 PROCESS_CONTROL) 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*. A relationship may also have attributes describing properties of the relationship.

Because any node may have many relationships representing many different classes of connections,

the concept of a *relation* is introduced to categorize the relationships. Relations identify the nature of relationships, and relationships are instances of relations. Certain basic relations are predefined by the CAIS. Their semantics are explained in the following sections. Additional predefined relations are introduced in Section 5 and are listed in Appendix A. Relations may also be defined by a user. The CAIS associates only the relation name with user-defined relations; no other semantics are supported.

Each relationship is identified by a relation name and a relationship key. The *relation name* identifies the relation, and the *relationship key* distinguishes between multiple relationships each bearing the same relation name and emanating from a given node.

Nodes in the environment are attainable by following relationships. Operations are provided to *traverse a relationship*, that is, to follow a relationship from its source node to its target node.

4.3.4.1. Kinds of relationships

There are two kinds of relationships: primary and secondary. When a node is created, an initial relationship is established from some other node to the newly created node. This initial relationship is called the *primary relationship* to this new node, and the source node of this initial relationship is called the *parent* node. In addition, the new node will be connected back to this parent via a relationship of the predefined relation PARENT. There is no requirement that all primary relationships emanating from a node have the same relation name. Primary relationships form a strictly hierarchical tree; that is, for every node (except the root) there is one and only one sequence of primary relationships leading to it from the node that is the root of the tree. No cycles can be constructed using only primary relationships.

The primary relationship is broken by DELETE_NODE or DELETE_TREE operations. After deletion of the primary relationship to a node, the node is said to be *unobtainable*. A *non-existing node* is one which has never been created. RENAME operations may be used to make the primary relationship to a node emanate from a different node which becomes the new parent of the node. The operations DELETE_NODE, DELETE_TREE, RENAME, and the operations creating nodes are the only operations that manipulate primary relationships. They maintain a state in which each node has exactly one parent and a unique primary pathname (see Section 4.3.5).

Secondary relationships are arbitrary connections which may be established between two existing nodes; secondary relationships may form an arbitrary directed graph. User-defined secondary relationships are created with the LINK procedure and broken with the UNLINK procedure. Secondary relationships may exist to unobtainable nodes.

4.3.4.2. Basic predefined relations

The CAIS predefines certain relations. Relationships belonging to a predefined relation cannot be created, modified, or deleted by means of the CAIS interfaces and their relationship keys are the empty string, except where explicitly noted. The semantics of the predefined relations which are basic to the node model, as well as related concepts of the CAIS, are explained in this Section and Section 4.4. The basic predefined relations explained in this Section are USER, DEVICE, JOB, CURRENT_JOB, CURRENT_USER and CURRENT_NODE.

The CAIS node model incorporates the notion of a user. A *user* may be an individual, project, or other organizational entity; this notion is not equated with only an individual person. Each user has one *top-level node*. This top-level node is a structural node which represents the user and from it the user can access other structural, file and process nodes.

The CAIS node model incorporates the notion of a system. This notion provides the means of administering all the entities represented within one CAIS implementation. This notion implies the existence of a *system-level node* which acts as the root of the CAIS primary relationship tree spanning the entire node structure. Each top-level node is reachable from the system-level node along a primary relationship of the predefined relation USER emanating from the system-level node. The key of this relationship is the *user name*. Each user name has a top-level node associated with it. The system-level node cannot be accessed explicitly by the user via the CAIS interfaces. It may only be manipulated by interfaces outside the CAIS, e.g., to add new relationships of the predefined relation USER emanating from the system-level node.

The CAIS node model incorporates the notion of devices. Each device is described by a file node. This file node is reachable from the system-level node along a primary relationship of the predefined relation DEVICE emanating from the system-level node. The key of this relationship is the *device name*. The CAIS does not define interfaces for creating nodes which represent devices; such interfaces are to be provided outside the CAIS.

The CAIS node model incorporates the notion of a job. When a user logs onto the APSE or calls the CREATE_JOB procedure, a *root process node* is created which often represents a command interpreter or other user-communication process. It is left to each CAIS implementation to set up methodology for users to log onto the APSE and for enforcing any constraints that limit the top-level nodes at which users may log on. After logging onto the APSE, the user will be regarded by the CAIS as the user associated with the top-level node at which he logged on. A process node tree, spanned by primary relationships, develops from the root process node as other processes (called *dependent processes*) are initiated for the user. A particular user may have several root processes nodes concurrently. Each corresponding process node tree is referred to as a *job*. The predefined JOB relation is provided for locating each of the root process nodes from the user's top-level node. A primary relationship of the predefined relation JOB emanates from each user's top-level node to the root process node of each of the user's jobs. The key of this relationship is assigned by the mechanism of interpreting the LATEST_KEY constant (see Section 4.3.5) unless otherwise specified in the CREATE_JOB procedure call.

While the CAIS does not specify an interface for creating the initial root process node when a user logs onto the APSE, the effect is to be the same as a call to the CREATE_JOB procedure. The secondary relationships which the implementation must establish are found in TABLE VII. In particular, secondary relationships of the predefined relations USER and DEVICE must be established, with the appropriate user and device names as keys. These relationships emanate from the root process node being created to an implementation-defined subset of top-level nodes and file nodes representing devices, respectively. Dependent process nodes in the job inherit these relationships. File nodes representing devices and top-level nodes of other users can be reached from a process node via a relationship of the relation DEVICE or USER and a relationship key which is interpreted as the respective device or user name.

CURRENT_JOB, CURRENT_USER, and CURRENT_NODE are predefined relations which provide a convenient means for identifying other CAIS nodes. The relationship of the predefined relation CURRENT_JOB always points to the root process node of a process node's job. The relationship of the predefined relation CURRENT_USER always points to the user's top-level node. The relationship of the predefined relation CURRENT_NODE can be used to point to a node which represents the process's current focus or context for its activities. The process node can thus use the CURRENT_NODE for a base node when specifying pathnames (see Section 4.3.5). The CAIS requires that, when a root process node is created, it has a relationship of the predefined relation CURRENT_NODE pointing to the top-level node for the user.

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

4.3.5. Paths and pathnames

Every accessible node may be reached by following a sequence of relationships; this sequence is called the *path* to the node. A path starts at a known (not necessarily top-level) node and follows a sequence of relationships to a desired node. The path from the system-level node to a given node traversing only primary relationships is called the *unique primary path* to the given node.

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 traversed.

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.

Relation names and relationship keys follow the syntax of Ada identifiers. Upper and lower case are treated as equivalent within such identifiers. If the relationship key of a path element is the empty string, the parentheses may be omitted. Thus, 'PARENT and 'PARENT() refer to the same node.

The CAIS predefines the relation DOT. If the relation name in a path element is DOT, then the path element may be represented simply by a dot ('.') followed by the relationship key. Thus, 'DOT(TRACKER) is the same as .TRACKER. Relationship keys of relationships of the DOT relation may not be the empty string. Instances of the DOT relation may be manipulated by the user within access right constraints. Relationships of the DOT relation are not restricted to be primary relationships and are not associated with any other CAIS-specific semantics.

The starting point for interpretation of a pathname is always the current process node. A pathname may begin simply with a relationship key, not prefixed by either an apostrophe or '.'. This is taken to mean interpretation following a relationship emanating from the current node with the relation name DOT and with the given key. Thus LANDING_SYSTEM is the same as 'CURRENT_NODE.LANDING_SYSTEM.

For example, all of the following are legal node pathnames, and they would all refer to the same node if the relationship of the predefined relation points to the same node as 'USER(JONES).TRACKER and the relationship of the predefined relation points to the same node as 'USER(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)

A pathname may also be a ":". This is interpreted as referring to the current process node.

By convention, a relationship key ending in '#' is taken to represent the LATEST_KEY (lexicographically last). When creating a node or relationship, use of '#' to end the final relationship

key of a pathname will cause a relationship key to be automatically assigned, lexicographically following all previous relationship keys for the same relation and initial relationship key character sequence of relationships emanating from that particular node.

Identification of a node is provided by a pathname or by a given node and an identification of a relationship emanating from the given node by means of its relation name and relationship key. The phrase *to identify* means to provide an identification for a node. A node identification is considered an *illegal identification* if either the pathname or the relationship key or the relation name is syntactically illegal with respect to the syntax defined in Table I. An illegal identification is treated as an identification for a non-existing node.

A pathname implies *traversal of a node* if a relationship emanating from the node is traversed; consequently all nodes on the path to a node are traversed, while the node at the end of the path is not traversed. An identification that would require traversal of an unobtainable or inaccessible node is treated as the identification for a non-existing node.

The pathname associated with the unique primary path is called the *unique primary pathname* of the node. The unique primary pathname of the node is syntactically identical to, and therefore can be used as, a pathname whose interpretation starts at the current process node. It always starts with 'USER(user_name).

When identifying a node, use of "#" to end any relationship key in the pathname is interpreted as the relationship key of an existing relationship, lexicographically following all other keys for the same relation and initial relationship key character sequence of relationships emanating from that particular node.

Table I. Pathname BNF

```

path_name ::= relationship_key { path_element } |
              path_element { path_element } | :

path_element ::= 'relation_name' ( ( relationship_key ) ) |
                 .relationship_key

relation_name ::= Identifier

relationship_key ::= Identifier | [ Identifier ] #

```

Note: the relation name DOT must have a non-empty relationship key.

Notation:

1. Words - syntactic categories
2. [] - optional items
3. { } - an item repeated zero or more times
4. | - separates alternatives

4.3.6. 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 has a name and has a list of the values assigned to it, represented using the LIST_UTILITIES type called LIST_TYPE (see Section 5.4.1).

Relation names and attribute names both have the same form (that is, the syntax of an Ada Identifier). Relation names and node attribute names for a given node must be different from each other; relationship attribute names are in a separate name space.

The CAIS predefines certain attributes which are discussed in Section 5 and listed in Appendix A. Predefined attributes cannot be created, modified or deleted by the user, except where explicitly noted. The user can also create and manipulate user-defined attributes (see Section 5.1.3).

4.4. Discretionary and mandatory access control

The CAIS specifies mechanisms for discretionary and mandatory access control (see [TCSEC]). These specifications are only recommendations. Alternate discretionary or mandatory access control mechanisms can be substituted by an implementation provided that the semantics of all interfaces in Section 5 (with the exception of Section 5.1.4) are implemented as specified.

In the CAIS, *access control* refers to all the aspects of controlling access to information. It consists of:

- a. *access control rights* Descriptions of the kinds of operations which can be performed.
- b. *access control rules* The rules describing the correlations between access rights and those rights required for an intended operation.
- c. *access checking* The operation of checking granted access rights against those rights required for the intended operation according to the access control rules, and either permitting or denying the intended operation.

All of the information required to perform access checking is collectively referred to as *access control information*. The resulting restrictions placed on certain kinds of operations by access control are called *access rights constraints*.

4.4.1. Node access

In the CAIS, the following operations constitute *access to a node*:

- a. reading or writing of the contents of the node,
- b. reading or writing of attributes of the node,
- c. reading or writing of relationships emanating from a node or of their attributes, and
- d. traversing a node (see Section 4.3.5).

The phrase "reading relationships" is a convenient short-hand meaning either traversing relationships or reading their attributes. To access a node, then, means to perform any of the above access operations. The phrase "to obtain access" to a node means being permitted to perform certain operations on the node within access right constraints. Access to a node by means of a pathname can only be achieved if the current process has the respective access rights to the node as well as to any node traversed on the path to the node.

In the CAIS, the following operations do not constitute access to a node: closing node handles to a node, opening a node with intent EXISTENCE (see TABLE V), reading or writing of relationships of which a node is the target or of the attributes of such relationships, querying the kind of a node and querying the status of node handles to a node.

A node is *inaccessible* if the current process does not have sufficient discretionary access control rights to have knowledge of the node's existence or if mandatory access controls prevent information flow from the node to the current process. The property of inaccessibility is always relative to the access rights of the currently executing process, while the property of unobtainability is a property of the node alone.

PROPOSED MIL-STD-CAIS
31 JANUARY 1985

4.4.2. Discretionary access control

Discretionary access control is a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject [TCSEC].

In the CAIS, an *object* is any node to be accessed and a *subject* is any process (acting on the behalf of a given user) performing an operation requiring access to an object. Discretionary access control is used to limit access to nodes by processes running programs on behalf of users or groups of users.

An object can have established for it a secondary relationship of the predefined relation ACCESS which specifies the kinds of operations which may be performed on it. A process node may have a secondary relationship of the ADOPTED_ROLE relation established to the same target node as a predefined relation relationship. The information provided by these two kinds of relationships determines the approved access rights which the process has to the object (see Section 4.4.2.3). When the process tries to open the object node, the access rights implied by the INTENT parameter (see Section 5.1) are checked against these approved access rights to determine whether the process can perform the operation on that node.

4.4.2.1. Establishing grantable access rights

An object may be the source node of zero or more secondary relationships of the predefined relation ACCESS (called *access relationships*). Each access relationship has a predefined attribute, called GRANT, which specifies what access rights to the object are grantable to processes (subjects).

In order to limit the set of nodes to which access relationships can be established, the CAIS discretionary access control model requires that, upon creation of a root process node, secondary relationships of the predefined relation ALLOW_ACCESS be created. These relationships emanate from the created root process node to an implementation-defined set of nodes. The CAIS implementation must establish at least the secondary relationship of the predefined relation ALLOW_ACCESS with the user name as key from the root process node to the user top-level node. All such relationships are inherited by the process nodes created under the root process node.

Access relationships and GRANT attributes are established for objects in one of two ways: using the interfaces provided in the package ACCESS_CONTROL or at node creation.

The SET_ACCESS_CONTROL procedure can be used by a process to establish an access relationship between two nodes and to set the value of the GRANT attribute. This procedure can also be used to change the value of the GRANT attribute of an existing access relationship.

Access relationships are also established at node creation. The ACCESS_CONTROL parameter provides the necessary information in two parts. One part provides relationship keys which are used to identify the nodes which will be the targets of the new access relationships. If the current process node has a relationship of the relation ALLOW_ACCESS whose key is one of the keys given in the parameter, then the node identified by that relationship becomes the target of a new access relationship from the created node.

The other part of the ACCESS_CONTROL parameter gives a set of access rights for each relationship key. These access rights become the value of the GRANT attribute of the access relationship created with the corresponding key.

The ACCESS_CONTROL parameter specifies the initial access control information to be established

for a node being created using named Ada aggregate syntax; that is, it consists of a list of items each of which has a name (identifying a target node for an access relationship) followed by a list of values for the GRANT attribute.

For every relationship key named in the list for which the current process node has a relationship of the predefined relation ALLOW_ACCESS, a relationship of the predefined relation ACCESS with the given relationship key and the given access rights value for its GRANT attribute value is created from the new node to the target of the relationship of the predefined relation ALLOW_ACCESS.

4.4.2.2. Adopting a role

In the CAIS, a *role* is associated with a set of access rights that a subject can acquire when it acts under authority of that role. Each role is associated with a CAIS user, a program being executed, or a particular group of users, programs or subgroups. A subject (process) may act under the authority of several roles. Roles can be acquired dynamically.

In the CAIS a role is represented by a node; the associated access rights are determined by access relationships as described in the following sections. This node may be a top-level node representing a user, a node containing the executable image of a program, or a structural node representing a group. The structural node representing a group has relationships emanating from it to the nodes which represent the group's members.

Each group member is identified either by a primary relationship of the predefined relation PERMANENT_MEMBER or by a secondary relationship of the predefined relation POTENTIAL_MEMBER emanating from the group node. The phrase *permanent member* of a group refers to any node reachable from a node representing the group via primary relationships of the predefined relation PERMANENT_MEMBER. The relation PERMANENT_MEMBER may be used to create a hierarchy of nodes representing roles by defining members of a group that are themselves groups. A user top-level node may not be the target of a primary relationship of the predefined relation PERMANENT_MEMBER emanating from a group node due to the restriction that user top-level nodes can only have a primary relationship from the system-level node.

Secondary relationships of the predefined relation POTENTIAL_MEMBER are used to identify those members that may dynamically acquire membership in the group. The phrase *potential member* of a group refers to any node that is the target of a relationship of the predefined relation POTENTIAL_MEMBER from that group or from any of that group's permanent members.

When a process *adopts* a particular role, a secondary relationship of the predefined relation ADOPTED_ROLE is created from the process node to the node representing the role. There may be multiple relationships of the predefined relation ADOPTED_ROLE emanating from a process node. Roles are adopted either at creation of the process node or explicitly. When a process is created, it implicitly adopts the role represented by the file node containing an executable image of the program it is executing. When a root process node is created, it implicitly adopts the role represented by its current user node. When any process node is created, it implicitly inherits the relationships of the relation ADOPTED_ROLE of the node of its creating process. A process may explicitly adopt a role associated with a group using the ADOPT procedure (Section 5.1.4.4). For a process to adopt a role associated with a given group, a node representing some other adopted role of the process must be a potential member of the given group.

4.4.2.3. Evaluating access rights

The value of the GRANT attribute is a list whose syntax is given by the BNF in TABLE II. The necessary right is an access right, and the resulting rights are a list of access rights. An access right name has the syntax of an Ada identifier.

Table II. GRANT attribute value BNF

```
grant_attribute_value ::= ( [ grant_item { , grant_item } ] )  
grant_item ::= ([ necessary_right = > ] resulting_rights_list)  
necessary_right ::= identifier  
resulting_rights_list ::= identifier |  
                        ( identifier { , identifier } )
```

Notation:

1. Words - syntactic categories
2. [] - optional items
3. { } - an item repeated zero or more times
4. | - separates alternatives

The syntax is consistent with that given in Section 5.4. The interfaces in Section 5.4 can be used to construct and manipulate values of the GRANT attribute.

Checking of discretionary access control rights involves relevant grant items and approved access rights, both of which are derived from the values of GRANT attributes. For a given subject and object, *relevant grant items* are the grant items in values of GRANT attributes of relationships of the relation ACCESS emanating from the object and pointing at any node representing a role which is an adopted role of the process subject or representing a group one of whose permanent members is an adopted role of the process subject. *Approved access rights* are access rights whose names appear in resulting rights lists of relevant grant items for which either (1) the necessary right is null or (2) the necessary right is an approved access right.

For example, given a process node SUBJECT, an object OBJECT, and two nodes ROLE1 and ROLE2 representing roles, the following relationships might exist:

- a. a relationship of the relation ACCESS from OBJECT to ROLE1 with a GRANT attribute value of (READMAIL = > (READ, WRITE)).
- b. a relationship of the relation ACCESS from OBJECT to ROLE2 with a GRANT attribute value of (READMAIL).

EXCLUSIVE

Exist = EXISTENCE	R = READ	W = WRITE
RC = READ_CONTENTS	WC = WRITE_CONTENTS	AC = APPEND_CONTENTS
RA = READ_ATTRIBUTES	WA = WRITE_ATTRIBUTES	AA = APPEND_ATTRIBUTES
RR = READ_RELATIONSHIPS	WR = WRITE_ATTRIBUTES	AR = APPEND_RELATIONSHIPS
C = CONTROL	Exec = EXECUTE	

Reproduced from
best available copy.

specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to read, write or append relationships or to read, write or append access control information.

EXECUTE: Open and change_intent operations are delayed if the node contents are locked against read operations. The established access right for subsequent operations is the permission to initiate a process taking the node contents as executable image.

Open node handles can block other attempts to open other node handles or to change the intent of other node handles according to the rules demonstrated in FIGURE 2.

For EXCLUSIVE_WRITE_ATTRIBUTES, the node is locked against opens with intent to read, write or append attributes as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to read, write or append attributes.

APPEND_ATTRIBUTES, EXCLUSIVE_APPEND_ATTRIBUTES:

Open and change_intent operations are delayed if the node or its attributes are locked against append operations. The established access right for subsequent operations is to create node attributes.

For EXCLUSIVE_APPEND_ATTRIBUTES, the node is locked against opens with intent to write or append attributes as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to write or append attributes.

READ_RELATIONSHIPS, EXCLUSIVE_READ_RELATIONSHIPS:

Open and change_intent operations are delayed if the node or its relationships are locked against read operations. The established access right for subsequent operations is to read node relationships, including their attributes.

For EXCLUSIVE_READ_RELATIONSHIPS, the node is locked against opens with intent to write relationships as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to write relationships.

WRITE_RELATIONSHIPS, EXCLUSIVE_WRITE_RELATIONSHIPS:

Open and change_intent operations are delayed if the node or its relationships are locked against write operations. The established access right for subsequent operations is to write or create node relationships, including their attributes.

For EXCLUSIVE_WRITE_RELATIONSHIPS, the node is locked against opens with intent to read, write or append relationships as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to read, or write append relationships.

APPEND_RELATIONSHIPS, EXCLUSIVE_APPEND_RELATIONSHIPS:

Open and change_intent operations are delayed if the node or its relationships are locked against append operations. The established access right for subsequent operations is to create node relationships, including their attributes.

For EXCLUSIVE_APPEND_RELATIONSHIPS, the node is locked against opens with intent to write or append relationships as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to write or append relationships.

CONTROL, EXCLUSIVE_CONTROL:

Open and change_intent operations are delayed if the node or its relationships are locked against write or control operations. The established access right for subsequent operations is to read, write or append access control information.

For EXCLUSIVE_CONTROL, the node is locked against opens to read, write, or append relationships or to read, write, or append access control information as

or relationships are locked against write operations. The established access right for subsequent operations is to write, create or append to node contents, attributes and relationships.

For EXCLUSIVE_WRITE, the node is locked against opens with any read, write or append intent as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with read, write or append intent.

READ_CONTENTS, EXCLUSIVE_READ_CONTENTS:

Open and change_intent operations are delayed if the node or its contents are locked against read operations. The established access right for subsequent operations is to read the node contents.

For EXCLUSIVE_READ_CONTENTS, the node contents are locked against all opens with write intent as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to write its contents.

WRITE_CONTENTS, EXCLUSIVE_WRITE_CONTENTS:

Open and change_intent operations are delayed if the node or its contents are locked against write operations. The established access right for subsequent operations is to write or append to the node contents.

For EXCLUSIVE_WRITE_CONTENTS, the node contents are locked against opens with read, write or append intent as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to read, write or append to its contents.

APPEND_CONTENTS, EXCLUSIVE_APPEND_CONTENTS:

Open and change_intent operations are delayed if the node or its contents are locked against append operations. The established access right for subsequent operations is to append to the node contents.

For EXCLUSIVE_APPEND_CONTENTS, the node contents are locked against opens with append or write intent as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to append or write to its contents.

READ_ATTRIBUTES, EXCLUSIVE_READ_ATTRIBUTES:

Open and change_intent operations are delayed if the node or its attributes are locked against read operations. The established access right for subsequent operations is to read node attributes.

For EXCLUSIVE_READ_ATTRIBUTES, the node is locked against opens with intent to write attributes as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with intent to write attributes.

WRITE_ATTRIBUTES, EXCLUSIVE_WRITE_ATTRIBUTES:

Open and change_intent operations are delayed if the node or its attributes are locked against write operations. The established access right for subsequent operations is to modify and create node attributes.

SECURITY_VIOLATION is raised whenever an operation is attempted which violates mandatory access controls for 'write' operations. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

5.1.2. Package NODE_MANAGEMENT

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

The operations defined in this package are applicable to all nodes, relationships and attributes except where explicitly stated otherwise. These operations do not include the creation of nodes. The creation of structural nodes is performed by the CREATE_NODE procedures of package STRUCTURAL_NODES (see Section 5.1.5), the creation of nodes for processes is performed by INVOKE_PROCESS, SPAWN_PROCESS and CREATE_JOB of package PROCESS_CONTROL (see Section 5.2.2), and the creation of nodes for files is performed by the CREATE procedures of the input and output packages (see Section 5.3).

Three CAIS interfaces for manipulating node handles are: OPEN opens a node handle, CLOSE closes the node handle, and CHANGE_INTENT alters the specification of the intention of node handle usage. In addition, GET_PARENT, GET_CURRENT_NODE, GET_NEXT, OPEN_FILE_NODE and the node creation procedures also open node handles. These interfaces perform access synchronization in accordance with an intent specified by the parameter INTENT.

Operations which open node handles or change their intent are central to general node administration since they manipulate node handles and most other interfaces take node handles as parameters. While such other interfaces may also be provided in overloaded versions, taking pathnames as node identification, these overloaded versions are to be understood as including implicit OPEN calls with appropriate intent specification and a default TIME_LIMIT parameter. Subsequent uses of the phrase 'open operation' may refer to any of the OPEN, GET_CURRENT_NODE, GET_PARENT, GET_NEXT and OPEN_FILE_NODE operations.

One or more of the intents defined in TABLE V can be expressed by the INTENT parameters.

Table V. Intents

EXISTENCE: The established access right for subsequent operations is to query properties of the node handle and existence of the node only. Locks on the node have no delaying effect.

READ, EXCLUSIVE_READ:

Open and CHANGE_INTENT operations are delayed if the node, its contents, attributes or relationships are locked against read operations. The established access right for subsequent operations is to read node contents, attributes and relationships.

For EXCLUSIVE_READ, the node is locked against opens with any write intent as specified in FIGURE 2. Open and change_intent operations are additionally delayed if there are open node handles to the node with write intent.

WRITE, EXCLUSIVE_WRITE:

Open and change_intent operations are delayed if the node, its contents, attributes

subtype FORM_STRING is STRING;

NODE_TYPE describes the type for node handles. NODE_KIND is the enumeration of the kinds of nodes. INTENT_SPECIFICATION describes the usage of node handles and is further explained in Section 5.1.2. INTENTION is the type of the parameter INTENT of CAIS procedures which open or change the intent of a node handle, as further explained in Section 5.1.2.

NAME_STRING, RELATIONSHIP_KEY, RELATION_NAME, and FORM_STRING are subtypes for pathnames, relationship keys, and relation names, as well as for form strings (see [LRM] 14). Value of these string subtypes are subject to certain syntactic restrictions whose violation causes exceptions to be raised.

```
CURRENT_USER : constant NAME_STRING := "CURRENT_USER";  
CURRENT_NODE : constant NAME_STRING := "CURRENT_NODE";  
CURRENT_PROCESS : constant NAME_STRING := ":";  
LATEST_KEY : constant RELATIONSHIP_KEY := "#";  
DEFAULT_RELATION : constant RELATION_NAME := "DOT";  
NO_DELAY : constant DURATION := DURATION'FIRST
```

CURRENT_USER, CURRENT_NODE, and CURRENT_PROCESS are standard pathnames for the current user's top-level node, current node, and current process, respectively. LATEST_KEY and DEFAULT_RELATION are standard names for the latest key and the default relation name, respectively. NO_DELAY is a constant of type DURATION (see [LRM] 9.6) used for time limits.

```
NAME_ERROR : exception;  
USE_ERROR : exception;  
STATUS_ERROR : exception;  
LOCK_ERROR : exception;  
INTENT_VIOLATION : exception;  
ACCESS_VIOLATION : exception;  
SECURITY_VIOLATION : exception;
```

NAME_ERROR is raised whenever an attempt is made to access a node via a pathname or node handle while the node does not exist, it is unobtainable, discretionary access control constraints for knowledge of existence of a node are violated, or mandatory access controls for 'read' operations are violated. This exception takes precedence over ACCESS_VIOLATION and SECURITY_VIOLATION exceptions.

USE_ERROR is raised whenever a restriction on the use of an interface is violated.

STATUS_ERROR is raised whenever the open status of a node handle does not conform to expectations.

LOCK_ERROR is raised whenever an attempt is made to modify or lock a locked node.

INTENT_VIOLATION is raised whenever an operation is attempted on an open node handle which is in violation of the intent associated with the open node handle.

ACCESS_VIOLATION is raised whenever an operation is attempted which violates access right constraints other than knowledge of existence of the node. ACCESS_VIOLATION is raised only if the conditions for NAME_ERROR are not present.

5. DETAILED REQUIREMENTS

The following detailed requirements shall be fulfilled in a manner consistent with the model descriptions given in Section 4 of this standard.

5.1. General node management

This section describes the CAIS interfaces for the general manipulation of nodes, relationships and attributes. These interfaces are defined in five CAIS packages: `NODE_DEFINITIONS` defines types, subtypes, exceptions, and constants used throughout the CAIS; `NODE_MANAGEMENT` defines interfaces for general operations on nodes and relationships; `ATTRIBUTES` defines interfaces for general operations on attributes; `ACCESS_CONTROL` defines interfaces for setting and adopting access rights; and `STRUCTURAL_NODES` defines interfaces for the creation of structural nodes.

Specialized interfaces for the manipulation of process and file nodes and of their relationships and attributes are defined in Sections 5.2 and 5.3, respectively.

To simplify manipulation by Ada programs, an Ada type `NODE_TYPE` is defined for values that represent an internal handle for a node (referred to as a *node handle*). Objects of this type can be associated with a node by means of CAIS procedures, causing an *open node handle* to be assigned to the object. While such an association is in effect, the node handle is said to be open; otherwise, the node handle is said to be closed. Most procedures expect either a parameter of type `NODE_TYPE`, a pathname, or a combination of a base node (specified by a parameter `BASE` of type `NODE_TYPE`) and a path element relative to it, to identify a node.

An open node handle is guaranteed always to refer to the same node, regardless of any changes to relationships that could cause pathnames to become invalid or to refer to different nodes. This behavior is referred to as the *tracking* of nodes by open node handles.

5.1.1. Package NODE_DEFINITIONS

This package defines the Ada type `NODE_TYPE`. It also defines certain enumeration and string types and exceptions useful for node manipulations.

```
type NODE_TYPE is limited private;

type NODE_KIND is (FILE, STRUCTURAL, PROCESS);

type INTENT_SPECIFICATION is
  (EXISTENCE, READ, WRITE, READ_ATTRIBUTES, WRITE_ATTRIBUTES,
   APPEND_ATTRIBUTES, READ_RELATIONSHIPS, WRITE_RELATIONSHIPS,
   APPEND_RELATIONSHIPS, READ_CONTENTS, WRITE_CONTENTS,
   APPEND_CONTENTS, CONTROL, EXECUTE, EXCLUSIVE_READ,
   EXCLUSIVE_WRITE, EXCLUSIVE_READ_ATTRIBUTES,
   EXCLUSIVE_WRITE_ATTRIBUTES, EXCLUSIVE_APPEND_ATTRIBUTES,
   EXCLUSIVE_READ_RELATIONSHIPS, EXCLUSIVE_WRITE_RELATIONSHIPS,
   EXCLUSIVE_APPEND_RELATIONSHIPS, EXCLUSIVE_READ_CONTENTS,
   EXCLUSIVE_WRITE_CONTENTS, EXCLUSIVE_APPEND_CONTENTS,
   EXCLUSIVE_CONTROL);

type INTENTION is array (POSITIVE range <>) of INTENT_SPECIFICATION;

subtype NAME_STRING is STRING;
subtype RELATIONSHIP_KEY is STRING;
subtype RELATION_NAME is STRING;
```

4.5.2. Pragmatics for SEQUENTIAL IO

A CAIS Implementation must support generic instantiation of this package with any (non-limited) constrained Ada type whose maximum size in bits (as defined by the attribute `ELEMENT_TYPE'SIZE`) is at least $2^{15}-1$. A conforming implementation must also support instantiation with unconstrained record types which have default constraints and a maximum size in bits of at least $2^{15}-1$. It may (but need not) use variable length elements to conserve space in the external file.

4.5.3. Pragmatics for 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 $2^{15}-1$.

A CAIS Implementation must support generic instantiation of this package with any (non-limited) constrained Ada type whose maximum size in bits (as defined by the attribute `ELEMENT_TYPE'SIZE`) is at least $2^{15}-1$. A conforming implementation must also support instantiation with unconstrained record types which have default constraints and a maximum size in bits of at least $2^{15}-1$. It may (but need not) use variable length elements to conserve space in the external file.

4.5.4. Pragmatics for TEXT IO

A CAIS Implementation must support files with at least $2^{15}-1$ records/lines in total and at least $2^{15}-1$ lines per page. A CAIS Implementation must support at least 255 columns per line.

assigned to the file node. The attribute `LOWEST_CLASSIFICATION` defines the lowest allowable object classification label that may be assigned to the file node.

When a file node representing the device is opened, the device inherits its security classification label from the first process performing the open operation. If it is not possible to label the node representing the device within the bounds of the attributes `HIGHEST_CLASSIFICATION` and `LOWEST_CLASSIFICATION`, the operation fails by raising the exception `SECURITY_VIOLATION`.

4.4.3.5. Mandatory access checking

When access control is enforced for a given operation, mandatory access control rules are checked. If mandatory access controls are not satisfied, the operation terminates by raising the exception `SECURITY_VIOLATION`, except where the indication of failure constitutes violation of mandatory access control rules for "read" operations, in which case `NAME_ERROR` may be raised.

4.5. Pragmatics

This section provides several minimum values for implementation-determined quantities and sizes.

4.5.1. Pragmatics for CAIS node model

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. However, it is important to establish certain minimum values for each type to enhance portability.

NAME_STRING

At least 255 characters must be supported in a CAIS pathname.

RELATIONSHIP_KEY

At least 80 leading characters must be significant in a relationship key.

ATTRIBUTE_NAME, RELATION_NAME

At least 80 leading characters must be significant in attribute and relation names.

Tree height At least 10 levels of hierarchy must be supported for the primary relationships.

Record size number

At least $2^{15}-1$ bits per record must be supported.

Open node count

Each process must be able to have at least 127 nodes open simultaneously.

List

At least $2^{15}-1$ bits per list must be supported.

Table IV. Classification attribute value BNF

```

object_classification ::= classification
subject_classification ::= classification
classification ::= ( hierarchical_classification,
                    non_hierarchical_categories )
hierarchical_classification ::= keyword
non_hierarchical_categories ::= ( [ keyword { , keyword } ] )
keyword ::= identifier

```

Notation:

1. Words - syntactic categories
2. [] - optional items
3. { } - an item repeated zero or more times
4. | - separates alternatives

4.4.3.2. Labeling of process nodes

When a root process is created, it is assigned subject and object classification labels. The method by which these initial labels are assigned is not specified; however, the labels *shall accurately represent security levels of the specific /users/ with which they are associated* [TCSEC]. When any non-root (dependent) process node is created, the creator may specify the classification attributes associated with the node. If no classification is specified, the classification is inherited from the creator. The assigned classification must adhere to the requirements for mandatory access control over write operations.

4.4.3.3. Labeling of non-process nodes

When a non-process object is created, it is assigned an object classification label. The classification label may be specified in the create operation, or it may be inherited from the parent. The assigned classification must adhere to the requirements for mandatory access control over write operations.

4.4.3.4. Labeling of nodes for devices

Certain file nodes representing devices may have a range of classification levels. The classification label of the node of the first process opening a handle to one of these nodes is assigned to the file node while there are any open node handles to the file node. Only when all open node handles have been closed can a new classification label be assigned to the file node.

The range of classification levels is specified by two predefined CAIS node attributes. The attribute HIGHEST_CLASSIFICATION defines the highest allowable object classification label that may be

Each subject and object is assigned zero or more non-hierarchical categories which represent coexisting classifications. A subject may obtain read access to an object if the set of non-hierarchical categories assigned to the subject contains each category assigned to the object. Likewise, a subject may obtain write access to an object if each of the non-hierarchical categories assigned to the subject are included in the set of categories assigned to the object.

A subject must satisfy both hierarchical and non-hierarchical access rights rules to obtain access to an object.

In the CAIS, subjects are CAIS processes, while an object may be any CAIS node. Operations are CAIS operations and are classified as read, write, or read/write operations. Access checking is performed at the time the operation is requested by comparing the classification of the subject with that of the object with respect to the type of operation.

4.4.3.1. Labeling of CAIS nodes

The labeling of nodes is provided by predefined node attributes. A predefined attribute, called SUBJECT_CLASSIFICATION, is assigned to each process node and represents the process' classification as a subject. A predefined attribute, called OBJECT_CLASSIFICATION, is assigned to each node and represents the node's classification as an object. These attributes have a limited function and cannot be read or written directly through the CAIS interfaces. The value of the attribute is a parenthesized list containing two items, the hierarchical classification level and the non-hierarchical category list. The hierarchical classification is a keyword member of the ordered set of hierarchical classification keywords. The non-hierarchical category list is a list of zero or more keyword members of the set of non-hierarchical categories. The hierarchical classification level set and the non-hierarchical category set are implementation-defined. For example, the following are possible classification attribute values:

(TOP_SECRET, (MAIL_USER, OPERATOR, STAFF))

(UNCLASSIFIED, ())

(SECRET, (STAFF))

The BNF for the value of a classification attribute (and of the LEVEL parameter which provides it at node creation) is given in Table IV.

READ	This is the union of READ_RELATIONSHIPS, READ_ATTRIBUTES, READ_CONTENTS and EXISTENCE access rights. This access right is necessary to open the object with intent READ. It is sufficient to open the object with intent READ_RELATIONSHIPS, READ_ATTRIBUTES or READ_CONTENTS.
WRITE	This is the union of WRITE_RELATIONSHIPS, WRITE_ATTRIBUTES, WRITE_CONTENTS and EXISTENCE access rights. This access right is necessary to open the object with intent WRITE. It is sufficient to open the object with intent WRITE_RELATIONSHIPS, WRITE_ATTRIBUTES or WRITE_CONTENTS.
APPEND	This is the union of APPEND_RELATIONSHIPS, APPEND_ATTRIBUTES, APPEND_CONTENTS and EXISTENCE access rights. This access right is necessary to open the object with intent APPEND. It is sufficient to open the object with intent APPEND_RELATIONSHIPS, APPEND_ATTRIBUTES or APPEND_CONTENTS.
EXECUTE	The subject may create a process that takes the contents of the object as its executable image; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with intent EXECUTE.
CONTROL	The subject may modify access control information of the object; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with intent CONTROL.

4.4.2.4. Discretionary access checking

CAIS access control rules state that any access right required for a subject to access an object must be contained in the set of approved access rights of that object with respect to that subject. The CAIS model allows discretionary access checking to be performed at the time a node handle is opened. At this point access rights implied by the INTENT parameter of the open operation must be a subset of the approved access rights. If this is not the case, the operation is terminated and an exception is raised. For subsequent access using the node handle, the access rights required may be compared to the rights implied by the intent, rather than the approved access rights.

4.4.3. Mandatory access control

Mandatory access control provides access controls based directly on a comparison of the individual's clearance or authorization for the information and the classification or sensitivity designation of the information being sought [TCSEC].

A mandatory access control classification may be either a hierarchical classification level or a non-hierarchical category. A hierarchical classification level is chosen from an ordered set of classification levels and represents either the sensitivity of the object or the trustworthiness of the subject. In hierarchical classification, the reading of information flows downward towards less sensitive areas, while the creating of information flows upward towards more trustworthy individuals. A subject may obtain read access to an object if the hierarchical classification of the subject is greater than or equal to that of the object. In turn, to obtain write access to the object, a subject's hierarchical classification must be less than or equal to the hierarchical classification of the object.

Table III. Predefined access rights

EXISTENCE The minimum access rights without which the object is inaccessible to the subject. Without additional access rights the subject may neither read nor write attributes, relationships or contents of the object.

READ_RELATIONSHIPS

The subject may read attributes of relationships emanating from the object or use it for traversal to another node; the access right **EXISTENCE** is implicitly granted. This access right is necessary to open the object with intent **READ_RELATIONSHIPS**.

WRITE_RELATIONSHIPS

The subject may create or delete relationships emanating from the object or may create, delete, or modify attributes of these relationships; the access right **EXISTENCE** is implicitly granted. This access right is necessary to open the object with intent **WRITE_RELATIONSHIPS**.

APPEND_RELATIONSHIPS

The subject may create relationships emanating from the object and attributes of these relationships; the access right **EXISTENCE** is implicitly granted. This access right is necessary to open the object with intent **APPEND_RELATIONSHIPS**.

READ_ATTRIBUTES

The subject may read attributes of the object; the access right **EXISTENCE** is implicitly granted. This access right is necessary to open the object with intent **READ_ATTRIBUTES**.

WRITE_ATTRIBUTES

The subject may create, write, or delete attributes of the object; the access right **EXISTENCE** is implicitly granted. This access right is necessary to open the object with intent **WRITE_ATTRIBUTES**.

APPEND_ATTRIBUTES

The subject may create attributes of the object; the access right **EXISTENCE** is implicitly granted. This access right is necessary to open the object with intent **APPEND_ATTRIBUTES**.

READ_CONTENTS

The subject may read contents of the object; the access right **EXISTENCE** is implicitly granted. This access right is necessary to open the object with intent **READ_CONTENTS**.

WRITE_CONTENTS

The subject may write contents of the object; the access right **EXISTENCE** is implicitly granted. This access right is necessary to open the object with intent **WRITE_CONTENTS** granted. This access right is necessary to open the object with intent **READ_CONTENTS**.

APPEND_CONTENTS

The subject may append contents of the object; the access right **EXISTENCE** is implicitly granted. This access right is necessary to open the object with intent **APPEND_CONTENTS**.

- c. a relationship of the relation **ADOPTED_ROLE** from **SUBJECT** to **ROLE1**, and
- d. a relationship of the relation **ADOPTED_ROLE** from **SUBJECT** to **ROLE2**.

The relevant grant items are *READMAIL* and *READMAIL=>(READ, WRITE)*. The approved access rights for **SUBJECT** to access **OBJECT** are (1) *READMAIL* because the necessary rights of the relevant grant item of the access relationship to **ROLE2** is null and (2) *READ* and *WRITE* because the necessary right, *READMAIL*, of the relevant grant item of the access relationship to **ROLE1** is approved. **FIGURE 1** shows a graphic representation of these relationships.

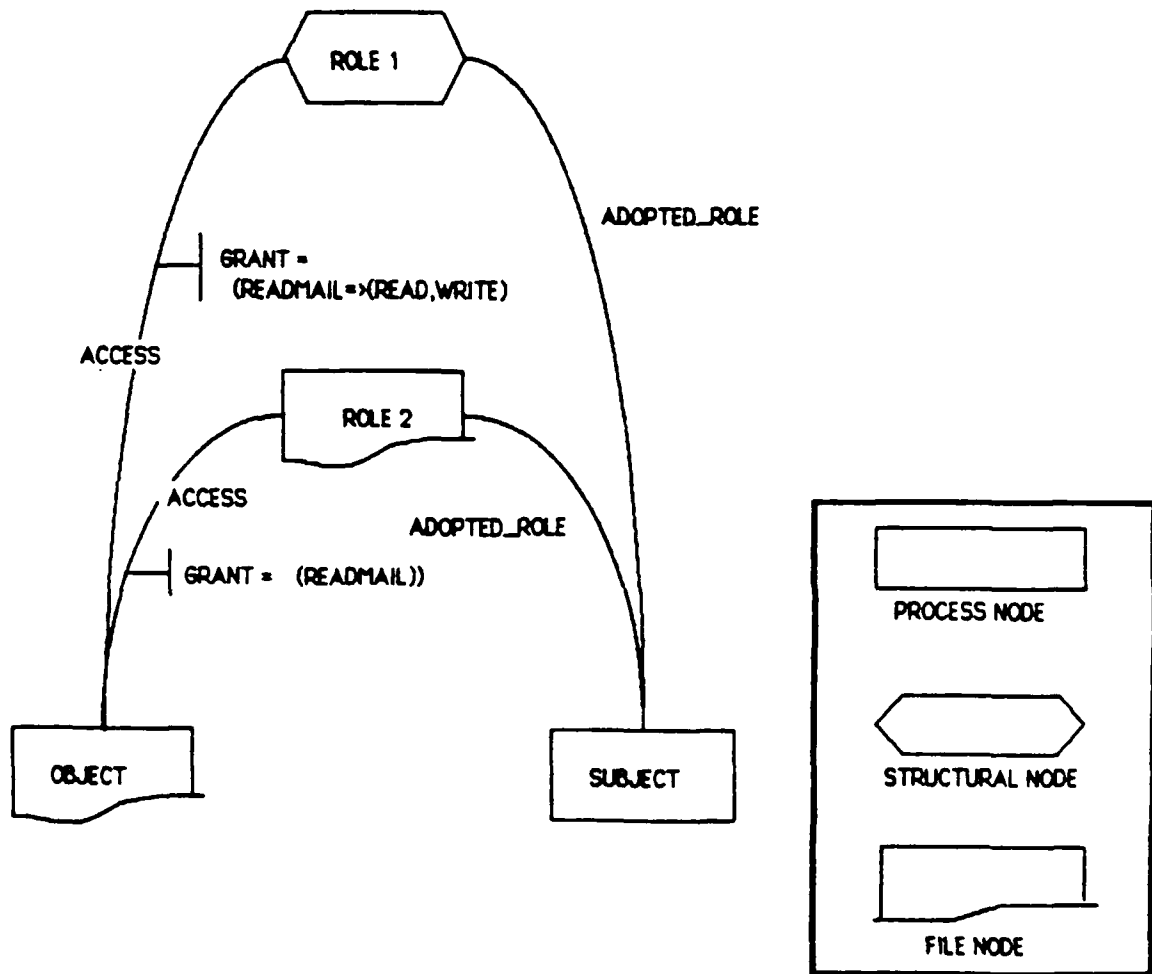


Figure 1. Access relationships

Access rights may be user-defined, but certain access rights have special significance to CAIS operations. In particular, the CAIS recognizes the access rights given in Table III and the kinds of access for which they are necessary or sufficient.

Reproduced from
best available copy.

5.1.2.1. Opening a node handle

```

procedure OPEN(NODE:      in out NODE_TYPE;
               NAME:      in  NAME_STRING;
               INTENT:     in  INTENTION :=
                               (1 => READ);
               TIME_LIMIT: in  DURATION := NO_DELAY);

procedure OPEN(NODE:      in out NODE_TYPE;
               BASE:      in  NODE_TYPE;
               KEY:       in  RELATIONSHIP_KEY;
               RELATION:  in  RELATION_NAME :=
                               DEFAULT_RELATION;
               INTENT:     in  INTENTION := (1 => READ);
               TIME_LIMIT: in  DURATION := NO_DELAY);

```

Purpose:

These procedures return an open node handle in NODE to the node identified by the pathname NAME or BASE/KEY/RELATION, respectively. The INTENT parameter determines the access rights available for subsequent uses of the node handle; it also establishes access synchronization with other users of the node. The TIME_LIMIT parameter allows the specification of a time limit for the delay imposed on OPEN by the existence of locks on the node. A delayed OPEN call completes after the node is unlocked or the specified time limit has elapsed. In the latter case, the exception LOCK_ERROR is raised.

Parameters:

NODE is a node handle, initially closed, to be opened to the identified node.

NAME is the pathname identifying the node to be opened.

BASE is an open node handle to a base node for node identification.

KEY is the relationship key for node identification.

RELATION is the relation name for node identification.

INTENT is the intent of subsequent operations on the node; the actual parameter takes the form of an array aggregate.

TIME_LIMIT is a value of type DURATION, specifying a time limit for the delay on waiting for the unlocking of a node in accordance with the desired INTENT.

Exceptions:

NAME_ERROR

is raised if the pathname specified by NAME is syntactically illegal or if any traversed node in the path specified by name is unobtainable, inaccessible or non-existent or if the relationship specified by RELATION and KEY or by the last path element of NAME does not exist. NAME_ERROR is also raised if the node to which a handle is to be opened is inaccessible or unobtainable and the given INTENT includes any intent other than EXISTENCE.

USE_ERROR is raised if the specified INTENT is an empty array.

STATUS_ERROR

is raised if the node handle **NODE** is already open prior to the call on **OPEN** or if **BASE** is not an open node handle.

LOCK_ERROR

is raised if the **OPEN** operation is delayed beyond the specified time limit due to the existence of locks in conflict with the specified **INTENT**. This includes any delays caused by locks on nodes traversed on the path specified by the pathname **NAME** or locks on the node identified by **BASE**, preventing the reading of relationships emanating from these nodes.

INTENT_VIOLATION

is raised if **BASE** was not opened with an intent establishing the right to read relationships.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to traverse the path specified by **NAME** by **BASE**, **KEY** and **RELATION** or to obtain access to the node consistent with the specified **INTENT**. **ACCESS_VIOLATION** is raised only if the conditions for **NAME_ERROR** are not present.

SECURITY_VIOLATION

is raised if the attempt to obtain access to the node with the specified **INTENT** represents a violation of mandatory access controls for the CAIS. **SECURITY_VIOLATION** is raised only if the conditions for other exceptions are not present.

Notes:

An open node handle acts as if the handle forms an unnamed temporary secondary relationship to the node; this means that, if the node identified by the open node handle is renamed (potentially by another process), the open node handle tracks the renamed node.

It is possible to open a node handle to an unobtainable node or to an inaccessible node. The latter is consistent with the fact that the existence of a relationship emanating from an accessible node to which the user has **READ_RELATIONSHIPS** rights cannot be hidden from the user.

5.1.2.2. Closing a node handle

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

Purpose:

This procedure severs any association between the node handle **NODE** and the node and releases any associated lock on the node imposed by the intent of the node handle **NODE**. Closing an already closed node handle has no effect.

Parameter:

NODE is a node handle, initially open, to be closed.

Exceptions:

none

Notes:

A NODE_TYPE variable must be closed before another OPEN can be called using the same NODE_TYPE variable as an actual parameter to the formal NODE parameter of OPEN.

5.1.2.3. Changing the intention regarding node handle usage

```
procedure CHANGE_INTENT (NODE:      in out NODE_TYPE;  
                        INTENT:      in  INTENTION;  
                        TIME_LIMIT:  in  DURATION :=  
                                      NO_DELAY);
```

Purpose:

This procedure changes the intention regarding use of the node handle NODE. It is semantically equivalent to closing the node handle and reopening the node handle to the same node with the INTENT and TIME_LIMIT parameters of CHANGE_INTENT, except that CHANGE_INTENT guarantees to return an open node handle that refers to the same node as the node handle input in NODE (see the issue explained in the note below).

Parameter:

NODE is an open node handle

INTENT is the intent of subsequent operations on the node; the actual parameter takes the form of an array aggregate.

TIME_LIMIT is a value of type DURATION, specifying a time limit for the delay on waiting for the unlocking of a node in accordance with the desired INTENT.

Exceptions:

NAME_ERROR

is raised if the node handle NODE refers to an unobtainable node and INTENT contains any intent specification other than EXISTENCE.

STATUS_ERROR

is raised if the node handle NODE is not an open node handle.

LOCK_ERROR

is raised if the operation is delayed beyond the specified time limit due to the existence of locks on the node in conflict with the specified INTENT.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to obtain access to the node consistent with the specified INTENT. ACCESS_VIOLATION is raised only if the condition for NAME_ERROR is not present.

SECURITY_VIOLATION

is raised if the attempt to obtain access consistent with the intention INTENT to the node specified by NODE represents a violation of mandatory access controls for

the CAIS. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Notes:

Use of the sequence of a CLOSE and an OPEN operation instead of a CHANGE_INTENT operation cannot guarantee that the same node is opened, since relationships, and therefore the node identification, may have changed since the previous OPEN on the node.

5.1.2.4. Examining the open status of a node handle

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

Purpose:

This function returns TRUE if the node handle NODE is open; otherwise, it returns FALSE.

Parameter:

NODE is a node handle.

Exceptions:

None.

5.1.2.5. Querying the intention of a node handle

```
function INTENT_OF(NODE: in NODE_TYPE)
    return INTENTION;
```

Purpose:

This function returns the intent with which the node handle NODE is open.

Parameter:

NODE is an open node handle.

Exception:

STATUS_ERROR
 is raised if the node handle NODE is not open.

5.1.2.6. Querying the kind of a node

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

Purpose:

This function returns the kind of a node, either FILE, PROCESS or STRUCTURAL.

Parameter:

NODE is an open node handle.

Exceptions:

STATUS_ERROR

is raised if the node handle NODE is not open.

5.1.2.7. Obtaining the unique primary pathname

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

Purpose:

This function returns the unique primary name of the node identified by NODE.

Parameter:

NODE is an open node handle identifying the node.

Exceptions:

NAME_ERROR

is raised if any node traversed on the primary path to the node is inaccessible.

STATUS_ERROR

is raised if the node handle NODE is not open.

LOCK_ERROR

is raised if access consistent with Intent READ_RELATIONSHIPS to any node traversed on the primary path cannot be obtained due to an existing lock on the node.

INTENT_VIOLATION

is raised if NODE was not opened with an Intent establishing the right to read relationships.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to traverse the node's primary path. ACCESS_VIOLATION is raised only if the conditions for NAME_ERROR are not present.

5.1.2.8. Obtaining the relationship key of a primary relationship

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

Purpose:

This function returns the relationship key of the last path element of the unique primary pathname of the node.

Parameter:

NODE is an open node handle identifying the node.

Exceptions:

NAME_ERROR

is raised if the parent node of the node identified by NODE is inaccessible.

STATUS_ERROR

is raised if the node handle NODE is not open.

LOCK_ERROR

is raised if the parent node is locked against reading relationships.

INTENT_VIOLATION

is raised if the node handle NODE was not opened with an intent establishing the right to read relationships.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to obtain access to the node's parent consistent with intent READ_RELATIONSHIP. ACCESS_VIOLATION is raised only if the conditions for NAME_ERROR are not present.

5.1.2.9. Obtaining the relation name of a primary relationship

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

Purpose:

This function returns the relation name of the last path element of the unique primary pathname of the node.

Parameter:

NODE is an open node handle identifying the node.

Exceptions:

NAME_ERROR

is raised if the parent node of the node identified by NODE is inaccessible.

STATUS_ERROR

is raised if the node handle NODE is not open.

LOCK_ERROR

is raised if the parent node is locked against reading relationships.

INTENT_VIOLATION

is raised if NODE was not opened with an intent establishing the right to read relationships.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to obtain access to the node's parent consistent with intent to READ_RELATIONSHIPS. ACCESS_VIOLATION is raised only if the conditions for NAME_ERROR are not present.

5.1.2.10. Obtaining the relationship key of the last relationship traversed

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

Purpose:

This function returns the relationship key of the relationship corresponding to the last path element of the pathname used in opening this node handle. Since a path element is a string, the relationship key is returned even if the relationship has been deleted.

Parameter:

NODE is an open node handle.

Exceptions:

STATUS_ERROR
is raised if the node handle NODE is not open.

5.1.2.11. Obtaining the relation name of the last relationship traversed

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

Purpose:

This function returns the relation name of the relationship corresponding to the last path element of the pathname used in opening this node handle. The relation name is returned even if the relationship has been deleted.

Parameter:

NODE is an open node handle.

Exceptions:

STATUS_ERROR
is raised if the node handle NODE is not open.

5.1.2.12. Obtaining a partial pathname

```
function BASE_PATH(NAME: in NAME_STRING)
    return NAME_STRING;
```

Purpose:

This function returns the pathname obtained by deleting the last path element from NAME. It does not establish whether the pathname identifies an existing node; only the syntactic properties of the pathname are examined. This function also checks the syntactic legality of the pathname NAME.

Parameters:

NAME is a pathname (not necessarily identifying a node).

Exceptions:

NAME_ERROR

is raised if NAME is a syntactically illegal pathname.

5.1.2.13. Obtaining the name of the last relationship in a pathname

```
function LAST_RELATION(NAME: in NAME_STRING)
  return RELATION_NAME;
```

Purpose:

This function returns the name of the relation of the last path element of the pathname NAME. It does not establish whether the pathname identifies an existing node; only the syntactic properties of the pathname are examined. This function also checks the syntactic legality of the pathname NAME.

Parameters:

NAME is a pathname (not necessarily identifying a node).

Exceptions:

NAME_ERROR

is raised if NAME is a syntactically illegal pathname.

5.1.2.14. Obtaining the key of the last relationship in a pathname

```
function LAST_KEY(NAME: in NAME_STRING)
  return RELATIONSHIP_KEY;
```

Purpose:

This function returns the relationship key of the last path element of the pathname NAME. It does not establish whether the pathname identifies an existing node; only the syntactic properties of the pathname are examined. This function checks the syntactic legality of the pathname NAME.

Parameters:

NAME is a pathname (not necessarily identifying a node).

Exceptions:

NAME_ERROR

is raised if NAME is a syntactically illegal pathname.

5.1.2.15. Querying the existence of a node

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

Purpose:

This function returns FALSE if the node identified by NODE is unobtainable or inaccessible. It returns TRUE otherwise.

Parameters:

NODE is an open node handle identifying the node.

Exceptions:

STATUS_ERROR
 is raised if NODE is not an open node handle.

Additional Interfaces:

```
function IS_OBTAINABLE(NAME: in NAME_STRING)
    return BOOLEAN
is
    NODE:  NODE_TYPE;
    RESULT: BOOLEAN;
begin
    OPEN(NODE, NAME, (1=>EXISTENCE));
    RESULT := IS_OBTAINABLE(NODE);
    CLOSE(NODE);
    return RESULT;
exception
    when others => return FALSE;
end IS_OBTAINABLE;

function IS_OBTAINABLE(BASE:      in NODE_TYPE;
                       KEY:        in RELATIONSHIP_KEY;
                       RELATION:   in RELATION_NAME := DEFAULT_RELATION)
    return BOOLEAN
is
    NODE:  NODE_TYPE;
    RESULT: BOOLEAN;
begin
    OPEN(NODE, BASE, KEY, RELATION, (1=>EXISTENCE));
    RESULT := IS_OBTAINABLE(NODE);
    CLOSE(NODE);
    return RESULT;
exception
    when others => return FALSE;
end IS_OBTAINABLE;
```

Notes:

OBTAINABLE can be used to determine whether a node identified via a secondary relationship has been made unobtainable by a DELETE operation or is inaccessible to the current process (see Note in Section 5.1.2.3).

5.1.2.10. Querying sameness

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

Purpose:

This function returns TRUE if the nodes identified by its arguments are the same node; otherwise, it returns FALSE.

Parameters:

NODE1 is an open node handle to a node.

NODE2 is an open node handle to a node.

Exceptions:

STATUS_ERROR

is raised if at least one of the node handles, NODE1 and NODE2, is not open.

Additional Interface:

```
function IS_SAME(NAME1: in NAME_STRING;
                 NAME2: in NAME_STRING)
    return BOOLEAN
is
    NODE1, NODE2: NODE_TYPE;
    RESULT:      BOOLEAN;
begin
    OPEN(NODE1, NAME1, (1=>EXISTENCE));
    begin
        OPEN(NODE2, NAME2, (1=>EXISTENCE));
    exception
        when others =>
            CLOSE(NODE1);
            raise;
    end;
    RESULT := IS_SAME(NODE1, NODE2);
    CLOSE(NODE1);
    CLOSE(NODE2);
    return RESULT;
end IS_SAME;
```

Notes:

Sameness is not to be confused with equality of attribute values, relationships and contents of nodes, which is a necessary but not a sufficient criterion for sameness.

5.1.2.17. Obtaining an open node handle to the parent node

```
procedure GET_PARENT(PARENT:    in out NODE_TYPE;
                    NODE:       in  NODE_TYPE;
                    INTENT:      in  INTENTION := (1=>READ);
                    TIME_LIMIT: in  DURATION  := NO_DELAY);
```

Purpose:

This procedure returns an open node handle in PARENT to the parent node of the node identified by the open node handle NODE. The intent under which the node handle PARENT is opened is specified by INTENT. A call on GET_PARENT is equivalent to a call OPEN(PARENT, NODE, "", PARENT, INTENT, TIME_LIMIT).

Parameters:

PARENT is a node handle, initially closed, to be opened to the parent node.

NODE is an open node handle identifying the node.

INTENT is the intent of subsequent operations on the node handle PARENT.

TIME_LIMIT is a value of type DURATION, specifying a time limit for the delay on waiting for the unlocking of the parent node in accordance with the desired INTENT.

Exceptions:

NAME_ERROR

is raised if the node identified by NODE is a top-level node or if its parent node is inaccessible.

USE_ERROR is raised if the specified INTENT is an empty array.

STATUS_ERROR

is raised if the node handle PARENT is open prior to the call or if the node handle NODE is not open.

LOCK_ERROR

is raised if the opening of the parent node is delayed beyond the specified TIME_LIMIT due to the existence of locks in conflict with the specified INTENT.

INTENT_VIOLATION

is raised if NODE was not opened with an intent establishing the right to read relationships.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to obtain access to the parent node with the specified INTENT. ACCESS_VIOLATION is raised only if the conditions for NAME_ERROR are not present.

SECURITY_VIOLATION

is raised if the attempt to gain with the specified INTENT access to the parent node represents a violation of mandatory access controls for the CAIS. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

5.1.2.18. Copying a node

```
procedure COPY_NODE (FROM:      in NODE_TYPE;  
                     TO_BASE:   in NODE_TYPE;  
                     TO_KEY:    in RELATIONSHIP_KEY;  
                     TO_RELATION: in RELATION_NAME  
                      :=DEFAULT_RELATION);
```

Purpose:

This procedure copies a file or structural node that does not have emanating primary relationships. The node copied is identified by the open node handle FROM and is copied to a newly created node. The new node is identified by the combination of the TO_BASE, TO_KEY and TO_RELATION parameters. The newly created node is of the same kind as the node identified by FROM. If the node is a file node, its contents are also copied, i.e., a new copied file is created. Any secondary relationships emanating from the original node, excepting the relationship of the predefined relation PARENT (which is appropriately adjusted), are recreated in the copy. If the target of the original node's relationship is the node itself, then the copy has an analogous relationship to itself. Any other secondary relationship whose target is the original node is unaffected. All attributes of the FROM node are also copied. Regardless of any locks on the node identified by FROM, the newly created node is unlocked.

Parameters:

FROM is an open node handle to the node to be copied.

TO_BASE is an open node handle to the base node for identification of the node to be created.

TO_KEY is a relationship key for the identification of the node to be created.

TO_RELATION
is a relation name for the identification of the node to be created.

Exceptions:

NAME_ERROR

is raised if the new node identification is illegal or if a node already exists with the identification given for the new node.

USE_ERROR is raised if the original node is not a file or structural node or if any primary relationships emanate from the original node. **USE_ERROR** is also raised if **TO_RELATION** is the name of a predefined relation that cannot be modified or created by the user.

STATUS_ERROR

is raised if the node handles **FROM** and **TO_BASE** are not both open.

INTENT_VIOLATION

is raised if **FROM** was not opened with an intent establishing the right to read contents, attributes, and relationships or if **TO_BASE** was not opened with an intent establishing the right to append relationships. **INTENT_VIOLATION** is not raised if the conditions for **NAME_ERROR** are present.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls and the conditions for other exceptions are not present.

Additional Interface:

```
procedure COPY_NODE(FROM: in NODE_TYPE;
                    TO: in NAME_STRING)
is
    TO_BASE: NODE_TYPE;
begin
    OPEN(TO_BASE, BASE_PATH(TO), (1=>APPEND_RELATIONSHIPS));
    COPY_NODE(FROM, TO_BASE, LAST_KEY(TO), LAST_RELATION(TO));
    CLOSE(TO_BASE);
exception
    when others =>
        CLOSE(TO_BASE);
        raise;
end COPY_NODE;
```


5.1.2.19. Copying trees

```
procedure COPY_TREE (FROM:      in NODE_TYPE;  
                     TO_BASE:   in NODE_TYPE;  
                     TO_KEY:    in RELATIONSHIP_KEY;  
                     TO_RELATION: in RELATION_NAME  
                      := DEFAULT_RELATION);
```

Purpose:

This procedure copies a tree of file or structural nodes formed by primary relationships emanating from the node identified by the open node handle FROM. Primary relationships are recreated between corresponding copied nodes. The root node of the newly created tree corresponding to the FROM node is the node identified by the combination of the TO_BASE, TO_KEY and TO_RELATION parameters. If an exception is raised by the procedure, none of the nodes are copied. Secondary relationships, attributes, and node contents are copied as described for COPY_NODE with the following additional rules: secondary relationships between two nodes which both are 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 original target node. Secondary relationships emanating from a node which is not copied, but which refer to nodes inside the tree being copied, are unaffected. If the node identified by TO_BASE is part of the tree to be copied, then the copy of the node identified by FROM will not be copied recursively.

Parameters:

FROM is an open node handle to the root node of the tree to be copied.

TO_BASE is an open node handle to the base node for identification of the node to be created as root of the new tree.

TO_KEY is a relationship key for the identification of the node to be created as root of the new tree.

TO_RELATION is a relation name for the identification of the node to be created as root of the new tree.

Exceptions:

NAME_ERROR is raised if the new node identification is illegal or if a node already exists with the identification given for the new node to be created as a copy of the node identified by FROM.

STATUS_ERROR is raised if the node handles FROM and TO_BASE are not both open.

USE_ERROR is raised if the original node is not a file or structural node. USE_ERROR is also raised if TO_RELATION is the name of a predefined relation that cannot be modified or created by the user.

LOCK_ERROR is raised if any node to be copied except the node identified by FROM, is locked against read access to attributes, relationships or contents.

INTENT_VIOLATION

is raised if FROM is not open with an intent establishing the right to read node contents, attributes and relationships or if TO_BASE is not open with an intent establishing the right to append relationships. INTENT_VIOLATION is only raised if the conditions for NAME_ERROR are not present.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to obtain access to each node to be copied with intent READ. ACCESS_VIOLATION is not raised if conditions for NAME_ERROR are present.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls and the conditions for other exceptions are not present.

Additional Interface:

```
procedure COPY_TREE(FROM: in NODE_TYPE;  
                    TO:   in NAME_STRING)  
is  
    TO_BASE: NODE_TYPE;  
begin  
    OPEN(TO_BASE, BASE_PATH(TO), (1=>APPEND_RELATIONSHIPS));  
    COPY_TREE(FROM, TO_BASE, LAST_KEY(TO), LAST_RELATION(TO));  
    CLOSE(TO_BASE);  
exception  
    when others =>  
        CLOSE(TO_BASE);  
        raise;  
end COPY_TREE;
```

5.1.2.20. Renaming the primary relationship of a node

```
procedure RENAME(NODE:      in NODE_TYPE;  
                 NEW_BASE:  in NODE_TYPE;  
                 NEW_KEY:   in RELATIONSHIP_KEY;  
                 NEW_RELATION: in RELATION_NAME  
                  :=DEFAULT_RELATION);
```

Purpose:

This procedure renames a file or structural node. It deletes the primary relationship to the node identified by NODE and installs a new primary relationship to the node, emanating from the node identified by NEW_BASE, with key and relation name given by the NEW_KEY and NEW_RELATION parameters. The parent relationship is changed accordingly. This changes the unique primary pathname of the node. Existing secondary relationships with the renamed node as target track the renaming, i.e., they have the renamed node as target.

Parameters:

NODE is an open node handle to the node to be renamed.

NEW_BASE is an open node handle to the base node from which the new primary relationship to the renamed node emanates.

NEW_KEY is a relationship key for the new primary relationship.

Parameters:

NODE is an open node handle to a node whose attribute is to be deleted.

ATTRIBUTE is the name of the attribute to be deleted.

Exceptions:

USE_ERROR is raised if the node does not have an attribute of the given name. **USE_ERROR** is also raised if **ATTRIBUTE** is the name of a predefined node attribute which cannot be modified or created by the user.

STATUS_ERROR
is raised if the node handle **NODE** is not open.

INTENT_VIOLATION
is raised if **NODE** was not opened with an intent establishing the right to write attributes.

SECURITY_VIOLATION
is raised if the operation represents a violation of mandatory access controls. **SECURITY_VIOLATION** is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure DELETE_NODE_ATTRIBUTE(NAME      : in NAME_STRING;  
                                ATTRIBUTE: in ATTRIBUTE_NAME)  
is  
  NODE: NODE_TYPE;  
begin  
  OPEN(NODE, NAME, (1=>WRITE_ATTRIBUTES));  
  DELETE_NODE_ATTRIBUTE(NODE, ATTRIBUTE);  
  CLOSE(NODE);  
exception  
  when others =>  
    CLOSE(NODE);  
    raise;  
end DELETE_NODE_ATTRIBUTE;
```

5.1.3.4. Deleting path attributes

```
procedure DELETE_PATH_ATTRIBUTE(BASE:      in NODE_TYPE;  
                                KEY:       in RELATIONSHIP_KEY;  
                                RELATION:  in RELATION_NAME  
                                :=DEFAULT_RELATION;  
                                ATTRIBUTE: in ATTRIBUTE_NAME);
```

Purpose:

This procedure deletes an attribute, named by **ATTRIBUTE**, of a relationship identified by the base node **BASE**, the relation name **RELATION** and the relationship key **KEY**.

Parameters:

BASE is an open node handle to the node from which the relationship emanates.

ATTRIBUTE is the attribute name.

VALUE is the initial value of the attribute.

Exceptions:

NAME_ERROR

is raised if the relationship identified by the BASE, KEY and RELATION parameters does not exist.

USE_ERROR is raised if the relationship already has an attribute of the given name or if the attribute name given is syntactically illegal. USE_ERROR is also raised if RELATION is the name of a predefined relation that cannot be modified by the user. USE_ERROR is also raised if ATTRIBUTE is the name of a predefined relationship attribute which cannot be created by the user.

STATUS_ERROR

is raised if the node handle BASE is not open.

INTENT_VIOLATION

is raised if BASE was not opened with an intent establishing the right to write relationships.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure CREATE_PATH_ATTRIBUTE(NAME:      in NAME_STRING;
                                ATTRIBUTE: in ATTRIBUTE_NAME;
                                VALUE:      in LIST_TYPE)
is
    BASE: NODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(NAME), (1=>WRITE_RELATIONSHIPS));
    CREATE_PATH_ATTRIBUTE(BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
                          ATTRIBUTE, VALUE);
    CLOSE(BASE);
exception
    when others =>
        CLOSE(BASE);
        raise;
end CREATE_PATH_ATTRIBUTE;
```

5.1.3.3. Deleting node attributes

```
procedure DELETE_NODE_ATTRIBUTE(NODE:      in NODE_TYPE;
                                ATTRIBUTE: in ATTRIBUTE_NAME);
```

Purpose:

This procedure deletes an attribute, named by ATTRIBUTE, of the node identified by the open node handle NODE.

name given is syntactically illegal. **USE_ERROR** is also raised if **ATTRIBUTE** is the name of a predefined node attribute which cannot be created by the user.

STATUS_ERROR

is raised if the node handle **NODE** is not open.

INTENT_VIOLATION

is raised if **NODE** was not opened with an intent establishing the right to append attributes.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. **SECURITY_VIOLATION** is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure CREATE_NODE_ATTRIBUTE(NAME:      in NAME_STRING;
                                ATTRIBUTE: in ATTRIBUTE_NAME;
                                VALUE:     in LIST_TYPE)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (1=>APPEND_ATTRIBUTES));
  CREATE_NODE_ATTRIBUTE(NODE, ATTRIBUTE, VALUE);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end CREATE_NODE_ATTRIBUTE;
```

5.1.3.2. Creating path attributes

```
procedure CREATE_PATH_ATTRIBUTE(BASE:      in NODE_TYPE;
                                KEY:       in RELATIONSHIP_KEY;
                                RELATION:  in RELATION_NAME
                                           :=DEFAULT_RELATION;
                                ATTRIBUTE: in ATTRIBUTE_NAME;
                                VALUE:     in LIST_TYPE);
```

Purpose:

This procedure creates an attribute, named by **ATTRIBUTE**, of a relationship and sets its initial value to **VALUE**. The relationship is identified by the base node identified by the open node handle **BASE**, the relation name **RELATION** and the relationship key **KEY**.

Parameters:

BASE is an open node handle to the node from which the relationship emanates.

KEY is the relationship key of the affected relationship.

RELATION is the relation name of the affected relationship.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to obtain access to the current node with the specified INTENT. ACCESS_VIOLATION is raised only if the conditions for NAME_ERROR are not present.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions other exceptions are not present.

Notes:

The call on GET_CURRENT_NODE is equivalent to OPEN(NODE, "CURRENT_NODE", (INTENT, TIME_LIMIT)).

5.1.3. Package ATTRIBUTES

This package supports the definition and manipulation of attributes for nodes and relationships. The name of an attribute follows the syntax of an Ada identifier. The value of each attribute is a list; the format of the list is defined by the package LIST_UTILITIES (see Section 5.4). Upper and lower case distinctions are not significant within the attribute names.

Unless stated otherwise, the attributes predefined by the CAIS cannot be created, deleted or modified by the user.

The operations defined for the manipulation of attributes identify the node to which an attribute belongs either by pathname or open node handle. They implicitly identify a relationship to which an attribute belongs by the last path element of a pathname or explicitly identify the relationship by base node, key and relation name identification.

5.1.3.1. Creating node attributes

```
procedure CREATE_NODE_ATTRIBUTE(NODE:      in NODE_TYPE;  
                                ATTRIBUTE: in ATTRIBUTE_NAME;  
                                VALUE:      in LIST_TYPE);
```

Purpose:

This procedure creates an attribute named by ATTRIBUTE of the node identified by the open node handle NODE and sets its initial value to VALUE.

Parameters:

NODE is an open node handle to a node to receive the new attribute.

ATTRIBUTE is the name of the attribute.

VALUE is the initial value of the attribute.

Exceptions:

USE_ERROR is raised if the node already has an attribute of the given name or if the attribute

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure SET_CURRENT_NODE(NAME: in NAME_STRING)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (1=>EXISTENCE));
  SET_CURRENT_NODE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end SET_CURRENT_NODE;
```

5.1.2.30. Opening a node handle to the current node.

```
procedure GET_CURRENT_NODE(NODE:      in out NODE_TYPE;
                           INTENT:    in   INTENTION:=
                                       (1=>EXISTENCE);
                           TIME_LIMIT: in DURATION:=NO_DELAY);
```

Purpose:

This procedure returns in NODE an open node handle to the current node of the current process; the intent with which the node handle is opened as specified by the INTENT parameter.

Parameter:

NODE is a node handle, initially closed, to be opened to the current node.

INTENT is the intent of subsequent operations on the node handle NODE.

TIME_LIMIT is a value of type DURATION specifying a time limit for the delay on waiting for the unlocking of the node in accordance with the desired INTENT.

Exceptions:

NAME_ERROR

is raised if the current node is inaccessible or if it is unobtainable and the INTENT is anything other than EXISTENCE.

USE_ERROR is raised if INTENT is an empty array.

STATUS_ERROR

is raised if NODE is an open node handle prior to the call.

LOCK_ERROR

is raised if access, with intent READ_RELATIONSHIPS, to the current process node cannot be obtained due to an existing lock on the node.

INTENT is the Intent of subsequent operations on the node handle NEXT_NODE.

TIME_LIMIT is a value of type DURATION, specifying a time limit for the delay on waiting for the unlocking of the node in accordance with the desired INTENT.

Exceptions:

NAME_ERROR

is raised if the node whose node handle is to be returned in by NEXT_NODE is unobtainable and if the INTENT includes any intent other than EXISTENCE.

USE_ERROR is raised if the ITERATOR has not been previously set by ITERATE or if the iterator is exhausted (i.e., MORE (ITERATOR)=FALSE) or if INTENT is an empty array.

LOCK_ERROR

is raised if the opening of the node is delayed beyond the specified TIME_LIMIT due to the existence of locks in conflict with the specified INTENT.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to obtain access to the next node with the specified INTENT. Access Violation is raised only if the conditions for NAME_ERROR are not present.

SECURITY_VIOLATION

is raised if the current process' attempt to obtain access to the next node with the specified INTENT represents a violation of mandatory access controls for the CAIS. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

5.1.2.29. Setting the current node relationship

procedure SET_CURRENT_NODE(NODE: in NODE_TYPE);

Purpose:

This procedure specifies the node identified by NODE as the current node. The relationship of the predefined relation CURRENT_NODE of the current process is changed accordingly.

Parameters:

NODE is an open node handle to a node to be the new target of the CURRENT_NODE relationship emanating from the current process node.

Exceptions:

STATUS_ERROR

is raised if the node handle NODE is not open.

LOCK_ERROR

is raised if access, with intent WRITE_RELATIONSHIPS, to the current process node cannot be obtained due to an existing lock on the node.


```
RELATION:    in RELATION_NAME_PATTERN
              := DEFAULT_RELATION;
PRIMARY_ONLY: in BOOLEAN := TRUE)

is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (1=>READ_RELATIONSHIPS));
  ITERATE(ITERATOR, NODE, KIND, KEY, RELATION, PRIMARY_ONLY);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end ITERATE;
```

Notes:

The functions PATH_KEY and PATH_RELATION may be used to determine the relationship which caused the node to be included in the iteration. The iteration interfaces can be used to determine relationships to inaccessible or unobtainable nodes.

5.1.2.27. Determining iteration status

```
function MORE (ITERATOR: in NODE_ITERATOR)
return BOOLEAN;
```

Purpose:

The function MORE returns FALSE if all nodes contained in the node iterator have been retrieved with the GET_NEXT procedure; otherwise it returns TRUE.

Parameters:

ITERATOR is a node iterator previously set by the procedure ITERATE.

Exceptions:

USE_ERROR is raised if the ITERATOR has not been previously set by the procedure ITERATE.

5.1.2.28. Getting the next node in an iteration

```
procedure GET_NEXT(ITERATOR: in out NODE_ITERATOR;
NEXT_NODE: in out NODE_TYPE;
INTENT: in INTENTION := (1=>EXISTENCE);
TIME_LIMIT: in DURATION := NO_DELAY);
```

Purpose:

The procedure GET_NEXT returns an open node handle to the next node in the parameter NEXT_NODE; the intent under which the node handle is opened is specified by the INTENT parameter. If NEXT_NODE is open prior to the call to GET_NEXT, it is closed prior to being opened to the next node. A time limit can be specified for the maximum delay permitted if the node to be opened is locked against access with the specified INTENT.

Parameters:

ITERATOR is a node iterator previously set by ITERATE.

NEXT_NODE

is a node handle to be opened to the next node on the ITERATOR.

relationship key. The effect on existing iterators of creation or deletion of relationships is implementation-defined.

5.1.2.26. Creating an iterator over nodes

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

Purpose:

This procedure establishes a node iterator ITERATOR over the set of nodes that are the targets of relationships emanating from a given node identified by NODE and matching the specified KEY and RELATION patterns. Nodes that are of a different kind than the KIND specified are omitted by subsequent calls to GET_NEXT using the resulting ITERATOR. If PRIMARY_ONLY is true, then the iterator will be based on only primary relationships.

Parameters:

ITERATOR is the node iterator returned.

NODE is an open node handle to a node whose relationships form the basis for constructing the iterator.

KIND is the kind of nodes on which the iterator is based.

KEY is the pattern for the relationship keys on which the iterator is based.

RELATION is the pattern for the relation names on which the iterator is based.

PRIMARY_ONLY is a boolean; if TRUE, the iterator will be based on only primary relationships; if FALSE, the iterator will be based on all relationships satisfying the patterns.

Exceptions:

USE_ERROR is raised if the pattern given in KEY or RELATION is syntactically illegal.

STATUS_ERROR is raised if NODE is not an open node handle.

INTENT_VIOLATION is raised if NODE was not opened with an intent establishing the right to read relationships.

Additional Interface:

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

RELATION is the relation name of the relationship to be deleted.

Exceptions:

NAME_ERROR

is raised if the relationship identified by **BASE**, **KEY** and **RELATION** does not exist.

USE_ERROR is raised if the specified relationship is a primary relationship. **USE_ERROR** is also raised if **RELATION** is the name of a predefined relation that cannot be modified or created by the user.

STATUS_ERROR

is raised if the **BASE** is not an open node handle.

INTENT_VIOLATION

is raised if **BASE** was not opened with an intent establishing the right to write relationships.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. **SECURITY_VIOLATION** is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure UNLINK(NAME: in NAME_STRING)
is
  BASE: NODE_TYPE;
begin
  OPEN(BASE, BASE_PATH(NAME), (1=>WRITE_RELATIONSHIPS));
  UNLINK(BASE, LAST_KEY(NAME), LAST_RELATION(NAME));
  CLOSE(BASE);
exception
  when others =>
    CLOSE(BASE);
    raise;
end UNLINK;
```

Notes:

UNLINK can be used to delete secondary relationships to nodes that have become unobtainable.

5.1.2.25. Node iteration types and subtypes

```
type NODE_ITERATOR is limited private;
subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
subtype RELATION_NAME_PATTERN is RELATION_NAME;
```

These types are used in the following interfaces for iterating over a set of nodes. **RELATIONSHIP_KEY_PATTERN** and **RELATION_NAME_PATTERN** follow the syntax of relationship keys and relation names, except that '?' will match any single character and '*' will match any string of characters. **NODE_ITERATOR** is a private type assumed to contain the bookkeeping information necessary for the implementation of the **MORE** and **GET_NEXT** functions. The nodes are returned by **GET_NEXT** in ASCII lexicographical order by relation name and then by

31 JANUARY 1985

NAME_ERROR

is raised if the relationship key or the relation name are illegal or if a node already exists with the identification given by NEW_BASE, NEW_KEY, and NEW_RELATION.

USE_ERROR is raised if NEW_RELATION is the name of a predefined relation that cannot be modified or created by the user.

STATUS_ERROR

is raised if the node handles NODE and NEW_BASE are not open.

INTENT_VIOLATION

is raised if NEW_BASE was not opened with an intent establishing the right to append relationships.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```

procedure LINK(NODE:      in NODE_TYPE;
                NEW_NAME: in NAME_STRING)
is
    NEW_BASE: NODE_TYPE;
begin
    OPEN(NEW_BASE, BASE_PATH(NEW_NAME), (1=>APPEND_RELATIONSHIPS));
    LINK(NODE, NEW_BASE, LAST_KEY(NEW_NAME),
                                                LAST_RELATION(NEW_NAME));
    CLOSE(NEW_BASE);
exception
    when others =>
        CLOSE(NEW_BASE);
        raise;
end LINK;

```

5.1.2.24. Deleting secondary relationships

```

procedure UNLINK(BASE:      in NODE_TYPE;
                  KEY:        in RELATIONSHIP_KEY;
                  RELATION: in RELATION_NAME
                           :=DEFAULT_RELATION);

```

Purpose:

This procedure deletes a secondary relationship identified by the BASE, KEY and RELATION parameters.

Parameters:

BASE Is an open node handle to the node from which the relationship emanates which is to be deleted.

KEY Is the relationship key of the relationship to be deleted.

31 JANUARY 1985

ACCESS_VIOLATION

is raised if the current process does not have sufficient discretionary access control rights to obtain access to the parent of the node specified by **NODE** with intent **WRITE_RELATIONSHIPS** or to obtain access to any target node of a primary relationship to be deleted with intent **EXCLUSIVE_WRITE** and the conditions for **NAME_ERROR** are not present.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. **SECURITY_VIOLATION** is raised only if the conditions for other exceptions are not present.

Additional Interface:

```

procedure DELETE_TREE(NAME: in NAME_STRING)
is
    NODE: NODE_TYPE;
begin
    OPEN(NODE, NAME, (EXCLUSIVE_WRITE,READ_RELATIONSHIPS));
    DELETE_TREE(NODE);
exception
    when others =>
        CLOSE(NODE);
        raise;
end DELETE_TREE;

```

Notes:

This operation can be used to delete more than one primary relationship in a single operation.

5.1.2.23. Creating secondary relationships

```

procedure LINK(NODE:      in NODE_TYPE;
               NEW_BASE:  in NODE_TYPE;
               NEW_KEY:    in RELATIONSHIP_KEY;
               NEW_RELATION: in RELATION_NAME
                               :=DEFAULT_RELATION);

```

Purpose:

This procedure creates a secondary relationship between two existing nodes. The procedure takes a node handle **NODE** on the target node, a node handle **NEW_BASE** on the source node, and an explicit key **NEW_KEY** and relation name **NEW_RELATION** for the relationship to be established from **NEW_BASE** to **NODE**.

Parameters:

NODE is an open node handle to the node to which the new secondary relationship points.

NEW_BASE is an open node handle to the base node from which the new secondary relationship to the node emanates.

NEW_KEY is the relationship key for the new secondary relationship.

NEW_RELATION
is the relation name for the new secondary relationship.

Exceptions:

```
        NODE: NODE_TYPE;  
begin  
    OPEN(NODE, NAME, (EXCLUSIVE_WRITE, READ_RELATIONSHIPS));  
    DELETE_NODE(NODE);  
exception  
    when others =>  
        CLOSE(NODE);  
        raise;  
end DELETE_NODE;
```

Notes:

The DELETE_NODE operations cannot be used to delete more than one primary relation node in a single operation. It is left to an implementation decision whether and when nodes whose primary relationships have been broken are actually removed. However, secondary relationships to such nodes must remain until they are explicitly deleted using the UNLINK procedures.

5.1.2.22. Deleting the primary relationships of a tree

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

Purpose:

This procedure effectively performs the DELETE_NODE operation for a specified node and recursively applies DELETE_TREE to all nodes reachable by a unique primary pathname from the designated node. The nodes whose primary relationships are to be deleted are opened with intent EXCLUSIVE_WRITE, thus locking them for other operations. The order in which the deletions of primary relationships is performed is not specified. If the DELETE_TREE operation raises an exception, none of the primary relationships is deleted.

Parameters:

NODE is an open node handle to the node at the root of the tree whose primary relationships are to be deleted.

Exceptions:

NAME_ERROR

is raised if the parent node of the node identified by NODE or any of the target nodes of primary relationships to be deleted are inaccessible.

USE_ERROR is raised if the primary relationship to the node identified by NODE belongs to a predefined relation that cannot be modified by the user.

STATUS_ERROR

is raised if the node handle NODE is not open prior to the call.

LOCK_ERROR

is raised if a node handle to the parent of the node specified by NODE cannot be opened with intent WRITE_RELATIONSHIPS or if a node handle identifying any node whose unique primary path traverses the node identified by NODE cannot be opened with intent EXCLUSIVE_WRITE.

INTENT_VIOLATION

is raised if the node handle NODE was not opened with an intent including EXCLUSIVE_WRITE and READ_RELATIONSHIPS.

```
        raise;  
    end RENAME;
```

Notes:

Open node handles from existing processes track the renamed node.

5.1.2.21. Deleting the primary relationship to a node

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

Purpose:

This procedure deletes the primary relationship to a node identified by NODE. The node becomes unobtainable. The node handle NODE is closed. If the node is a process node and the process is not yet TERMINATED (see Section 5.2), DELETE_NODE aborts the process.

Parameters:

NODE is an open node handle to the node which is the target of the primary relationship to be deleted.

Exceptions:

NAME_ERROR
 is raised if the parent node of the node identified by NODE is inaccessible.

USE_ERROR is raised if any primary relationships emanate from the node.

STATUS_ERROR
 is raised if the node handle NODE is not open prior to the call.

LOCK_ERROR
 is raised if access, with intent WRITE_RELATIONSHIPS, to the parent of the node to be deleted cannot be obtained due to an existing lock on the node.

INTENT_VIOLATION
 is raised if the node handle NODE was not opened with an intent including EXCLUSIVE_WRITE and READ_RELATIONSHIPS.

ACCESS_VIOLATION
 is raised if the current process does not have sufficient discretionary access control rights to obtain access to the parent of the node to be deleted with intent WRITE_RELATIONSHIPS and the conditions for NAME_ERROR are not present.

SECURITY_VIOLATION
 is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure DELETE_NODE(NAME: in NAME_STRING)  
is
```

NEW_RELATION

is a relation name for the new primary relationship.

Exceptions:

NAME_ERROR

is raised if the new node identification is illegal or if a node already exists with the identification given for the new node.

USE_ERROR is raised if the node identified by NODE is not a file or structural node or if the renaming cannot be accomplished while still maintaining acircularity of primary relationships (e.g., if the new parent node would be the renamed node). USE_ERROR is also raised if NEW_RELATION is the name of a predefined relation that cannot be modified or created by the user or if the primary relationship to be deleted belongs to a predefined relation that cannot be modified by the user.

STATUS_ERROR

is raised if the node handles NODE and NEW_BASE are not open.

LOCK_ERROR

is raised if access, with intent WRITE_RELATIONSHIPS, to the parent of the node to be deleted cannot be obtained due to an existing lock on the node.

INTENT_VIOLATION

is raised if NODE was not opened with an intent establishing the right to write relationships or if NEW_BASE was not opened with an intent establishing the right to append relationships.

ACCESS_VIOLATION

is raised if the current process does not have sufficient discretionary access control rights to obtain access to the parent of the node to be renamed with intent WRITE_RELATIONSHIPS and the conditions for NAME_ERROR are not present.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure RENAME(NODE: in NODE_TYPE;
                 NEW_NAME: in NAME_STRING)
is
  NEW_BASE: NODE_TYPE;
begin
  OPEN(NEW_BASE, BASE_PATH(NEW_NAME), (1=>APPEND_RELATIONSHIPS));
  RENAME (NODE, NEW_BASE, LAST_KEY(NEW_NAME),
          LAST_RELATION(NEW_NAME));
  CLOSE(NEW_BASE);
exception
  when others =>
    CLOSE(NEW_BASE);
```


Parameters:

NODE is an open node handle to a node the value of whose attribute named by **ATTRIBUTE** is to be set.

ATTRIBUTE is the name of the attribute.

VALUE is the new value of the attribute.

Exceptions:

USE_ERROR is raised if the node has no attribute of the given name. **USE_ERROR** is also raised if **ATTRIBUTE** is the name of a predefined node attribute which cannot be modified by the user.

STATUS_ERROR
is raised if **NODE** is not an open node handle.

INTENT_VIOLATION
is raised if **NODE** was not opened with an intent establishing the right to write attributes.

SECURITY_VIOLATION
is raised if the operation represents a violation of mandatory access controls. **SECURITY_VIOLATION** is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure SET_NODE_ATTRIBUTE(NAME:      in NAME_STRING;
                             ATTRIBUTE: in ATTRIBUTE_NAME;
                             VALUE:     in LIST_TYPE)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (1=>WRITE_ATTRIBUTES));
  SET_NODE_ATTRIBUTE(NODE, ATTRIBUTE, VALUE);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end SET_NODE_ATTRIBUTE;
```

5.1.3.6. Setting path attributes

```
procedure SET_PATH_ATTRIBUTE(BASE      : in NODE_TYPE;
                              KEY       : in RELATIONSHIP_KEY;
                              RELATION  : in RELATION_NAME
                                      :=DEFAULT_RELATION;
                              ATTRIBUTE : in ATTRIBUTE_NAME;
                              VALUE     : in LIST_TYPE);
```

Purpose:

This procedure sets the value of the relationship attribute named by **ATTRIBUTE** to the value

specified by VALUE. The relationship is identified explicitly by the base node BASE, the relation name RELATION and the relationship key KEY.

Parameters:

BASE is an open node handle to the node from which the relationship emanates.

KEY is the relationship key of the affected relationship.

RELATION is the relation name of the affected relationship.

ATTRIBUTE is the name of the attribute.

VALUE is the new value of the attribute.

Exceptions:

NAME_ERROR

is raised if the relationship identified by the BASE, KEY and RELATION parameters does not exist.

USE_ERROR is raised if the node does not have an attribute of the given name. USE_ERROR is also raised if RELATION is the name of a predefined relation that cannot be modified by the user. USE_ERROR is also raised if ATTRIBUTE is the name of a predefined relationship attribute which cannot be modified by the user.

STATUS_ERROR

is raised if the node handle BASE is not open.

INTENT_VIOLATION

is raised if BASE was not opened with an intent establishing the right to write relationships.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure SET_PATH_ATTRIBUTE(NAME:      in NAME_STRING;
                             ATTRIBUTE: in ATTRIBUTE_NAME;
                             VALUE:     in LIST_TYPE)
is
  BASE: NODE_TYPE;
begin
  OPEN(BASE, BASE_PATH(NAME), (1=>WRITE_RELATIONSHIPS));
  SET_PATH_ATTRIBUTE(BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
                    ATTRIBUTE, VALUE);
  CLOSE(BASE);
exception
  when others =>
    CLOSE(BASE);
    raise;
end SET_PATH_ATTRIBUTE;
```

31 JANUARY 1985

5.1.3.7. Getting node attributes

```

procedure GET_NODE_ATTRIBUTE (NODE:      in  NODE_TYPE;
                              ATTRIBUTE: in  ATTRIBUTE_NAME;
                              VALUE:     in out LIST_TYPE);

```

Purpose:

This procedure returns the value of the node attribute named by ATTRIBUTE in the parameter VALUE. The node is identified by open node handle NODE.

Parameters:

NODE is an open node handle to a node the value of whose attribute ATTRIBUTE is to be retrieved.

ATTRIBUTE is the name of the attribute.

VALUE is the result parameter containing the value of the attribute.

Exceptions:

USE_ERROR is raised if the node has no attribute of name ATTRIBUTE.

STATUS_ERROR

is raised if NODE is not an open node handle.

INTENT_VIOLATION

is raised if NODE was not opened with an intent establishing the right to read attributes.

Additional Interface:

```

procedure GET_NODE_ATTRIBUTE (NAME:      in  NAME_STRING;
                              ATTRIBUTE: in  ATTRIBUTE_NAME;
                              VALUE:     in out LIST_TYPE)
is
    NODE: NODE_TYPE;
begin
    OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));
    GET_NODE_ATTRIBUTE(NODE, ATTRIBUTE, VALUE);
    CLOSE(NODE);
exception
    when others =>
        CLOSE(NODE);
        raise;
end GET_NODE_ATTRIBUTE;

```

5.1.3.8. Getting path attributes

```

procedure GET_PATH_ATTRIBUTE (BASE:      in  NODE_TYPE;
                              KEY:        in  RELATIONSHIP_KEY;
                              RELATION:   in  RELATION_NAME;
                                          :=DEFAULT_RELATION;
                              ATTRIBUTE: in  ATTRIBUTE_NAME;
                              VALUE:     in out LIST_TYPE);

```

Purpose:

This procedure assigns the value of the relationship attribute named by **ATTRIBUTE** to the parameter **VALUE**. The relationship is identified explicitly by the base node **BASE**, the relation name **RELATION** and the relationship key **KEY**.

Parameters:

BASE is an open node handle to the node from which the relationship emanates.

KEY is the relationship key of the accessed relationship.

RELATION is the relation name of the accessed relationship.

ATTRIBUTE is the name of the attribute.

VALUE is the result parameter containing the value of the attribute.

Exceptions:

NAME_ERROR
is raised if the relationship identified by the **BASE**, **KEY** and **RELATION** parameters does not exist.

USE_ERROR is raised if the relationship does not have an attribute of the given name.

STATUS_ERROR
is raised if the node handle **BASE** is not open.

INTENT_VIOLATION
is raised if **BASE** was not opened with an intent establishing the right to read relationships.

Additional Interface:

```
procedure GET_PATH_ATTRIBUTE(NAME:      in  NAME_STRING;
                             ATTRIBUTE: in  ATTRIBUTE_NAME;
                             VALUE:     in out LIST_TYPE)
is
    BASE: NODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(NAME), (1=>READ_RELATIONSHIPS));
    GET_PATH_ATTRIBUTE(BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
                      ATTRIBUTE, VALUE);
    CLOSE(BASE);
exception
    when others =>
        CLOSE(BASE);
        raise;
end GET_PATH_ATTRIBUTE;
```

5.1.3.9. Attribute iteration types and subtypes

```
subtype ATTRIBUTE_NAME is STRING;
type     ATTRIBUTE_ITERATOR is limited private;
subtype ATTRIBUTE_PATTERN is STRING;
```

These types are used in the following interfaces for iteration over a set of attributes of nodes or relationships. `ATTRIBUTE_NAME` is a subtype for the names of attributes. An `ATTRIBUTE_PATTERN` has the same syntax as an `ATTRIBUTE_NAME`, except that '?' will match any single character and '*' will match any string of characters. `ATTRIBUTE_ITERATOR` is a private type assumed to contain the bookkeeping information necessary for the implementation of the `MORE` and `GET_NEXT` functions. The attributes are returned by `GET_NEXT` in ASCII lexicographical order by attribute name. The effect on existing iterators of creation or deletion of attributes or relationships is implementation-defined.

5.1.3.10. Creating an iterator over node attributes

```
procedure NODE_ATTRIBUTE_ITERATE(ITERATOR: out ATTRIBUTE_ITERATOR;
                                NODE:      in  NODE_TYPE;
                                PATTERN:   in  ATTRIBUTE_PATTERN
                                :="*");
```

Purpose:

The procedure `NODE_ATTRIBUTE_ITERATE` returns in the parameter `ITERATOR` an attribute iterator according to the semantic rules for attribute selection given in Section 5.1.3.9.

Parameters:

`ITERATOR` is the attribute iterator returned.

`NODE` is an open node handle to a node over whose attributes the iterator is to be constructed.

`PATTERN` is a pattern for attribute names as described in Section 5.1.3.9.

Exceptions:

`USE_ERROR` is raised if the `PATTERN` is syntactically illegal.

`STATUS_ERROR`

is raised if `NODE` is not an open node handle.

`INTENT_VIOLATION`

is raised if `NODE` is not open with an intent establishing the right to read attributes.

Additional Interface:

```
procedure NODE_ATTRIBUTE_ITERATE(ITERATOR: out ATTRIBUTE_ITERATOR;
                                NAME:      in  NAME_STRING;
                                PATTERN:   in  ATTRIBUTE_PATTERN
                                :="*")
is
  NODE: NODE_TYPE;
begin
```

```
OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));  
MODE_ATTRIBUTE_ITERATE(ITERATOR, NODE, PATTERN);  
CLOSE(NODE);  
exception  
  when others =>  
    CLOSE(NODE);  
    raise;  
end MODE_ATTRIBUTE_ITERATE;
```

Notes:

By using the pattern '*', it is possible to iterate over all attributes of a node.

5.1.3.11. Creating an iterator over relationship attributes

```
procedure PATH_ATTRIBUTE_ITERATE(ITERATOR: out ATTRIBUTE_ITERATOR;  
                                BASE: in MODE_TYPE;  
                                KEY: in RELATIONSHIP_KEY;  
                                RELATION: in RELATION_NAME  
                                           :=DEFAULT_RELATION;  
                                PATTERN: in ATTRIBUTE_PATTERN  
                                           :="*");
```

Purpose:

This procedure is provided to obtain an attribute iterator for relationship attributes. The relationship is identified explicitly by the base node BASE, the relation name RELATION and the relationship key KEY. The procedure returns an attribute iterator in ITERATOR according to the semantic rules for attribute selection applied to the attributes of the identified relationship. This iterator can then be processed by means of the MORE and GET_NEXT interfaces.

Parameters:

ITERATOR is the attribute iterator returned.

BASE is an open node handle to the node from which the relationship emanates.

KEY is the relationship key of the affected relationship.

RELATION is the relation name of the affected relationship.

PATTERN is a pattern for attribute names (see Section 5.1.3.9).

Exceptions:

NAME_ERROR
is raised if the relationship identified by the BASE, KEY and RELATION parameters does not exist.

USE_ERROR is raised if the PATTERN is syntactically illegal.

STATUS_ERROR
is raised if BASE is not an open node handle.

INTENT_VIOLATION
is raised if BASE was not opened with an intent establishing the right to read relationships.

Additional Interface:

```
procedure PATH_ATTRIBUTE_ITERATE(ITERATOR: out ATTRIBUTE_ITERATOR;  
                                NAME: in NAME_STRING;  
                                PATTERN: in ATTRIBUTE_PATTERN  
                                :="")  
is  
    BASE: NODE_TYPE;  
begin  
    OPEN(BASE, BASE_PATH(NAME), (1=>READ_RELATIONSHIPS));  
    PATH_ATTRIBUTE_ITERATE(ITERATOR, BASE, LAST_KEY(NAME),  
                          LAST_RELATION(NAME), PATTERN);  
    CLOSE(BASE);  
exception  
    when others =>  
        CLOSE(BASE);  
        raise;  
end PATH_ATTRIBUTE_ITERATE;
```

5.1.3.12. Determining iteration status

```
function MORE(ITERATOR: in ATTRIBUTE_ITERATOR)  
    return BOOLEAN;
```

Purpose:

The function MORE returns FALSE if all attributes contained in the attribute iterator have been retrieved with the procedure GET_NEXT; otherwise, it returns TRUE.

Parameters:

ITERATOR is an attribute iterator previously constructed.

Exceptions:

USE_ERROR is raised if the ITERATOR has not been previously set by the procedures NODE_ATTRIBUTE_ITERATE or PATH_ATTRIBUTE_ITERATE.

5.1.3.13. Getting the next attribute

```
procedure GET_NEXT(ITERATOR: in out ATTRIBUTE_ITERATOR;  
                  ATTRIBUTE: out ATTRIBUTE_NAME;  
                  VALUE: in out LIST_TYPE);
```

Purpose:

The procedure GET_NEXT returns, in its parameters ATTRIBUTE and VALUE, both the name and the value of the next attribute in the iterator.

Parameters:

ITERATOR is an attribute iterator previously constructed.

ATTRIBUTE is a result parameter containing the name of an attribute.

VALUE is a result parameter containing the value of the attribute named by ATTRIBUTE.

Exceptions:

USE_ERROR is raised if the ITERATOR has not been previously set by the procedures

NODE_ATTRIBUTE_ITERATE or PATH_ATTRIBUTE_ITERATE or if the
iterator is exhausted, i.e., MORE(ITERATOR)= FALSE.

5.1.4. Package ACCESS_CONTROL

This package provides primitives for manipulating discretionary access control information for CAIS nodes. In addition, certain CAIS subprograms declared elsewhere allow the specification of initial access control information. The CAIS specifies mechanisms for discretionary and mandatory access control (see [TCSEC]). These mechanisms are only recommendations. Alternate discretionary or mandatory access control mechanisms can be substituted by an implementation provided that the semantics of all interfaces in Section 5 (with the exception of Section 5.1.4) are implemented as specified.

5.1.4.1. Subtypes

subtype GRANT_VALUE is CAIS.LIST_UTILITIES.LIST_TYPE;

GRANT_VALUE is a subtype for values of GRANT attributes; it is a list in the syntax described in Table II.

5.1.4.2. Setting access control

```
procedure SET_ACCESS_CONTROL (NODE:      in NODE_TYPE;  
                             ROLE_NODE: in NODE_TYPE;  
                             GRANT:      in GRANT_VALUE);
```

Purpose:

This procedure sets access control information for a given node. If a relationship of the predefined relation ACCESS does not exist from the node identified by NODE to the node identified by ROLE_NODE, such a relationship with an implementation-defined relationship key is created from the node specified by NODE to the node specified by ROLE_NODE. If necessary, the predefined attribute GRANT is created on this relationship. The value of the GRANT attribute is set to the value of the GRANT parameter (see Table II for the syntax). The effect is to grant the access specified by GRANT to processes that have adopted the role ROLE_NODE.

Parameters:

NODE is an open node handle to the node whose access control information is to be set.

ROLE_NODE is an open node handle to the node representing the role.

GRANT is a list describing what access rights can be granted.

Exceptions:

USE_ERROR is raised if GRANT is not in valid syntax.

STATUS_ERROR
 is raised if NODE and ROLE_NODE are not both open node handles.

INTENT_VIOLATION
 is raised if NODE was not opened with intent CONTROL.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```

procedure SET_ACCESS_CONTROL(NAME:      in NAME_STRING;
                             ROLE_NAME: in NAME_STRING;
                             GRANT:      in GRANT_VALUE)
is
    NODE, ROLE_NODE: NODE_TYPE;
begin
    OPEN(NODE, NAME, (1=>CONTROL));
    OPEN(ROLE_NODE, ROLE_NAME, (1=>EXISTENCE));
    SET_ACCESS_CONTROL(NODE, ROLE_NODE, GRANT);
    CLOSE(NODE);
    CLOSE(ROLE_NODE);
exception
    when others =>
        CLOSE(NODE);
        CLOSE(ROLE_NODE);
        raise;
end SET_ACCESS_CONTROL;

```

5.1.4.3. Examining access rights

```

function IS_GRANTED(OBJECT_NODE : in NODE_TYPE;
                    ACCESS_RIGHT: in NAME_STRING)
    return BOOLEAN;

```

Purpose:

This function returns TRUE if the current process as a subject has an approved access right ACCESS_RIGHT to the OBJECT_NODE as an object. Otherwise it returns FALSE.

Parameters:**OBJECT_NODE**

is an open node handle to the object node.

ACCESS_RIGHT

is the name of a predefined or user-defined access right.

Exceptions:

USE_ERROR is raised if ACCESS_RIGHT is not a valid Ada identifier.

STATUS_ERROR

is raised if OBJECT_NODE is not an open node handle.

INTENT_VIOLATION

is raised if OBJECT_NODE was not opened with an intent establishing the right to read relationships or to read access control information.

Additional Interface:

```
function IS_GRANTED(OBJECT_NAME: in NAME_STRING;  
                   ACCESS_RIGHT: in NAME_STRING)  
    return BOOLEAN  
is  
    OBJECT_NODE: NODE_TYPE;  
    RESULT:      BOOLEAN;  
begin  
    OPEN(OBJECT_NODE, OBJECT_NAME, (1=>READ_RELATIONSHIPS));  
    RESULT := IS_GRANTED(OBJECT_NODE, ACCESS_RIGHT);  
    CLOSE(OBJECT_NODE);  
    return RESULT;  
exception  
    when others =>  
        CLOSE(OBJECT_NODE);  
        raise;  
end IS_GRANTED;
```

5.1.4.4. Adopting a role

```
procedure ADOPT(ROLE_NODE: in NODE_TYPE;  
               ROLE_KEY:   in RELATIONSHIP_KEY:=LATEST_KEY);
```

Purpose:

This procedure causes the current process to adopt the group specified by the ROLE_NODE. A relationship of the predefined relation ADOPTED_ROLE with relationship key ROLE_KEY is created from the calling process node to the node identified by ROLE_NODE. In order for the current process to adopt the group, a node representing some other adopted role of the current process must be a potential member of the group to be adopted.

Parameters:

ROLE_NODE
 is an open node handle to a node representing the group.

ROLE_KEY is a relationship key to be used in creating the relationship.

Exceptions:

USE_ERROR is raised if there is no adopted role of the current process that is a potential member of the group represented by ROLE_NODE or if there already exists a relationship of the predefined relation ADOPTED_ROLE with relationship key ROLE_KEY emanating from the current process node. USE_ERROR is also raised if the node identified by ROLE_NODE is inaccessible or unobtainable.

STATUS_ERROR
 is raised if ROLE_NODE is not an open node handle.

LOCK_ERROR
 is raised if access with intent APPEND_RELATIONSHIPS to the current process node cannot be obtained due to an existing lock on the node.

SECURITY_VIOLATION
 is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

5.1.4.5. Unlinking an adopted role

procedure UNADOPT(ROLE_KEY: in RELATIONSHIP_KEY);

Purpose:

This procedure deletes the relationship of the predefined relation ADOPTED_ROLE with relationship key ROLE_KEY emanating from the current process node. If there is no such relationship, the procedure has no effect.

Parameters:

ROLE_KEY is the relationship key of the relation ADOPTED_ROLE.

Exception:

USE_ERROR is raised if the target node of the relationship to be deleted is the top-level node identified by "CURRENT_USER". In this case the relationship is not deleted.

LOCK_ERROR

is raised if access, with intent WRITE_RELATIONSHIPS, to the current process node cannot be obtained due to an existing lock on the node.

5.1.5. Package STRUCTURAL_NODES

Structural nodes are special nodes in the sense that they do not have contents as the other nodes of the CAIS model do. 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 STRUCTURAL_NODES defines the primitive operations for creating structural nodes.

5.1.5.1. Creating structural nodes

```
procedure CREATE_NODE(NODE:           in out NODE_TYPE;  
                      BASE:           in  NODE_TYPE;  
                      KEY:            in  RELATIONSHIP_KEY  
                      :=LATEST_KEY;  
                      RELATION:       in  RELATION_NAME  
                      :=DEFAULT_RELATION;  
                      ATTRIBUTES:     in LIST_TYPE := EMPTY_LIST;  
                      ACCESS_CONTROL: in LIST_TYPE := EMPTY_LIST;  
                      LEVEL:          in LIST_TYPE := EMPTY_LIST);
```

Purpose:

This procedure creates a structural node and installs the primary relationship to it. The relation name and relationship key of the primary relationship to the node and the base node from which it emanates are given by the parameters RELATION, KEY, and BASE. An open node handle to the newly created node with WRITE intent is returned in NODE.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node. The ACCESS_CONTROL parameter specifies initial access control information to be established for the created node (see Section 4.4).

The LEVEL parameter specifies the security level at which the node is to be created.

Parameters:

NODE is a node handle, initially closed, to be opened to the newly created node.

BASE is an open node handle to the node from which the primary relationship to the new node is to emanate.

KEY is the relationship key of the primary relationship to be created.

RELATION is the relation name of the primary relationship to be created.

ATTRIBUTES is a named list (see Section 5.4) whose elements are used to establish initial values for attributes of the newly created node; each named item specifies an attribute name and the value to be given to that attribute.

ACCESS_CONTROL is the initial access control information associated with the created node; it is a named list (see Section 5.4) each of whose named items specifies a relationship key followed by a list of access rights.

LEVEL is the classification label for the created node (see TABLE IV).

Exceptions:

NAME_ERROR is raised if a node already exists for the node identification given, if the node identification is illegal, or if any node identifying a group specified in the given ACCESS_CONTROL parameter is unobtainable or inaccessible.

USE_ERROR is raised if the ACCESS_CONTROL or LEVEL parameters do not adhere to the required syntax or if the ATTRIBUTES parameter contains references to predefined attributes which cannot be modified or created by the user. USE_ERROR is also raised if RELATION is the name of a predefined relation that cannot be modified or created by the user.

STATUS_ERROR is raised if BASE is not an open node handle or if NODE is an open node handle prior to the call.

INTENT_VIOLATION is raised if BASE was not opened with an intent establishing the right to append relationships.

SECURITY_VIOLATION is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interfaces:

```
procedure CREATE_NODE(NODE:      in out NODE_TYPE;  
                      NAME:      in  NAME_STRING;
```

```

        ATTRIBUTES:    in LIST_TYPE := EMPTY_LIST;
        ACCESS_CONTROL: in LIST_TYPE := EMPTY_LIST;
        LEVEL:         in LIST_TYPE := EMPTY_LIST);
is
    BASE: NODE_TYPE;

begin
    OPEN(BASE, BASE_PATH(NAME), (1=>APPEND_RELATIONSHIPS));
    CREATE_NODE(NODE, BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
        ATTRIBUTES, ACCESS_CONTROL, LEVEL);
    CLOSE(BASE);
exception
    when others =>
        CLOSE(NODE);
        CLOSE(BASE);
        raise;
end CREATE_NODE;

procedure CREATE_NODE(BASE:      in NODE_TYPE;
                      KEY:       in RELATIONSHIP_KEY
                                :=LATEST_KEY;
                      RELATION:  in RELATION_NAME
                                :=DEFAULT_RELATION;
                      ATTRIBUTES: in LIST_TYPE := EMPTY_LIST;
                      ACCESS_CONTROL: in LIST_TYPE := EMPTY_LIST;
                      LEVEL:     in LIST_TYPE := EMPTY_LIST);
is
    NODE: NODE_TYPE;
begin
    CREATE_NODE(NODE, KEY, RELATION, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
    CLOSE(NODE);
end CREATE_NODE;

procedure CREATE_NODE (NAME:      in NAME_STRING;
                      ATTRIBUTES: in LIST_TYPE := EMPTY_LIST;
                      ACCESS_CONTROL: in LIST_TYPE := EMPTY_LIST;
                      LEVEL:     in LIST_TYPE := EMPTY_LIST);
is
    NODE: NODE_TYPE;
begin
    CREATE_NODE(NODE, NAME, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
    CLOSE(NODE);
end CREATE_NODE;

```

Notes:

Use of the sequence of a CREATE_NODE call that does not return an open node handle followed by a call on OPEN for the created node, using the node identification of the created node, cannot guarantee that a handle to the node just created is opened; this is because relationships, and therefore the node identification, may have changed since the CREATE_NODE call.

5.2. CAIS process nodes

This section describes the semantics of the execution of Ada programs as represented by CAIS processes and the facilities provided by the CAIS for initiating and controlling processes. The major stages in a process' life are initiation, running (which may include suspension or resumption), and termination or abortion. The CAIS defines facilities to control and coordinate the initiation, suspension, resumption, and termination or abortion of processes (see Section 4.3.2). Each CAIS process has a current status associated with it which changes with certain events as specified in TABLE VI.

A process is said to be *terminated* when its main program (in the sense of [LRM] 10.1) has terminated (in the sense of [LRM] 9.4). See also the notes in [LRM] 9.4. Thus, termination of a process takes place when the main program has been completed and all tasks dependent on the main program have terminated. A process may be *aborted* either by itself or by another process. When a process has terminated or has been aborted, all of its dependent processes which have not already terminated or been aborted will be aborted but its process node remains until explicitly deleted. Any open node handles of a process are closed when the process terminates or is aborted.

Two mechanisms for a process to initiate another process are provided:

- a. **Spawn** - the procedure `SPAWN_PROCESS` returns after initiating the specified program. The initiating process and the initiated process run in parallel, and, within each of them, their tasks may execute in parallel.
- b. **Invoke** - the procedure `INVOKE_PROCESS` returns control to the calling task after the initiated process has terminated or aborted. Execution of the calling task is blocked until termination or abortion of the initiated process, but other tasks in the initiating process may execute in parallel with the initiated process and its tasks.

Every process node has several predefined attributes. Three of these are: `RESULTS`, which can be used to store user-defined strings giving intermediate results of the process; `PARAMETERS`, which contains the parameters with which the process was initiated; and `CURRENT_STATUS`, which gives the current status of the process (see TABLE VI). In addition, every process node has several predefined attributes which provide information for standardized debugging and performance measurement of processes within the CAIS implementation. One of these predefined attributes, `HANDLES_OPEN`, has an implementation-independent value which gives the number of node handles the process currently has open. The remaining predefined attributes have implementation-dependent values and should not be used for comparison with values from other CAIS implementations. `START_TIME` and `FINISH_TIME` give the time of activation and the time of termination or abortion of the process. `MACHINE_TIME` gives the length of time the process was active on the logical processor, if the process has terminated or aborted, or zero, if the process has not terminated or aborted. `IO_UNITS` gives the number of GET and PUT operations that have been performed by the process. The `CURRENT_STATUS`, `HANDLES_OPEN`, `START_TIME`, `FINISH_TIME`, `MACHINE_TIME`, and `IO_UNITS` predefined attributes are maintained by the implementation and cannot be set using CAIS interfaces.

When a process has terminated or aborted, the final status, recorded in the predefined process node attribute `CURRENT_STATUS`, will persist as long as the process node exists. `CURRENT_STATUS` may also be examined by the CAIS procedures `STATUS_OF_PROCESS` and `GET_RESULTS`. The process status of a process will be returned to any task awaiting the termination or abortion of the process whenever the process is terminated or aborted. If the process has already been terminated or aborted at the time a call to `AWAIT_PROCESS_COMPLETION` is made, then the final status is immediately available.

For purposes of input and output, every process node has one relationship of each of the following predefined relations: `STANDARD_INPUT`, `STANDARD_OUTPUT`, `STANDARD_ERROR`, `CURRENT_INPUT`, `CURRENT_OUTPUT`, and `CURRENT_ERROR`. `STANDARD_INPUT`, `STANDARD_OUTPUT` and `STANDARD_ERROR` are relation names of relationships established at job creation to the default input, output and error files, respectively. The `STANDARD_INPUT` and `STANDARD_OUTPUT` files conform to the semantics given for these in [LRM] 14.3.2. `CURRENT_INPUT`, `CURRENT_OUTPUT` and `CURRENT_ERROR` are relation names of relationships established by a process to alternative files to be used as the default input, output and error files, respectively. `CURRENT_INPUT` and `CURRENT_OUTPUT` also conform to the

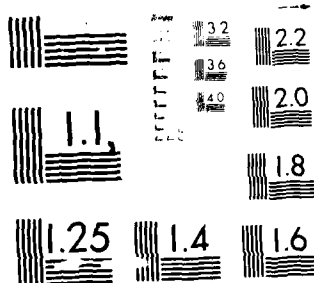
AD-A157-589

MILITARY STANDARD COMMON APSE (ADA PROGRAMMING SUPPORT
ENVIRONMENT) INTERFACE SET (CAIS) (U) ADA JOINT PROGRAM
OFFICE ARLINGTON VA JAN 85

2/4

UNCLASSIFIED

F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

semantics of [LRM] 14.3.2. Interfaces are provided in the CAIS input and output packages (see Section 5.3) to read relationships of these predefined relations and to change the relationships of the relations CURRENT_INPUT, CURRENT_OUTPUT, and CURRENT_ERROR.

Table VI. Process status transition table

status: event	non_existent	READY	SUSPENDED	ABORTED	TERMINATED
process creation	READY	N/A	N/A	N/A	N/A
termination of main program	N/A	TERM- INATED	N/A	N/A	N/A
ABORT_ PROCESS	N/A	ABOR- TED	ABORTED	---	---
SUSPEND_ PROCESS	N/A	SUS- PENDED	---	---	---
RESUME_ PROCESS	N/A	---	READY	---	---

N/A: marks events that are not applicable to the status specified.

---: marks events that have no effect on the status.

upper case: status which are values of the enumeration type PROCESS_STATUS (e.g., READY) and for events which are caused by calling CAIS interfaces (e.g., ABORT_PROCESS).

lower case: other status (i.e., non-existent) and other events (e.g., termination of the main program).

5.2.1. Package PROCESS_DEFINITIONS

This package defines the types and exceptions associated with process nodes.

```
type PROCESS_STATUS is
    (READY, SUSPENDED, ABORTED, TERMINATED);
```

An object of type PROCESS_STATUS is the status of a process.

```
subtype RESULTS_LIST is CAIS.LIST_UTILITIES.LIST_TYPE;
subtype RESULTS_STRING is STRING;
subtype PARAMETER_LIST is CAIS.LIST_UTILITIES.LIST_TYPE;
```

An object of type RESULTS_LIST is a list of results from a process. The elements of this list are of type RESULTS_STRING. An object of type PARAMETER_LIST is a list containing process parameter information.

```

ROOT_PROCESS : constant NAME_STRING := "CURRENT_JOB";
CURRENT_INPUT : constant NAME_STRING := "CURRENT_INPUT";
CURRENT_OUTPUT : constant NAME_STRING := "CURRENT_OUTPUT";
CURRENT_ERROR : constant NAME_STRING := "CURRENT_ERROR";

```

ROOT_PROCESS is a standard pathname for the root process node of the current job. CURRENT_INPUT, CURRENT_OUTPUT and CURRENT_ERROR are standard pathnames for the current process' input, output and error files, respectively.

5.2.2. Package PROCESS CONTROL

This package specifies interfaces for the creation and termination of processes and examination and modification of process node attributes.

As part of the creation of process nodes, new secondary relationships are built as described in TABLE VII.

Table VII. Created and Inherited relationships

A secondary relationship of the predefined relation:	is created to the node identified by:
CURRENT_INPUT	the interface parameter INPUT_FILE
CURRENT_OUTPUT	the interface parameter OUTPUT_FILE
CURRENT_ERROR	the interface parameter ERROR_FILE
ADOPTED_ROLE	the interface parameter FILE_NODE
CURRENT_NODE	the interface parameter ENVIRONMENT_NODE
PARENT	the predefined constant CURRENT_PROCESS (for dependent process nodes) the predefined constant CURRENT_USER (for root process nodes)

The created process node inherits all secondary relationships
of the following predefined relations from the creating process
node:

```

CURRENT_USER
USER
ALLOW_ACCESS
DEVICE
STANDARD_INPUT
STANDARD_OUTPUT
STANDARD_ERROR
ADOPTED_ROLE [1]
CURRENT_JOB [2]

```

1. For CREATE_JOB, only the relationship of the predefined relation ADOPTED_ROLE with the CURRENT_USER as target is inherited from the creating process node.
2. For CREATE_JOB, a relationship of the predefined relation CURRENT_JOB is created with the new node as both source and target instead of being inherited from the creating process node.

5.2.2.1. Spawning a process

```

procedure SPAN_PROCESS
  (NODE:      in out NODE_TYPE;
   FILE_NODE: in    NODE_TYPE;
   INPUT_PARAMETERS: in  PARAMETER_LIST := EMPTY_LIST;
   KEY:       in    RELATIONSHIP_KEY := LATEST_KEY;
   RELATION:  in    RELATION_NAME := DEFAULT_RELATION;
   ACCESS_CONTROL: in  LIST_TYPE := EMPTY_LIST;
   LEVEL:     in    LIST_TYPE := EMPTY_LIST;
   ATTRIBUTES: in  LIST_TYPE := EMPTY_LIST;
   INPUT_FILE: in    NAME_STRING := CURRENT_INPUT;
   OUTPUT_FILE: in  NAME_STRING := CURRENT_OUTPUT;
   ERROR_FILE: in  NAME_STRING := CURRENT_ERROR;
   ENVIRONMENT_NODE: in  NAME_STRING := CURRENT_NODE);

```

Purpose:

This procedure creates a new process node whose contents represent the execution of the program contained in the specified file node. Control returns to the calling task after the new node is created. The process node containing the calling task must have execution rights for the file node. An open node handle **NODE** on the new node is returned, with an intent (**I=>READ_ATTRIBUTES**). The new process, as a subject, has all discretionary access rights to its own process node (the object). When the parent process terminates or aborts, the child process will be aborted.

Secondary relationships emanating from the new process node are created and inherited as described in TABLE VII.

The **ACCESS_CONTROL** parameter specifies the initial access control information to be established for the created node. If the CAIS models of discretionary and mandatory access control are used, then, in addition to the relationships established using the information in the **ACCESS_CONTROL** parameter, an access relationship is established from the created process node to the current user node, with a **GRANT** attribute value ((**READ, WRITE, CONTROL**)).

The **LEVEL** parameter specifies the security level at which the node is to be created.

Parameters:

NODE is a node handle returned open on the newly created process node.

FILE_NODE is an open node handle on the file node containing the executable image whose execution will be represented by the new process.

INPUT_PARAMETERS

is a list containing process parameter information. The list is constructed and parsed using the tools provided in **CAIS.LIST_UTILITIES** (see Section 5.4). The value of **INPUT_PARAMETERS** is stored in a predefined attribute **PARAMETERS** of the new node.

KEY is the relationship key of the primary relationship from the current process node to the new process node. The default is supplied by the mechanism of interpreting the **LATEST_KEY** constant.

RELATION is the relation name of the primary relationship from the current process node to the new process node. The default is **DEFAULT_RELATION**.

31 JANUARY 1985

ACCESS_CONTROL

is a string defining the initial access control information associated with the created node.

LEVEL

is a string defining the classification label for the created node (see TABLE IV).

ATTRIBUTES

is a list which can be used to set attributes of the new node. It could be used by an implementation to establish allocation of resources.

INPUT_FILE, OUTPUT_FILE, ERROR_FILE

are pathnames to file nodes.

ENVIRONMENT_NODE

is the node the new process will have as its initial current node. The default value is the CURRENT_NODE of the initiating process.

Exceptions:

NAME_ERROR

is raised if a node already exists for the relationship specified by KEY and RELATION. NAME_ERROR is also raised if any of the nodes identified by INPUT_FILE, OUTPUT_FILE, ERROR_FILE, or ENVIRONMENT_NODE do not exist. It is also raised if KEY or RELATION is syntactically illegal or if any node identifying a group specified in the given ACCESS_CONTROL parameter is unobtainable or inaccessible.

USE_ERROR

is raised if it can be determined that the node indicated by FILE_NODE does not contain an executable image. USE_ERROR is also raised if any of the parameters INPUT_PARAMETERS, LEVEL, ACCESS_CONTROL, or ATTRIBUTES is syntactically or semantically illegal. USE_ERROR is also raised if RELATION is the name of a predefined relation or if the ATTRIBUTES parameter contains references to a predefined attribute which cannot be modified or created by the user.

STATUS_ERROR

is raised if NODE is an open node handle prior to the call or if FILE_NODE is not an open node handle.

LOCK_ERROR

is raised if access with intent APPEND_RELATIONSHIPS to the current process node cannot be obtained due to an existing lock on the node.

INTENT_VIOLATION

is raised if the node designated by FILE_NODE was not opened with an intent establishing the right to execute its contents.

Notes:

SPAWN_PROCESS does not return results or process status. If coordination between any task and the new process is desired, AWAIT_PROCESS_COMPLETION or the techniques provided in CAIS input and output (see Section 5.3) must be used.

5.2.2.2. Awaiting termination or abortion of another process

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

Purpose:

This procedure suspends the calling task and waits for the process identified by NODE to terminate or abort. The calling task is suspended until the identified process terminates or aborts or until the time limit is exceeded

Parameters:

NODE is an open node handle for the process to be awaited.

TIME_LIMIT is the limit on the time that the calling task will be suspended awaiting the process. When the limit is exceeded the calling task resumes execution. The default is the implementation-dependent maximum value for DURATION.

Exceptions:

NAME_ERROR
is raised if the node identified by NODE is inaccessible or unobtainable.

STATUS_ERROR
is raised if NODE is not an open node handle.

INTENT_VIOLATION
is raised if NODE was not opened with an intent establishing the right to read attributes.

Additional interface:

```
procedure AWAIT_PROCESS_COMPLETION
  (NODE          : in  NODE_TYPE;
   RESULTS_RETURNED : in out RESULTS_LIST;
   STATUS        : out PROCESS_STATUS;
   TIME_LIMIT     : in  DURATION := DURATION'LAST)
is
begin
  AWAIT_PROCESS_COMPLETION (NODE, TIME_LIMIT);
  GET_RESULTS (NODE, RESULTS_RETURNED);
  STATUS := STATUS_OF_PROCESS (NODE);
end AWAIT_PROCESS_COMPLETION;
```

5.2.2.3. Invoking a new process

```
procedure INVOKE_PROCESS
  (NODE:          in out NODE_TYPE;
   FILE_NODE:     in  NODE_TYPE;
   RESULTS_RETURNED: in out RESULTS_LIST;
   STATUS:        out PROCESS_STATUS;
   INPUT_PARAMETERS: in  PARAMETER_LIST;
   KEY:           in  RELATIONSHIP_KEY := LATEST_KEY;
   RELATION:      in  RELATION_NAME := DEFAULT_RELATION;
```

ACCESS_CONTROL:	in	LIST_TYPE := EMPTY_LIST;
LEVEL:	in	LIST_TYPE := EMPTY_LIST;
ATTRIBUTES:	in	LIST_TYPE := EMPTY_LIST;
INPUT_FILE:	in	NAME_STRING := CURRENT_INPUT;
OUTPUT_FILE:	in	NAME_STRING := CURRENT_OUTPUT;
ERROR_FILE:	in	NAME_STRING := CURRENT_ERROR;
ENVIRONMENT_NODE:	in	NAME_STRING := CURRENT_NODE;
TIME_LIMIT:	in	DURATION := DURATION_LAST);

Purpose:

This procedure provides the functionality described by the following Ada fragment except that the implementation must guarantee that only exceptions raised by the call to SPAWN_PROCESS in this fragment are raised by INVOKE_PROCESS.

```

SPAWN_PROCESS (NODE, FILE_NODE, INPUT_PARAMETERS, KEY, RELATION,
               ACCESS_CONTROL, LEVEL, ATTRIBUTES, INPUT_FILE,
               OUTPUT_FILE, ERROR_FILE, ENVIRONMENT_NODE);
AWAIT_PROCESS_COMPLETION (NODE, TIME_LIMIT);
GET_RESULTS (NODE, RESULTS_RETURNED);
STATUS := STATUS_OF_PROCESS (NODE);

```

Parameters:

NODE is a node handle returned open on the newly created process node.

FILE_NODE is an open node handle on the file node containing the executable image whose execution will be represented by the new process.

RESULTS_RETURNED

is a list of results which are represented by strings from the new process. The individual results may be extracted from the list using the tools of CAIS.LIST_UTILITIES.

STATUS gives the process status of the process. If termination or abortion of the identified process can be reported within the specified time limit, STATUS will have the value ABORTED or TERMINATED. If the process does not terminate or abort within the time limit, STATUS will have the value READY or SUSPENDED.

INPUT_PARAMETERS

is a list containing process parameter information. The list is constructed and parsed using the list handling tools of CAIS.LIST_UTILITIES. The value of INPUT_PARAMETERS is stored in the predefined attribute PARAMETERS of the new node.

KEY is the relationship key of the primary relationship from the current process node to the new process node. The default is supplied by the mechanism of interpreting the LATEST_KEY constant.

RELATION is the relation name of the primary relationship from the current process node to the new node. The default is DEFAULT_RELATION.

ACCESS_CONTROL

is a string defining the initial access control information associated with the created node.

LEVEL is a string defining the classification label for the created node (see TABLE IV).

ATTRIBUTES is a list which can be used to set attributes of the new node. It could be used by an implementation to establish allocation of resources.

INPUT_FILE, OUTPUT_FILE, ERROR_FILE
are pathnames to file nodes.

ENVIRONMENT_NODE
is the node the new process will have as its current node.

TIME_LIMIT is the limit on the time that the calling task will be suspended awaiting the new process. When the limit is exceeded, the calling task resumes execution. The default is the implementation dependent maximum value for **DURATION**.

Exceptions:

NAME_ERROR

is raised if a node already exists for the relationship specified by **KEY** and **RELATION**. **NAME_ERROR** is also raised if any of the nodes identified by **INPUT_FILE**, **OUTPUT_FILE**, **ERROR_FILE** or **ENVIRONMENT_NODE** do not exist. It is also raised if **KEY** or **RELATION** is syntactically illegal or if any node identifying a group specified in the given **ACCESS_CONTROL** parameter is unobtainable or inaccessible.

USE_ERROR is raised if it can be determined that the node indicated by **FILE_NODE** does not contain an executable image. **USE_ERROR** is also raised if any of the parameters **INPUT_PARAMETERS**, **LEVEL**, **ACCESS_CONTROL**, or **ATTRIBUTES** is syntactically or semantically illegal. **USE_ERROR** is also raised if **RELATION** is the name of a predefined relation or if the **ATTRIBUTES** parameter contains references to a predefined attribute which cannot be modified or created by the user.

STATUS_ERROR

is raised if **NODE** is an open node handle prior to the call or if **FILE_NODE** is not an open node handle.

LOCK_ERROR

is raised if access with intent **APPEND_RELATIONSHIPS** cannot be obtained to the current process node due to an existing lock on the node.

INTENT_VIOLATION

is raised if the node designated by **FILE_NODE** was not opened with an intent establishing the right to execute contents.

Notes:

Both control and data (results and status) are returned to the calling task upon termination or abortion of the invoked process or when the **TIME_LIMIT** is exceeded.

5.2.2.4. Creating a new job

```
procedure CREATE_JOB
(FILE_NODE:      in NODE_TYPE;
 INPUT_PARAMETERS: in PARAMETER_LIST:=EMPTY_LIST;
 KEY:            in RELATIONSHIP_KEY := LATEST_KEY;
 ACCESS_CONTROL:  in LIST_TYPE := EMPTY_LIST;
 LEVEL:          in LIST_TYPE := EMPTY_LIST;
 ATTRIBUTES:     in LIST_TYPE := EMPTY_LIST;
 INPUT_FILE:      in NAME_STRING := CURRENT_INPUT;
 OUTPUT_FILE:     in NAME_STRING := CURRENT_OUTPUT;
 ERROR_FILE:      in NAME_STRING := CURRENT_ERROR;
 ENVIRONMENT_NODE: in NAME_STRING := CURRENT_USER);
```

Purpose:

This procedure creates a new root process node whose contents represent the execution of the program contained in the specified file node. Control returns to the calling task after the new job is created. The process node containing the calling task must have execution rights for the file node and sufficient rights to append relationships to the node identified by "CURRENT_USER". A new primary relationship of the predefined relation JOB is established from the current user node to the root process node of the new job. The new root process as a subject can acquire all discretionary access rights to its own process node (the object). Secondary relationships emanating from the new process node are created and inherited as described in TABLE VII.

The ACCESS_CONTROL parameter specifies the initial access control information to be established for the created node. If the CAIS models of discretionary and mandatory access control are used, then, in addition to the relationships established using the information in the ACCESS_CONTROL parameter, an access relationship is established from the created process node to the current user node, with a GRANT attribute value ((READ, WRITE, CONTROL)).

The LEVEL parameter specifies the security level at which the node is to be created.

Parameters:

FILE_NODE is an open node handle on the file node containing the executable image whose execution will be represented by the new process.

INPUT_PARAMETERS

is a list containing process parameter information. The list is constructed and parsed using the tools provided in CAIS.LIST_UTILITIES. INPUT_PARAMETERS is stored in the predefined attribute PARAMETERS of the new node.

KEY

is the relationship key of the primary relationship of the predefined relation JOB from the current user node to the new process node. The default is supplied by the mechanism of interpreting the LATEST_KEY constant.

ACCESS_CONTROL

is a string defining the initial access control information associated with the created node.

LEVEL

is a string defining the classification label for the created node (see TABLE IV).

ATTRIBUTES is a list which can be used to set attributes of the new node. It could be used by an implementation to establish allocation of resources.

INPUT_FILE, OUTPUT_FILE, ERROR_FILE
are pathnames to file nodes.

ENVIRONMENT_NODE
is the node the new process will have as its initial current node.

Exceptions:

NAME_ERROR
is raised if a node already exists for the relationship specified by **KEY** and the relation **JOB**. **NAME_ERROR** is also raised if any of the nodes identified by **INPUT_FILE**, **OUTPUT_FILE**, **ERROR_FILE** or **ENVIRONMENT_NODE** does not exist. It is also raised if **KEY** is syntactically illegal or if any node identifying a group specified in the **ACCESS_CONTROL** parameter is unobtainable or inaccessible.

USE_ERROR is raised if it can be determined that the node indicated by **FILE_NODE** does not contain an executable image. **USE_ERROR** is also raised if any of the parameters **INPUT_PARAMETERS**, **LEVEL**, **ACCESS_CONTROL**, or **ATTRIBUTES** is syntactically or semantically illegal. **USE_ERROR** is also raised if the **ATTRIBUTES** parameter contains references to a predefined attribute which cannot be modified or created by the user.

STATUS_ERROR
is raised if **FILE_NODE** is not an open node handle.

LOCK_ERROR
is raised if access to the current user node or the current process node with intent **APPEND_RELATIONSHIPS** cannot be obtained due to an existing lock on the node.

INTENT_VIOLATION
is raised if the node designated by **FILE_NODE** was not opened with an intent establishing the right to execute contents.

ACCESS_VIOLATION
is raised if the current process does not have sufficient discretionary access rights to open the current user node with **APPEND_RELATIONSHIPS** intent. **ACCESS_VIOLATION** is raised only if the conditions for raising **NAME_ERROR** are not satisfied.

SECURITY_VIOLATION
is raised if the attempt to obtain access to the node identified by **CURRENT_USER** represents a violation of mandatory access controls for the CAIS. **SECURITY_VIOLATION** is raised only if the conditions for raising the other exceptions are not satisfied.

31 JANUARY 1985

Notes:

CREATE_JOB does not return results or process status to the calling program unit. If coordination between any program unit and the new process is desired, AWAIT_PROCESS_COMPLETION or the techniques provided in CAIS input and output (see Section 5.3) must be used.

The relation name for the primary relationship to the new node is JOB.

5.2.2.5. Appending results

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

Purpose:

This procedure inserts the value of its RESULTS parameter as the last item in to the list which is the value of the RESULTS attribute of the current process node.

Parameters:

RESULTS is a string to be appended to the RESULTS attribute value of the current process node.

Exceptions:

LOCK_ERROR

is raised if access with intent WRITE_ATTRIBUTES to the current process node cannot be obtained due to an existing lock on the node.

5.2.2.6. Overwriting results

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

Purpose:

This procedure replaces the value of the RESULTS attribute of the current process node with a list containing a single item which is the value of the parameter RESULTS.

Parameters:

RESULTS is a string to be stored in the RESULTS attribute of the current process node.

Exceptions:

LOCK_ERROR

is raised if access with intent WRITE_ATTRIBUTES to the current process node cannot be obtained due to an existing lock on the node.

5.2.2.7. Getting results from a process

```
procedure GET_RESULTS(NODE:            in    NODE_TYPE;
                     RESULTS:        in out RESULTS_LIST);
```

Purpose:

This procedure returns the value of the attribute RESULTS of the process node identified by NODE. The process need not have terminated or aborted. The empty list is returned in RESULTS if WRITE_RESULTS or APPEND_RESULTS has not been called by the process contained in the node identified by NODE.

Parameters:

NODE is an open node handle on a process node.

RESULTS is an unnamed list of strings which returns the value of the **RESULTS** attribute of the process node identified by **NODE**. The individual strings may be extracted from the list using the tools of CAIS.LIST_UTILITIES (see Section 5.4).

Exceptions:

USE_ERROR is raised if the node identified by **NODE** is not a process node.

STATUS_ERROR

is raised if **NODE** is not an open node handle.

INTENT_VIOLATION

is raised if the **NODE** was not opened with an intent establishing the right to read attributes.

Additional Interfaces:

```

procedure GET_RESULTS(NODE: in  MODE_TYPE;
                      RESULTS: in out RESULTS_LIST;
                      STATUS: out PROCESS_STATUS)

```

```

is
begin
  GET_RESULTS(NODE, RESULTS);
  STATUS:=STATUS_OF_PROCESS(NODE);
end GET_RESULTS;

```

```

procedure GET_RESULTS(NAME: in  NAME_STRING;
                      RESULTS: in out RESULTS_LIST;
                      STATUS: out PROCESS_STATUS)

```

```

is
  MODE: MODE_TYPE;
begin
  OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));
  GET_RESULTS(NODE, RESULTS);
  STATUS:=STATUS_OF_PROCESS(NODE);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end GET_RESULTS;

```

```

procedure GET_RESULTS(NAME: in  NAME_STRING;
                      RESULTS: in out RESULTS_LIST)

```

```

is
  MODE: MODE_TYPE;
begin
  OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));
  GET_RESULTS(NODE, RESULTS);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end GET_RESULTS;

```

5.2.2.8. Determining the status of a process

```
function STATUS_OF_PROCESS(NODE: in NODE_TYPE)
    return PROCESS_STATUS;
```

Purpose:

This function returns the current status of the process represented by NODE. It returns the value of the attribute CURRENT_STATUS associated with the process node identified by NODE.

Parameters:

NODE is an open node handle identifying the node of the process whose status is to be queried.

Exceptions:

USE_ERROR is raised if the node identified by NODE is not a process node.

STATUS_ERROR
 is raised if NODE is not an open node handle.

INTENT_VIOLATION
 is raised if the node handle NODE was not opened with an intent establishing the right to read attributes.

Additional Interface:

```
function STATUS_OF_PROCESS(NAME: in NAME_STRING)
    return PROCESS_STATUS
is
    NODE: NODE_TYPE;
    RESULT: PROCESS_STATUS;
begin
    OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := STATUS_OF_PROCESS(NODE);
    CLOSE(NODE);
    return RESULT;
exception
    when others =>
        CLOSE(NODE);
        raise;
end STATUS_OF_PROCESS;
```

5.2.2.9. Getting the parameter list

```
procedure GET_PARAMETERS(PARAMETERS: in out PARAMETER_LIST);
```

Purpose:

This procedure returns the value of the predefined attribute PARAMETERS of the current process node.

Parameters:

PARAMETERS

ACCESS_CONTROL

defines the initial access control information associated with the created node.

LEVEL

defines the classification label for the created node.

Exceptions:

NAME_ERROR

is raised if a node already exists for the node specified by KEY and RELATION, if KEY or RELATION is syntactically illegal, or if any node identifying a group specified in the given ACCESS_CONTROL parameter is unobtainable or inaccessible.

USE_ERROR is raised if any of the parameters ACCESS_CONTROL, LEVEL or ATTRIBUTES is syntactically or semantically illegal. USE_ERROR is also raised if interpretation of the ATTRIBUTES parameter would result in modification or creation of any predefined attribute. USE_ERROR is also raised if RELATION is the name of a predefined relation that cannot be modified or created by the user.

STATUS_ERROR

is raised if BASE is not an open node handle or if FILE is an open file handle prior to the call.

INTENT_VIOLATION

is raised if BASE was not opened with an intent establishing the right to append relationships.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```

procedure CREATE(FILE:          in out FILE_TYPE;
                  NAME:         in  NAME_STRING;
                  MODE:         in  FILE_MODE := INPUT_FILE;
                  FORM:         in  LIST_TYPE := EMPTY_LIST;
                  ATTRIBUTES:   in  LIST_TYPE := EMPTY_LIST;
                  ACCESS_CONTROL: in  LIST_TYPE := EMPTY_LIST;
                  LEVEL:        in  LIST_TYPE := EMPTY_LIST)
is
    BASE : MODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(NAME), (1=>APPEND_RELATIONSHIPS));
    CREATE(FILE, BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
           MODE, FORM, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
    CLOSE(BASE);
exception
    when others =>
        CLOSE(FILE);
        CLOSE(BASE);
        raise;
end CREATE;

```

FILE_MODE indicates whether input operations, output operations or both can be performed on the direct-access file.

5.3.2.2. Creating a direct input or output file

```
procedure CREATE(FILE:          in out FILE_TYPE;  
                  BASE:         in  MODE_TYPE;  
                  KEY:          in  RELATIONSHIP_KEY :=  
                                LATEST_KEY;  
                  RELATION:     in  RELATION_NAME :=  
                                DEFAULT_RELATION;  
                  MODE:         in  FILE_MODE := INOUT_FILE;  
                  FORM:         in  LIST_TYPE := EMPTY_LIST;  
                  ATTRIBUTES:   in  LIST_TYPE := EMPTY_LIST;  
                  ACCESS_CONTROL: in  LIST_TYPE := EMPTY_LIST;  
                  LEVEL:        in  LIST_TYPE := EMPTY_LIST);
```

Purpose:

This procedure creates a file and its file node; each element of the file is directly addressable by an index. The [LRM] defines what constitutes an *element*. The attribute ACCESS_METHOD is assigned the value "(DIRECT, SEQUENTIAL)" as part of the creation.

The FORM parameter is used to provide file characteristics concerning the creation of the file. The predefined file characteristic ESTIMATED_SIZE may be used to specify an approximation to the number of storage units (i.e., bytes or blocks) that should be writable to the file. The ESTIMATED_SIZE characteristic is specified as "(ESTIMATED_SIZE => n)", where "n" is any NATURAL number.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node. The ACCESS_CONTROL parameter specifies initial access control information to be established for the created node (see Section 4.4.2.1 for details).

The LEVEL parameter specifies the security level at which the file node is to be created.

The value of the attribute FILE_KIND for the file node will be SECONDARY_STORAGE.

Parameters:

FILE is a file handle, initially closed, to be opened.

BASE is an open node handle to the node which will be the source of the primary relationship to the new node.

KEY is the relationship key of the primary relationship to be created.

RELATION is the relation name of the primary relationship to be created.

MODE indicates the mode of the file.

FORM indicates file characteristics

ATTRIBUTES defines initial values for attributes of the newly created node.

5.3.1. Package IO_DEFINITIONS

This package defines the types and exceptions associated with file nodes.

```
type CHARACTER_ARRAY is array (CHARACTER) of BOOLEAN;  
type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE, APPEND_FILE);  
type FILE_TYPE is limited private;  
type FUNCTION_KEY_DESCRIPTOR (LENGTH: POSITIVE) is private;  
type TAB_ENUMERATION is (HORIZONTAL, VERTICAL);  
type POSITION_TYPE is  
  record  
    ROW      : NATURAL;  
    COLUMN   : NATURAL;  
  end record;
```

CHARACTER_ARRAY provides information concerning the characters that can be obtained during a GET operation. FILE_MODE indicates the type of operations that are to be permitted on a file. Analogous to the [LRM] type FILE_TYPE and the CAIS type NODE_TYPE, the CAIS provides a type FILE_TYPE whose values are references to internal files. FILE_TYPE is used for controlling the operations on all files. FUNCTION_KEY_DESCRIPTOR is used to determine the function keys entered from a terminal. TAB_ENUMERATION is used to specify the kind of tab stop to be set. POSITION_TYPE is used to specify a position on a terminal.

This package also provides the definitions for all exceptions generated by the input and output packages. These definitions are comparable to those specified in the package IO_EXCEPTIONS in the [LRM].

5.3.2. Package DIRECT_IO

This package provides facilities for direct-access input and output to CAIS files comparable to those described in the DIRECT_IO package of [LRM]. Files written with the CAIS.DIRECT_IO are also readable by CAIS.SEQUENTIAL_IO, if the two packages are instantiated with the same generic data type.

The package specification and semantics of the CAIS.DIRECT_IO are comparable to those of the [LRM] package DIRECT_IO. All subprograms present in the [LRM] package DIRECT_IO are present in this CAIS package. The following sections demonstrate only the specifications and semantics that differ.

5.3.2.1. Subtypes and constants

```
subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;  
subtype FILE_MODE is CAIS.IO_DEFINITIONS.FILE_MODE;
```

FILE_TYPE describes the type for file handles for all direct input and output operations.

Table IX. File node predefined attributes, attribute values and relation

	Secondary Storage	Queue	Terminal	Magnetic Tape
ACCESS_METHOD	A	A	A	A
SEQUENTIAL	V	V		
DIRECT	V			
TEXT	V	V	V	V
FILE_KIND	A	A	A	A
SECONDARY_STORAGE	V			
QUEUE		V		
TERMINAL			V	
MAGNETIC_TAPE				V
QUEUE_KIND		A		
SOLD		V		
MIMIC		V		
COPY		V		
TERMINAL_KIND			A	
SCROLL			V	
PAGE			V	
FORM			V	
COUPLE		A		

A = an attribute or relation which applies to the file node
V = an attribute value which the attribute can have for the file node.

The input and output operations in the packages in this section are expressed as operations on objects of some file type, rather than directly in terms of the external files. These objects are files which are internal to a CAIS process (*internal files*). Internal files are identified by *file handles*. Throughout this document, the word *file* is used to mean an Ada external file, while in the [LRM] the word file is used to mean an internal file. The mode of a file determines the intents with which its associated file node can be opened. These corresponding modes and intents are given in TABLE X.

Table X. Modes and intents

If the MODE is:	the INTENT must establish the right to:
IN_FILE	read contents
OUT_FILE	write contents
INFILE	read and write contents
APPEND_FILE	append contents

It is possible that a single file node may have more than one access method, as specified by the predefined attribute ACCESS_METHOD. The value of the attribute ACCESS_METHOD determines the packages that may operate upon the file. The predefined values for the attribute ACCESS_METHOD are SEQUENTIAL, DIRECT, and TEXT or any list combination of these. A value of SEQUENTIAL indicates that the CAIS.SEQUENTIAL_IO package may be used. A value of DIRECT indicates that the package CAIS.DIRECT_IO may be used. A value of TEXT indicates that the package CAIS.TEXT_IO may be used.

The attribute FILE_KIND denotes the kind of file that is represented by the contents of the file node. The predefined values for the attribute FILE_KIND are SECONDARY_STORAGE, QUEUE, TERMINAL, and MAGNETIC_TAPE. These values determine which packages may be used to operate on files, as shown in TABLE VIII.

File nodes with a FILE_KIND value of QUEUE also have a predefined attribute QUEUE_KIND. The predefined values for the attribute QUEUE_KIND are SOLO, MIMIC, and COPY.

File nodes with a FILE_KIND value of TERMINAL also have a predefined attribute TERMINAL_KIND. The values SCROLL, PAGE, and FORM are predefined for this attribute. In addition, terminal file nodes will have a value of TEXT for the attribute ACCESS_METHOD.

When a QUEUE file node is created with QUEUE_KIND of COPY or MIMIC, a relationship of the predefined relation COUPLE is established from the QUEUE node to the file node which provides the queue's initial contents.

The above discussion is summarized in TABLE IX.

31 JANUARY 1985

Table VIII. Input and output packages for file kinds

	Secondary Storage	Queue	Terminal	Magnetic Tape
CAIS.IO_CONTROL	X	X	X	X
CAIS.IO_DEFINITIONS	X	X	X	X
CAIS.SEQUENTIAL_IO	X	X		
CAIS.DIRECT_IO	X			
CAIS.TEXT_IO	X	X	X	X
CAIS.SCROLL_TERMINAL			X	
CAIS.PAGE_TERMINAL			X	
CAIS.FORM_TERMINAL			X	
CAIS.MAGNETIC_TAPE				X

A secondary storage file in the CAIS represents a disk or other random access storage file. Secondary storage files may be created by use of the CREATE procedures specified in the packages CAIS.SEQUENTIAL_IO, CAIS.DIRECT_IO, and CAIS.TEXT_IO.

A queue file in the CAIS represents a sequence of information that is accessed in a first-in, first-out manner. There are three kinds of CAIS queue files: solo, copy and mimic. A solo queue operates like a simple queue, initially empty, in which all writes append information to the end and all reads are destructive. A copy queue operates like a solo queue except that it has initial contents which are copied from another file; after the creation of the copy queue, it is independent of the file. A mimic queue operates like a solo queue except that it has initial contents that are the same as the contents of another file; after the creation of the mimic queue, the mimic queue and the file are mutually dependent. This means that, if information is written to the mimic queue file, it is appended to the other file as well at an implementation defined time which is no later than CLOSE of the mimic queue file; the effect on the mimic queue file of writing or appending to the other file is implementation defined. Solo queue files may be created by use of the CREATE procedures in the packages CAIS.SEQUENTIAL_IO and CAIS.TEXT_IO. Copy and mimic queue files may be created by use of the COUPLE procedure in the package IO_CONTROL.

A terminal file in the CAIS represents an interactive terminal device. Three kinds of terminal devices are distinguished in the CAIS: scroll, page and form. These are distinguished because they have different characteristics which require specialized interfaces. Scroll and page terminals may be represented either by a single terminal file for input and output or by two terminal files, one for input and one for output. The implementation determines, for each physical terminal, whether it will be represented by one or two terminal files. If two terminal files are used to represent the terminal input and output, then the implementation maintains an implicit association between the two files. A form terminal is represented by a single terminal file for both input and output.

A magnetic tape drive file in the CAIS represents a magnetic tape drive. Operations on magnetic tape drive files can affect either the magnetic tape or the drive. Interfaces must be provided outside the CAIS for the creation of terminal files and magnetic tape drive files.

Several predefined attributes are applicable to file nodes. The attributes ACCESS_METHOD, FILE_KIND, QUEUE_KIND, and TERMINAL_KIND provide information about the contents of a file node and how it may be accessed.

5.2.2.17. Determining the time a process has been active

```
function MACHINE_TIME (NODE : in NODE_TYPE)
    return DURATION;
```

Purpose:

This function returns a value of type DURATION representing the value of the predefined attribute MACHINE_TIME of the process node identified by NODE.

Parameters:

NODE is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

USE_ERROR is raised if the node identified by NODE is not a process node.

STATUS_ERROR

is raised if NODE is not an open node handle.

INTENT_VIOLATION

is raised if the node handle NODE was not opened with an intent establishing the right to read attributes.

Additional Interface:

```
function MACHINE_TIME (NAME : in NAME_STRING)
    return DURATION
is
    NODE: NODE_TYPE;
    RESULT: DURATION;
begin
    OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := MACHINE_TIME(NODE);
    CLOSE(NODE);
    return RESULT;
exception
    when others =>
        CLOSE(NODE);
        raise;
end MACHINE_TIME;
```

5.3. CAIS input and output

The CAIS defines four kinds of files: secondary storage files, queue files, terminal files and magnetic tape drive files. CAIS files are supported by CAIS input and output packages as described in TABLE VIII.

```

    RESULT: TIME;
begin
    OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := START_TIME(NODE);
    CLOSE(NODE);
    return RESULT;
exception
    when others =>
        CLOSE(NODE);
        raise;
end START_TIME;

```

5.2.2.16. Determining the time of termination or abortion

```

function FINISH_TIME (NODE : in NODE_TYPE)
    return TIME;

```

Purpose:

This function returns a value of type *TIME* representing the value of the predefined attribute *FINISH_TIME* of the process node identified by *NODE*.

Parameters:

NODE is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

USE_ERROR is raised if the node identified by *NODE* is not a process node.

STATUS_ERROR

is raised if *NODE* is not an open node handle.

INTENT_VIOLATION

is raised if the node handle *NODE* was not opened with an intent establishing the right to read attributes.

Additional Interface:

```

function FINISH_TIME (NAME : in NAME_STRING)
    return TIME
is
    NODE: NODE_TYPE;
    RESULT: TIME;
begin
    OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := FINISH_TIME(NODE);
    CLOSE(NODE);
    return RESULT;
exception
    when others =>
        CLOSE(NODE);
        raise;
end FINISH_TIME;

```

is raised if the node handle was not opened with an intent establishing the right to read attributes.

Additional Interface:

```
function IO_UNITS (NAME : in NAME_STRING)
    return NATURAL
is
    NODE: NODE_TYPE;
    RESULT: NATURAL;
begin
    OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := IO_UNITS(NODE);
    CLOSE(NODE);
    return RESULT;
exception
    when others =>
        CLOSE(NODE);
        raise;
end IO_UNITS;
```

5.2.2.15. Determining the time of activation

```
function START_TIME (NODE : in NODE_TYPE)
    return TIME;
```

Purpose:

This function returns a value of type TIME representing the value of the predefined attribute START_TIME of the process node identified by NODE.

Parameters:

NODE is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

USE_ERROR is raised if the node identified by NODE is not a process node.

STATUS_ERROR

is raised if NODE is not an open node handle.

LOCK_ERROR

is raised if the node is locked against reading attributes.

INTENT_VIOLATION

is raised if the node handle NODE was not opened with an intent establishing the right to read attributes.

Additional Interface:

```
function START_TIME (NAME : in NAME_STRING)
    return TIME
is
    NODE: NODE_TYPE;
```

Exceptions:

USE_ERROR is raised if the node identified by NODE is not a process node.

STATUS_ERROR

is raised if NODE is not an open node handle.

INTENT_VIOLATION

is raised if the node handle NODE was not opened with an intent establishing the right to read attributes.

Additional Interface:

```
function HANDLES_OPEN (NAME : in NAME_STRING)
    return NATURAL
is
    NODE: NODE_TYPE;
    RESULT: NATURAL;
begin
    OPEN(NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := HANDLES_OPEN(NODE);
    CLOSE(NODE);
    return RESULT;
exception
    when others =>
        CLOSE(NODE);
        raise;
end HANDLES_OPEN;
```

5.2.2.14. Determining the number of input and output units used

```
function IO_UNITS (NODE : in NODE_TYPE)
    return NATURAL;
```

Purpose:

This function returns a natural number representing the value of the predefined attribute IO_UNITS of the process node identified by NODE.

Parameters:

NODE is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

USE_ERROR is raised if the node identified by NODE is not a process node.

STATUS_ERROR

is raised if NODE is not an open node handle.

LOCK_ERROR

is raised if the node is locked against reading attributes.

INTENT_VIOLATION

NODE is an open node handle identifying the node of the process to be resumed.

Exceptions:

USE__ERROR is raised if the node identified by **NODE** is not a process node.

STATUS__ERROR

is raised if **NODE** is not an open node handle.

INTENT__VIOLATION

is raised if the node handle **NODE** was not opened with an intent establishing rights to read relationships and to write attributes and contents.

ACCESS__VIOLATION

is raised if the current process does not have sufficient discretionary access rights to obtain access to any node of a process to be suspended with intent including **READ__RELATIONSHIPS**, **WRITE__ATTRIBUTES** and **WRITE__CONTENTS**.

SECURITY__VIOLATION

is raised if the attempt to obtain access to the node identified by **NODE** represents a violation of the mandatory access controls for the CAIS. **SECURITY__VIOLATION** is raised only if the conditions for raising the other exceptions are not satisfied.

Additional Interface:

```
procedure RESUME_PROCESS(NAME: in NAME_STRING)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (READ_RELATIONSHIPS, WRITE_ATTRIBUTES,
                    WRITE_CONTENTS));
  RESUME_PROCESS(NODE);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end RESUME_PROCESS;
```

5.2.2.13. Determining the number of open node handles

```
function HANDLES_OPEN (NODE : in NODE_TYPE)
  return NATURAL;
```

Purpose:

This function returns a natural number representing the value of the predefined attribute **HANDLES__OPEN** of the process node identified by **NODE**.

Parameters:

NODE is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

USE_ERROR is raised if the node identified by **NODE** is not a process node.

STATUS_ERROR

is raised if **NODE** is not an open node handle.

INTENT_VIOLATION

is raised if the node handle **NODE** was not opened with an intent establishing rights to read relationships and to write attributes and contents.

ACCESS_VIOLATION

is raised if the current process does not have sufficient discretionary access rights to obtain access to any node of a process to be suspended with intent including **READ_RELATIONSHIPS**, **WRITE_ATTRIBUTES** and **WRITE_CONTENTS**.

SECURITY_VIOLATION

is raised if the attempt to obtain access to the node identified by **NODE** represents a violation of the mandatory access controls for the CAIS. **SECURITY_VIOLATION** is raised only if conditions for raising the other exceptions are not satisfied.

Additional Interface:

```
procedure SUSPEND_PROCESS(NAME: in NAME_STRING)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (READ_RELATIONSHIPS, WRITE_ATTRIBUTES,
                   WRITE_CONTENTS));
  SUSPEND_PROCESS(NODE);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end SUSPEND_PROCESS;
```

Notes:

SUSPEND_PROCESS can be used by a task to suspend the process that contains it.

5.2.2.1. Resuming a process

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

Purpose:

This procedure causes the process represented by **NODE** to resume execution. **RESUME_PROCESS** does not change the process status if the process is not suspended. After **RESUME_PROCESS** is called, the **PROCESS_STATUS** of the identified process is **READY** provided that the process was in the **SUSPENDED** status at the time that the resumption took effect. If the node identified by **NODE** is the parent of other process nodes, the other processes are likewise resumed. If an exception is raised, none of the processes is resumed.

Parameters:

a violation of mandatory access controls in the CAIS. SECURITY_VIOLATION is raised only if the conditions for raising the other exceptions are not satisfied.

Additional Interfaces:

```
procedure ABORT_PROCESS(NAME: in NAME_STRING;
                        RESULTS: in RESULTS_STRING)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (READ_RELATIONSHIPS, WRITE_CONTENTS,
                    WRITE_ATTRIBUTES));
  ABORT_PROCESS(NODE, RESULTS);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end ABORT_PROCESS;

procedure ABORT_PROCESS(NODE: in NODE_TYPE)
is
begin
  ABORT_PROCESS(NODE, "ABORTED");
end ABORT_PROCESS;

procedure ABORT_PROCESS(NAME: in NAME_STRING)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (READ_RELATIONSHIPS, WRITE_CONTENTS, WRITE_ATTRIBUTES));
  ABORT_PROCESS(NODE, "ABORTED");
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end ABORT_PROCESS;
```

Notes:

ABORT_PROCESS can be used by a task to abort the process that contains it. It is intentional that LOCK_ERROR will not be raised by this procedure.

5.2.2.11. Suspending a process

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

Purpose:

This procedure suspends the process represented by NODE. After SUSPEND_PROCESS is called, the CURRENT_STATUS of the identified process is SUSPENDED, provided that the process was in the READY status at the time that the suspension took effect. SUSPEND_PROCESS does not change the process status if the process is not in the READY state. If the node identified by NODE is the parent of other process nodes, the other processes are likewise suspended. If an exception is raised, none of the processes are suspended.

Parameters:

NODE is an open node handle identifying the node of the process to be suspended.

is a list containing parameter information. The list is constructed and can be manipulated using the tools provided in CAIS.LIST_UTILITIES.

Exceptions:

LOCK_ERROR

is raised if access with intent READ_ATTRIBUTES to the current process node cannot be obtained due to an existing lock on the node.

Notes:

The value of the predefined attribute PARAMETERS is set during process node creation; see the interfaces SPAWN_PROCESS, INVOKE_PROCESS and CREATE_JOB.

5.2.2.10. Aborting a process

```
procedure ABORT_PROCESS(NODE:      in NODE_TYPE;
                        RESULTS: in RESULTS_STRING);
```

Purpose:

This procedure aborts the process represented by NODE and forces any processes in the subtree rooted at the identified process to be aborted. The order of the process abortions is not specified. If the state of the process represented by NODE after return of ABORT_PROCESS is examined, it will be ABORTED or TERMINATED; it will be TERMINATED only if the process terminated before ABORT_PROCESS took effect. The node associated with the aborted process remains until explicitly deleted. If an exception is raised, none of the processes are aborted.

Parameters:

NODE is an open node handle for the node of the process to be aborted.

RESULTS is a string to be appended to the RESULTS attribute of the node represented by NODE.

Exceptions:

USE_ERROR is raised if the node identified by NODE is not a process node.

STATUS_ERROR

is raised if NODE is not an open node handle.

INTENT_VIOLATION

is raised if the node was not opened with an intent establishing rights to read relationships and to write attributes and contents.

ACCESS_VIOLATION

is raised if the current process does not have sufficient discretionary access control rights to obtain access to any node of a process to be aborted with intent including READ_RELATIONSHIPS, WRITE_ATTRIBUTES and WRITE_CONTENTS.

SECURITY_VIOLATION

is raised if the attempt to obtain access to the node identified by NODE represents

5.3.2.3. Opening a direct input or output file

```
procedure OPEN(FILE: in out FILE_TYPE;  
               MODE: in  MODE_TYPE;  
               MODE: in  FILE_MODE);
```

Purpose:

This procedure opens a file handle on a file, given an open node handle to the file node; each element of the file is directly addressable by an index.

Parameters:

FILE is a file handle, initially closed, to be opened.

NODE is an open node handle to the file node.

MODE indicates the mode of the file.

Exceptions:

USE_ERROR is raised if the attribute ACCESS_METHOD of the file node does not have the value DIRECT, if the element type of the file does not correspond with the element type of this instantiation of the CAIS.DIRECT_IO package, or if the mode is APPEND_FILE.

STATUS_ERROR

is raised if FILE is an open file handle at the time of the call on OPEN or if NODE is not an open node handle.

INTENT_VIOLATION

is raised if NODE was not opened with an intent establishing the access rights required for the MODE, as specified in TABLE X.

Additional Interface:

```
procedure OPEN(FILE: in out FILE_TYPE;  
               NAME: in  NAME_STRING;  
               MODE: in  FILE_MODE )  
is  
  NODE : MODE_TYPE;  
begin  
  case MODE is  
    when IN_FILE => OPEN(NODE, NAME, (1=>READ_CONTENTS));  
    when OUT_FILE => OPEN(NODE, NAME, (1=>WRITE_CONTENTS));  
    when INOUT_FILE => OPEN(NODE, NAME,  
                           (READ_CONTENTS, WRITE_CONTENTS));  
    when APPEND_FILE => raise USE_ERROR;  
  end case;  
  OPEN(FILE, NODE, MODE)  
  CLOSE(NODE);  
exception  
  when others =>  
    CLOSE(FILE);  
    CLOSE(NODE);  
  raise;  
end OPEN;
```

Notes:

The effects on the open file handle of closing an open node handle on its node are implementation-defined. In particular, no assumption can be made about the access protection provided by the node model.

5.3.2.4. Deleting a direct input or output file

procedure DELETE(FILE:in out FILE_TYPE):

Purpose:

In addition to the semantics specified in [LRM], if the node associated with the open file handle FILE is not already unobtainable, this node is made unobtainable as if a call to the DELETE_NODE procedure had been made. If this node is already unobtainable by this call, no exception other than STATUS_ERROR may be raised by this procedure.

Parameters:

FILE is an open file handle on the file being deleted.

Exceptions:

NAME_ERROR

is raised if the parent node of the node associated with the file identified by FILE is inaccessible.

USE_ERROR is raised if any primary relationships emanate from the node associated with the file identified by FILE.

STATUS_ERROR

is raised if FILE is not an open file handle.

LOCK_ERROR

is raised if access with intent WRITE_RELATIONSHIPS to the parent of the node to be deleted cannot be obtained due to an existing lock on the node.

ACCESS_VIOLATION

is raised if the current process does not have sufficient discretionary access control rights to obtain access to the parent of the node to be deleted with intent WRITE_RELATIONSHIPS or to obtain access to the node to be deleted with intent EXCLUSIVE_WRITE. ACCESS_VIOLATION is only raised if the conditions for NAME_ERROR are not present.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

5.3.3. Package SEQUENTIAL_IO

This package provides facilities for sequentially accessing data elements in CAIS files. [LRM] defines what constitutes an element. These facilities are comparable to those described in the SEQUENTIAL_IO package of [LRM].

The package specification and semantics of the CAIS.SEQUENTIAL_IO are comparable to those of the [LRM] package SEQUENTIAL_IO. All subprograms present in the [LRM] package SEQUENTIAL_IO are present in this CAIS package. The following sections demonstrate only the specifications and semantics that differ.

5.3.3.1. Subtypes and constants

subtype FILE_TYPE **is** CAIS.IO_DEFINITIONS.FILE_TYPE;

subtype FILE_MODE **is** CAIS.IO_DEFINITIONS.FILE_MODE;

FILE_TYPE describes the type for file handles for all sequential input and output operations. FILE_MODE indicates whether input operations, output operations or both can be performed on the sequential-access file. A mode of APPEND_FILE causes any elements that are written to the specified file to be appended to the elements that are already in the file.

5.3.3.2. Creating a sequential input or output file

```
procedure CREATE(FILE:           in out FILE_TYPE;
                  BASE:          in  MODE_TYPE;
                  KEY:           in  RELATIONSHIP_KEY := LATEST_KEY;
                  RELATION:      in  RELATION_NAME := DEFAULT_RELATION;
                  MODE:          in  FILE_MODE := INOUT_FILE;
                  FORM:          in  LIST_TYPE := EMPTY_LIST;
                  ATTRIBUTES:    in  LIST_TYPE := EMPTY_LIST;
                  ACCESS_CONTROL: in  LIST_TYPE := EMPTY_LIST;
                  LEVEL:         in  LIST_TYPE := EMPTY_LIST);
```

Purpose:

This procedure creates a file and its file node; each element of the file is sequentially accessible. The attribute ACCESS_METHOD is assigned the value "(SEQUENTIAL)" as part of the creation.

The FORM parameter is used to provide file characteristics concerning the creation of the file. The predefined file characteristic ESTIMATED_SIZE may be used to specify an approximation to the number of storage units (e.g., bytes or blocks) that should be writable to the file. The ESTIMATED_SIZE characteristic is specified as "(ESTIMATED_SIZE => n)", where "n" is any NATURAL number.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node. The ACCESS_CONTROL parameter specifies initial access control information to be established for the created node (see Section 4.4.2.1 for details).

The LEVEL parameter specifies the security level at which the file node is to be created.

The default value for the attribute FILE_KIND for the file node is SECONDARY_STORAGE. The default value may be overridden by explicitly specifying a value of QUEUE in the attributes parameter (i.e., "(FILE_KIND => QUEUE)"), in which case the value of the attribute QUEUE_KIND is SOLO.

Parameters:

FILE is a file handle, initially closed, to be opened.

BASE is an open node handle to the node which will be the source of the primary relationship to the new node.

KEY is the relationship key of the primary relationship to be created.

RELATION is the relation name of the primary relationship to be created.

MODE indicates the mode of the file.

FORM indicates file characteristics.

ATTRIBUTES defines initial values for attributes of the newly created node.

ACCESS_CONTROL
defines the initial access control information associated with the created node.

LEVEL defines the classification label for the created node.

Exceptions:

NAME_ERROR
is raised if a node already exists for the node specified by KEY and RELATION or if KEY or RELATION is syntactically illegal or if any node identifying a group specified in the given ACCESS_CONTROL parameter is unobtainable or inaccessible.

USE_ERROR is raised if any of the parameters ACCESS_CONTROL, LEVEL or ATTRIBUTES is syntactically or semantically illegal. USE_ERROR is also raised if interpretation of the ATTRIBUTES parameter would result in creation of any predefined attribute other than FILE_KIND. USE_ERROR is also raised if RELATION is the name of a predefined relation that cannot be created by the user.

STATUS_ERROR
is raised if BASE is not an open node handle or if FILE is an open file handle prior to the call.

INTENT_VIOLATION
is raised if BASE was not opened with an intent establishing the right to append relationships.

31 JANUARY 1985

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```

procedure CREATE(FILE:      in out FILE_TYPE;
                  NAME:      in   NAME_STRING;
                  MODE:      in   FILE_MODE := INOUT_FILE;
                  FORM:      in   LIST_TYPE := EMPTY_LIST;
                  ATTRIBUTES: in   LIST_TYPE := EMPTY_LIST;
                  ACCESS_CONTROL: in LIST_TYPE := EMPTY_LIST;
                  LEVEL:      in   LIST_TYPE := EMPTY_LIST)
is
    BASE : NODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(NAME), (1=>APPEND_RELATIONSHIPS));
    CREATE(FILE, BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
           MODE, FORM, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
    CLOSE(BASE);
exception
    when others =>
        CLOSE(FILE);
        CLOSE(BASE);
        raise;
end CREATE;

```

5.3.3.3. Opening a sequential input or output file

```

procedure OPEN(FILE:      in out FILE_TYPE;
                NODE:      in   NODE_TYPE;
                MODE:      in   FILE_MODE);

```

Purpose:

This procedure opens a file handle on a file, given an open node handle on the file node; each element of the file is sequentially accessible.

Parameters:

FILE is a file handle, initially closed, to be opened.

NODE is an open node handle to the file node.

MODE indicates the mode of the file.

Exceptions:

USE_ERROR is raised if the attribute **ACCESS_METHOD** of the file node does not have the value **SEQUENTIAL** or if the element type of the file does not correspond with the element type of this instantiation of the **CAIS.SEQUENTIAL_IO** package. **USE_ERROR** is also raised if the node identified by **NODE** has a value of **QUEUE** for the attribute **FILE_KIND** and a value of **MIMIC** for the attribute **QUEUE_KIND** and the mimic queue file identified by **FILE** is being opened with **MODE** other than **IN_FILE** but the coupled file (see Section 5.3.5.13) has been deleted.

STATUS_ERROR

is raised if FILE is an open file handle at the time of the call on OPEN or if NODE is not an open node handle.

INTENT_VIOLATION

is raised if NODE was not opened with an intent establishing the access rights required for the MODE, as specified in TABLE X.

Additional Interface:

```
procedure OPEN(FILE: in out FILE_TYPE;  
               NAME: in   NAME_STRING;  
               MODE: in   FILE_MODE )  
is  
  NODE : MODE_TYPE;  
begin  
  case MODE is  
    when IN_FILE => OPEN(NODE, NAME, (1=>READ_CONTENTS));  
    when OUT_FILE => OPEN(NODE, NAME, (1=>WRITE_CONTENTS));  
    when INOUT_FILE=>OPEN(NODE,NAME,  
                          (READ_CONTENTS,WRITE_CONTENTS));  
    when APPEND_FILE => OPEN(NODE, NAME, (1=>APPEND_CONTENTS));  
  end case;  
  OPEN(FILE, NODE, MODE);  
  CLOSE(NODE);  
exception  
  when others =>  
    CLOSE(FILE);  
    CLOSE(NODE);  
  raise;  
end OPEN;
```

5.3.3.4. Deleting a sequential input or output file

```
procedure DELETE(FILE: in out FILE_TYPE);
```

Purpose:

In addition to the semantics specified in [LRM], if the node associated with the open file handle FILE is not already unobtainable, this node is made unobtainable as if a call to the DELETE_NODE procedure had been made. If this node is already unobtainable by this call, no exception other than STATUS_ERROR may be raised by this procedure.

Parameters:

FILE is an open file handle on the file being deleted.

Exceptions:

NAME_ERROR

is raised if the parent node of the node associated with the file identified by FILE is inaccessible.

USE_ERROR is raised if any primary relationships emanate from the node associated with the file identified by FILE.

STATUS_ERROR

is raised if FILE is not open file handle.

LOCK_ERROR

is raised if access with intent WRITE_RELATIONSHIPS to the parent of the node to be deleted cannot be obtained due to an existing lock on the node.

ACCESS_VIOLATION

is raised if the current process does not have sufficient discretionary access control rights to obtain access to the parent of the node to be deleted with intent WRITE_RELATIONSHIPS or to obtain access to the node to be deleted with intent EXCLUSIVE_WRITE. ACCESS_VIOLATION is only raised if the conditions for NAME_ERROR are not present.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

5.3.4. Package TEXT_IO

This package provides facilities for the input and output of textual data to CAIS files. [LRM] defines what constitutes an element of data. These facilities are comparable to those specified in the package TEXT_IO in [LRM]. All subprograms present in the [LRM] package TEXT_IO are present in this CAIS package. The following sections demonstrate only the specifications and semantics that differ.

5.3.4.1. Subtypes and constants

subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;

subtype FILE_MODE is CAIS.IO_DEFINITIONS.FILE_MODE;

FILE_TYPE describes the type for file handles for all text input and output operations. FILE_MODE indicates whether input operations, output operations or both can be performed on the text file. A mode of APPEND_FILE causes any text written to the specified file to be appended to the text that is already in the file.

5.3.4.2. Creating a text input or output file

```
procedure CREATE(FILE:      in out FILE_TYPE;
                     BASE:    in      MODE_TYPE;
                     KEY:      in      RELATIONSHIP_KEY := LATEST_KEY;
                     RELATION: in      RELATION_NAME := DEFAULT_RELATION;
                     MODE:     in      FILE_MODE := INOUT_FILE;
                     FORM:      in      LIST_TYPE := EMPTY_LIST;
                     ATTRIBUTES: in      LIST_TYPE := EMPTY_LIST;
                     ACCESS_CONTROL: in      LIST_TYPE := EMPTY_LIST;
                     LEVEL:     in      LIST_TYPE := EMPTY_LIST);
```

Purpose:

This procedure creates a file and its file node; the file is textual. The attribute ACCESS_METHOD is assigned the value "(TEXT)" as part of the creation.

The FORM parameter is used to provide file characteristics concerning the creation of the external file. The predefined file characteristic ESTIMATED_SIZE may be used to specify an approximation to the number of storage units (e.g., bytes or blocks) that should be writable to the file. The ESTIMATED_SIZE characteristic is specified as "(ESTIMATED_SIZE => n)", where "n" is any NATURAL number.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node. The ACCESS_CONTROL parameter specifies initial access control information to be established for the created node (see Section 4.4.2.1 for details).

The LEVEL parameter specifies the security level at which the file node is to be created.

The default value for the attribute FILE_KIND is SECONDARY_STORAGE. The default value may be overridden by explicitly specifying a value of QUEUE in the ATTRIBUTES parameter i.e., "(FILE_KIND => QUEUE)". If the value of FILE_KIND is QUEUE, the default value of the attribute QUEUE_KIND is SOLO.

Parameters:

FILE	is a file handle, initially closed, to be opened.
BASE	is an open node handle to the node which will be the source of the primary relationship to the new node.
KEY	is the relationship key of the primary relationship to be created.
RELATION	is the relation name of the primary relationship to be created.
MODE	indicates the mode of the file.
FORM	indicates file characteristics.
ATTRIBUTES	defines initial values for attributes of the newly created node.
ACCESS_CONTROL	defines the initial access control information associated with the created node.
LEVEL	defines the classification label for the created node.

Exceptions:

NAME_ERROR	is raised if a node already exists for the node specified by KEY and RELATION or if KEY or RELATION is syntactically illegal or if any node identifying a group specified in the given ACCESS_CONTROL parameter is unobtainable.
USE_ERROR	is raised if any of the parameters ACCESS_CONTROL, LEVEL or ATTRIBUTES is syntactically or semantically illegal. USE_ERROR is also raised if interpretation of the ATTRIBUTES parameter would result in modification or creation of a predefined attribute other than FILE_KIND. USE_ERROR is also raised if RELATION is the name of a predefined relation which cannot be created by the user.

STATUS_ERROR

is raised if BASE is not an open node handle or if FILE is an open file handle prior to the call.

INTENT_VIOLATION

is raised if BASE was not opened with an intent establishing the right to append relationships.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls
SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure CREATE(FILE:          in out FILE_TYPE;
                  NAME:         in   NAME_STRING;
                  MODE:         in   FILE_MODE := INOUT_FILE;
                  FORM:         in   LIST_TYPE := EMPTY_LIST;
                  ATTRIBUTES:   in   LIST_TYPE := EMPTY_LIST;
                  ACCESS_CONTROL: in   LIST_TYPE := EMPTY_LIST;
                  LEVEL:        in   LIST_TYPE := EMPTY_LIST)
is
    BASE : NODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(NAME), (1=>APPEND_RELATIONSHIPS));
    CREATE(FILE, BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
           MODE, FORM, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
    CLOSE(BASE);
exception
    when others =>
        CLOSE(FILE);
        CLOSE(BASE);
        raise;
end CREATE;
```

5.3.4.3. Opening a text input or output file

```
procedure OPEN(FILE:          in out FILE_TYPE;
               MODE:         in   MODE_TYPE;
               MODE:         in   FILE_MODE);
```

Purpose:

This procedure opens a file handle on a file that has textual contents, given an open node handle on the file node.

Parameters:

FILE is a file handle, initially closed, to be opened.

NODE is an open node handle to the file node.

MODE indicates the mode of the file.

Exceptions:

USE_ERROR is raised if the attribute **ACCESS_METHOD** of the file node does not have the value **TEXT** or the element type of the file does not correspond with the element type of this instantiation of the **CAIS.TEXT_IO** package. **USE_ERROR** is also raised if the node identified by **NODE** has a value of **QUEUE** for the attribute **FILE_KIND** and a value of **MIMIC** for the attribute **QUEUE_KIND** and the mimic queue file identified by **FILE** is being opened with **MODE** other than **IN_FILE** but the coupled file (see Section 5.3.5.13) has been deleted. **USE_ERROR** is also raised if the node identified by **NODE** has a value of **TERMINAL** or **MAGNETIC_TAPE** for the attribute **FILE_KIND** and the **MODE** is **APPEND_FILE**.

STATUS_ERROR

is raised if **FILE** is an open file handle at the time of the call on **OPEN** or if **NODE** is not an open node handle.

INTENT_VIOLATION

is raised if **NODE** has not been opened with an intent establishing the access rights required for the **MODE**, as specified in TABLE X.

Additional Interface:

```
procedure OPEN(FILE: in out FILE_TYPE;
               NAME: in   NAME_STRING;
               MODE: in   FILE_MODE)
is
  NODE : MODE_TYPE;
begin
  case MODE is
    when IN_FILE => OPEN(NODE, NAME, (1=>READ_CONTENTS));
    when OUT_FILE => OPEN(NODE, NAME, (1=>WRITE_CONTENTS));
    when INOUT_FILE => OPEN(NODE, NAME,
                           (READ_CONTENTS, WRITE_CONTENTS));
    when APPEND_FILE => OPEN(NODE, NAME, (1=>APPEND_CONTENTS));
  end case;
  OPEN(FILE, NODE, MODE);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(FILE);
    CLOSE(NODE);
  raise;
end OPEN;
```

Notes:

If the file identified by **FILE** is a mimic queue file which is being opened to read and its coupled file (see Section 5.3.5.13) has been deleted or has fewer elements than expected to be in the mimic queue file (e.g., if some of the contents of the coupled file have been deleted), read operations on the mimic queue file will encounter an end of file.

5.3.4.4. Deleting a text input or output file

```
procedure DELETE(FILE: in out FILE_TYPE);
```

Purpose:

In addition to the semantics specified in [LRM], the node associated with the open file handle **FILE** is made unobtainable as if a call to the **DELETE_NODE** procedure had been made.

Parameters:

FILE is an open file handle on the file being deleted.

Exceptions:**NAME_ERROR**

is raised if the parent node of the node associated with the file identified by **FILE** is inaccessible.

USE_ERROR is raised if any primary relationships emanate from the node associated with the file identified by **FILE**.

STATUS_ERROR

is raised if **FILE** is not an open file handle.

LOCK_ERROR

is raised if access with intent **WRITE_RELATIONSHIPS** to the parent of the node to be deleted cannot be obtained due to an existing lock on the node.

ACCESS_VIOLATION

is raised if the current process does not have sufficient discretionary access control rights to obtain access to the parent of the node to be deleted with intent **WRITE_RELATIONSHIPS** or to obtain access to the node to be deleted with intent **EXCLUSIVE_WRITE**. **ACCESS_VIOLATION** is only raised if the conditions for **NAME_ERROR** are not present.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. **SECURITY_VIOLATION** is raised only if the conditions for other exceptions are not present.

5.3.4.5. Resetting a text file

```
procedure RESET(FILE: in out FILE_TYPE;  
                MODE: in    FILE_MODE);
```

Purpose:

In addition to the semantics specified in [LRM], application of this procedure to a file which represents a magnetic tape drive will cause the magnetic tape to be rewound to the filemark immediately preceding the current tape position. See Section 5.3.9 for more information on magnetic tapes.

Parameters:

FILE is an open file handle on the file begin reset.

MODE indicates the mode of the file.

Exceptions:

USE_ERROR is raised if the node associated with the file identified by **FILE** has a value of **TERMINAL** or **MAGNETIC_TAPE** for the attribute **FILE_KIND** and the **MODE** is **APPEND_FILE**.

PROPOSED MIL-STD-CAIS
31 JANUARY 1985

5.3.4.6. Reading from a text file

procedure GET(...);

Purpose:

These procedures read characters from the specified text file.

For all values of the attribute FILE_KIND the CAIS defines only reading of the printable ASCII characters plus the format effectors called horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed. All of the printable characters plus the horizontal tabulation and vertical tabulation characters may be read as characters. The characters carriage return and line feed are to be treated as line terminators whether encountered singly or together (i.e., CR, LF, CRLF, and LFCR are line terminators). The character form feed is to be treated as the page terminator.

When text is being read from a file whose file node attribute FILE_KIND has the value TERMINAL, it is expected that most implementations will provide facilities for editing the input entered by the user before making the characters available to a program for reading.

5.3.4.7. Writing to a text file

procedure PUT(...);

Purpose:

These procedures write characters to the specified file.

The CAIS supports the transfer of information to and from a single magnetic tape volume. Data transferred to and from magnetic tapes may consist of the following characters:

Characters	Representation of Characters
all printable characters	corresponding ASCII characters
horizontal tab	ASCII.HT
vertical tab	ASCII.VT
carriage return	ASCII.CR
line terminator	ASCII.LF
page terminator	ASCII.FF
file terminator	zero or more fill characters followed immediately by a tape mark
fill character	ASCII.WUL

Use of other characters is not defined.

5.3.4.8. Setting the input file

procedure SET_INPUT(FILE : in FILE_TYPE);

Purpose:

In addition to the semantics specified in the [LRM], the file node associated with the file identified by FILE becomes the target of the relationship of the predefined relation CURRENT_INPUT of the current process node.

Parameters:

FILE is an open file handle.

Exceptions:

MODE_ERROR

is raised if the mode of the file identified by FILE is OUT_FILE or APPEND_FILE.

STATUS_ERROR

is raised if FILE is not an open file handle.

LOCK_ERROR

is raised if the current process node is LOCKed against writing relationships.

5.3.4.9. Setting the output file

procedure SET_OUTPUT(FILE : in FILE_TYPE);

Purpose:

In addition to the semantics specified in the [LRM], the file node associated with FILE becomes the target of the relationship of the predefined relation CURRENT_OUTPUT of the current process node.

Parameters:

FILE is an open file handle.

Exceptions:

MODE_ERROR

is raised if the mode of the file identified by FILE is IN_FILE.

STATUS_ERROR

is raised if FILE is not an open file handle.

LOCK_ERROR

is raised if the current process node is LOCKed against writing relationships.

5.3.4.10. Setting the error file

procedure SET_ERROR(FILE : in FILE_TYPE);

Purpose:

The file node associated with the file identified by FILE becomes the target of the relationship of the predefined relation CURRENT_ERROR of the current process node.

Parameters:

FILE is an open file handle.

Exceptions:

MODE_ERROR

is raised if the mode of the file identified by FILE is IN_FILE.

STATUS_ERROR

is raised if FILE is not an open file handle.

LOCK_ERROR

is raised if the current process node is LOCKED against writing relationships.

5.3.4.11. Determining the standard error file

function STANDARD_ERROR return FILE_TYPE;

Purpose:

This function returns an open file handle to the target node of the relationship of the predefined relation STANDARD_ERROR that was set at the start of program execution.

Parameters:

None.

Exceptions:

LOCK_ERROR

is raised if the current process node is locked against reading relationships.

5.3.4.12. Determining the current error file

function CURRENT_ERROR return FILE_TYPE;

Purpose:

This function returns an open file handle to the target node of the relationship of the predefined relation CURRENT_ERROR which is either the standard error file or the file specified in the most recent invocation of SET_ERROR in the current process.

Parameters:

None.

Exceptions:

LOCK_ERROR

is raised if the current process node is locked against reading relationships.

5.3.5. Package IO_CONTROL

This package defines facilities that may be used to modify or query the functionality of CAIS files. It provides for association of input and output text files with an output logging file. It also provides facilities for forcing data from an internal file to its associated external file, for manipulation of function keys and prompt strings and for creating mimic and copy queues.

Parameters:

TERMINAL is an open file handle on an output terminal file.

KIND is the kind (horizontal or vertical) of tab stop to be removed.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **SCROLL** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**, or if there is no tab stop of the designated kind at the active position.

MODE_ERROR
is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR
is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR
is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface

```
procedure CLEAR_TAB(KIND in TAB_ENUMERATION := HORIZONTAL)
is
begin
    CLEAR_TAB(CURRENT_OUTPUT, KIND);
end CLEAR_TAB;
```

5.3.6.7. Advancing to the next tab position

```
procedure TAB(TERMINAL in FILE_TYPE;
              KIND      in TAB_ENUMERATION := HORIZONTAL;
              COUNT     in POSITIVE := 1);
```

Purpose:

This procedure advances the active position **COUNT** tab stops. Horizontal advancement causes a change in only the column number of the active position. Vertical advancement causes a change in only the row number of the active position.

Parameters:

TERMINAL is an open file handle on an output terminal file.

KIND is the kind (horizontal or vertical) of tab to be advanced.

COUNT is a positive integer indicating the number of tab stops the active position is to advance.

Exceptions:

```
    return TERMINAL_SIZE(CURRENT_OUTPUT);  
end TERMINAL_SIZE;
```

5.3.6.5. Setting a tab stop

```
procedure SET_TAB(TERMINAL: in FILE_TYPE;  
                  KIND: in TAB_ENUMERATION := HORIZONTAL);
```

Purpose:

This procedure establishes a horizontal tab stop at the column of the active position if KIND is HORIZONTAL, or a vertical tab stop at the row of the active position if KIND is VERTICAL.

Parameters:

TERMINAL is an open file handle on an output terminal file.

KIND is the kind (horizontal or vertical) of tab stop to be set.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or SCROLL is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL. USE_ERROR is also raised if the number of rows for the terminal is unlimited and KIND is VERTICAL.

MODE_ERROR

is raised if the file identified by TERMINAL is of mode IN_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure SET_TAB (KIND: in TAB_ENUMERATION := HORIZONTAL)  
is  
begin  
    SET_TAB(CURRENT_OUTPUT, KIND);  
end SET_TAB;
```

5.3.6.6. Clearing a tab stop

```
procedure CLEAR_TAB(TERMINAL: in FILE_TYPE;  
                   KIND: in TAB_ENUMERATION := HORIZONTAL);
```

Purpose:

This procedure removes a horizontal tab stop from the column of the active position if KIND is HORIZONTAL or a vertical tab stop from the row of the active position if KIND is VERTICAL.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function GET_POSITION
    return POSITION_TYPE
is
begin
    return GET_POSITION(CURRENT_OUTPUT);
end GET_POSITION;
```

5.3.6.4. Determining the size of the terminal

```
function TERMINAL_SIZE(TERMINAL: in FILE_TYPE)
    return POSITION_TYPE;
```

Purpose:

This function returns the maximum row and maximum column of the output terminal file identified by **TERMINAL**. A value of zero for the row number indicates that the row number is unlimited.

Parameters:

TERMINAL is an open file handle on an output terminal file.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **SCROLL** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function TERMINAL_SIZE
    return POSITION_TYPE
is
begin
```

Parameters:

TERMINAL is an open file handle on an output terminal file.

POSITION is the new active position in the output terminal file.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND**, or **SCROLL** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR
is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR
is raised if **TERMINAL** is not an open file handle.

LAYOUT_ERROR
is raised if the position does not exist on the terminal or the position precedes the active position.

DEVICE_ERROR
is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure SET_POSITION (POSITION: in POSITION_TYPE)
is
begin
  SET_POSITION(CURRENT_OUTPUT, POSITION);
end SET_POSITION;
```

5.3.6.3. Determining the active position

```
function GET_POSITION(TERMINAL: in FILE_TYPE)
  return POSITION_TYPE;
```

Purpose:

This function returns the active position of the output terminal file identified by **TERMINAL**.

Parameters:

TERMINAL is an open file handle on an output terminal file.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **SCROLL** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

to open with MODE other than IN-FILE a mimic queue file whose coupled file has been deleted will raise a USE_ERROR exception.

5.3.6. Package SCROLL_TERMINAL

This package provides the functionality of a scroll terminal. A scroll terminal consists of two devices: an input device (keyboard) and an associated output device (a printer or display). A scroll terminal may be accessed either as a single file of mode INOUT_FILE or as two files: one of mode IN_FILE (the keyboard) and the other of mode OUT_FILE (the printer or display). As keys are pressed on the scroll terminal keyboard, the transmitted characters are made available for reading by the CAIS.SCROLL_TERMINAL package. As characters are written to the scroll terminal file, they are displayed on the output device.

The output devices for scroll terminals have *positions* in which printable ASCII characters may be graphically displayed. The positions are arranged into horizontal rows and vertical columns. Each position is identifiable by the combination of a positive row number and a positive column number. An output device for a scroll terminal has a fixed number of columns and might have a fixed number of rows. The rows are incrementally indexed starting with one after performing the NEW_PAGE (see Section 5.3.6.19) operation. The columns are incrementally indexed starting with one at the left side of the output device.

The *active position* on the output device of a scroll terminal is the position at which the next operation will be performed. The active position is said to *advance* if (1) the row number of the new position is greater than the row number of the old position or (2) the row number of the new position is the same as the row number of the old position and the new position has a greater column number. Similarly, a position is said to *precede* the active position if (1) the row number of the position is less than the row number of the active position or (2) the row number of the position is the same as the row number of the active position and the column number of the position is smaller than the column number of the active position.

5.3.6.1. Subtypes

```
subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;  
  
subtype FUNCTION_KEY_DESCRIPTOR is  
    CAIS.IO_DEFINITIONS.FUNCTION_KEY_DESCRIPTOR;  
  
subtype POSITION_TYPE is CAIS.IO_DEFINITIONS.POSITION_TYPE;  
  
subtype TAB_ENUMERATION is CAIS.IO_DEFINITIONS.TAB_ENUMERATION;
```

FILE_TYPE describes the type for file handles. FUNCTION_KEY_DESCRIPTOR is used to obtain information about function keys read from a terminal. POSITION_TYPE describes the type of a position on a terminal. TAB_ENUMERATION is used to specify the kind of tab stop to be set.

5.3.6.2. Setting the active position

```
procedure SET_POSITION(TERMINAL: in FILE_TYPE;  
    POSITION: in POSITION_TYPE);
```

Purpose:

This procedure advances the active position to the specified POSITION in the output terminal file identified by TERMINAL.

```

CLOSE(BASE);
exception
  when others =>
    CLOSE(BASE);
  raise;
end COUPLE;

procedure COUPLE(Queue_Base: in Mode_Type;
  Queue_Key: in Relationship_Key :=
    Latest_Key;
  Queue_Relation: in Relation_Name :=
    Default_Relation;
  File_Name: in Name_String;
  Form: in List_Type := Empty_List;
  Attributes: in List_Type;
  Access_Control: in List_Type := Empty_List;
  Level: in List_Type := Empty_List)
is
  File_Node : Mode_Type;
begin
  OPEN(File_Node, File_Name,
    (READ_ATTRIBUTES, READ_CONTENTS));
  COUPLE(Queue_Base, Queue_Key, Queue_Relation,
    File_Node, Form, Attributes, Access_Control, Level);
  CLOSE(File_Node);
exception
  when others =>
    CLOSE(File_Node);
  raise;
end COUPLE;

procedure COUPLE(Queue_Name: in Name_String;
  File_Name: in Name_String;
  Form: in List_Type := Empty_List;
  Attributes: in List_Type;
  Access_Control: in List_Type := Empty_List;
  Level: in List_Type := Empty_List)
is
  File_Node : Mode_Type;
  Queue_Base : Mode_Type;
begin
  OPEN(Queue_Base, Base_Path(Queue_Name),
    (1=>APPEND_RELATIONSHIPS));
  OPEN(File_Node, File_Name,
    (READ_ATTRIBUTES, READ_CONTENTS));
  COUPLE(Queue_Base, Last_Key(Queue_Name),
    Last_Relation(Queue_Name),
    File_Node, Form, Attributes, Access_Control, Level);
  CLOSE(Queue_Base);
  CLOSE(File_Node);
exception
  when others =>
    CLOSE(Queue_Base);
    CLOSE(File_Node);
  raise;
end COUPLE;

```

Notes:

Read operations on a mimic queue file whose coupled file has been deleted or has fewer elements than expected in the mimic queue file (e.g., if some of the contents of the coupled file have been deleted) will encounter an end of file. Attempts to open a mimic queue file whose coupled file has been deleted with MODE other than IN_FILE raises a USE_ERROR exception. Attempts

31 JANUARY 1985

ATTRIBUTES defines initial values for attributes of the newly created node.

ACCESS_CONTROL

defines the initial access control information associated with the created node.

LEVEL

defines the classification label for the created node.

Exceptions:

NAME_ERROR

is raised if a node already exists for the node identification given by QUEUE_BASE, QUEUE_KEY and QUEUE_RELATION or if this node identification is illegal. NAME_ERROR is also raised if any node identifying a group specified in the ACCESS_CONTROL parameter is unobtainable or inaccessible.

USE_ERROR is raised if the node identified by FILE_NODE is not a file node, does not have a FILE_KIND attribute value of SECONDARY_STORAGE, or has an ACCESS_METHOD attribute value of DIRECT or if the ATTRIBUTES parameter either has no value for the QUEUE_KIND attribute or has the value QUEUE_KIND => SOLO. USE_ERROR is also raised if the FORM, LEVEL, ACCESS_CONTROL or ATTRIBUTES parameters do not adhere to the required syntax. USE_ERROR is also raised if interpretation of the ATTRIBUTES parameter would result in modification or creation of any predefined attributes other than QUEUE_KIND, or if QUEUE_RELATION is the name of a predefined relation which cannot be created by the user.

STATUS_ERROR

is raised if QUEUE_BASE and FILE_NODE are not both open node handles.

INTENT_VIOLATION

is raised if QUEUE_BASE was not opened with an intent establishing the right to append relationships or if FILE_NODE was not opened with an intent establishing the right to read contents and attributes.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interfaces:

```

procedure COUPLE(QUEUE_NAME:    in NAME_STRING;
                  FILE_NODE:     in NODE_TYPE;
                  FORM:           in LIST_TYPE := EMPTY_LIST;
                  ATTRIBUTES:    in LIST_TYPE;
                  ACCESS_CONTROL: in LIST_TYPE := EMPTY_LIST;
                  LEVEL:         in LIST_TYPE := EMPTY_LIST)
is
    BASE : NODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(QUEUE_NAME), (1=>APPEND_RELATIONSHIPS));
    COUPLE(BASE, LAST_KEY(QUEUE_NAME), LAST_RELATION(QUEUE_NAME),
           FILE_NODE, FORM, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
end COUPLE;
```

STATUS_ERROR

is raised if **TERMINAL** is not an open file node.

5.3.5.13. Creating a queue file node

```
procedure COUPLE
(Queue_BASE      :in NODE_TYPE;
 Queue_KEY       :in RELATIONSHIP_KEY := LATEST_KEY;
 Queue_RELATION  :in RELATION_NAME := DEFAULT_RELATION;
 File_NODE       :in NODE_TYPE;
 FORM           :in LIST_TYPE := EMPTY_LIST;
 ATTRIBUTES      :in LIST_TYPE; -- intentionally not defaulted
 ACCESS_CONTROL  :in LIST_TYPE := EMPTY_LIST;
 LEVEL           :in LIST_TYPE := EMPTY_LIST);
```

Purpose:

This procedure creates a queue file node and its contents and installs the primary relationship to it. The relation name and relationship key of the primary relationship to the node and the base node from which it emanates are given by the parameters **QUEUE_RELATION**, **QUEUE_KEY** and **QUEUE_BASE**. A secondary relationship of the predefined relation **COUPLE** is created from the created queue file node to the file node identified by **FILE_NODE**.

The initial contents of the queue file is the contents of the file associated with the file node identified by **FILE_NODE** at the time the queue file is created. The queue file node is created with the same **ACCESS_METHOD** attribute value as the node identified by **FILE_NODE**. **DIRECT** may not be a value of this **ACCESS_METHOD** attribute. The **FILE_KIND** attribute of the created queue file node has the value **QUEUE**. The **QUEUE_KIND** attribute of the created queue file node is set by the appropriate value in the **ATTRIBUTES** parameter. **ATTRIBUTES** must include a list item that is either **QUEUE_TYPE => COPY** or **QUEUE_TYPE => MIMIC**. **COUPLE** is the only interface that can be used to create a mimic or copy queue.

The **ATTRIBUTES** parameter defines and provides initial values for attributes of the node. The **ACCESS_CONTROL** parameter specifies initial access control information to be established for the created node.

The **LEVEL** parameter specifies the security level at which the file node is to be created.

Parameters:

QUEUE_BASE

is an open node handle to the node from which the primary relationship to the new node is to emanate.

QUEUE_KEY

is the relationship key of the primary relationship to be created.

QUEUE_RELATION

is the relation name of the primary relationship to be created.

FILE_NODE is an open node handle to the file node with which the queue is to be coupled.

FORM indicates file characteristics.

5.3.5.11. Enabling and disabling function key usage

```
procedure ENABLE_FUNCTION_KEYS (TERMINAL :in FILE_TYPE;  
                                ENABLE    :in BOOLEAN);
```

Purpose:

This procedure establishes whether data read as the result of pressing a function key on the physical input terminal is to appear in the input terminal file as ASCII character sequences or as function key identification numbers. A value of TRUE for ENABLE indicates that the function keys should appear as numbered values. A value of FALSE indicates that the function keys should appear as ASCII character sequences. The function keys are said to have been enabled if the value of ENABLE is TRUE.

Parameters:

TERMINAL is an open file handle on an input terminal file.

ENABLE indicates how function keys are to appear.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the attribute FILE_KIND of the node associated with the file identified by the parameter TERMINAL. USE_ERROR is also raised if the file identified by TERMINAL is of mode OUT_FILE or APPEND_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

Notes:

This procedure has no effect on read operations of the CAIS.TEXT_IO package.

5.3.5.12. Determining function key usage

```
function FUNCTION_KEYS_ENABLED (TERMINAL :in FILE_TYPE)  
return BOOLEAN;
```

Purpose:

This function returns TRUE if the function keys are enabled, i.e., they appear in the input terminal file as numbered values; otherwise it returns FALSE.

Parameters:

TERMINAL is an open file handle on an input terminal file.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the attribute FILE_KIND of the node associated with the file identified by the parameter TERMINAL. USE_ERROR is also raised if the file identified by TERMINAL is of mode OUT_FILE or APPEND_FILE.

5.3.5.9. Determining the prompt string

```
function GET_PROMPT (TERMINAL :in FILE_TYPE)
    return STRING;
```

Purpose:

This function returns the current prompt string for the input terminal file identified by TERMINAL.

Parameters:

TERMINAL is an open file handle identifying an input terminal file.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the attribute FILE_KIND or if SCROLL or PAGE is not a value of the attribute TERMINAL_KIND of the node associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is not of mode IN_FILE or INOUT_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

5.3.5.10. Determining intercepted characters

```
function INTERCEPTED_CHARACTERS (TERMINAL :in FILE_TYPE)
    return CHARACTER_ARRAY;
```

Purpose:

This function returns the array CHARACTER_ARRAY that indicates the characters that can never appear in the input terminal file identified by TERMINAL due to characteristics of the underlying system and the individual physical terminal. A value of TRUE indicates that the character can appear; a value of FALSE indicates that it cannot appear.

Parameters:

TERMINAL is an open file handle on an input terminal file.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the attribute FILE_KIND of the node associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is not of mode IN_FILE or INOUT_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

USE_ERROR is raised if the file identified by FILE has no log file.

STATUS_ERROR

is raised if FILE is not an open file handle.

5.3.5.7. Determining the file size

```
function NUMBER_OF_ELEMENTS (FILE :in FILE_TYPE)
    return NATURAL;
```

Purpose:

This function returns the number of data elements contained in the file identified by FILE. The package that was used to write the elements determines what constitutes a data element.

Parameters:

FILE is an open file handle on a secondary storage or queue file.

Exceptions:

USE_ERROR is raised if the value of the attribute FILE_KIND of the node associated with the file identified by FILE is TERMINAL or MAGNETIC_TAPE.

STATUS_ERROR

is raised if FILE is not an open file handle.

5.3.5.8. Setting the prompt string

```
procedure SET_PROMPT (TERMINAL :in FILE_TYPE;
    PROMPT :in STRING);
```

Purpose:

This procedure sets the prompt string for the output terminal file associated with the input terminal file identified by TERMINAL. All future requests for a line of input from the input terminal file identified by TERMINAL will first output the prompt string to the associated output terminal file.

Parameters:

TERMINAL is an open file handle identifying an input terminal file.

PROMPT is the new value of the prompt string.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the attribute FILE_KIND or if SCROLL or PAGE is not a value of the attribute TERMINAL_KIND of the node associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is not of mode IN_FILE or INOUT_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

5.3.5.4. Removing a log file

procedure CLEAR_LOG(FILE :in FILE_TYPE);

Purpose:

This procedure removes the association established between the file identified by FILE and its log file.

Parameters:

FILE is an open file handle on a file that has a log file.

Exceptions:

STATUS_ERROR
is raised if FILE is not an open file handle.

Notes:

If FILE is an open file handle and there is no log file, this procedure has no effect.

5.3.5.5. Determining whether logging is specified

**function LOGGING (FILE :in FILE_TYPE)
return BOOLEAN;**

Purpose:

This function returns TRUE if the file identified by FILE has a log file associated with it; otherwise, it returns FALSE.

Parameters:

FILE is an open file handle.

Exceptions:

STATUS_ERROR
is raised if FILE is not an open file handle.

5.3.5.6. Determining the log file

**function GET_LOG (FILE :in FILE_TYPE)
return FILE_TYPE;**

Purpose:

This function returns an open file handle on the log file currently associated with the file identified by FILE.

Parameters:

FILE is an open file handle.

Exceptions:

5.3.5.2. Synchronizing program files with system files

procedure SYNCHRONIZE(FILE :in FILE_TYPE);

Purpose:

This procedure forces all data that has been written to the internal file identified by FILE to be transmitted to the external file with which it is associated.

Parameters:

FILE is an open file handle on the internal file to be synchronized.

Exceptions:

USE_ERROR is raised if the file identified by FILE is of mode IN_FILE.

STATUS_ERROR

is raised if FILE is not an open file handle.

5.3.5.3. Establishing a log file

procedure SET_LOG(FILE :in FILE_TYPE;
LOG_FILE :in FILE_TYPE);

Purpose:

This procedure associates a log file identified by LOG_FILE with the file identified by FILE. All elements written to the internal file identified by FILE are also written to the file identified by LOG_FILE.

Parameters:

FILE is an open file handle on the file which is to have a log file.

LOG_FILE is an open file handle on the file to which the log should be written.

Exceptions:

MODE_ERROR

is raised if the mode of either of the files identified by FILE or LOG_FILE is IN_FILE.

USE_ERROR is raised if the nodes associated with the files identified by FILE and LOG_FILE do not have the same values for the attribute ACCESS_METHOD or if the files do not have compatible elements (implementation-defined).

STATUS_ERROR

is raised if FILE and LOG_FILE are not both open file handles.

5.3.5.1. Obtaining an open node handle from a file handle

```
procedure OPEN_FILE_NODE(FILE:      in    FILE_TYPE;  
                          NODE:      in out NODE_TYPE;  
                          INTENT:    in    INTENTION;  
                          TIME_LIMIT: in    DURATION:=NO_DELAY);
```

Purpose:

This procedure returns an open node handle for the node associated with the file identified by FILE.

Parameters:

FILE is an open file handle.

NODE is a node handle, initially closed, to be opened.

INTENT is the intent of subsequent operations on the node; the actual parameter takes the form of an array aggregate.

TIME_LIMIT specifies a time limit for the delay on waiting for the unlocking of a node in accordance with the desired INTENT.

Exceptions:

NAME_ERROR

is raised if the node to which a handle is to be opened is inaccessible or if it is unobtainable and the given INTENT includes any intent other than EXISTENCE.

USE_ERROR is raised if the specified INTENT is an empty array.

STATUS_ERROR

is raised if FILE is not an open file handle or if NODE is an open node handle.

LOCK_ERROR

is raised if the OPEN_FILE_NODE operation is delayed beyond the specified time limit due to the existence of locks in conflict with the specified intent.

ACCESS_VIOLATION

is raised if the current process' discretionary access control rights are insufficient to obtain access to the node consistent with the specified INTENT. ACCESS_VIOLATION is raised only if the conditions for NAME_ERROR are not present.

SECURITY_VIOLATION

is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND**, **SCROLL** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**, or there are fewer than **COUNT** tab stops of the designated kind after the active position.

MODE_ERROR
is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR
is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR
is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure TAB(KIND: in TAB_ENUMERATION := HORIZONTAL;  
              COUNT: in POSITIVE := 1)  
is  
begin  
  TAB(CURRENT_OUTPUT, KIND, COUNT);  
end TAB;
```

5.3.6.8. Sounding a terminal bell

```
procedure BELL(TERMINAL: in FILE_TYPE);
```

Purpose:

This procedure sounds the bell (beeper) on the terminal represented by the output terminal file identified by **TERMINAL**.

Parameters:

TERMINAL is an open file handle on an output terminal file.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **SCROLL** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR
is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR
is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR
is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure BELL
is
begin
  BELL(CURRENT_OUTPUT);
end BELL;
```

5.3.6.9. Writing to the terminal

```
procedure PUT(TERMINAL: in FILE_TYPE;
              ITEM:      in CHARACTER);
```

Purpose:

This procedure writes a single character to the output terminal file identified by TERMINAL and advances the active position by one column.

Parameter:

TERMINAL is an open file handle on an output terminal file.

ITEM is the character to be written.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or SCROLL is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is of mode IN_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interfaces:

```
procedure PUT(ITEM: in CHARACTER)
is
begin
  PUT(CURRENT_OUTPUT, ITEM);
end PUT;

procedure PUT(TERMINAL: in FILE_TYPE;
              ITEM:      in STRING)
is
begin
  for INDEX in ITEM'FIRST .. ITEM'LAST loop
    PUT(TERMINAL, ITEM(INDEX));
  end loop;
end PUT;

procedure PUT(ITEM: in STRING)
is
```



```
begin
  PUT(CURRENT_OUTPUT, ITEM);
end PUT;
```

Notes:

After a character is written in the rightmost position of a row, the active position is the first position of the next row.

5.3.6.10. Enabling echo on a terminal

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

Purpose:

This procedure establishes whether characters which appear in the input terminal file identified by TERMINAL are echoed to its associated output terminal file. When TO is TRUE, each character is echoed to the output terminal file. When TO is FALSE, each character which appears in the input terminal file is not echoed to its associated output terminal file.

Parameters:

TERMINAL is an open file handle on an input terminal file.

TO indicates whether or not to echo input characters.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or SCROLL is not a value of the predefined attribute TERMINAL_KIND of the file mode associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is of mode OUT_FILE or APPEND_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure SET_ECHO(TO: in BOOLEAN := TRUE)
is
begin
  SET_ECHO(CURRENT_INPUT, TO);
end SET_ECHO;
```

5.3.6.11. Querying echo on a terminal

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

Purpose:

This function returns TRUE if echo is enabled; otherwise it returns FALSE.

Parameters:

TERMINAL is an open file handle on an input terminal file.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or SCROLL is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is of mode OUT_FILE or APPEND_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function ECHO return BOOLEAN
is
begin
    return ECHO(CURRENT_INPUT);
end ECHO;
```

5.3.6.12. Determining the number of function keys

```
function MAXIMUM_FUNCTION_KEY(TERMINAL: in FILE_TYPE)
    return NATURAL;
```

Purpose:

This function returns the maximum function key identification number that can be returned by a GET operation on the input terminal file identified by TERMINAL.

Parameters:

TERMINAL is an open file handle on an input terminal file.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or

SCROLL is not a value of the predefined attribute `TERMINAL_KIND` of the file node associated with the file identified by the parameter `TERMINAL`.

MODE_ERROR

is raised if the file identified by `TERMINAL` is of mode `OUT_FILE` or `APPEND_FILE`.

STATUS_ERROR

is raised if `TERMINAL` is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function MAXIMUM_FUNCTION_KEY return NATURAL  
is  
begin  
  return MAXIMUM_FUNCTION_KEY(CURRENT_INPUT);  
end MAXIMUM_FUNCTION_KEY;
```

5.3.6.13. Reading a character from a terminal

```
procedure GET(TERMINAL: in FILE_TYPE;  
              ITEM: out CHARACTER;  
              KEYS: out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:

This procedure reads either a single character into `ITEM` or a single function key identification number into `KEYS` from the input terminal file identified by `TERMINAL`.

Parameters:

TERMINAL is an open file handle on an input terminal file.

ITEM is the character that was read.

KEYS is the description of the function key identification number that was read.

Exceptions:

USE_ERROR is raised if `TERMINAL` is not the value of the predefined attribute `FILE_KIND` or `SCROLL` is not a value of the predefined attribute `TERMINAL_KIND` of the file node associated with the file identified by the parameter `TERMINAL`.

MODE_ERROR

is raised if the file identified by `TERMINAL` is of mode `OUT_FILE` or `APPEND_FILE`.

STATUS_ERROR

is raised if `TERMINAL` is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure GET (ITEM: out CHARACTER;  
              KEYS: out FUNCTION_KEY_DESCRIPTOR)  
is  
begin  
  GET (CURRENT_INPUT, ITEM, KEYS);  
end GET;
```

Notes:

This procedure will only return function key identification numbers in KEYS if function keys have been enabled (see Section 5.3.5.11). Otherwise the characters in the ASCII character sequence representing the function key will appear one at a time in ITEM.

5.3.6.14. Reading all available characters from a terminal

```
procedure GET (TERMINAL: in FILE_TYPE;  
              ITEM: out STRING;  
              LAST: out NATURAL;  
              KEYS: out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:

This procedure successively reads characters and function key identification numbers into ITEM and KEYS respectively, until either all positions of ITEM or KEYS are filled or there are no more characters available in the input terminal file. Upon completion, LAST contains the index of the last position in ITEM to contain a character that has been read.

Parameters:

TERMINAL is an open file handle on an input terminal file.

ITEM is the string of characters that were read.

LAST is the position of the last character read in ITEM.

KEYS is a description of the function key identification numbers that were read.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **SCROLL** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **OUT_FILE** or **APPEND_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure GET(ITEM: out STRING;  
             LAST: out NATURAL;  
             KEYS: out FUNCTION_KEY_DESCRIPTOR)  
is  
begin  
  GET(CURRENT_INPUT, ITEM, LAST, KEYS);  
end GET;
```

Notes:

This procedure will only return function key identification numbers in KEYS if function keys have been enabled (see Section 5.3.5.11). Otherwise the characters in the ASCII character sequence representing the function key will appear in ITEM. If there are no elements available for reading from the input terminal file, then LAST has a value one less than ITEM.FIRST and FUNCTION_KEY_COUNT(KEYS) (see Section 5.3.6.15) is equal to zero.

5.3.6.15. Determining the number of function keys that were read

```
function FUNCTION_KEY_COUNT(KEYS: in FUNCTION_KEY_DESCRIPTOR)  
  return NATURAL;
```

Purpose:

This function returns the number of function keys described in KEYS.

Parameters:

KEYS is the function key descriptor being queried.

Exceptions:

None

5.3.6.16. Determining function key usage

```
procedure FUNCTION_KEY(KEYS:                    in FUNCTION_KEY_DESCRIPTOR;  
                      INDEX:                   in POSITIVE;  
                      KEY_IDENTIFIER:         out POSITIVE;  
                      POSITION:                out NATURAL);
```

Purpose:

This procedure returns the identification number of a function key and the position in the string (read at the same time as the function keys) of the character following the function key.

Parameters:

KEYS is the description of the function key identification numbers that were read.

INDEX is the index in KEYS of the function key to be queried.

KEY_IDENTIFIER

is the identification number of a function key.

POSITION

is the position of the character read after the function key.

Exceptions:

CONSTRAINT_ERROR

is raised if INDEX is greater than FUNCTION_KEY_COUNT(KEYS).

5.3.6.17. Determining the name of a function key

```
procedure FUNCTION_KEY_NAME (TERMINAL: in FILE_TYPE;  
                             KEY_IDENTIFIER: in POSITIVE;  
                             KEY_NAME: out STRING;  
                             LAST: out POSITIVE);
```

Purpose:

This function returns (in KEY_NAME) the string identification of the function key sequence designated by KEY_IDENTIFIER. It also returns the index of the last character of the function key name in LAST.

Parameters:

TERMINAL is an open file handle on an input terminal file.

KEY_IDENTIFIER

is the identification number of a function key.

KEY_NAME is the name of the key designated by KEY_IDENTIFIER.

LAST is the position in KEY_NAME of the last character of the function key name.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or SCROLL is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is of mode OUT_FILE or APPEND_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

CONSTRAINT_ERROR

is raised if the value of KEY_IDENTIFIER is greater than

MAXIMUM_FUNCTION_KEY(TERMINAL) or the string identification of the function key sequence is longer than the string KEY_NAME.

Additional Interface:

```
procedure FUNCTION_KEY_NAME
(KEY_IDENTIFIER: in POSITIVE;
 KEY_NAME: out STRING;
 LAST: out POSITIVE)
is
begin
FUNCTION_KEY_NAME(CURRENT_INPUT,
 KEY_IDENTIFIER, KEY_NAME, LAST);
end FUNCTION_KEY_NAME;
```

5.3.6.18. Advancing the active position to the next line

```
procedure NEW_LINE(TERMINAL: in FILE_TYPE;
COUNT: in POSITIVE := 1);
```

Purpose:

This procedure advances the active position in the output terminal file to column one, COUNT lines after the active position.

Parameters:

TERMINAL is an open file handle on an output terminal file.

COUNT is the number of lines to advance.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or SCROLL is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is of mode IN_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure NEW_LINE(COUNT: in POSITIVE := 1)
is
begin
NEW_LINE(CURRENT_OUTPUT, COUNT);
end NEW_LINE;
```

5.3.6.19. Advancing the active position to the next page

procedure NEW_PAGE (TERMINAL: in FILE_TYPE);

Purpose:

This procedure advances the active position in the output terminal file to the first column of the first line of a new page.

Parameters:

TERMINAL is an open file handle on an output terminal file.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **SCROLL** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure NEW_PAGE  
is  
begin  
    NEW_PAGE (CURRENT_OUTPUT);  
end NEW_PAGE;
```

5.3.7. Package PAGE_TERMINAL

This package provides the functionality of a page terminal. A page terminal consists of two devices: an input device (keyboard) and an associated output device (display). A page terminal may be accessed either as a single file of mode **INOUT_FILE** or as two files: one of mode **IN_FILE** (the keyboard) and the other of mode **OUT_FILE** (the display). As keys are pressed on the page terminal keyboard, the transmitted characters are made available for reading by the **CAIS.PAGE_TERMINAL** package. As characters are written to the page terminal file, they are displayed on the output device.

The display for a page terminal has positions in which printable ASCII characters may be graphically displayed. The positions are arranged into horizontal rows and vertical columns. Each position is identifiable by the combination of a row number and a column number. A display has a fixed number of rows and columns. The rows and columns of a display are identified by positive numbers. The rows are incrementally indexed starting with one at the top of the display. The columns are incrementally indexed starting with one at the left side of the display.

The active position on the display of a page terminal is the position at which the next operation will be performed. The active position is said to advance if (1) the row number of the new position is greater than the row number of the old position or (2) the row number of the new position is the same as the row number of the old position and the new position has a greater column number. Similarly, a position is said to precede the active position if (1) the row number of the position is less than the row number of the active position or (2) the row number of the position is the same as the row number of the active position and the column number of the position is smaller than the column number of the active position.

5.3.7.1. Types, subtypes and constants

```
subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;

subtype FUNCTION_KEY_DESCRIPTOR is
    CAIS.IO_DEFINITIONS.FUNCTION_KEY_DESCRIPTOR;

subtype POSITION_TYPE is CAIS.IO_DEFINITIONS.POSITION_TYPE;

subtype TAB_ENUMERATION is CAIS.IO_DEFINITIONS.TAB_ENUMERATION;

type SELECT_ENUMERATION is
    (FROM_ACTIVE_POSITION_TO_END,
     FROM_START_TO_ACTIVE_POSITION,
     ALL_POSITIONS);

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

type GRAPHIC_RENDITION_ARRAY is array (GRAPHIC_RENDITION_ENUMERATION)
    of BOOLEAN;

DEFAULT_GRAPHIC_RENDITION : constant GRAPHIC_RENDITION_ARRAY
    := (PRIMARY_RENDITION => TRUE, BOLD..REVERSE_IMAGE => FALSE);
```

FILE_TYPE describes the type for file handles. FUNCTION_KEY_DESCRIPTOR is used to obtain information about function keys read from a terminal. POSITION_TYPE describes the type of a position on a terminal. TAB_ENUMERATION is used to specify the kind of tab stop to be set. SELECT_ENUMERATION is used in ERASE_IN_DISPLAY and ERASE_IN_LINE to determine the portion of the display or line to be erased. GRAPHIC_RENDITION_ENUMERATION, GRAPHIC_RENDITION_ARRAY, and DEFAULT_GRAPHIC_RENDITION are used to determine display characteristics of printable characters.

5.3.7.2. Setting the active position

```
procedure SET_POSITION (TERMINAL : in FILE_TYPE;
                        POSITION : in POSITION_TYPE);
```

Purpose:

This procedure advances the active position to the specified POSITION on the output terminal file identified by TERMINAL.

Parameters:

TERMINAL is an open file handle on an output terminal file.

POSITION is the new active position in the output terminal file.

Exceptions:

USE__ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE__ERROR
 is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS__ERROR
 is raised if **TERMINAL** is not an open file handle.

LAYOUT__ERROR
 is raised if the position does not exist on the terminal.

DEVICE__ERROR
 is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure SET_POSITION(POSITION : in POSITION_TYPE)
is
begin
    SET_POSITION(CURRENT_OUTPUT, POSITION);
end SET_POSITION;
```

5.3.7.3. Determining the active position

```
function GET_POSITION(TERMINAL : in FILE_TYPE)
return POSITION_TYPE;
```

Purpose:

This function returns the active position of the output terminal file identified by **TERMINAL**.

Parameters:

TERMINAL is an open file handle on an output terminal file.

Exceptions:

USE__ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE__ERROR
 is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function GET_POSITION return POSITION_TYPE
is
begin
    return GET_POSITION(CURRENT_OUTPUT);
end GET_POSITION;
```

5.3.7.4. Determining the size of the terminal

```
function TERMINAL_SIZE(TERMINAL : in FILE_TYPE)
    return POSITION_TYPE;
```

Purpose:

This function returns the maximum row and maximum column of the output terminal file identified by **TERMINAL**.

Parameters:

TERMINAL is an open file handle on a terminal file.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function TERMINAL_SIZE
    return POSITION_TYPE
is
begin
    return TERMINAL_SIZE(CURRENT_OUTPUT);
end TERMINAL_SIZE;
```

5.3.7.5. Setting a tab stop

```
procedure SET_TAB(TERMINAL : in FILE_TYPE;  
                  KIND      : in TAB_ENUMERATION := HORIZONTAL);
```

Purpose:

This procedure establishes a horizontal tab stop at the column of the active position if KIND is HORIZONTAL, or a vertical tab stop at the row of the active position if KIND is VERTICAL.

Parameters:

TERMINAL is an open file handle on a terminal file.

KIND is the kind (horizontal or vertical) of tab to be set.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or PAGE is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

MODE_ERROR is raised if the file identified by TERMINAL is of mode IN_FILE.

STATUS_ERROR is raised if the file identified by TERMINAL is not an open file handle.

DEVICE_ERROR is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure SET_TAB(KIND : in TAB_ENUMERATION := HORIZONTAL)  
is  
begin  
    SET_TAB(CURRENT_OUTPUT, KIND);  
end SET_TAB;
```

5.3.7.6. Clearing a tab stop

```
procedure CLEAR_TAB(TERMINAL : in FILE_TYPE;  
                   KIND      : in TAB_ENUMERATION := HORIZONTAL);
```

Purpose:

This procedure removes a horizontal tab stop from the column of the active position if KIND is HORIZONTAL or a VERTICAL tab stop from the row of the active position if KIND is VERTICAL.

Parameters:

TERMINAL is an open file handle on a terminal file.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure ERASE_IN_LINE(SELECTION: in SELECT_ENUMERATION)
is
begin
    ERASE_IN_LINE(CURRENT_OUTPUT, SELECTION);
end ERASE_IN_LINE;
```

5.3.7.23. Inserting space characters in a line

```
procedure INSERT_SPACE(TERMINAL: in FILE_TYPE;
                       COUNT:    in POSITIVE := 1);
```

Purpose:

This procedure inserts **COUNT** space characters into the active line at the active position. The character at the active position and adjacent characters are shifted to the right. The **COUNT** rightmost characters on the line are lost. The active position is not changed.

Parameters:

TERMINAL is an open file handle on an output terminal file.

COUNT is the number of space characters to be inserted.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**, or if the value of **COUNT** is greater than the number of columns including and following the active position.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

TERMINAL is an open file handle on an output terminal file.

SELECTION is the portion of the display to be erased.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR
is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR
is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR
is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure ERASE_IN_DISPLAY(SELECTION : in SELECT_ENUMERATION)
is
begin
    ERASE_IN_DISPLAY(CURRENT_OUTPUT, SELECTION);
end ERASE_IN_DISPLAY;
```

5.3.7.22. Erasing characters in a line

```
procedure ERASE_IN_LINE(TERMINAL: in FILE_TYPE;
                        SELECTION: in SELECT_ENUMERATION);
```

Purpose:

This procedure erases the characters in the active line as determined by the active position and the given **SELECTION** (including the active position). After erasure erased positions have space characters. The active position is not changed.

Parameters:

TERMINAL is an open file handle on an output terminal file.

SELECTION is the portion of the line to be erased.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**, or if the value of **COUNT** is greater than the number of columns including and following the active position.

5.3.7.20. Erasing characters in a line

```
procedure ERASE_CHARACTER(TERMINAL: in FILE_TYPE;  
                          COUNT: in POSITIVE := 1);
```

Purpose:

This procedure replaces COUNT characters on the active line with space characters starting at the active position and advancing toward the end position. The active position is not changed.

Parameters:

TERMINAL is an open file handle on an output terminal file.

COUNT is the number of characters to be erased

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or PAGE is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL, or if the value of COUNT is greater than the number of positions in the active line including and after the active position.

MODE_ERROR
is raised if the file identified by TERMINAL is of mode IN_FILE.

STATUS_ERROR
is raised if the file identified by TERMINAL is not an open file handle.

DEVICE_ERROR
is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interaces:

```
procedure ERASE_CHARACTER(COUNT : in POSITIVE :=1)  
is  
begin  
    ERASE_CHARACTER(CURRENT_OUTPUT, COUNT);  
end ERASE_CHARACTER;
```

5.3.7.21. Erasing characters in a display

```
procedure ERASE_IN_DISPLAY(TERMINAL: in FILE_TYPE;  
                           SELECTION: in SELECT_ENUMERATION);
```

Purpose:

This procedure erases the characters in the display as determined by the active position and the given SELECTION (including the active position). After erasure erased positions have space characters. The active position is not changed.

Parameters:

Additional interface:

```
procedure DELETE_CHARACTER(COUNT: in POSITIVE :=1)
is
begin
  DELETE_CHARACTER(CURRENT_OUTPUT, COUNT);
end DELETE_CHARACTER;
```

5.3.7.19. Deleting lines

```
procedure DELETE_LINE(TERMINAL: in FILE_TYPE;
                      COUNT:    in POSITIVE:=1);
```

Purpose:

This procedure deletes COUNT lines starting at the active position and advancing toward the end position. Adjacent lines are shifted from the bottom toward the active position. Open space at the bottom of the display is filled with erased lines. The active position is not changed.

Parameters:

TERMINAL is an open file handle on an output terminal file.

COUNT is the number of lines to be deleted.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or PAGE is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL, or if the value of COUNT is greater than the number of rows including and following the active position.

MODE_ERROR is raised if the file identified by TERMINAL is of mode IN_FILE.

STATUS_ERROR is raised if TERMINAL is not an open file handle.

DEVICE_ERROR is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional interface:

```
procedure DELETE_LINE(COUNT: in POSITIVE := 1)
is
begin
  DELETE_LINE(CURRENT_OUTPUT, COUNT);
end DELETE_LINE;
```


DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

CONSTRAINT_ERROR

is raised if the value of KEY_IDENTIFIER is greater than MAXIMUM_FUNCTION_KEY(TERMINAL) or the string identification of the function key sequence is longer than the string KEY_NAME.

Additional Interface:

```
procedure FUNCTION_KEY_NAME (KEY_IDENTIFIER : in    POSITIVE;  
                             KEY_NAME       : out STRING;  
                             LAST          : out POSITIVE;)  
is  
begin  
    FUNCTION_KEY_NAME(CURRENT_INPUT,  
                      KEY_IDENTIFIER, KEY_NAME, LAST);  
end FUNCTION_KEY_NAME;
```

5.3.7.18. Deleting characters

```
procedure DELETE_CHARACTER(TERMINAL: in FILE_TYPE;  
                          COUNT:    in POSITIVE := 1);
```

Purpose:

This procedure deletes COUNT characters on the active line starting at the active position and advancing toward the end 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.

Parameters:

TERMINAL is an open file handle on an output terminal file.

COUNT is the number of characters to be deleted.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or PAGE is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL, or if the value of COUNT is greater than the number of positions in the active line including and following the active position.

MODE_ERROR

is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR

is raised if the file identified by TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Parameters:

KEYS is the description of the function key numbers that were read.

INDEX is the index in **KEYS** of the function key to be queried.

KEY_IDENTIFIER
is the identification number of a function key.

POSITION is the position of the character read after the function key.

Exceptions:

CONSTRAINT_ERROR
is raised if **INDEX** is greater than **FUNCTION_KEY_COUNT(KEYS)**.

5.3.7.17. Determining the name of a function key

```
procedure FUNCTION_KEY_NAME (TERMINAL      : in   FILE_TYPE;  
                             KEY_IDENTIFIER : in   POSITIVE;  
                             KEY_NAME       : out STRING;  
                             LAST          : out POSITIVE);
```

Purpose:

This function returns (in **KEY_NAME**) the string identification of the function key designated by **KEY_IDENTIFIER**. It also returns the index of the last character of the function key name in **LAST**.

Parameters:

TERMINAL is an open file handle on an input terminal file.

KEY_IDENTIFIER
is the identification number of a function key.

KEY_NAME is the name of the key designated by **KEY_IDENTIFIER**.

LAST is the position in **KEY_NAME** of the last character of the function key name.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file mode associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR
is raised if the file identified by **TERMINAL** is of mode **OUT_FILE** or **APPEND_FILE**.

STATUS_ERROR
is raised if **TERMINAL** is not an open file handle.

is raised if the file identified by `TERMINAL` is of mode `OUT_FILE` or `APPEND_FILE`.

STATUS__ERROR

is raised if `TERMINAL` is not an open file handle.

DEVICE__ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure GET(ITEM : out STRING;  
              LAST : out NATURAL;  
              KEYS : out FUNCTION_KEY_DESCRIPTOR)  
is  
begin  
  GET(CURRENT_INPUT, ITEM, LAST, KEYS);  
end GET;
```

Notes:

This procedure will only return function key identification numbers in `KEYS` if function keys have been enabled (see Section 5.3.5.11). Otherwise the characters in the ASCII character sequence representing the function key will appear in `ITEM`. If there are no elements available for reading from the input terminal file, then `LAST` has a value one less than `ITEM'FIRST` and `FUNCTION_KEY_COUNT(KEYS)` (see Section 5.3.7.15) is equal to zero.

5.3.7.15. Determining the number of function keys that were read

```
function FUNCTION_KEY_COUNT(KEYS : in FUNCTION_KEY_DESCRIPTOR)  
  return NATURAL;
```

Purpose:

This function returns the number of function keys described in `KEYS`.

Parameters:

`KEYS` is the function key descriptor being queried.

Exceptions:

None.

5.3.7.16. Determining function key usage

```
procedure FUNCTION_KEY(KEYS          : in FUNCTION_KEY_DESCRIPTOR;  
                       INDEX         : in POSITIVE;  
                       KEY_IDENTIFIER : out POSITIVE;  
                       POSITION       : out NATURAL);
```

Purpose:

This procedure returns the identification number of a function key and the position in the string (read at the same time as the function keys) of the character following the function key.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a misfunction of the underlying system.

Additional Interface:

```
procedure GET (ITEM : out CHARACTER;  
              KEYS : out FUNCTION_KEY_DESCRIPTOR)  
is  
begin  
  GET (CURRENT_INPUT, ITEM, KEYS);  
end GET;
```

Notes:

This procedure will only return function key identification numbers in **KEYS** if function keys have been enabled (see Section 5.3.5.12). Otherwise the characters in the ASCII character sequence representing the function key will appear one at a time in **ITEM**.

5.3.7.14. Reading all available characters from a terminal

```
procedure GET (TERMINAL : in    FILE_TYPE;  
              ITEM      : out STRING;  
              LAST      : out NATURAL;  
              KEYS      : out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:

This procedure successively reads characters and function key identification numbers into **ITEM** and **KEYS** respectively until either all positions of **ITEM** or **KEYS** are filled or there are no more characters available in the input terminal file. Upon completion, **LAST** contains the index of the last position in **ITEM** to contain a character that has been read.

Parameters:

TERMINAL is an open file handle on an input terminal file.

ITEM is a string of the characters that were read.

LAST is the position of the last character read in **ITEM**.

KEYS is the description of the function key identification numbers that were read.

Exceptions:

USE_ERROR is raised if the file identified by **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **OUT_FILE** or **APPEND_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function MAXIMUM_FUNCTION_KEY
    return NATURAL
is
begin
    return MAXIMUM_FUNCTION_KEY(CURRENT_INPUT);
end MAXIMUM_FUNCTION_KEY;
```

5.3.7.13. Reading a character from a terminal

```
procedure GET(TERMINAL : in    FILE_TYPE;
              ITEM      : out CHARACTER;
              KEYS      : out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:

This procedure reads either a single character into **ITEM** or a single function key identification number into **KEYS** from the input terminal file identified by **TERMINAL**.

Parameters:

TERMINAL is an open file handle on an input terminal file.

ITEM is the character that was read.

KEYS describes the function key that was read.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **OUT_FILE** or **APPEND_FILE**.

5.3.7.11. Querying echo on a terminal

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

Purpose:

This function returns TRUE if echo is enabled; otherwise it returns FALSE.

Parameters:

TERMINAL is an open file handle on an input terminal file.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **OUT_FILE** or **APPEND_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function ECHO
    return BOOLEAN
is
begin
    return ECHO(CURRENT_INPUT);
end ECHO;
```

5.3.7.12. Determining the number of function keys

```
function MAXIMUM_FUNCTION_KEY(TERMINAL : in FILE_TYPE)
    return NATURAL;
```

Purpose:

This function returns the maximum function key identification number that can be returned by a GET operation in the input terminal file identified by **TERMINAL**.

Parameters:

TERMINAL is an open file handle on an input terminal file.

Exceptions:

Notes:

After a character is written in the rightmost position of a row, the active position is the first position of the next row.

5.3.7.10. Enabling echo on a terminal

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

Purpose:

This procedure establishes whether characters which appear in the input terminal file identified by TERMINAL are echoed to its associated output terminal file. When TO is TRUE, each character which appears in the input terminal file is echoed to the output terminal file. When TO is FALSE, each character which appears in the input terminal file is not echoed to its associated output terminal file.

Parameters:

TERMINAL is an open file handle on an input terminal file.

TO indicates whether or not to echo input characters.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or PAGE is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

MODE_ERROR is raised if the file identified by TERMINAL is of mode OUT_FILE or APPEND_FILE.

STATUS_ERROR is raised if TERMINAL is not an open file handle.

DEVICE_ERROR is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure SET_ECHO(TO : in BOOLEAN := TRUE)  
is  
begin  
  SET_ECHO(CURRENT_INPUT, TO);  
end SET_ECHO;
```

end BELL;

5.3.7.9. Writing to the terminal

```
procedure PUT(TERMINAL : in FILE_TYPE;  
              ITEM      : in CHARACTER);
```

Purpose:

This procedure writes a single character to the output terminal file identified by TERMINAL and advances the active position by one column.

Parameter:

TERMINAL is an open file handle on an output terminal file.

ITEM is the character to be written.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or PAGE is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is of mode IN_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interfaces:

```
procedure PUT(ITEM : in CHARACTER)
is
begin
  PUT(CURRENT_OUTPUT, ITEM);
end PUT;

procedure PUT(TERMINAL : in FILE_TYPE;
              ITEM      : in STRING)
is
begin
  for INDEX in ITEM'FIRST .. ITEM'LAST loop
    PUT(TERMINAL, ITEM(INDEX));
  end loop;
end PUT;

procedure PUT(ITEM : in STRING)
is
begin
  PUT(CURRENT_OUTPUT, ITEM);
end PUT;
```


MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if the file identified by **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure TAB(KIND : in TAB_ENUMERATION := HORIZONTAL;  
              COUNT : in POSITIVE := 1);  
is  
begin  
  TAB(CURRENT_OUTPUT, KIND, COUNT);  
end TAB;
```

5.3.7.8. Sounding a terminal bell

```
procedure BELL(TERMINAL : in FILE_TYPE);
```

Purpose:

This procedure sounds the bell (beeper) on the terminal represented by the output terminal file identified by **TERMINAL**.

Parameters:

TERMINAL is an open file handle on an output terminal file.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure BELL  
is  
begin  
  BELL(CURRENT_OUTPUT);
```

31 JANUARY 1985

KIND is the kind (horizontal or vertical) of tab stop to be removed.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**, or if there is no tab stops of the designated kind at the active position.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure CLEAR_TAB(KIND : in TAB_ENUMERATION := HORIZONTAL)
is
begin
  CLEAR_TAB(CURRENT_OUTPUT, KIND);
end CLEAR_TAB;
```

5.3.7.7. Advancing to the next tab position

```
procedure TAB(TERMINAL : in FILE_TYPE;
  KIND      : in TAB_ENUMERATION := HORIZONTAL;
  COUNT     : in POSITIVE := 1);
```

Purpose:

This procedure advances the active position **COUNT** tab stops. Horizontal advancement causes a change in only the column number of the active position. Vertical advancement causes a change in only the row number of the active position.

Parameters:

TERMINAL is an open file handle on an output terminal file.

KIND is the kind (horizontal or vertical) of tab stop to be advanced.

COUNT is a positive integer indicating the number of tab stops the active position is to advance.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**, or there are fewer than **COUNT** tab stops of the designated kind after the active position.

Additional Interface:

```
procedure INSERT_SPACE(COUNT: in POSITIVE := 1)
is
begin
  INSERT_SPACE(CURRENT_OUTPUT, COUNT);
end INSERT_SPACE;
```

5.3.7.24. Inserting blank lines in the output terminal file

```
procedure INSERT_LINE(TERMINAL: in FILE_TYPE;
                      COUNT:    in POSITIVE := 1);
```

Purpose:

This procedure inserts COUNT blank lines into the output terminal file at the active line. The lines at and below the active position are shifted down. The COUNT bottom lines of the display are lost. The active line is not changed. The column of the active position is changed to one.

Parameters:

TERMINAL is an open file handle on an output terminal file.

COUNT is the number of blank lines to be inserted.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or PAGE is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL, or the value of COUNT is greater than the number of rows including and following the active position.

MODE_ERROR is raised if the file identified by TERMINAL is of mode IN_FILE.

STATUS_ERROR is raised if TERMINAL is not an open file handle.

DEVICE_ERROR is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
procedure INSERT_LINE(COUNT: in POSITIVE := 1)
is
begin
  INSERT_LINE(CURRENT_OUTPUT, COUNT);
end INSERT_LINE;
```

5.3.7.25. Determining graphic rendition support

```
function GRAPHIC_RENDITION_SUPPORT(TERMINAL: in FILE_TYPE;  
                                   RENDITION: in GRAPHIC_RENDITION_ARRAY)  
  
    return BOOLEAN;
```

Purpose:

This function returns TRUE if the RENDITION of combined graphic renditions is supported by the physical terminal associated with the output terminal file identified by TERMINAL; otherwise it returns FALSE.

Parameters:

TERMINAL is an open file handle on an output terminal file.

RENDITION is a combination of graphic renditions.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or PAGE is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL, or if the selected graphic renditions are not supported by the physical terminal associated with the output terminal file identified by TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is of mode IN_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function GRAPHIC_RENDITION_SUPPORT(RENDITION:  
                                   in GRAPHIC_RENDITION_ARRAY)  
  
    return BOOLEAN  
  
is  
begin  
    return GRAPHIC_RENDITION_SUPPORT(CURRENT_OUTPUT, RENDITION);  
end GRAPHIC_RENDITION_SUPPORT;
```

5.3.7.26. Selecting the graphic rendition

```
procedure SELECT_GRAPHIC_RENDITION(TERMINAL: in FILE_TYPE;  
                                   RENDITION: in GRAPHIC_RENDITION_ARRAY  
                                   := DEFAULT_GRAPHIC_RENDITION);
```

Purpose:

This procedure sets the graphic rendition for subsequent characters to be output to the output terminal file.

Parameters:

TERMINAL is an open file handle on an output terminal file.

RENDITION is the graphic rendition to be used in subsequent output operations.

Exceptions:

USE_ERROR is raised if **TERMINAL** is not the value of the predefined attribute **FILE_KIND** or **PAGE** is not a value of the predefined attribute **TERMINAL_KIND** of the file node associated with the file identified by the parameter **TERMINAL**, or if the selected graphic renditions are not supported by the physical terminal associated with the output terminal file identified by **TERMINAL**.

MODE_ERROR

is raised if the file identified by **TERMINAL** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```

procedure SELECT_GRAPHIC_RENDITION(RENDITION : in
                                     GRAPHIC_RENDITION_ARRAY :=
                                     DEFAULT_GRAPHIC_RENDITION)
is
begin
    SELECT_GRAPHIC_RENDITION(CURRENT_OUTPUT, RENDITION);
end SELECT_GRAPHIC_RENDITION;
```

5.3.8. Package FORM TERMINAL

This package provides the functionality of a form terminal (e.g., an IBM 327x terminal). A form terminal consists of a single device (inasmuch as a programmer is concerned).

The scenario for usage of a form terminal has two active agents: a process and a user. Each interaction with the form terminal consists of a three step sequence. First, the process creates and writes a form to the terminal. Second, the user modifies the form. Third, the process reads the modified form.

A *form* is a two-dimensional matrix of character positions. The rows of a form are indexed by positive numbers starting with row one at the top of the display. The columns of a form are indexed by positive numbers starting with column one at the left side of the form. The position identified by row one, column one, is called the *start position* of the form. The position with the highest row and column index term is called the *end position* of the form.

The position at which an operation is to be performed is called the *active position*. The active position is said to advance toward the end position of the form when the indices of its position are

incremented. The column index is incremented until it attains the highest value permitted for the form. The next position is determined by incrementing the row index of the active position and resetting the column index to 1.

A form is divided into *qualified areas*. A qualified area identifies a contiguous group of positions that share a common set of characteristics. A qualified area begins at the position designated by an *area qualifier* and ends at the position preceding the next area qualifier toward the end of the form. Depending on the form, the position of the area qualifier may or may not be considered to be in a qualified area. The characteristics of a qualified area consist of such things as protection (from modification by the user), display renditions (e.g., intensity), and permissible values (e.g., numeric only, alphabetic only). Each position in a qualified area contains a single printable ASCII character.

5.3.8.1. Types and subtypes

```
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);

type FORM_TYPE
  (ROW                               : POSITIVE;
   COLUMN                           : POSITIVE;
   AREA_QUALIFIER_REQUIRES_SPACE : BOOLEAN)
is private;

subtype FILE_TYPE is CAIS.ID_DEFINITIONS.FILE_TYPE;

subtype PRINTABLE_CHARACTERS is CHARACTER range ' ' .. '~';
```

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. FORM_TYPE describes characteristics of forms. FILE_TYPE describes the type for file handles. PRINTABLE_CHARACTERS describes the characters that can be output to a form terminal.

5.3.8.2. Determining the number of function keys

```
function MAXIMUM_FUNCTION_KEY(TERMINAL: in FILE_TYPE)
    return NATURAL;
```

Purpose:

This function returns the maximum function key identifier that can be returned by the function TERMINATION_KEY (see Section 5.3.8.13).

Parameters:

TERMINAL is an open file handle on a terminal file.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or FORM is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

MODE_ERROR

is raised if the file identified by TERMINAL is of mode OUT_FILE or APPEND_FILE.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function MAXIMUM_FUNCTION_KEY return NATURAL
is
begin
    return MAXIMUM_FUNCTION_KEY(CURRENT_INPUT);
end MAXIMUM_FUNCTION_KEY;
```

5.3.8.3. Defining a qualified area

```
procedure DEFINE_QUALIFIED_AREA
(FORM: in out FORM_TYPE;
 INTENSITY: in AREA_INTENSITY := NORMAL;
 PROTECTION: in AREA_PROTECTION := PROTECTED;
 INPUT: in AREA_INPUT := GRAPHIC_CHARACTERS;
 VALUE: in AREA_VALUE := NO_FILL);
```

Purpose:

This procedure places an area qualifier with the designated attributes at the active position of the form. A qualified area consists of the character positions between two area qualifiers. The area is qualified by the area qualifier that precedes the area. A qualified area may or may not include the position of its area qualifier.

Parameters:

FORM is the form on which the qualified area is being defined.

INTENSITY indicates the intensity at which the qualified area is to be displayed.

PROTECTION indicates the protection for the qualified area.

INPUT indicates the permissible input characters for the qualified area.

VALUE indicates the initial value of the qualified area.

Exceptions:

STATUS_ERROR
 is raised if the active position is already defined as an area qualifier.

5.3.8.4. Removing an area qualifier

procedure REMOVE_AREA_QUALIFIER(FORM: in out FORM_TYPE);

Purpose:

This procedure removes an area qualifier from the active position of the form.

Parameters:

FORM is the form from which the qualified area is to be removed.

Exceptions:

USE_ERROR is raised if the active position does not have an area qualifier.

STATUS_ERROR
 is raised if the active position does not contain an area qualifier.

5.3.8.5. Changing the active position

procedure SET_POSITION(FORM: in out FORM_TYPE;
 POSITION: in POSITION_TYPE);

Purpose:

This procedure indicates the position on the form that is to become the active position.

Parameters:

FORM is the form on which to change the active position.

POSITION is the new active position on the form.

Exceptions:

LAYOUT_ERROR
 is raised if POSITION does not identify a position in FORM.

5.3.8.6. Moving to the next qualified area

```
procedure NEXT_QUALIFIED_AREA(FORM: in out FORM_TYPE;  
                             COUNT: in POSITIVE := 1);
```

Purpose:

This procedure advances the active position COUNT qualified areas toward the end of the form.

Parameters:

FORM is the form on which the active position is being advanced.

COUNT is the number of qualified areas the active position is to be advanced.

Exceptions:

USE_ERROR is raised if FORM has fewer than COUNT qualified areas after the active position.

5.3.8.7. Writing to a form

```
procedure PUT(FORM: in out FORM_TYPE;  
             ITEM: in PRINTABLE_CHARACTER);
```

Purpose:

This procedure places ITEM at the active position of FORM and advances the active position one position toward the end position. If the active position is the end position, the active position is not changed.

Parameters:

FORM is the form being written.

ITEM is the character to be written to the form.

Exceptions:

USE_ERROR is raised if the active position contains an area qualifier and AREA_QUALIFIER_REQUIRES_SPACE of FORM was set to TRUE.

Additional interface:

```
procedure PUT(FORM: in out FORM_TYPE;  
             ITEM: in STRING)  
is  
begin  
  for INDEX in ITEM'FIRST .. ITEM'LAST loop  
    PUT(FORM, ITEM(INDEX));  
  end loop;  
end PUT;
```

5.3.8.8. Erasing a qualified area

procedure ERASE_AREA(FORM: in out FORM_TYPE);

Purpose:

This procedure places space characters in all positions of the area in which the active position of the form is located.

Parameters:

FORM is the form on which the qualified area is being erased.

Exceptions:

STATUS_ERROR

is raised if no area qualifiers have been defined for FORM.

5.3.8.9. Erasing a form

procedure ERASE_FORM(FORM: in out FORM_TYPE);

Purpose:

This procedure removes all area qualifiers and places SPACE characters in all positions of the form.

Parameters:

FORM is the form to be erased.

Exceptions:

None.

5.3.8.10. Activating a form on a terminal

**procedure ACTIVATE(TERM: in FILE_TYPE;
FORM: in out FORM_TYPE);**

Purpose:

This procedure activates the form on the terminal. The contents of the terminal file is modified to reflect the contents of the form. When the user of the terminal enters a termination key, the modified contents of the terminal file is copied back to the form and returned. This operation may not result in the modification of protected areas.

Parameters:

TERM is an open file handle on a terminal file.

FORM is the form to be activated.

Exceptions:

USE_ERROR is raised if TERM is not the value of the predefined attribute FILE_KIND or FORM is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERM.

STATUS_ERROR

is raised if **TERMINAL** is not an open file handle.

5.3.8.11. Reading from a form

```
procedure GET(FORM: in out FORM_TYPE;  
              ITEM: out PRINTABLE_CHARACTER);
```

Purpose:

This procedure reads a character from **FORM** at the active position and advances the active position forward one position (unless the active position is the end position). An area qualifier (on a form on which the area qualifier requires space) is read as the **SPACE** character.

Parameters:

FORM is the form to be read.

ITEM is the character that was read.

Exceptions:

None.

Additional Interface:

```
procedure GET(FORM: in out FORM_TYPE;  
              ITEM: out STRING)  
is  
begin  
  for INDEX in ITEM'FIRST .. ITEM'LAST loop  
    GET(FORM, ITEM(INDEX));  
  end loop;  
end GET;
```

5.3.8.12. Determining changes to a form

```
function IS_FORM_UPDATED(FORM: in FORM_TYPE)  
  return BOOLEAN;
```

Purpose:

This function returns **TRUE** if the value of any position on the form was modified during the last activate operation in which the form was used; otherwise it returns **FALSE**.

Parameters:

FORM is the form to be queried.

Exceptions:

None.

5.3.8.13. Determining the termination key

```
function TERMINATION_KEY(FORM: in FORM_TYPE)
return NATURAL;
```

Purpose:

This function returns a number that indicates which (implementation-dependent) key terminated the ACTIVATE procedure for the FORM. A value of zero indicates the normal termination key (e.g., the ENTER key).

Parameters:

FORM is the form to be queried.

Exceptions:

None.

5.3.8.14. Determining the size of a form

```
function FORM_SIZE(FORM: in FORM_TYPE)
return POSITION_TYPE;
```

Purpose:

This function returns the position of the last column of the last row of the form.

Parameters:

FORM is the form to be queried.

Exceptions:

None.

5.3.8.15. Determining the size of a terminal

```
function TERMINAL_SIZE(TERMINAL: in FILE_TYPE)
return POSITION_TYPE;
```

Purpose:

This function returns the position of the last column of the last row of the terminal file.

Parameters:

TERMINAL is an open file handle on a terminal file.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or FORM is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function TERMINAL_SIZE
  return POSITION_TYPE
is
begin
  return TERMINAL_SIZE(CURRENT_OUTPUT);
end TERMINAL_SIZE;
```

5.3.8.16. Determining if the area qualifier requires space in the form

```
function AREA_QUALIFIER_REQUIRES_SPACE(FORM: in FORM_TYPE)
  return BOOLEAN;
```

Purpose:

This function returns TRUE if the area qualifier requires space in the form; otherwise it returns FALSE.

Parameters:

FORM is the form to be queried.

Exceptions:

None.

5.3.8.17. Determining if the area qualifier requires space on a terminal

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

Purpose:

This function returns TRUE if the area qualifier requires space on the physical terminal associated with the terminal file identified by TERMINAL; otherwise it returns FALSE.

Parameters:

TERMINAL is an open file handle on a terminal file.

Exceptions:

USE_ERROR is raised if TERMINAL is not the value of the predefined attribute FILE_KIND or FORM is not a value of the predefined attribute TERMINAL_KIND of the file node associated with the file identified by the parameter TERMINAL.

STATUS_ERROR

is raised if TERMINAL is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function AREA_QUALIFIER_REQUIRES_SPACE  
    return BOOLEAN  
is  
begin  
    return AREA_QUALIFIER_REQUIRES_SPACE(CURRENT_OUTPUT);  
end AREA_QUALIFIER_REQUIRES_SPACE;
```

5.3.9. Package MAGNETIC_TAPE

This package provides interfaces for the support of input and output operations on both labeled and unlabeled magnetic tapes. Interfaces for labeled tapes are designed with careful consideration of level II of the [ANSI 78] standard. These interfaces only support single-volume magnetic tape files.

To use a tape drive, a file handle on the file representing the tape drive must be obtained (see OPEN in Section 5.3.4.3). The first time a tape is used, it must be initialized either as a labeled tape or as an unlabeled tape. All initialized tapes may be loaded as unlabeled tapes; however, only initialized labeled tapes may be loaded as labeled tapes. Once a tape has been loaded, CAIS.TEXT_IO routines are used to get information to and from the tape.

When information transfer is completed, the tape is unloaded and dismounted using the UNLOAD and DISMOUNT procedures.

Once a tape is dismounted, another tape may be mounted. When the user is finished utilizing the drive, he closes the file handle on the file representing the tape on the drive (see Section 5.3.4).

Magnetic tape drive files can only be created by the implementation. Implementation-defined file characteristics must be supported by the implementation and will include the densities and block sizes supported by the tape drive, whether or not a tape is mounted on the drive and whether the tape was loaded as a labeled or unlabeled tape. Each block of a file may be terminated by zero or more null characters.

An unlabeled tape is read according to the format:

BOT file * file * ... * file **

where * represents a tape mark, ** represents the logical end of tape, and BOT is the beginning of the tape. For the CAIS, a file on a magnetic tape is either a text file or a label group. A labeled tape may be mounted as an unlabeled tape, which causes each label group to be considered as a file. A label group can be one of the following: a volume header label and a file header label, or a file header label, or an end-of-file label.

A labeled tape is read according to the format:

BOT VOL1 HDR * file * EOF * HDR * file * EOF *...* HDR * file * EOF**

where * represents a tape mark, ** represents the logical end of tape, BOT is the beginning of the tape, VOL1 is the volume header label, HDR is the file header label, and EOF is the end-of-file label.

5.3.9.1. Types and subtypes

```
type TAPE_POSITION is
  (BEGINNING_OF_TAPE,
   PHYSICAL_END_OF_TAPE,
   TAPE_MARK,
   OTHER);
```

```
subtype REEL_NAME is STRING;
subtype VOLUME_STRING is STRING(1..8);
subtype FILE_STRING is STRING(1..17);
subtype LABEL_STRING is STRING (1..80);
subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;
```

TAPE_POSITION describes the position of the tape on the tape drive; a value of TAPE_MARK means that the tape is positioned just after a tape mark. That is, a read in this position will read the next file or label. A read starting in position TAPE_MARK will only read a tape mark if there are two consecutive tape marks on the tape at this location.

REEL_NAME describes the type used for the external name of a tape (i.e., the name written on the tape container).

VOLUME_STRING and FILE_STRING both have the syntax of an Ada Identifier. LABEL_STRING describes the type used for reading volume header labels, file header labels and end-of-file labels. FILE_TYPE describes the type for file handles, which are used for controlling all operations on tape drives.

5.3.9.2. Mounting a tape

```
procedure MOUNT(TAPE_DRIVE: in FILE_TYPE;
                 TAPE_NAME: in REEL_NAME;
                 DENSITY: in POSITIVE);
```

Purpose:

This procedure generates an implementation-defined request that the tape whose external name is TAPE_NAME be mounted on the tape drive represented by the file identified by TAPE_DRIVE. It also requests that the tape drive density be set to DENSITY. Following completion of the requested operations, the function IS_MOUNTED(TAPE_DRIVE) will return TRUE.

Parameters:

TAPE_DRIVE

is an open file handle on the file representing the tape drive.

TAPE_NAME

is an external name which identifies the tape to be mounted on the tape drive.

DENSITY is the density in characters per inch (e.g., 800, 1600, 6250).

Exceptions:

USE_ERROR is raised if MAGNETIC_TAPE is not the value of the attribute FILE_KIND of the node associated with the file identified by TAPE_DRIVE or if IS_MOUNTED(TAPE_DRIVE) is TRUE at the time of the call.

STATUS_ERROR

is raised if TAPE_DRIVE is not an open file handle.

DEVICE_ERROR

is raised if this operation cannot be completed because of a malfunction of the underlying system.

5.3.9.3. Loading an unlabeled tape

```
procedure LOAD_UNLABELED(TAPE_DRIVE: in FILE_TYPE;  
                          DENSITY:    in POSITIVE;  
                          BLOCK_SIZE: in POSITIVE);
```

Purpose:

This procedure loads the tape on the tape drive represented by the file identified by TAPE_DRIVE. The tape is positioned at the beginning of tape. The DENSITY is validated against the settings of the tape drive. The block size for subsequent reads and writes is set to the value of BLOCK_SIZE. Following completion of this procedure, the function IS_LOADED(TAPE_DRIVE) will return true.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the drive.

DENSITY is the density in characters per inch (e.g., 800, 1600, 6250) at which the tape is to be read or written.

BLOCK_SIZE is the size of each data block which is to be read from or written to the file identified by TAPE_DRIVE.

Exceptions:

USE_ERROR is raised if IS_LOADED(TAPE_DRIVE) is TRUE or IS_MOUNTED(TAPE_DRIVE) is FALSE at the time of the call, or if DENSITY is not the same as the density of the tape drive, or if the block size cannot be supported by the tape drive.

STATUS_ERROR

is raised if TAPE_DRIVE is not an open file handle.

DEVICE_ERROR

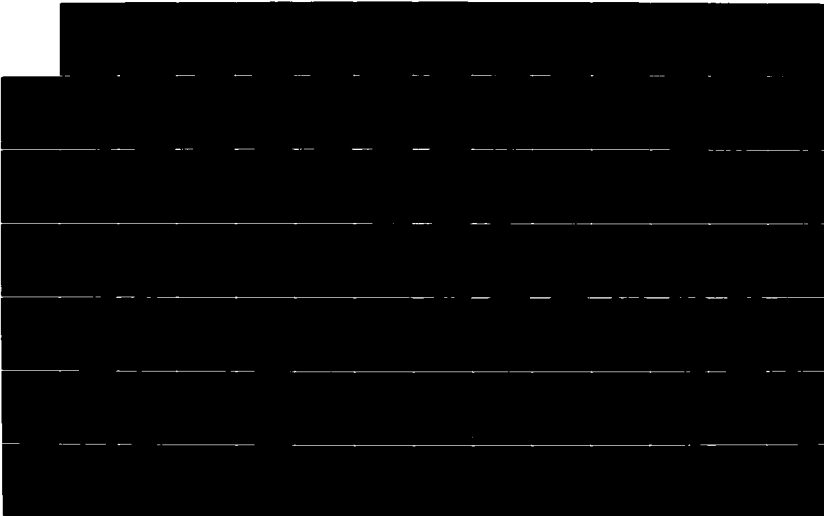
is raised if an input or output operation cannot be completed because of a malfunction of the underlying system or if the tape is uninitialized.

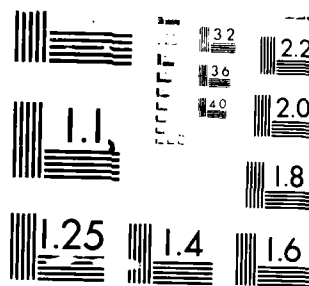
AD-A157-589

MILITARY STANDARD COMMON APSE (ADA PROGRAMMING SUPPORT 3/4
ENVIRONMENT) INTERFACE SET (CAIS) (U) ADA JOINT PROGRAM
OFFICE ARLINGTON VA JAN 85

UNCLASSIFIED

F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

5.3.9.4. Initializing an unlabeled tape

```
procedure INITIALIZE_UNLABELED(TAPE_DRIVE: in FILE_TYPE;  
                               DENSITY: in POSITIVE;  
                               BLOCK_SIZE: in POSITIVE);
```

Purpose:

This procedure initializes the tape which is mounted on the tape drive represented by the file identified by TAPE_DRIVE. The tape drive must have been mounted but not loaded. If the tape is not positioned at the beginning of tape, then the tape is rewound to it. Two adjacent tape marks are written following the beginning of tape mark. The DENSITY is validated against the settings of the tape drive. The block sign for subsequent reads and writes is set to the value of BLOCK_SIZE. The tape is positioned at the beginning of the tape. Initialization places the logical end of tape at the beginning of the tape. The resulting tape is an initialized unlabeled tape.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the drive.

DENSITY is the density in characters per inch (e.g., 800, 1600, 6250)

BLOCK_SIZE is the size of each data block which is to be read from or written to the file identified by TAPE_DRIVE.

Exceptions:

USE_ERROR is raised if MAGNETIC_TAPE is not the value of the attribute FILE_KIND of the node associated with the file identified by TAPE_DRIVE, or DENSITY is not the same as the density of the tape drive, or if the block size cannot be supported by the tape drive.

MODE_ERROR

is raised if the file identified by TAPE_DRIVE is of mode IN_FILE.

STATUS_ERROR

is raised if TAPE_DRIVE is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Notes:

The first file is written immediately following the beginning of tape mark, overwriting the two tape marks written at initialization.

5.3.9.5. Loading a labeled tape

```
procedure LOAD_LABELED(TAPE_DRIVE: in FILE_TYPE;  
                      VOLUME_IDENTIFIER: in VOLUME_STRING;  
                      DENSITY: in POSITIVE;  
                      BLOCK_SIZE: in POSITIVE);
```

Purpose:

This procedure loads the labeled tape on the tape drive represented by the file identified by TAPE_DRIVE. It checks to see that the first block on the volume is a volume header label ("VOL1"). The VOLUME_IDENTIFIER in the parameter list must match the volume identifier in the volume header label on the tape. The tape is positioned at the beginning of tape. The DENSITY is validated against the settings of the tape drive. The block size for subsequent reads and writes is set to the value of BLOCK_SIZE. Following completion of this procedure, the function IS_LOADED(TAPE_DRIVE) (see Section 5.3.9.6) will return TRUE.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

VOLUME_IDENTIFIER

is the name which identifies the volume.

DENSITY

is the density in characters per inch (e.g., 800, 1600, 6250) at which the tape is to be read or written.

BLOCK_SIZE is the size of each data block which is to be read from or written to the file identified by TAPE_DRIVE.

Exceptions:

USE_ERROR is raised if IS_LOADED(TAPE_DRIVE) is TRUE or IS_MOUNTED(TAPE_DRIVE) is FALSE prior to the call, or the VOLUME_IDENTIFIER does not match the volume identifier in the volume header label on the tape, or if the tape is unlabeled. USE_ERROR is also raised if the block size cannot be supported by the tape drive or, if DENSITY is not the same as the density of the tape drive.

STATUS_ERROR

is raised if TAPE_DRIVE is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.6. Initializing a labeled tape

```
procedure INITIALIZE_LABELED(TAPE_DRIVE:      in FILE_TYPE;  
                             VOLUME_IDENTIFIER: in VOLUME_STRING;  
                             DENSITY:          in POSITIVE;  
                             BLOCK_SIZE:       in POSITIVE;  
                             ACCESSIBILITY:    in CHARACTER:= ' ');
```

Purpose:

This procedure initializes the tape which is mounted on the tape drive represented by the file identified by TAPE_DRIVE. The tape drive must have been mounted but not loaded. If the tape is not positioned at the beginning of tape, then the tape is rewound to it. A volume header label is written, followed by two tape marks. The tape is positioned following the volume header label. Initialization places the logical end of tape after the volume header label. The DENSITY is validated against the settings of the tape drive. The block size for subsequent reads and writes is set to the value of BLOCK_SIZE. The resulting tape is an initialized labeled tape.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

VOLUME_IDENTIFIER

is a six-character string giving the volume name.

DENSITY

is the density in characters per inch (e.g., 800, 1600, 6250) at which the tape is to be read or written.

BLOCK_SIZE is the size of each data block which is to be read from or written to the file identified by **TAPE_DRIVE**.

ACCESSIBILITY

is a character representing restrictions on access to the tape, in accordance with [ANSI 78]; a SPACE indicates no access control.

Exceptions:

USE_ERROR is raised if **MAGNETIC_TAPE** is not the value of the attribute **FILE_KIND** of the node associated with the file identified by **TAPE_DRIVE**, or the **VOLUME_IDENTIFIER** does not match the volume identifier in the volume header label on the tape, or if the tape is unlabeled.

MODE_ERROR

is raised if the file identified by **TAPE_DRIVE** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TAPE_DRIVE** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Notes:

When the first file is written on the tape, the file header label will follow the volume header created by this procedure.

5.3.9.7. Unloading a tape

procedure UNLOAD(TAPE_DRIVE: in FILE_TYPE);

Purpose:

This procedure unloads the tape on the tape drive represented by the file identified by **TAPE_DRIVE**. It rewinds the tape to the beginning of tape and releases the established block size. Following completion of this procedure, the function **IS_LOADED(TAPE_DRIVE)** will return FALSE.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

31 JANUARY 1985

Exceptions:

STATUS_ERROR

is raised if TAPE_DRIVE is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Notes:

If no conditions for these exceptions exist and there is no tape loaded on the tape drive, this procedure has no effect.

5.3.9.8. Dismounting a tape

procedure DISMOUNT(TAPE_DRIVE: in FILE_TYPE);

Purpose:

This procedure generates an implementation-defined request that the tape on the tape drive represented by the file identified by TAPE_DRIVE be removed from the drive. It makes the tape available for removal and releases the established density. Following the completion of this procedure, the function IS_MOUNTED (TAPE_DRIVE) will return FALSE.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

Exceptions:

USE_ERROR is raised if MAGNETIC_TAPE is not the value of the attribute FILE_KIND of the node associated with the file identified by TAPE_DRIVE.

STATUS_ERROR

is raised if TAPE_DRIVE is not an open file handle.

DEVICE_ERROR

is raised if this operation cannot be completed because of a malfunction of the underlying system.

Notes:

If no conditions for these exceptions exist and there is no tape mounted on the tape drive, this procedure has no effect.

5.3.9.9. Determining if the tape drive is loaded

**function IS_LOADED(TAPE_DRIVE: in FILE_TYPE)
return BOOLEAN;**

Purpose:

This function returns TRUE if the tape on the tape drive represented by the file identified by TAPE_DRIVE has been loaded; otherwise it returns FALSE.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

Exceptions:

USE_ERROR is raised if MAGNETIC_TAPE is not the value of the attribute FILE_KIND of the node associated with the file identified by TAPE_DRIVE.

STATUS_ERROR

is raised if TAPE_DRIVE is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.10. Determining if a tape is mounted

```
function IS_MOUNTED(TAPE_DRIVE: in FILE_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if a tape is mounted on the tape drive represented by the file identified by TAPE_DRIVE; otherwise it returns FALSE.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

Exceptions:

USE_ERROR is raised if MAGNETIC_TAPE is not the value of the attribute FILE_KIND of the node associated with the file identified by TAPE_DRIVE.

STATUS_ERROR

is raised if TAPE_DRIVE is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.11. Determining the position of the tape

```
function TAPE_STATUS(TAPE_DRIVE: in FILE_TYPE)
    return TAPE_POSITION;
```

Purpose:

This function returns current tape position information.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

Exceptions:

USE__ERROR is raised if MAGNETIC__TAPE is not the value of the attribute FILE__KIND of the node associated with the file identified by TAPE__DRIVE.

STATUS__ERROR

is raised if TAPE__DRIVE is not an open file handle.

DEVICE__ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.12. Rewinding the tape

procedure REWIND_TAPE(TAPE_DRIVE: in FILE_TYPE);

Purpose:

This procedure positions the tape at the beginning of tape.

Parameters:

TAPE__DRIVE is an open file handle on the file representing the tape drive.

Exceptions:

USE__ERROR is raised if MAGNETIC__TAPE is not the value of the attribute FILE__KIND of the node associated with the file identified by TAPE__DRIVE.

STATUS__ERROR

is raised if TAPE__DRIVE is not an open file handle.

DEVICE__ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.13. Skipping tape marks

procedure SKIP_TAPE_MARKS(TAPE_DRIVE: in FILE_TYPE;
NUMBER: in INTEGER:=1;
TAPE_STATE: out TAPE_POSITION);

Purpose:

This procedure provides a method of skipping over tape marks. A positive NUMBER indicates forward skipping, while a negative NUMBER indicates backward skipping. If NUMBER is zero, the tape position does not change.

Following a call to SKIP_TAPE_MARKS, if NUMBER is positive, the tape is positioned immediately following the appropriate tape mark. Following a call to SKIP_TAPE_MARKS, if NUMBER is negative, the tape is positioned immediately preceding the appropriate tape mark (i.e., at the end of a file or label). If two consecutive tape marks are encountered, the tape is positioned immediately following the second one, even if fewer than NUMBER tape marks have been skipped. Additionally, the current column, current line and current page numbers (see [IRM] 14.3) are set to one.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

NUMBER is the number of tape marks to skip and the direction of movement.

TAPE_STATE
is the position of the tape after skipping the specified number of tape marks.

Exceptions:

USE_ERROR is raised if **MAGNETIC_TAPE** is not the value of the attribute **FILE_KIND** of the node associated with the file identified by **TAPE_DRIVE**.

STATUS_ERROR
is raised if **TAPE_DRIVE** is not an open file handle.

DEVICE_ERROR
is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.14. Writing a tape mark

```
procedure WRITE_TAPE_MARK(TAPE_DRIVE: in    FILE_TYPE;  
                           NUMBER:      in    POSITIVE := 1;  
                           TAPE_STATE:   out TAPE_POSITION);
```

Purpose:

This procedure writes **NUMBER** consecutive tape marks on the tape which is mounted on the tape drive represented by the file identified by **TAPE_DRIVE**. The tape is stopped following the last tape mark written.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

NUMBER is the number of consecutive tape marks to be written.

TAPE_STATE
is the new position of the tape.

Exceptions:

USE_ERROR is raised if **MAGNETIC_TAPE** is not the value of the attribute **FILE_KIND** of the node associated with the file identified by **TAPE_DRIVE** or if **IS_LOADED(TAPE_DRIVE)** is **FALSE**.

MODE_ERROR
is raised if the file identified by **TAPE_DRIVE** is of mode **IN_FILE**.

STATUS_ERROR
is raised if **TAPE_DRIVE** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.15. Writing a volume header label

```

procedure VOLUME_HEADER(TAPE_DRIVE:      in FILE_TYPE;
                        VOLUME_IDENTIFIER: in VOLUME_STRING;
                        ACCESSIBILITY:     in CHARACTER := ' ');

```

Purpose:

This procedure writes a volume header label, as described in TABLE XI on the tape loaded on the tape drive represented by the file identified by TAPE_DRIVE.

The accessibility character is obtained from the ACCESSIBILITY parameter. The owner identification is the user name indicated by 'CURRENT_USER'. The Label-Standard Version, which is 3, indicates the ANSI standard version to which these labels conform.

Table XI. Volume header label

Character Position	Field Name	Content
1 to 3	Label Identifier	VOL
4	Label Number	1
5 to 10	Volume Identifier	Assigned permanently by owner to identify volume
11	Accessibility	Indicates restrictions on access to the information on the volume
12 to 37	Reserved for Future Standardization	Spaces
38 to 51	Owner Identity	Identifies owner of volume
52 to 79	Reserved for Future Standardization	Spaces
80	Label-Standard Version	Indicates the version of the ANSI standard to which the labels and data formats on this volume conform

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

VOLUME_IDENTIFIER

is a six-character string giving the volume name.

ACCESSIBILITY

is a character representing restrictions on access to the tape. In accordance with [ANSI 78]; a SPACE indicates no access control.

Exceptions:

USE_ERROR is raised if **MAGNETIC_TAPE** is not the value of the attribute **FILE_KIND** of the node associated with the file identified by **TAPE_DRIVE**. **USE_ERROR** is also raised if the tape on the tape drive represented by the file identified by **TAPE_DRIVE** was loaded as an unlabeled tape or if the value of **VOLUME_IDENTIFIER** does not conform to the syntax of an Ada identifier. **USE_ERROR** is also raised if **IS_LOADED(TAPE_DRIVE)** is **FALSE** at the time of the call.

MODE_ERROR

is raised if the file identified by **TAPE_DRIVE** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TAPE_DRIVE** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.16. Writing a file header label

```
procedure FILE_HEADER(TAPE_DRIVE:      in FILE_TYPE;  
                      FILE_IDENTIFIER: in FILE_STRING;  
                      EXPIRATION_DATE: in STRING := " 99366";  
                      ACCESSIBILITY  : in CHARACTER := ' ');
```

Purpose:

This procedure writes a file header label, as described in **TABLE XII**, on the tape loaded on the tape drive represented by the file identified by **TAPE_DRIVE**.

Parameters:

Table XII. File header label

Character Position	Field Name	Content
1 to 3	Label Identifier	HDR
4	Label Number	1
5 to 21	File Identifier	Assigned permanently by system to identify file
22 TO 27	File Set Identifier	The VOLUME_IDENTIFIER in the file set
28 to 31	File Section Number	0001
32 to 35	File Sequence Number	Distinguishes files in a file set. First file in set gets '0001'. For each file after, sequence number is incremented by one base 10.
36 to 39	Generation Number	0001
40 to 41	Generation Version Number	00
42 to 47	Creation Date	Date file header is written
48 to 53	Expiration Date	Date on which file may be overwritten
54	Accessibility	Indicates restrictions on access to information in file
55 to 60	Block COUNT	000000
61 to 73	System Code	Spaces
74 to 80	Reserved for Future Standardization	Spaces

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

FILE_IDENTIFIER

is a 17-character string giving the file name.

EXPIRATION_DATE

is a string identifying the date (8 characters 'YYDDD' where YY is the year and DDD is the day (001-366)) the file may be overwritten. When the expiration date is

a space followed by 5 zeroes, the file has expired. ACCESSIBILITY is a character representing restrictions on access to the tape. In accordance with [ANSI 78]; a SPACE indicates no access control.

Exceptions:

USE_ERROR is raised if MAGNETIC_TAPE is not the value of the attribute FILE_KIND of the node associated with the file identified by TAPE_DRIVE. USE_ERROR is also raised if the tape on the tape drive represented by the file identified by TAPE_DRIVE was loaded as an unlabeled tape or if FILE_IDENTIFIER does not conform to the syntax of an Ada identifier. USE_ERROR is also raised if IS_LOADED(TAPE_DRIVE) is FALSE at the time of the call.

MODE_ERROR

is raised if the file identified by TAPE_DRIVE is of mode IN_FILE.

STATUS_ERROR

is raised if TAPE_DRIVE is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.17. Writing an end of file label

procedure END_FILE_LABEL(TAPE_DRIVE: in FILE_TYPE);

Purpose:

This procedure writes an end of file label, as shown in TABLE XIII on the tape loaded on the tape drive represented by the file identified by TAPE_DRIVE.

Table XIII. End of file label

Character Position	Field Name	Contents
1 to 3	Label Identifier	EOF
4	Label Number	1
5 to 54	Same as corresponding fields in file header label	Same as corresponding fields in file header label
55 to 60	Block COUNT	Number of blocks in file
61 to 80	Same as corresponding fields in file header label	Same as corresponding fields in file header label

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

Exceptions:

USE_ERROR is raised if **MAGNETIC_TAPE** is not the value of the attribute **FILE_KIND** of the node associated with the file identified by **TAPE_DRIVE**. **USE_ERROR** is also raised if **IS_LOADED(TAPE_DRIVE)** is **FALSE** at the time of call or if the tape on the tape drive represented by the file identified by **TAPE_DRIVE** was loaded as an unlabeled tape.

MODE_ERROR

is raised if the file identified by **TAPE_DRIVE** is of mode **IN_FILE**.

STATUS_ERROR

is raised if **TAPE_DRIVE** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

5.3.9.18. Reading a label on a labeled tape

```
procedure READ_LABEL (TAPE_DRIVE: in FILE_TYPE;  
                      LABEL:      out LABEL_STRING);
```

Purpose:

This procedure obtains the first 80 characters of the next available block and returns them in **LABEL**.

Parameters:

TAPE_DRIVE is an open file handle on the file representing the tape drive.

LABEL is the 80-character string read from the tape.

Exceptions:

USE_ERROR is raised if the attempt to read eighty characters encounters a tape mark or if **MAGNETIC_TAPE** is not the value of the attribute **FILE_KIND** of the node associated with the file identified by **TAPE_DRIVE** or if **IS_LOADED(TAPE_DRIVE)** is **FALSE** at the time of the call. **USE_ERROR** is also raised if the tape on the tape drive represented by the file identified by **TAPE_DRIVE** was loaded as an unlabeled tape.

STATUS_ERROR

is raised if **TAPE_DRIVE** is not an open file handle.

DEVICE_ERROR

is raised if an input or output operation cannot be completed because of a malfunction of the underlying system or if the tape is uninitialized.

SEARCH_ERROR

is raised if there is no item with the name NAMED.

5.4.1.18. Inserting a list-type item into a list

```
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in    LIST_TYPE;  
                 POSITION:   in    COUNT);  
  
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in    LIST_TYPE;  
                 NAMED     : in    NAME_STRING;  
                 POSITION   : in    COUNT);  
  
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in    LIST_TYPE;  
                 NAMED     : in    TOKEN_TYPE;  
                 POSITION   : in    COUNT);
```

Purpose:

This procedure inserts a list-type item into a list after the list item specified by POSITION. A value of zero in POSITION specifies a position at the head of the list. Subsequent modifications to the values of LIST or of LIST_ITEM do not affect the other value.

Parameters:

LIST is the list into which the item will be inserted.

LIST_ITEM is the value of the item to be inserted.

POSITION is the position in the list after which the item is to be inserted.

NAMED is the name of the new item. It may only be used with named or empty lists.

Exceptions:

USE_ERROR is raised if an attempt is made to insert a named item into an unnamed list or, conversely, an attempt is made to insert an unnamed item into a named list or if LIST is a named list that already contains an item with the name NAMED or if POSITION specifies a value larger than the current length of the list.

5.4.1.19. Identifying a list-type item by value within a list

```
function POSITION_BY_VALUE(LIST:      in LIST_TYPE;  
                          VALUE:     in LIST_TYPE;  
                          START_POSITION: in POSITION_COUNT  
                          := POSITION_COUNT.FIRST;  
                          END_POSITION : in POSITION_COUNT  
                          := POSITION_COUNT.LAST)  
return POSITION_COUNT;
```

Purpose:

This function returns the position at which the next list-type item of the given value is located. The search begins at the START_POSITION and ends when either an item of value VALUE is found, the last item of the list has been examined, or the item at the END_POSITION has been examined, whichever comes first.

Parameters:

This function locates a list-type item in a list and returns in LIST_ITEM a copy of it. Subsequent modifications to the values of LIST or to the value returned in LIST_ITEM do not affect the other value.

Parameters:

LIST is the list containing the item to be extracted.

POSITION is the position within the list that identifies the item to be extracted.

LIST_ITEM is the value of the list-type item extracted.

NAMED is the name of the item to be extracted. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if the list is empty or if POSITION has a value larger than the current length of the list. USE_ERROR is also raised if NAMED is used with an unnamed list or if the POSITION specification or the name NAMED identifies an item not of list-type kind.

SEARCH_ERROR

is raised if there is not item with the name NAMED.

5.4.1.17. Replacing a list-type item in a list

```
procedure REPLACE(LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in   LIST_TYPE;  
                  POSITION : in   POSITION_COUNT);  
  
procedure REPLACE(LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in   LIST_TYPE;  
                  NAMED   : in   NAME_STRING);  
  
procedure REPLACE(LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in   LIST_TYPE;  
                  NAMED   : in   TOKEN_TYPE);
```

Purpose:

This procedure replaces the value of a list-type item in a list. Subsequent modifications to the values of LIST or of LIST_ITEM does not affect the other value.

Parameters:

LIST is the list containing the item to be replaced.

LIST_ITEM is the value of the new item.

POSITION is the position within the list that identifies the item to be replaced.

NAMED is the name of the item to be replaced. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if NAMED is used with an unnamed list, if the POSITION specification or the name NAMED identifies an item not of list-type kind, if the list is empty or or if POSITION has a value larger than the current length of the list.

Parameters:

LIST is the list of interest.

POSITION is the position within the list that identifies the item.

NAME is the token representation of the name of the item in the named list.

Exceptions:

USE_ERROR is raised if LIST is not a named list. If POSITION has a value larger than the current length of LIST.

5.4.1.15. Determining the position of a named item

```
function POSITION_BY_NAME(LIST: in LIST_TYPE;  
                          NAMED: in NAME_STRING)  
  return POSITION_COUNT;
```

```
function POSITION_BY_NAME(LIST: in LIST_TYPE;  
                          NAMED: in TOKEN_TYPE)  
  return POSITION_COUNT;
```

Purpose:

This function returns the position at which an item with the given name NAMED is located in LIST. It may only be used with named lists.

Parameters:

LIST is the list in which the position of an item is to be found by name.

NAMED is the name.

Exceptions:

USE_ERROR is raised if LIST is not a named list or if the list is empty.

SEARCH_ERROR

is raised if NAMED is not a name of an item contained in the list.

5.4.1.16. Extracting a list-type item from a list

```
procedure EXTRACT(LIST: in LIST_TYPE;  
                  POSITION: in POSITION_COUNT;  
                  LIST_ITEM: out LIST_TYPE);
```

```
procedure EXTRACT(LIST: in LIST_TYPE;  
                  NAMED: in NAME_STRING;  
                  LIST_ITEM: out LIST_TYPE);
```

```
procedure EXTRACT(LIST: in LIST_TYPE;  
                  NAMED: in TOKEN_TYPE;  
                  LIST_ITEM: out LIST_TYPE);
```

Purpose:

Parameters:

LIST is the list of interest.

Exceptions:

None.

5.4.1.13. Determining the length of a string representing a list or a list item

```
function TEXT_LENGTH(LIST:     in LIST_TYPE)
    return NATURAL;

function TEXT_LENGTH(LIST:     in LIST_TYPE;
                     POSITION: in POSITION_COUNT)
    return POSITIVE;

function TEXT_LENGTH(LIST:     in LIST_TYPE;
                     NAMED:   in NAME_STRING)
    return POSITIVE;

function TEXT_LENGTH(LIST:     in LIST_TYPE;
                     NAMED:   in TOKEN_TYPE)
    return POSITIVE;
```

Purpose:

This function returns the length of a string representing either a list or the list item identified by POSITION or NAMED in a list.

Parameters:

LIST is the list of interest.

POSITION is the position within the list that identifies the item.

NAMED is the name of the list item.

Exceptions:

USE_ERROR is raised if POSITION has a value larger than the (existing) length of the list or if the parameter NAMED is used with an unnamed list.

SEARCH_ERROR
is raised if there is no item with the name NAMED.

5.4.1.14. Determining the name of a named item

```
procedure ITEM_NAME(LIST:     in     LIST_TYPE;
                   POSITION: in     POSITION_COUNT;
                   NAME:       out TOKEN_TYPE);
```

Purpose:

This procedure returns in NAME the token representation of the name of the item in the named list, as specified by POSITION.

empty list. The values of FRONT and BACK are not affected. Subsequent modifications to the values of FRONT or BACK or to the value of the returned RESULT list do not affect the other list.

Parameters:

FRONT is the first list to be merged.

BACK is the second list to be merged.

RESULT is the list produced by the merge; it has the list items of FRONT in its initial sublist and those of BACK as the rest of its items.

Exceptions:

USE_ERROR is raised if FRONT and BACK are not of the same kind and neither of them is an empty list. **USE_ERROR** is also raised if FRONT and BACK are both named and contain an item with the same name.

5.4.1.11. Extracting a sublist of items from a list

```
function SET_EXTRACT(LIST:     in LIST_TYPE;  
                      POSITION: in POSITION_COUNT;  
                      LENGTH:  in POSITIVE := POSITIVE'LAST)  
return LIST_TEXT;
```

Purpose:

This function allows a (sub)list to be extracted from a list. The returned value is a copy of the list subset that starts at the item at POSITION and has LENGTH items in it. If there are fewer than LENGTH items in this part of the list, the subset extends to the tail of the list.

Parameters:

LIST is the list containing the subset to be extracted.

POSITION is the position within the list that identifies the subset to be extracted.

LENGTH is the length of the subset.

Exceptions:

USE_ERROR is raised if POSITION has a value larger than the current length of the list.

5.4.1.12. Determining the length of a list

```
function LENGTH(LIST: in LIST_TYPE)  
return COUNT;
```

Purpose:

This function returns a count of the number of items in LIST. If LIST is empty, LENGTH returns zero.

LIST is the list of interest.

POSITION is the position within the list that identifies the item.

NAMED is the name of the list item.

Exceptions:

USE_ERROR is raised if the parameter **NAMED** is used with an unnamed list, if the list is empty, if there is no item with the name **NAMED** or if **POSITION** has a value larger than the current length of **LIST**.

SEARCH_ERROR is raised if there is no item with the name **NAMED**.

5.4.1.9. Inserting a sublist of items into a list

```
procedure SPLICE(LIST: in out LIST_TYPE;  
                 POSITION: in POSITION_COUNT;  
                 SUB_LIST: in LIST_TEXT);
```

```
procedure SPLICE(LIST: in out LIST_TYPE;  
                 POSITION: in POSITION_COUNT;  
                 SUB_LIST: in LIST_TYPE);
```

Purpose:

This procedure allows a list to be inserted into a list. The items in the list to be inserted will become items in the resulting list. Subsequent modifications to the value of **LIST** or to the value of **SUB_LIST** do not affect the other list.

Parameters:

LIST is the list into which a list is to be inserted.

POSITION is the position after which the new items will be inserted.

SUB_LIST is the list to be inserted.

Exceptions:

USE_ERROR is raised if **SUB_LIST** as **LIST_TEXT** does not conform to the syntax specified in TABLE XIV. **USE_ERROR** is also raised if **LIST** and **SUB_LIST** are not of the same kind and neither of them is an empty list. **USE_ERROR** is also raised if **LIST** and **SUB_LIST** are both named and contain an item of the same name or if **POSITION** has a value larger than the current length of the list.

5.4.1.10. Merging two lists

```
procedure MERGE(FRONT: in LIST_TYPE;  
               BACK: in LIST_TYPE;  
               RESULT: in out LIST_TYPE);
```

Purpose:

This procedure returns in **RESULT** a list constructed by concatenating **BACK** to **FRONT**. The lists **FRONT** and **BACK** must be of the same kind or either **FRONT** or **BACK** must be an

Purpose:

This procedure deletes the item specified by POSITION or NAMED from LIST. If this was the last item in the list, the kind of the list changes to EMPTY.

Parameters:

LIST is the list from which the item will be deleted.

POSITION is the position within the list that identifies the item to be deleted.

NAMED is the name of the list item to be deleted.

Exceptions:

USE_ERROR is raised if the parameter NAMED is used with an unnamed list, if the list is empty, if there is no item with the name NAMED or if POSITION has a value larger than the current length of LIST.

SEARCH_ERROR
is raised if there is no item with the name NAMED.

5.4.1.7. Determining the kind of list

```
function GET_LIST_KIND(LIST: in LIST_TYPE)
return LIST_KIND;
```

Purpose:

This function returns the kind of the referenced list.

Parameters:

LIST is the list of interest.

Exceptions:

None.

5.4.1.8. Determining the kind of list item

```
function GET_ITEM_KIND(LIST: in LIST_TYPE;
                       POSITION: in POSITION_COUNT)
return ITEM_KIND;
```

```
function GET_ITEM_KIND(LIST: in LIST_TYPE;
                       NAMED: in NAME_STRING)
return ITEM_KIND;
```

```
function GET_ITEM_KIND(LIST: in LIST_TYPE;
                       NAMED: in TOKEN_TYPE)
return ITEM_KIND;
```

Purpose:

This function returns the kind of an item in the referenced list.

Parameters:

5.4.1.4. Converting to an external list representation

```
function TO_TEXT (LIST_ITEM: in LIST_TYPE)
  return LIST_TEXT;
```

Purpose:

This function returns the external representation of the value of the LIST_ITEM parameter. The representation is the string representation defined in Section 5.4.

Parameters:

LIST_ITEM is the list to be converted.

Exceptions:

None.

5.4.1.5. Determining the equality of two lists

```
function IS_EQUAL (LIST1: in LIST_TYPE;
  LIST2: in LIST_TYPE)
  return BOOLEAN;
```

Purpose:

This function returns TRUE if the values of the two lists LIST1 and LIST2 are equal according to the following rules; otherwise, it returns FALSE.

Two values of type LIST_TYPE are equal if and only if:

- a. both lists are of the same kind (i.e., named, unnamed or empty), and
- b. both lists contain the same number of list items, and
- c. for each position, the values of list items at this position, as obtained by an EXTRACT operation, are of the same kind and are equal under the equality defined for this kind, and
- d. in the case of named lists, for each position, the names of the list items at this position are equal under TOKEN_TYPE equality (i.e., IS_EQUAL).

Parameters:

LIST1, LIST2 are the lists whose equality is to be determined.

Exceptions:

None.

5.4.1.6. Deleting an item from a list

```
procedure DELETE (LIST: in out LIST_TYPE;
  POSITION: in POSITION_COUNT);

procedure DELETE (LIST: in out LIST_TYPE;
  NAMED: in NAME_STRING);

procedure DELETE (LIST: in out LIST_TYPE;
  NAMED: in TOKEN_TYPE);
```

CONSTRAINT_ERROR is raised if an attempt is made to convert a value to a numeric type when the value does not satisfy the constraints for that type.

5.4.1.2. Copying a list

```
procedure COPY(TO_LIST:      out LIST_TYPE;  
              FROM_LIST: in  LIST_TYPE);
```

Purpose:

This procedure returns in the parameter TO_LIST a copy of the list value of the parameter FROM_LIST. Subsequent modifications of either list do not affect the other list.

Parameters:

TO_LIST is the list returned as a copy of the value of FROM_LIST.

FROM_LIST is the list to be copied.

Exceptions:

None.

5.4.1.3. Converting to an internal list representation

```
procedure TO_LIST(LIST_STRING: in STRING;  
                 LIST:      out LIST_TYPE);
```

Purpose:

This procedure converts the string representation of a list into the internal list representation. It establishes the list as a named, unnamed, or empty list. The individual list items are classified according to their external representation. For a numeric item value, the item is classified as an integer item if the numeric value can be interpreted as a literal of universal_integer type; otherwise, the numeric item is classified as a floating point item. Blanks, format effectors and non-printing characters are allowed in the value of the parameter LIST_STRING.

Parameters:

LIST_STRING
is the string to be interpreted as a list value.

LIST
is the list built and returned according to the contents of LIST_STRING.

Exceptions:

USE_ERROR is raised if the value of the parameter LIST_STRING does not conform to the syntax of TABLE XIV. Blanks, format effectors and non-printing characters are allowed between lexical or syntactic elements of this syntax.

CONSTRAINT_ERROR

is raised if a numeric literal in the LIST_STRING parameter designates a value which cannot be represented as the value of an item in the LIST result.

- c. For an identifier list item or the name of a list item, the external string representation is the identifier string in upper case characters.
- d. For a quoted string list item, the external string representation is the string literal representing the value of the list item (i.e., the string value enclosed by quotation characters and with inner quotation characters doubled).
- e. for a list as a list item, the external string representation is the external representation of the value of the list.
- f. for a list, the external string representation of its value is the string representation composed of the external representation of its list items according to the syntax of Table XIV without blanks, format effectors or non-printing characters between the lexical or syntactic constituents of the syntax.

5.4.1. Package LIST_UTILITIES

This package defines types, subtypes, constants, exceptions and general list manipulation interfaces. The latter are supplemented by generic subpackages for the manipulation of list items of numeric type.

5.4.1.1. Types and subtypes

```
type LIST_TYPE      is limited private;
type TOKEN_TYPE     is limited private;
type LIST_KIND      is (UNNAMED, NAMED, EMPTY);
type ITEM_KIND      is (LIST_ITEM, STRING_ITEM, INTEGER_ITEM,
                        FLOAT_ITEM, IDENTIFIER_ITEM);
subtype LIST_TEXT    is STRING;
subtype NAME_STRING  is STRING;
type COUNT          is range 0 .. INTEGER'LAST;
subtype POSITION_COUNT is COUNT range COUNT'FIRST + 1 .. COUNT'LAST;
```

LIST_TYPE describes the type for lists. TOKEN_TYPE describes the type for internal representations of identifiers. LIST_KIND enumerates the kinds of lists. ITEM_KIND enumerates the kinds of list items. LIST_TEXT is the type of a list's external representation. NAME_STRING is the type of an identifier or of an item's name in a named item in its external representation. COUNT describes the type for the length of a list. POSITION_COUNT describes the type for the position of an item in a non-empty list.

EMPTY_LIST : constant LIST_TYPE;

EMPTY_LIST is a deferred constant denoting the value of an empty list. Any implementation of the CAIS must ensure that IS_EQUAL(EMPTY_LIST, X) is TRUE for any object X of type LIST_TYPE whose value is an empty list.

SEARCH_ERROR : exception;

CONSTRAINT_ERROR: exception;

SEARCH_ERROR is raised if a search for an item fails because the item is not present in the list.

(sub)list. Operations to delete an item or a set of items are also provided. Insertion and deletion operations will adjust the ordinal positions of items after the inserted or deleted items.

The value of an entity of type LIST_TYPE can be represented externally to the package LIST_UTILITIES as a string. Interfaces are provided to convert between entities of type STRING, containing a string value consistent with the syntax of this external representation, and entities of type LIST_TYPE. An object of type LIST_TYPE has as its initial value the empty list. The BNF for a list's external representation is given in TABLE XIV.

Table XIV. List external representation BNF

```
list ::= named_list
      | unnamed_list
      | empty_list
named_list ::= (named_item { , named_item } )
unnamed_list ::= (item { , item } )
empty_list ::= ()
named_item ::= name_string => item
item ::= list
      | quoted_string
      | integer_number
      | float_number
      | identifier
integer_number ::= integer
float_number ::= decimal_literal
quoted_string ::= string_literal
name_string ::= identifier
```

Notation:

1. Words - syntactic categories
2. [] - optional items
3. { } - an item repeated zero or more times
4. | - separates alternatives

The CAIS defines a canonical external string representation for values of type LIST_TYPE. The string subtype LIST_TEXT is used in the CAIS interfaces for string values that adhere to this canonical external representation. This external representation is obtained by applying the TO_TEXT operation to a value of type LIST_TYPE or to a value that is a legal value of a list item.

The canonical external string representation of a value of type LIST_TYPE and of its list items is defined as follows:

- a. For an integer list item, the external string representation is the decimal representation of its numeric value without leading zeroes.
- b. For a floating point list item, the external string representation is the string image of its numeric value in decimal notation with a format as obtained under implementation-defined settings of the FORE, AFT, and EXP parameters in PUT operations of Ada TEXT_IO (see [LRM] 14.3.8). These settings of FORE, AFT, and EXP must guarantee that quality of the external representation implies equality of the internal representation and vice versa within the limitations imposed by the accuracy of numeric comparisons in Ada.

5.4. CAIS Utilities

This section defines the abstract data type LIST_TYPE for use by other CAIS interfaces. The value of an entity of type LIST_TYPE (referred to as a *list*) is a linearly ordered set of data elements called *list items*.

It is possible to associate a name with a list item. If no name is associated with a list item, the item is an *unnamed item*. If a name is associated with a list item, the item is a *named item*. A list can either contain all unnamed items, in which case it is called an *unnamed list*, or all named items, in which case it is called a *named list*, but not both. If a list contains all named items, names among these items must be unique. An empty list is a list which contains no items. Such a list is not considered to be either named or unnamed. An empty list can be obtained by using the EMPTY_LIST constant or the DELETE procedure. The type LIST_KIND enumerates these three classifications of lists.

Associated with each list item is a classification, or kind. List items are classified as strings, integers, float numbers, identifiers and lists. The kind of an item is a value of the enumeration type ITEM_KIND. The CAIS interfaces allow, but do not require, an individual implementation of the CAIS to employ efficient mechanisms for representing identifiers as part of lists. Towards this purpose, a private type TOKEN_TYPE is introduced, which allows identifiers to be manipulated as internal representations called *tokens*. Interfaces are provided to transform identifiers in the form of a NAME_STRING into a TOKEN_TYPE and vice versa. NAME_STRING is a subtype of STRING, whose values are assumed to conform to the syntax of Ada identifiers. Tokens are equal if and only if their external representations are equal under string comparison, excepting differences in upper and lower case notation.

The names of list items in a named list may be internally represented as tokens. Overloaded interfaces are provided in the CAIS that allow the names of list items within a named list to be specified by parameters of either NAME_STRING or TOKEN_TYPE type.

The specifications within this package allow for the manipulation of lists which are of unnamed, named or empty kind. If a parameter of an interface specifies an item by position, then that interface may be used with either unnamed lists or named lists. If, however, a parameter specifies an item by name, then the associated interface may only be used with named lists.

Items of a list can be manipulated by:

- a. extracting items from a list,
- b. replacing or changing values of items in a list, and
- c. inserting new items into a list.

These operations are provided by the EXTRACT, REPLACE, and INSERT subprograms, respectively. Packages are provided to allow such operations to be performed directly on strings, identifiers and lists. Operations on the numeric types are provided with generic packages.

The positions in the list where these operations are specified to take place are usually designated by the parameter POSITION. With named lists a particular item can be specified by a name. This is possible since such names by definition are unique. Specifying a particular item by name is only permitted with EXTRACT and REPLACE operations.

Insertion operations can also be performed on sets of items. A set would then effectively constitute a

```
IMPORT (NODE, HOST_FILE_NAME);  
CLOSE (NODE);  
exception  
  when others =>  
    CLOSE (NODE);  
    raise;  
end IMPORT;
```

5.3.10.2. Exporting a file

```
procedure EXPORT (NODE:          in NODE_TYPE;  
                  HOST_FILE_NAME: in STRING);
```

Purpose:

This procedure creates a new file named HOST_FILE_NAME in the host file system and copies the contents of the file node identified by NODE into it.

Parameters:

NODE is an open node handle on the file node.

HOST_FILE_NAME
 is the name of the host file to be created.

Exceptions:

NAME_ERROR
 is raised if the node identified by NODE is inaccessible.

USE_ERROR is raised if HOST_FILE_NAME does not adhere to the required syntax for file names in the host file system or if HOST_FILE_NAME cannot be created in the host file system. USE_ERROR is also raised if FILE is not the value of the attribute KIND of the node identified by NODE.

STATUS_ERROR
 is raised if NODE is not an open node handle.

INTENT_VIOLATION
 is raised if NODE was not opened with an intent establishing the right to read contents.

Additional Interface:

```
procedure EXPORT (NAME:          in NAME_STRING;  
                  HOST_FILE_NAME: in STRING);  
is  
  NODE: NODE_TYPE;  
begin  
  OPEN (NODE, NAME, (1=>READ_CONTENTS));  
  EXPORT (NODE, HOST_FILE_NAME);  
  CLOSE (NODE);  
exception  
  when others =>  
    CLOSE (NODE);  
    raise;  
end EXPORT;
```

5.3.10. Package FILE_IMPORT_EXPORT

The CAIS allows a particular CAIS implementation to maintain files separately from files maintained by the host file system. This package provides the capability to transfer files between these two systems.

5.3.10.1. Importing a file

```
procedure IMPORT(NODE:      in NODE_TYPE;  
                 HOST_FILE_NAME: in STRING);
```

Purpose:

This procedure searches for a file in the host file system named HOST_FILE_NAME and copies its contents into a CAIS file which is the contents of the node identified by NODE. It also copies any file characteristic information which must be maintained by the CAIS implementation.

Parameters:

NODE is an open node handle on the file node.

HOST_FILE_NAME
 is the name of the host file to be copied.

Exceptions:

NAME_ERROR
 is raised if the node identified by NODE is inaccessible.

USE_ERROR is raised if HOST_FILE_NAME does not adhere to the required syntax for file names in the host file system or if HOST_FILE_NAME does not exist in the host file system. USE_ERROR is also raised if FILE is not the value of the attribute KIND of the node identified by NODE.

STATUS_ERROR
 is raised if NODE is not an open node handle.

INTENT_VIOLATION
 is raised if NODE was not opened with an intent establishing the right to write contents.

SECURITY_VIOLATION
 is raised if the operation represents a violation of mandatory access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interface:

```
procedure IMPORT(NAME:      in NAME_STRING;  
                 HOST_FILE_NAME: in STRING)  
is  
  NODE: NODE_TYPE;  
begin  
  OPEN(NODE, NAME, (1=>WRITE_CONTENTS));
```

LIST is the list in which the position of an item is to be found.

VALUE is the list-type item value.

START_POSITION
is the position of the first item to be considered in the search.

END_POSITION
is the position beyond which the search will not proceed; the search may terminate prior to reaching **END_POSITION** should the sought list-type item be found or should the last element of the list be considered.

Exceptions:

USE_ERROR is raised if **START_POSITION** specifies a value larger than the current length of the list, if the list is empty or if **END_POSITION** is less than **START_POSITION**.

SEARCH_ERROR
is raised if the **VALUE** specified is not found within the region specified by **START_POSITION** and **END_POSITION**.

5.4.1.20. Package IDENTIFIER_ITEM

This package provides interfaces for the manipulation of list items whose values are identifiers and of names of list items. Such names and values are represented internally as values of type **TOKEN_TYPE**.

5.4.1.20.1 Converting an identifier to a token

```
procedure TO_TOKEN (IDENTIFIER: in NAME_STRING;  
                    TOKEN:      out TOKEN_TYPE);
```

Purpose:

This procedure converts the string representation of an identifier into the corresponding internal token representation.

Parameters:

IDENTIFIER is the string to be converted to a token.

TOKEN is the token built and returned according to the value of **IDENTIFIER**.

Exceptions:

USE_ERROR is raised if the value of the parameter **IDENTIFIER** does not conform to the syntax of an Ada identifier.

5.4.1.20.2 Converting a token to an identifier

```
function TO_TEXT (LIST_ITEM: in TOKEN_TYPE)
return NAME_STRING;
```

Purpose:

This function returns the external representation of the value of the LIST_ITEM parameter. The external representation is the string representation defined in Section 5.4. It adheres to the syntax required for NAME_STRING..

Parameters:

LIST_ITEM is the item expressed as a token.

Exceptions:

None.

5.4.1.20.3 Determining the equality of two tokens

```
function IS_EQUAL (TOKEN1: in TOKEN_TYPE;
                  TOKEN2: in TOKEN_TYPE);
return BOOLEAN;
```

Purpose:

This function returns TRUE if the two tokens TOKEN1 and TOKEN2 represent Ada identifiers whose string representation is equal under string comparison, excepting differences in upper and lower case notation; otherwise, it returns FALSE.

Parameters:

TOKEN1, TOKEN2
are the tokens whose equality is to be determined.

Exceptions:

None.

5.4.1.20.4 Extracting an identifier item from a list

```
procedure EXTRACT (LIST: in LIST_TYPE;
                  POSITION: in POSITION_COUNT;
                  TOKEN: out TOKEN_TYPE);

procedure EXTRACT (LIST: in LIST_TYPE;
                  NAMED: in NAME_STRING;
                  TOKEN: out TOKEN_TYPE);

procedure EXTRACT (LIST: in LIST_TYPE;
                  NAMED: in TOKEN_TYPE;
                  TOKEN: out TOKEN_TYPE);
```

Purpose:

This function locates an identifier item in a list and returns in TOKEN a copy of its token.

Parameters:

LIST is the list containing the item to be extracted.

POSITION is the position within the list that identifies the item to be extracted.

TOKEN is the token representation of the identifier item.

NAMED is the name of the item to be extracted. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if **NAMED** is used with an unnamed list, if the **POSITION** specification or the name **NAMED** identifies an item not of token type if the list is empty or if **POSITION** has a value larger than the current length of the list.

SEARCH_ERROR
is raised if there is no item with the name **NAMED**.

5.4.1.20.5 Replacing an identifier item in a list

```
procedure REPLACE(LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in   TOKEN_TYPE;  
                  POSITION:   in   POSITION_COUNT);  
  
procedure REPLACE(LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in   TOKEN_TYPE;  
                  NAMED      : in   NAME_STRING);  
  
procedure REPLACE(LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in   TOKEN_TYPE;  
                  NAMED      : in   TOKEN_TYPE);
```

Purpose:

This procedure replaces the value of an identifier item in a list.

Parameters:

LIST is the list containing the item to be replaced.

LIST_ITEM is the new value of the item.

POSITION is the position within the list that identifies the item to be replaced.

NAMED is the name of the item to be replaced. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if **NAMED** is used with an unnamed list, if the **POSITION** specification or the name **NAMED** identifies an item not of identifier kind, if the list is empty, or if **POSITION** has a value larger than the current length of the list.

SEARCH_ERROR
is raised if there is no item with the name **NAMED**.

5.4.1.20.6 Inserting an identifier item into a list

```
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in   TOKEN_TYPE;  
                 POSITION:   in   COUNT);  
  
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in   TOKEN_TYPE;  
                 NAMED:     in   NAME_STRING;  
                 POSITION:   in   COUNT);  
  
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in   TOKEN_TYPE;  
                 NAMED      : in   TOKEN_TYPE;  
                 POSITION    : in   COUNT);
```

Purpose:

This procedure inserts an identifier item into a list after the list item specified by POSITION.
A value of zero in POSITION specifies a position at the head of the list.

Parameters:

LIST is the list into which the item will be inserted.

LIST_ITEM is the value of the item to be inserted.

POSITION is the position in the list after which the item is to be inserted.

NAMED is the name of the new item. It may only be used with named or empty lists.

Exceptions:

USE_ERROR is raised if an attempt is made to insert a named item into an unnamed list or, conversely, an attempt is made to insert an unnamed item into a named list or if LIST is a named list that already contains an item with the name NAMED.
USE_ERROR is also raised if POSITION specifies a value larger than the current length of the list.

5.4.1.20.7 Identifying an identifier item by value within a list

```
function POSITION_BY_VALUE(LIST:      in LIST_TYPE;  
                         VALUE:     in TOKEN_TYPE;  
                         START_POSITION: in POSITION_COUNT  
                         := POSITION_COUNT.FIRST;  
                         END_POSITION : in POSITION_COUNT  
                         := POSITION_COUNT.LAST)  
return POSITION_COUNT;
```

Purpose:

This function returns the position at which the next identifier item of the given value is located. The search begins at the START_POSITION and ends when either an item of value VALUE is found, the last item of the list has been examined, or the item at the END_POSITION has been examined, whichever comes first.

Parameters:

LIST is the list in which the position of an item is to be found by value.

VALUE is the Identifier Item value (token).

START_POSITION
is the position of the first item to be considered in the search.

END_POSITION
is the position beyond which the search will not proceed; the search may terminate prior to reaching **END_POSITION** should the sought Identifier Item be found or should the last element of the list be considered.

Exceptions:

USE_ERROR is raised if **START_POSITION** specifies a value larger than the current length of the list, if the list is empty or if **END_POSITION** is less than **START_POSITION**.

SEARCH_ERROR
is raised if the **VALUE** specified is not found within the region specified by **START_POSITION** and **END_POSITION**.

5.4.1.21. Generic package INTEGER_ITEM

This is a generic package for manipulating list items which are integers. This package must be instantiated for the appropriate integer type (indicated by **NUMBER** in the specification).

5.4.1.21.1 Converting an integer item to its canonical external representation

```
function TO_TEXT(LIST_ITEM: in NUMBER)
return STRING;
```

Purpose:

This function returns the external representation of the value of the **LIST_ITEM** parameter. The external representation is the string representation defined in Section 5.4.

Parameters:

LIST_ITEM is the integer item whose external representation is to be returned.

Exceptions:

None.

5.4.1.21.2 Extracting an integer item from a list

```
function EXTRACT(LIST: in LIST_TYPE;
POSITION: in POSITION_COUNT)
return NUMBER;
```

```
function EXTRACT(LIST : in LIST_TYPE;
NAMED: in NAME_STRING)
return NUMBER;
```

```
function EXTRACT(LIST : in LIST_TYPE;
NAMED: in TOKEN_TYPE)
return NUMBER;
```

31 JANUARY 1985

Purpose:

This function locates an Integer Item in a list and returns a copy of its numeric value.

Parameters:

- LIST** is the list containing the item to be extracted.
- POSITION** is the position within the list that identifies the item to be extracted.
- NAMED** is the name of the item to be extracted. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if NAMED is used with an unnamed list, if the POSITION specification or the name NAMED identifies an item not of integer kind, if the list is empty or if POSITION has a value larger than the current length of the list.

SEARCH_ERROR

is raised if there is no item with the name NAMED.

CONSTRAINT_ERROR

is raised if the value to be extracted violates the constraints of the type designated by NUMBER.

5.4.1.21.3 Replacing an integer item in a list

```
procedure REPLACE(LIST:      in out LIST_TYPE;
                  LIST_ITEM: in    NUMBER;
                  POSITION:   in    POSITION_COUNT);
```

```
procedure REPLACE(LIST:      in out LIST_TYPE;
                  LIST_ITEM: in    NUMBER;
                  NAMED:     in    NAME_STRING);
```

```
procedure REPLACE(LIST:      in out LIST_TYPE;
                  LIST_ITEM: in    NUMBER;
                  NAMED      : in    TOKEN_TYPE);
```

Purpose:

This procedure replaces the value of an Integer Item in a list.

Parameters:

- LIST** is the list containing the item to be replaced.
- LIST_ITEM** is the new value of the item.
- POSITION** is the position within the list that identifies the item to be replaced.
- NAMED** is the name of the item to be replaced. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if NAMED is used with an unnamed list or if the POSITION specification or the name NAMED identifies an item not of integer kind, if the list is empty or if POSITION has a value larger than the current length of the list.

SEARCH_ERROR

is raised if there is no item with the name NAMED.

5.4.1.21.4 Inserting an integer item into a list

```
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in    NUMBER;  
                 POSITION : in    COUNT);  
  
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in    NUMBER;  
                 NAMED      : in    NAME_STRING;  
                 POSITION : in    COUNT);  
  
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in    NUMBER;  
                 NAMED      : in    TOKEN_TYPE;  
                 POSITION : in    COUNT);
```

Purpose:

This procedure inserts an integer item into a list after the list item specified by POSITION. A value of zero in POSITION specifies a position at the head of the list.

Parameters:

LIST is the list into which the item will be inserted.

LIST_ITEM is the value of the item to be inserted.

POSITION is the position within the list after which the item is to be inserted.

NAMED is the name of the new item. It may only be used with named or empty lists.

Exceptions:

USE_ERROR is raised if an attempt is made to insert a named item into an unnamed list or, conversely, an attempt is made to insert an unnamed item into a named list or if LIST is a named list that already contains an item with the name NAMED. USE_ERROR is also raised if POSITION specifies a value larger than the current length of the list.

5.4.1.21.5 Identifying an integer item by value within a list

```
function POSITION_BY_VALUE(LIST:      in LIST_TYPE;  
                         VALUE:      in NUMBER;  
                         START_POSITION: in POSITION_COUNT  
                         := POSITION_COUNT'FIRST;  
                         END_POSITION:  in POSITION_COUNT  
                         := POSITION_COUNT'LAST)  
return POSITION_COUNT;
```

Purpose:

This function returns the position at which the next integer item of the given value is located. The search begins at the START_POSITION and ends when either an item of value VALUE is found, the last item of the list has been examined, or the item at the END_POSITION has been examined, whichever comes first.

Parameters:

LIST is the list in which the position of an item is to be found.

VALUE is the integer item value.

START_POSITION
is the position of the first item to be considered in the search.

END_POSITION
is the position beyond which the search will not proceed; the search may terminate prior to reaching **END_POSITION** should the sought integer item be found or should the last element of the list be considered.

Exceptions:

USE_ERROR is raised if **START_POSITION** specifies a value larger than the current length of the list, if the list is empty or if **END_POSITION** is less than **START_POSITION**.

SEARCH_ERROR
is raised if the **VALUE** specified is not found within the region specified by **START_POSITION** and **END_POSITION**.

5.4.1.22. Generic package FLOAT ITEM

This is a generic package for manipulating list items which are floating point numbers. This package must be instantiated for the appropriate type (indicated by **NUMBER** in the specification).

5.4.1.22.1 Converting a floating point item to its canonical external representation

```
function TO_TEXT(LIST_ITEM: in NUMBER)
  return STRING;
```

Purpose:

This function returns the external representation of the value of the **LIST_ITEM** parameter. The external representation is the string representation defined in Section 5.4.

Parameters:

LIST_ITEM is the floating point item whose external representation is to be returned.

Exceptions:

None.

5.4.1.22.2 Extracting a floating point item from a list

```
function EXTRACT(LIST: in LIST_TYPE;
  POSITION: in POSITION_COUNT)
  return NUMBER;
```

```
function EXTRACT(LIST: in LIST_TYPE;
  NAMED: in NAME_STRING)
  return NUMBER;
```

```
function EXTRACT(LIST: in LIST_TYPE;
  NAMED: in TOKEN_TYPE)
  return NUMBER;
```

Purpose:

This function locates a floating point item in a list and returns a copy of its numeric value.

Parameters:

- LIST** is the list containing the item to be extracted.
- POSITION** is the position within the list that identifies the item to be extracted.
- NAMED** is the name of the item to be extracted. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if **NAMED** is used with an unnamed list, if the **POSITION** specification or the name **NAMED** identifies an item not of floating point kind, if the list is empty or if **POSITION** has a value larger than the current length of the list.

SEARCH_ERROR

is raised if there is no item with the name **NAMED**.

CONSTRAINT_ERROR

is raised if the value to be extracted violates the constraints of the type designated by **NUMBER**.

5.4.1.22.3 Replacing a floating point item into a list

```
procedure REPLACE(LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in      NUMBER;  
                  POSITION:   in      POSITION_COUNT);  
  
procedure REPLACE(LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in      NUMBER;  
                  NAMED:     in      NAME_STRING);  
  
procedure REPLACE(LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in      NUMBER;  
                  NAMED:     in      TOKEN_TYPE);
```

Purpose:

This procedure replaces the value of a floating point item in a list.

Parameters:

- LIST** is the list containing the item to be replaced.
- LIST_ITEM** is the new value of the item.
- POSITION** is the position within the list that identifies the item to be replaced.
- NAMED** is the name of the item to be replaced. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if **NAMED** is used with an unnamed list, if the **POSITION** specification or

the name NAMED identifies an item not of floating point kind, if the list is empty or if POSITION has a value larger than the current length of the list.

SEARCH_ERROR

is raised if there is no item with the name NAMED.

5.4.1.22.4 Inserting a floating point item into a list

```
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in      NUMBER;  
                 POSITION:   in      COUNT);  
  
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in      NUMBER;  
                 NAMED:     in      NAME_STRING;  
                 POSITION:   in      COUNT);  
  
procedure INSERT(LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in      NUMBER;  
                 NAMED:     in      TOKEN_TYPE;  
                 POSITION:   in      COUNT);
```

Purpose:

This procedure inserts a floating point item into a list after the list item specified by POSITION. A value of zero in POSITION specifies a position at the head of the list.

Parameters:

LIST is the list into which the item will be inserted.

LIST_ITEM is the value of the item to be inserted.

POSITION is the position in the list after which the item is to be inserted.

NAMED is the name of the new item. It may only be used with named or empty lists.

Exceptions:

USE_ERROR is raised if an attempt is made to insert a named item into an unnamed list or, conversely, an attempt is made to insert an unnamed item into a named list or if LIST is a named list that already contains an item with the name NAMED. **USE_ERROR** is also raised if POSITION specifies a value larger than the current length of the list.

5.4.1.22.5 Identifying a floating point item by value within a list

```
function POSITION_BY_VALUE(LIST:      in LIST_TYPE;  
                         VALUE:     in NUMBER;  
                         START_POSITION: in POSITION_COUNT;  
                         END_POSITION:  in POSITION_COUNT;  
                         := POSITION_COUNT'FIRST;  
                         := POSITION_COUNT'LAST)  
return POSITION_COUNT;
```

Purpose:

This function returns the position at which the next floating point item of the given value is

located. The search begins at the `START_POSITION` and ends when either an item of value `VALUE` is found, the last item of the list has been examined, or the item at the `END_POSITION` has been examined, whichever comes first.

Parameters:

`LIST` is the list in which the position of an item is to be found.

`VALUE` is the floating point item value.

`START_POSITION`
is the position of the first item to be considered in the search.

`END_POSITION`
is the position beyond which the search will not proceed; the search may terminate prior to reaching `END_POSITION` should the sought floating point item be found or should the last element of the list be considered.

Exceptions:

`USE_ERROR` is raised if `START_POSITION` specifies a value larger than the current length of the list, or if `END_POSITION` is less than `START_POSITION`.

`SEARCH_ERROR`
is raised the `VALUE` specified is not found within the region specified by `START_POSITION` and `END_POSITION`.

5.4.1.23. Package STRING_ITEM

This is a package for manipulating list items which are strings. The external representation of the value of a string item is the string returned by an `EXTRACT` operation applied to the string item.

5.4.1.23.1 Extracting a string item from a list

```
function EXTRACT(LIST: in LIST_TYPE,  
                 POSITION: in POSITION_COUNT)  
  return STRING;
```

```
function EXTRACT(LIST: in LIST_TYPE;  
                 NAMED: in NAME_STRING)  
  return STRING;
```

```
function EXTRACT(LIST: in LIST_TYPE;  
                 NAMED: in TOKEN_TYPE)  
  return STRING;
```

Purpose:

This function locates a string item in a list and returns a copy of it.

Parameters:

`LIST` is the list containing the item to be extracted.

`POSITION` is the position within the list that identifies the item to be extracted.

NAMED is the name of the item to be extracted. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if **NAMED** is used with an unnamed list, if the **POSITION** specification or the name **NAMED** identifies an item not of string kind, if the list is empty or if **POSITION** has a value larger than the current length of the list.

SEARCH_ERROR

is raised if there is no item with the name **NAMED**.

5.4.1.23.2 Replacing a string item in a list

```
procedure REPLACE (LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in  STRING;  
                  POSITION:   in  POSITION_COUNT);
```

```
procedure REPLACE (LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in  STRING;  
                  NAMED:     in  NAME_STRING);
```

```
procedure REPLACE (LIST:      in out LIST_TYPE;  
                  LIST_ITEM: in  STRING;  
                  NAMED      : in  TOKEN_TYPE);
```

Purpose:

This procedure replaces the value of a string item in a list.

Parameters:

LIST is the list containing the item to be replaced.

LIST_ITEM is the new value of the item.

POSITION is the position within the list that identifies the item to be replaced.

NAMED is the name of the item to be replaced. It may only be used with named lists.

Exceptions:

USE_ERROR is raised if **NAMED** is used with an unnamed list or if the **POSITION** specification or the name **NAMED** identifies an item not of string kind, if the list is empty or if **POSITION** has a value larger than the current length of the list.

SEARCH_ERROR

is raised if there is no item with the name **NAMED**.

5.4.1.23.3 Inserting a string item into a list

```
procedure INSERT (LIST:      in out LIST_TYPE;  
                 LIST_ITEM: in  STRING;  
                 POSITION:   in  COUNT);
```

```
procedure INSERT (LIST:      in out LIST_TYPE;
```



```
LIST_ITEM: in    STRING;  
NAMED:      in    NAME_STRING;  
POSITION:   in    COUNT);
```

```
procedure INSERT(LIST:      in out LIST_TYPE;  
LIST_ITEM:   in    STRING;  
NAMED:       in    TOKEN_TYPE;  
POSITION :   in    COUNT);
```

Purpose:

This procedure inserts a string item into a list after the list item specified by POSITION. A value of zero in POSITION specifies a position at the head of the list.

Parameters:

LIST is the list into which the item will be inserted.

LIST_ITEM is the value of the item to be inserted.

POSITION is the position in the list after which the item is to be inserted.

NAMED is the name of the new item. It may only be used with named or empty lists.

Exceptions:

USE_ERROR is raised if an attempt is made to insert a named item into an unnamed list or, conversely, an attempt is made to insert an unnamed item into a named list or if LIST is a named list that already contains an item with the name NAMED. USE_ERROR is also raised if POSITION specifies a value larger than the current length of the list.

5.4.1.23.4 Identifying a string item by value within a list

```
function POSITION_BY_VALUE(LIST:      in LIST_TYPE;  
VALUE:      in STRING;  
START_POSITION: in POSITION_COUNT  
              :=POSITION_COUNT'FIRST;  
END_POSITION  : in POSITION_COUNT  
              :=POSITION_COUNT'LAST)  
return POSITION_COUNT;
```

Purpose:

This function returns the position at which the next string item of the given value is located. The search begins at the START_POSITION and ends when either an item of value VALUE is found, the last item of the list has been examined, or the item at the END_POSITION has been examined, whichever comes first.

Parameters:

LIST is the list in which the position of an item is to be found by value.

VALUE is the string item value.

START_POSITION is the position of the first item to be considered in the search

END_POSITION

is the position beyond which the search will not proceed; the search may terminate prior to reaching END_POSITION should the sought string item be found or should the last element of the list be considered.

Exceptions:

USE_ERROR is raised if START_POSITION specifies a value larger than the current length of the list, if the list is empty or if END_POSITION is less than START_POSITION.

SEARCH_ERROR

is raised if the VALUE specified is not found within the region specified by START_POSITION and END_POSITION.

```

        TIME_LIMIT:    DURATION := NO_DELAY);
procedure SET_CURRENT_NODE(NODE:  NODE_TYPE);
procedure SET_CURRENT_NODE(NAME:  NAME_STRING);
procedure GET_CURRENT_NODE
    (NODE:  in out NODE_TYPE;
     INTENT:    INTENTION := (1 => EXISTENCE);
     TIME_LIMIT:    DURATION := NO_DELAY);

private
    type NODE_ITERATOR is
        (IMPLEMENTATION_DEFINED);
    -- should be defined by implementor

end NODE_MANAGEMENT;

package ATTRIBUTES is
    use NODE_DEFINITIONS;
    use LIST_UTILITIES;

    subtype ATTRIBUTE_NAME is STRING;

    type ATTRIBUTE_ITERATOR is limited private;

    subtype ATTRIBUTE_PATTERN is STRING;

    procedure CREATE_NODE_ATTRIBUTE(NODE:  NODE_TYPE;
                                     ATTRIBUTE:  ATTRIBUTE_NAME;
                                     VALUE:  LIST_TYPE);
    procedure CREATE_NODE_ATTRIBUTE(NAME:  NAME_STRING;
                                     ATTRIBUTE:  ATTRIBUTE_NAME;
                                     VALUE:  LIST_TYPE);
    procedure CREATE_PATH_ATTRIBUTE(BASE:  NODE_TYPE;
                                     KEY:  RELATIONSHIP_KEY;
                                     RELATION:  RELATION_NAME :=
                                         DEFAULT_RELATION;
                                     ATTRIBUTE:  ATTRIBUTE_NAME;
                                     VALUE:  LIST_TYPE);
    procedure CREATE_PATH_ATTRIBUTE(NAME:  NAME_STRING;
                                     ATTRIBUTE:  ATTRIBUTE_NAME;
                                     VALUE:  LIST_TYPE);
    procedure DELETE_NODE_ATTRIBUTE(NODE:  NODE_TYPE;
                                     ATTRIBUTE:  ATTRIBUTE_NAME);
    procedure DELETE_NODE_ATTRIBUTE(NAME:  NAME_STRING;
                                     ATTRIBUTE:  ATTRIBUTE_NAME);
    procedure DELETE_PATH_ATTRIBUTE(BASE:  NODE_TYPE;
                                     KEY:  RELATIONSHIP_KEY;
                                     RELATION:  RELATION_NAME :=
                                         DEFAULT_RELATION;
                                     ATTRIBUTE:  ATTRIBUTE_NAME);
    procedure DELETE_PATH_ATTRIBUTE(NAME:  NAME_STRING;
                                     ATTRIBUTE:  ATTRIBUTE_NAME);
    procedure SET_NODE_ATTRIBUTE(NODE:  NODE_TYPE;
                                 ATTRIBUTE:  ATTRIBUTE_NAME;
                                 VALUE:  LIST_TYPE);
    procedure SET_NODE_ATTRIBUTE(NAME:  NAME_STRING;
                                 ATTRIBUTE:  ATTRIBUTE_NAME;
                                 VALUE:  LIST_TYPE);
    procedure SET_PATH_ATTRIBUTE(BASE:  NODE_TYPE;
                                 KEY:  RELATIONSHIP_KEY;
                                 RELATION:  RELATION_NAME :=
                                     DEFAULT_RELATION;
                                 ATTRIBUTE:  ATTRIBUTE_NAME;
                                 VALUE:  LIST_TYPE);
    procedure SET_PATH_ATTRIBUTE(NAME:  NAME_STRING;
                                 ATTRIBUTE:  ATTRIBUTE_NAME);

```

```

        NODE2: NODE_TYPE)
    return BOOLEAN;
function IS_SAME(NAME1: NAME_STRING;
                NAME2: NAME_STRING)
    return BOOLEAN;
procedure GET_PARENT
    (PARENT:      in out NODE_TYPE;
     NODE:        NODE_TYPE;
     INTENT:      INTENTION := (1 => READ);
     TIME_LIMIT:  DURATION := NO_DELAY);
procedure COPY_NODE
    (FROM:        NODE_TYPE;
     TO_BASE:     NODE_TYPE;
     TO_KEY:      RELATIONSHIP_KEY;
     TO_RELATION: RELATION_NAME := DEFAULT_RELATION);
procedure COPY_NODE(FROM: NODE_TYPE;
                    TO:   NAME_STRING);
procedure COPY_TREE
    (FROM:        NODE_TYPE;
     TO_BASE:     NODE_TYPE;
     TO_KEY:      RELATIONSHIP_KEY;
     TO_RELATION: RELATION_NAME := DEFAULT_RELATION);
procedure COPY_TREE(FROM: NODE_TYPE;
                    TO:   NAME_STRING);
procedure RENAME
    (NODE:        NODE_TYPE;
     NEW_BASE:    NODE_TYPE;
     NEW_KEY:     RELATIONSHIP_KEY;
     NEW_RELATION: RELATION_NAME := DEFAULT_RELATION);
procedure RENAME(NODE: NODE_TYPE;
                 NEW_NAME: NAME_STRING);
procedure DELETE_NODE(NODE: in out NODE_TYPE);
procedure DELETE_NODE(NAME: NAME_STRING);
procedure DELETE_TREE(NODE: in out NODE_TYPE);
procedure DELETE_TREE(NAME: NAME_STRING);
procedure LINK(NODE: NODE_TYPE;
               NEW_BASE: NODE_TYPE;
               NEW_KEY: RELATIONSHIP_KEY;
               NEW_RELATION: RELATION_NAME := DEFAULT_RELATION);
procedure LINK(NODE: NODE_TYPE;
               NEW_NAME: NAME_STRING);
procedure UNLINK(BASE: NODE_TYPE;
                 KEY: RELATIONSHIP_KEY;
                 RELATION: RELATION_NAME := DEFAULT_RELATION);
procedure UNLINK(NAME: NAME_STRING);
procedure ITERATE
    (ITERATOR: out NODE_ITERATOR;
     NODE:     NODE_TYPE;
     KIND:     NODE_KIND;
     KEY:      RELATIONSHIP_KEY_PATTERN := "**";
     RELATION: RELATION_NAME_PATTERN := DEFAULT_RELATION;
     PRIMARY_ONLY: BOOLEAN := TRUE);
procedure ITERATE
    (ITERATOR: out NODE_ITERATOR;
     NAME:     NAME_STRING;
     KIND:     NODE_KIND;
     KEY:      RELATIONSHIP_KEY_PATTERN := "**";
     RELATION: RELATION_NAME_PATTERN := DEFAULT_RELATION;
     PRIMARY_ONLY: BOOLEAN := TRUE);
function MORE(ITERATOR: NODE_ITERATOR)
    return BOOLEAN;
procedure GET_NEXT
    (ITERATOR: in out NODE_ITERATOR;
     NEXT_NODE: in out NODE_TYPE;
     INTENT:    INTENTION := (1 => EXISTENCE);

```

```

type TOKEN_TYPE is (IMPLEMENTATION_DEFINED);
-- should be defined by implementor
type LIST_TYPE is (IMPLEMENTATION_DEFINED);
-- should be defined by implementor
EMPTY_LIST : constant LIST_TYPE := (IMPLEMENTATION_DEFINED);
-- should be defined by implementor

```

end LIST_UTILITIES;

```

package NODE_MANAGEMENT is
use NODE_DEFINITIONS;
use LIST_UTILITIES;

```

```

type NODE_ITERATOR is limited private;
subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
subtype RELATION_NAME_PATTERN is RELATION_NAME;

```

```

procedure OPEN
(NODE:          in out NODE_TYPE;
 NAME:          NAME_STRING;
 INTENT:        INTENTION := (1 => READ);
 TIME_LIMIT:    DURATION := NO_DELAY);

procedure OPEN
(NODE:          in out NODE_TYPE;
 BASE:          NODE_TYPE;
 KEY:           RELATIONSHIP_KEY;
 RELATION:      RELATION_NAME := DEFAULT_RELATION;
 INTENT:        INTENTION := (1 => READ);
 TIME_LIMIT:    DURATION := NO_DELAY);

procedure CLOSE(NODE: in out NODE_TYPE);

procedure CHANGE_INTENT
(NODE:          in out NODE_TYPE;
 INTENT:        INTENTION;
 TIME_LIMIT:    DURATION := NO_DELAY);

function IS_OPEN(NODE: NODE_TYPE)
return BOOLEAN;

function INTENT_OF(NODE: NODE_TYPE)
return INTENTION;

function KIND(NODE: NODE_TYPE)
return NODE_KIND;

function PRIMARY_NAME(NODE: NODE_TYPE)
return NAME_STRING;

function PRIMARY_KEY(NODE: NODE_TYPE)
return RELATIONSHIP_KEY;

function PRIMARY_RELATION(NODE: NODE_TYPE)
return RELATION_NAME;

function PATH_KEY(NODE: NODE_TYPE)
return RELATIONSHIP_KEY;

function PATH_RELATION(NODE: NODE_TYPE)
return RELATION_NAME;

function BASE_PATH(NAME: NAME_STRING)
return NAME_STRING;

function LAST_RELATION(NAME: NAME_STRING)
return RELATION_NAME;

function LAST_KEY(NAME: NAME_STRING)
return RELATIONSHIP_KEY;

function IS_OBTAINABLE(NODE: NODE_TYPE)
return BOOLEAN;

function IS_OBTAINABLE(NAME: NAME_STRING)
return BOOLEAN;

function IS_OBTAINABLE(BASE:      NODE_TYPE,
                       KEY:        RELATIONSHIP_KEY,
                       RELATION:   RELATION_NAME := DEFAULT_RELATION)
return BOOLEAN;

function IS_SAME(NODE1: NODE_TYPE;

```

```

        NAMED:          NAME_STRING);
procedure REPLACE(LIST:      in out LIST_TYPE;
                  LIST_ITEM: NUMBER;
                  NAMED:     TOKEN_TYPE);
procedure INSERT(LIST:      in out LIST_TYPE;
                 LIST_ITEM: NUMBER;
                 POSITION:    COUNT);
procedure INSERT(LIST:      in out LIST_TYPE;
                 LIST_ITEM: NUMBER;
                 NAMED:     NAME_STRING;
                 POSITION:    COUNT);
procedure INSERT(LIST:      in out LIST_TYPE;
                 LIST_ITEM: NUMBER;
                 NAMED:     TOKEN_TYPE;
                 POSITION:    COUNT);
function POSITION_BY_VALUE
  (LIST:      LIST_TYPE;
   VALUE:     NUMBER;
   START_POSITION: POSITION_COUNT
    := POSITION_COUNT*FIRST;
   END_POSITION: POSITION_COUNT
    := POSITION_COUNT*LAST)
  return POSITION_COUNT;
end FLOAT_ITEM;
package STRING_ITEM is
  function EXTRACT(LIST:      LIST_TYPE;
                  POSITION:    POSITION_COUNT)
    return STRING;
  function EXTRACT(LIST:      LIST_TYPE;
                  NAMED:     NAME_STRING)
    return STRING;
  function EXTRACT(LIST:      LIST_TYPE;
                  NAMED:     TOKEN_TYPE)
    return STRING;
  procedure REPLACE(LIST:      in out LIST_TYPE;
                  LIST_ITEM:   STRING;
                  POSITION:      POSITION_COUNT);
  procedure REPLACE(LIST:      in out LIST_TYPE;
                  LIST_ITEM:   STRING;
                  NAMED:       NAME_STRING);
  procedure REPLACE(LIST:      in out LIST_TYPE;
                  LIST_ITEM:   STRING;
                  NAMED:       TOKEN_TYPE);
  procedure INSERT(LIST:      in out LIST_TYPE;
                  LIST_ITEM:   STRING;
                  POSITION:      COUNT);
  procedure INSERT(LIST:      in out LIST_TYPE;
                  LIST_ITEM:   STRING;
                  NAMED:       NAME_STRING;
                  POSITION:      COUNT);
  procedure INSERT(LIST:      in out LIST_TYPE;
                  LIST_ITEM:   STRING;
                  NAMED:       TOKEN_TYPE;
                  POSITION:      COUNT);
  function POSITION_BY_VALUE
    (LIST:      LIST_TYPE;
     VALUE:     STRING;
     START_POSITION: POSITION_COUNT
      := POSITION_COUNT*FIRST;
     END_POSITION: POSITION_COUNT
      := POSITION_COUNT*LAST)
    return POSITION_COUNT;
end STRING_ITEM;
private

```

PROPOSED MIL-STD-CAIS
31 JANUARY 1985

```

end IDENTIFIER_ITEM;
generic
  type NUMBER is range <>;
package INTEGER_ITEM is
  function TO_TEXT(LIST_ITEM: NUMBER)
    return STRING;
  function EXTRACT(LIST: LIST_TYPE;
    POSITION: POSITION_COUNT)
    return NUMBER;
  function EXTRACT(LIST: LIST_TYPE;
    NAMED: NAME_STRING)
    return NUMBER;
  function EXTRACT(LIST: LIST_TYPE;
    NAMED: TOKEN_TYPE)
    return NUMBER;
  procedure REPLACE(LIST: in out LIST_TYPE;
    LIST_ITEM: NUMBER;
    POSITION: POSITION_COUNT);
  procedure REPLACE(LIST: in out LIST_TYPE;
    LIST_ITEM: NUMBER;
    NAMED: NAME_STRING);
  procedure REPLACE(LIST: in out LIST_TYPE;
    LIST_ITEM: NUMBER;
    NAMED: TOKEN_TYPE);
  procedure INSERT(LIST: in out LIST_TYPE;
    LIST_ITEM: NUMBER;
    POSITION: COUNT);
  procedure INSERT(LIST: in out LIST_TYPE;
    LIST_ITEM: NUMBER;
    NAMED: NAME_STRING;
    POSITION: COUNT);
  procedure INSERT(LIST: in out LIST_TYPE;
    LIST_ITEM: NUMBER;
    NAMED: TOKEN_TYPE;
    POSITION: COUNT);
  function POSITION_BY_VALUE
    (LIST: LIST_TYPE;
     VALUE: NUMBER;
     START_POSITION: POSITION_COUNT;
     END_POSITION: POSITION_COUNT)
    return POSITION_COUNT;
end INTEGER_ITEM;
generic
  type NUMBER is digits <>;
package FLOAT_ITEM is
  function TO_TEXT(LIST_ITEM: NUMBER)
    return STRING;
  function EXTRACT(LIST: LIST_TYPE;
    POSITION: POSITION_COUNT)
    return NUMBER;
  function EXTRACT(LIST: LIST_TYPE;
    NAMED: NAME_STRING)
    return NUMBER;
  function EXTRACT(LIST: LIST_TYPE;
    NAMED: TOKEN_TYPE)
    return NUMBER;
  procedure REPLACE(LIST: in out LIST_TYPE;
    LIST_ITEM: NUMBER;
    POSITION: POSITION_COUNT);
  procedure REPLACE(LIST: in out LIST_TYPE;
    LIST_ITEM: NUMBER;

```

```

        POSITION: POSITION_COUNT)
    return POSITIVE;
function TEXT_LENGTH(LIST: LIST_TYPE;
    NAMED: NAME_STRING)
    return POSITIVE;
function TEXT_LENGTH(LIST: LIST_TYPE;
    NAMED: TOKEN_TYPE)
    return POSITIVE;

package IDENTIFIER_ITEM is
    procedure TO_TOKEN(IDENTIFIER: NAME_STRING;
        TOKEN: out TOKEN_TYPE);

    function TO_TEXT(LIST_ITEM: TOKEN_TYPE)
        return NAME_STRING;

    function IS_EQUAL(TOKEN1: TOKEN_TYPE;
        TOKEN2: TOKEN_TYPE)
        return BOOLEAN;

    procedure EXTRACT(LIST: LIST_TYPE;
        POSITION: POSITION_COUNT;
        TOKEN: out TOKEN_TYPE);
    procedure EXTRACT(LIST: LIST_TYPE;
        NAMED: NAME_STRING;
        TOKEN: out TOKEN_TYPE);

    procedure EXTRACT(LIST: LIST_TYPE;
        NAMED: TOKEN_TYPE;
        TOKEN: out TOKEN_TYPE);

    procedure REPLACE(LIST: in out LIST_TYPE;
        LIST_ITEM: TOKEN_TYPE;
        POSITION: POSITION_COUNT);

    procedure REPLACE(LIST: in out LIST_TYPE;
        LIST_ITEM: TOKEN_TYPE;
        NAMED: NAME_STRING);

    procedure REPLACE(LIST: in out LIST_TYPE;
        LIST_ITEM: TOKEN_TYPE;
        NAMED: TOKEN_TYPE);

    procedure INSERT(LIST: in out LIST_TYPE;
        LIST_ITEM: TOKEN_TYPE;
        POSITION: COUNT);

    procedure INSERT(LIST: in out LIST_TYPE;
        LIST_ITEM: TOKEN_TYPE;
        NAMED: NAME_STRING;
        POSITION: COUNT);

    procedure INSERT(LIST: in out LIST_TYPE;
        LIST_ITEM: TOKEN_TYPE;
        NAMED: TOKEN_TYPE;
        POSITION: COUNT);

    function POSITION_BY_VALUE(LIST: LIST_TYPE;
        VALUE: TOKEN_TYPE;
        START_POSITION: POSITION_COUNT
            := POSITION_COUNT'FIRST;
        END_POSITION: POSITION_COUNT
            := POSITION_COUNT'LAST)
        return POSITION_COUNT;

```



```

procedure INSERT(LIST:      in out LIST_TYPE;
                 LIST_ITEM: LIST_TYPE;
                 NAMED:      TOKEN_TYPE;
                 POSITION:     COUNT);

function POSITION_BY_VALUE(LIST: LIST_TYPE;
                          VALUE: LIST_TYPE;
                          START_POSITION: POSITION_COUNT
                                := POSITION_COUNT'FIRST;
                          END_POSITION:   POSITION_COUNT
                                := POSITION_COUNT'LAST)
    return POSITION_COUNT;

function SET_EXTRACT(LIST:      LIST_TYPE;
                     POSITION:    POSITION_COUNT;
                     LENGTH:     POSITIVE := POSITIVE'LAST)
    return LIST_TEXT;

procedure SPLICE(LIST:      in out LIST_TYPE;
                 POSITION:    POSITION_COUNT;
                 SUB_LIST:   LIST_TEXT);

procedure SPLICE(LIST:      in out LIST_TYPE;
                 POSITION:    POSITION_COUNT;
                 SUB_LIST:   LIST_TYPE);

procedure DELETE(LIST:      in out LIST_TYPE;
                 POSITION:    POSITION_COUNT);

procedure DELETE(LIST:      in out LIST_TYPE;
                 NAMED:      NAME_STRING);

procedure DELETE(LIST:      in out LIST_TYPE;
                 NAMED:      TOKEN_TYPE);

function GET_LIST_KIND(LIST: LIST_TYPE)
    return LIST_KIND;

function GET_ITEM_KIND(LIST:      LIST_TYPE;
                       POSITION:    POSITION_COUNT)
    return ITEM_KIND;

function GET_ITEM_KIND(LIST:      LIST_TYPE;
                       NAMED:      NAME_STRING)
    return ITEM_KIND;

function GET_ITEM_KIND(LIST:      LIST_TYPE;
                       NAMED:      TOKEN_TYPE)
    return ITEM_KIND;

procedure MERGE(FRONT:      LIST_TYPE;
                BACK:       LIST_TYPE;
                RESULT: in out LIST_TYPE);

function LENGTH(LIST: LIST_TYPE) return COUNT;

procedure ITEM_NAME(LIST:      LIST_TYPE;
                   POSITION:    POSITION_COUNT;
                   NAME:       out TOKEN_TYPE);

function POSITION_BY_NAME(LIST:      LIST_TYPE;
                        NAMED:      NAME_STRING)
    return POSITION_COUNT;

function POSITION_BY_NAME(LIST:      LIST_TYPE;
                        NAMED:      TOKEN_TYPE)
    return POSITION_COUNT;

function TEXT_LENGTH(LIST: LIST_TYPE)
    return NATURAL;

function TEXT_LENGTH(LIST:      LIST_TYPE;

```

31 JANUARY 1985

```

LOCK_ERROR:      exception;
ACCESS_VIOLATION: exception;
INTENT_VIOLATION: exception;
SECURITY_VIOLATION: exception;

private
  type NODE_TYPE is
    (IMPLEMENTATION_DEFINED);
    -- should be defined by implementor
end NODE_DEFINITIONS;

package LIST_UTILITIES is
  use NODE_DEFINITIONS;
  type LIST_TYPE is limited private;
  type TOKEN_TYPE is limited private;
  type LIST_KIND is (UNNAMED, NAMED, EMPTY);
  type ITEM_KIND is
    (LIST_ITEM,      STRING_ITEM,
     INTEGER_ITEM,   FLOAT_ITEM,   IDENTIFIER_ITEM);
  type COUNT is range 0 .. INTEGER'LAST;

  subtype LIST_TEXT is STRING;

  subtype POSITION_COUNT is COUNT range COUNT'FIRST + 1 .. COUNT'LAST;

  EMPTY_LIST:      constant LIST_TYPE;
  SEARCH_ERROR:    exception;
  CONSTRAINT_ERROR: exception;

  procedure COPY(TO_LIST: out LIST_TYPE;
                 FROM_LIST: LIST_TYPE);

  procedure TO_LIST(LIST_STRING: STRING;
                   LIST: out LIST_TYPE);
  function TO_TEXT(LIST_ITEM: LIST_TYPE)
    return LIST_TEXT;
  function IS_EQUAL(LIST1: LIST_TYPE;
                   LIST2: LIST_TYPE)
    return BOOLEAN;

  procedure EXTRACT(LIST: LIST_TYPE;
                   POSITION: POSITION_COUNT;
                   LIST_ITEM: out LIST_TYPE);
  procedure EXTRACT(LIST: LIST_TYPE;
                   NAMED: NAME_STRING;
                   LIST_ITEM: out LIST_TYPE);
  procedure EXTRACT(LIST: LIST_TYPE;
                   NAMED: TOKEN_TYPE;
                   LIST_ITEM: out LIST_TYPE);
  procedure REPLACE(LIST: in out LIST_TYPE;
                   LIST_ITEM: LIST_TYPE;
                   POSITION: POSITION_COUNT);
  procedure REPLACE(LIST: in out LIST_TYPE;
                   LIST_ITEM: LIST_TYPE;
                   NAMED: NAME_STRING);
  procedure REPLACE(LIST: in out LIST_TYPE;
                   LIST_ITEM: LIST_TYPE;
                   NAMED: TOKEN_TYPE);
  procedure INSERT(LIST: in out LIST_TYPE;
                   LIST_ITEM: LIST_TYPE;
                   POSITION: COUNT);
  procedure INSERT(LIST: in out LIST_TYPE;
                   LIST_ITEM: LIST_TYPE;
                   NAMED: NAME_STRING;
                   POSITION: COUNT);

```

Appendix B CAIS Specification

This appendix contains a set of Ada package specifications of the CAIS Interfaces which complies correctly. It brings together the Interfaces found in Section 5 using the Nested Generic Subpackages Implementation approach. Although the interfaces are not necessarily shown here in the order in which they are discussed in the text, this appendix provides a reference listing of the CAIS as well as an illustration of the generics approach.

```
with CALENDAR;
use CALENDAR;
package CAIS is
  package NODE_DEFINITIONS is

    type NODE_TYPE is limited private;
    type NODE_KIND is (FILE, STRUCTURAL, PROCESS);
    type INTENT_SPECIFICATION is
      (EXISTENCE,
       READ,
       WRITE,
       READ_ATTRIBUTES,
       WRITE_ATTRIBUTES,
       APPEND_ATTRIBUTES,
       READ_RELATIONSHIPS,
       WRITE_RELATIONSHIPS,
       APPEND_RELATIONSHIPS,
       READ_CONTENTS,
       WRITE_CONTENTS,
       APPEND_CONTENTS,
       CONTROL,
       EXECUTE,
       EXCLUSIVE_READ,
       EXCLUSIVE_WRITE,
       EXCLUSIVE_READ_ATTRIBUTES,
       EXCLUSIVE_WRITE_ATTRIBUTES,
       EXCLUSIVE_APPEND_ATTRIBUTES,
       EXCLUSIVE_READ_RELATIONSHIPS,
       EXCLUSIVE_WRITE_RELATIONSHIPS,
       EXCLUSIVE_APPEND_RELATIONSHIPS,
       EXCLUSIVE_READ_CONTENTS,
       EXCLUSIVE_WRITE_CONTENTS,
       EXCLUSIVE_APPEND_CONTENTS,
       EXCLUSIVE_CONTROL);
    type INTENTION is array (POSITIVE range <>)
      of INTENT_SPECIFICATION;

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

    CURRENT_USER: constant NAME_STRING := "CURRENT_USER";
    CURRENT_NODE: constant NAME_STRING := "CURRENT_NODE";
    CURRENT_PROCESS: constant NAME_STRING := "";
    LATEST_KEY: constant RELATIONSHIP_KEY := "*";
    DEFAULT_RELATION: constant RELATION_NAME := "DOT";
    NO_DELAY: constant DURATION := DURATION_FIRST;

    NAME_ERROR: exception;
    USE_ERROR: exception;
    STATUS_ERROR: exception;
```

applies to process nodes; designates the classification of the node's process as a subject; values are implementation-defined.

TERMINAL_KIND:

applies to file nodes with a FILE_KIND attribute value of TERMINAL; designates the kind of terminal which is represented by the node's contents; possible values are SCROLL, PAGE and FORM.

Predefined Attribute Values:

ABORTED
APPEND
APPEND_ATTRIBUTES
APPEND_CONTENTS
APPEND_RELATIONSHIPS
CONTROL
COPY
DIRECT
EXECUTE
EXISTENCE
FILE
FORM
MAGNETIC_TAPE
MIMIC
PAGE
PROCESS
QUEUE
READ
READ_ATTRIBUTES
READ_CONTENTS
READ_RELATIONSHIPS
READY
SCROLL
SECONDARY_STORAGE
SEQUENTIAL
SOLO
STRUCTURAL
SUSPENDED
TERMINAL
TERMINATED
TEXT
WRITE
WRITE_ATTRIBUTES
WRITE_CONTENTS
WRITE_RELATIONSHIPS

FINISH_TIME:

applies to process nodes; designates the implementation-defined time at which the process terminated or aborted.

GRANT:

applies to relationships of the predefined relation ACCESS; designates the access rights which can be granted via the access relationship; values are lists of relevant grant items as specified in TABLE II.

HANDLES_OPEN:

applies to process nodes; designates the number of node handles the node's process currently has opened.

HIGHEST_CLASSIFICATION:

applies to file nodes; designates the highest allowable object classification label that may be assigned to the node; values are implementation-defined.

IO_UNITS:

applies to process nodes; designates the number of GET and PUT operations that have been performed by the node's process.

KIND:

applies to all relationships; designates the kind of the target node; possible values are STRUCTURAL, PROCESS and FILE.

LOWEST_CLASSIFICATION:

applies to file nodes; designates the lowest allowable object classification label that may be assigned to the node; values are implementation-defined.

MACHINE_TIME:

applies to process nodes; designates the length of time the process was active on the logical processor, if the process has terminated or aborted, or zero, if the process has not terminated or aborted.

OBJECT_CLASSIFICATION:

applies to all nodes; designates the node's classification as an object; values are implementation-defined.

PARAMETERS:

applies to process nodes; designates the parameters with which the process was initiated.

QUEUE_KIND:

applies to file nodes with a FILE_KIND attribute value of QUEUE; designates the kind of queue file; possible values are SOLO, MIMIC and COPY.

RESULTS:

applies to process nodes; designates the intermediate results of the process; values are user-defined.

START_TIME:

applies to process nodes; designates the implementation-defined time of activation of the process.

SUBJECT_CLASSIFICATION:

representing a device to which the process has access. Also designates a primary relationship from the system-level node to a node representing a device.

DOT: designates the default relation name to be used when none is provided. Special rules apply for pathname abbreviations in the presence of path elements whose relation name is DOT. No other semantics are associated with DOT.

JOB: designates a primary relationship from the top-level node of a user to the root process node of a job.

PARENT: designates the secondary relationship from a given node to the node which is the source of the unique primary relationship pointing to the given node.

PERMANENT_MEMBER:
designates a primary relationship from a node representing a group to the node representing a permanent member of the group.

POTENTIAL_MEMBER:
designates a secondary relationship from a node representing a group to the node representing a potential member of the group.

STANDARD_ERROR:
designates the secondary relationship from a process node to a file node representing the standard device for error messages for the whole job.

STANDARD_INPUT:
designates the secondary relationship from a process node to a file representing the standard input device for the whole job.

STANDARD_OUTPUT:
designates the secondary relationship from a process node to a file node representing the standard output device for the whole job.

USER: designates a secondary relationship from a process node to a user's top-level node. Also designates a primary relationship from the system-level node to a top-level node representing a user.

Predefined Attributes:

ACCESS_METHOD:
applies to file nodes; designates the kind of access which can be used on the node's contents; possible values are SEQUENTIAL, DIRECT and TEXT.

CURRENT_STATUS:
applies to process nodes; designates the current status of the node's contents; possible values are READY, SUSPENDED, ABORTED and TERMINATED.

FILE_KIND: applies to file nodes; designates the kind of file that is the node's contents; possible values are SECONDARY_STORAGE, QUEUE, TERMINAL and MAGNETIC_TAPE.

Appendix A PREDEFINED RELATIONS, ATTRIBUTES AND ATTRIBUTE VALUES

Predefined Relations:

- ACCESS:** designates a secondary relationship from an object node to a node representing a role; the access rights that can be granted to adopters of the role are given in the GRANT attribute of this relationship.
- ADOPTED_ROLE:** designates a secondary relationship from a subject (process) node to a node representing a role; indicates that the process has adopted the role represented by the node.
- ALLOW_ACCESS:** designates a secondary relationship from a process node to a node representing a role; indicates that the process can create relationships of the predefined relation ACCESS from an object to this node representing the role.
- COUPLE:** designates a secondary relationship from a node representing a queue file to the node representing that file's coupled file; indicates that the queue file and the other file are coupled; for copy queue files, this means the contents of the file are the initial contents of the queue file; for mimic queue files, this means that the contents of the file are the initial contents of the queue file and subsequent writes to the queue file are appended to the other file as well.
- CURRENT_ERROR:** designates a secondary relationship from a process node to a file node representing the file to which error messages are to be written.
- CURRENT_INPUT:** designates a secondary relationship from a process node to a file node representing the file which is currently the source of process inputs.
- CURRENT_JOB:** designates a secondary relationship from a process node to the root process node of the tree which contains the process node.
- CURRENT_NODE:** designates a secondary relationship from a process node to the node representing the current focus of attention or context for the process' activities.
- CURRENT_OUTPUT:** designates a secondary relationship from a process node to a file node representing the file to which outputs are currently being directed.
- CURRENT_USER:** designates a secondary relationship from a process node to a top-level node representing the user on whose behalf the process was initiated.
- DEVICE:** designates a secondary relationship from a process node to a top-level node

6. NOTES

6.1. Keywords

The following list represents the keywords applicable to this standard. These keywords may be used to categorize the concepts presented within this standard and assist in automatic retrieval of appropriate data used in automated document retrieval systems.

- Ada
- APSE
- CAIS
- Common APSE Interface Set
- computer file system
- KAPSE
- high level languages
- interfaces
- interoperability
- operating system
- portability
- programming support environment
- software engineering environment
- transportability
- virtual operating system

31 JANUARY 1985

```

                                VALUE:      LIST_TYPE);
procedure GET_NODE_ATTRIBUTE(NODE:      NODE_TYPE;
                                ATTRIBUTE: ATTRIBUTE_NAME;
                                VALUE:    in out LIST_TYPE);
procedure GET_NODE_ATTRIBUTE(NAME:      NAME_STRING;
                                ATTRIBUTE: ATTRIBUTE_NAME;
                                VALUE:    in out LIST_TYPE);
procedure GET_PATH_ATTRIBUTE(BASE:      NODE_TYPE;
                                KEY:      RELATIONSHIP_KEY;
                                RELATION:  RELATION_NAME :=
                                            DEFAULT_RELATION;
                                ATTRIBUTE: ATTRIBUTE_NAME;
                                VALUE:    in out LIST_TYPE);
procedure GET_PATH_ATTRIBUTE(NAME:      NAME_STRING;
                                ATTRIBUTE: ATTRIBUTE_NAME;
                                VALUE:    in out LIST_TYPE);
procedure NODE_ATTRIBUTE_ITERATE(ITERATOR: out ATTRIBUTE_ITERATOR;
                                NODE:      NODE_TYPE;
                                PATTERN:    ATTRIBUTE_PATTERN := "");
procedure NODE_ATTRIBUTE_ITERATE(ITERATOR: out ATTRIBUTE_ITERATOR;
                                NAME:      NAME_STRING;
                                PATTERN:    ATTRIBUTE_PATTERN := "");
procedure PATH_ATTRIBUTE_ITERATE(ITERATOR: out ATTRIBUTE_ITERATOR;
                                BASE:      NODE_TYPE;
                                KEY:      RELATIONSHIP_KEY;
                                RELATION:  RELATION_NAME :=
                                            DEFAULT_RELATION;
                                PATTERN:    ATTRIBUTE_PATTERN := "");
procedure PATH_ATTRIBUTE_ITERATE(ITERATOR: out ATTRIBUTE_ITERATOR;
                                NAME:      NAME_STRING;
                                PATTERN:    ATTRIBUTE_PATTERN := "");
function MORE(ITERATOR: ATTRIBUTE_ITERATOR)
    return BOOLEAN;
procedure GET_NEXT(ITERATOR: in out ATTRIBUTE_ITERATOR;
                                ATTRIBUTE: out ATTRIBUTE_NAME;
                                VALUE:    in out LIST_TYPE);
private
    type ATTRIBUTE_ITERATOR is (IMPLEMENTATION_DEFINED);
    -- should be defined by implementor
end ATTRIBUTES;

package ACCESS_CONTROL is
    use NODE_DEFINITIONS;

    subtype GRANT_VALUE is CAIS.LIST_UTILITIES.LIST_TYPE;
    procedure SET_ACCESS_CONTROL(NODE:      NODE_TYPE;
                                ROLE_NODE:  NODE_TYPE;
                                GRANT:      GRANT_VALUE);
    procedure SET_ACCESS_CONTROL(NAME:      NAME_STRING;
                                ROLE_NAME:  NAME_STRING;
                                GRANT:      GRANT_VALUE);
    function IS_GRANTED(OBJECT_NODE:  NODE_TYPE;
                        ACCESS_RIGHT:  NAME_STRING)
        return BOOLEAN;
    function IS_GRANTED(OBJECT_NAME:  NAME_STRING;
                        ACCESS_RIGHT:  NAME_STRING)
        return BOOLEAN;
    procedure ADOPT(ROLE_NODE:  NODE_TYPE;
                    ROLE_KEY:    RELATIONSHIP_KEY := LATEST_KEY);
    procedure UNADOPT(ROLE_KEY:  RELATIONSHIP_KEY);
end ACCESS_CONTROL;

package STRUCTURAL_NODES is
    use NODE_DEFINITIONS;

```

```

use LIST_UTILITIES;

procedure CREATE_NODE
  (NODE:
   BASE:
   KEY:
   RELATION:
   ATTRIBUTES:
   ACCESS_CONTROL:
   LEVEL:
  in out NODE_TYPE;
   NODE_TYPE;
   RELATIONSHIP_KEY := LATEST_KEY;
   RELATION_NAME := DEFAULT_RELATION;
   LIST_TYPE := EMPTY_LIST;
   LIST_TYPE := EMPTY_LIST;
   LIST_TYPE := EMPTY_LIST);

procedure CREATE_NODE
  (NODE:
   NAME:
   ATTRIBUTES:
   ACCESS_CONTROL:
   LEVEL:
  in out NODE_TYPE;
   NAME_STRING;
   LIST_TYPE := EMPTY_LIST;
   LIST_TYPE := EMPTY_LIST;
   LIST_TYPE := EMPTY_LIST);

procedure CREATE_NODE
  (BASE:
   KEY:
   RELATION:
   ATTRIBUTES:
   ACCESS_CONTROL:
   LEVEL:
  NODE_TYPE;
   RELATIONSHIP_KEY := LATEST_KEY;
   RELATION_NAME := DEFAULT_RELATION;
   LIST_TYPE := EMPTY_LIST;
   LIST_TYPE := EMPTY_LIST;
   LIST_TYPE := EMPTY_LIST);

procedure CREATE_NODE
  (NAME:
   ATTRIBUTES:
   ACCESS_CONTROL:
   LEVEL:
  NAME_STRING;
   LIST_TYPE := EMPTY_LIST;
   LIST_TYPE := EMPTY_LIST;
   LIST_TYPE := EMPTY_LIST);

end STRUCTURAL_NODES;

package PROCESS_DEFINITIONS is
  use NODE_DEFINITIONS;
  use LIST_UTILITIES;

  type PROCESS_STATUS is
    (READY, SUSPENDED, ABORTED, TERMINATED);

  subtype RESULTS_LIST is CAIS.LIST_UTILITIES.LIST_TYPE;
  subtype RESULTS_STRING is STRING;
  subtype PARAMETER_LIST is CAIS.LIST_UTILITIES.LIST_TYPE;

  ROOT_PROCESS: constant NAME_STRING := "CURRENT_JOB";
  CURRENT_INPUT: constant NAME_STRING := "CURRENT_INPUT";
  CURRENT_OUTPUT: constant NAME_STRING := "CURRENT_OUTPUT";
  CURRENT_ERROR: constant NAME_STRING := "CURRENT_ERROR";

end PROCESS_DEFINITIONS;

package PROCESS_CONTROL is
  use NODE_DEFINITIONS;
  use LIST_UTILITIES;
  use PROCESS_DEFINITIONS;

  procedure SPAWN_PROCESS
    (NODE:
     FILE_NODE:
     INPUT_PARAMETERS:
     KEY:
     RELATION:
     ACCESS_CONTROL:
     LEVEL:
     ATTRIBUTES:
    in out NODE_TYPE;
     NODE_TYPE;
     PARAMETER_LIST := EMPTY_LIST;
     RELATIONSHIP_KEY := LATEST_KEY;
     RELATION_NAME := DEFAULT_RELATION;
     LIST_TYPE := EMPTY_LIST;
     LIST_TYPE := EMPTY_LIST;
     LIST_TYPE :=

```

31 JANUARY 1985

```

                                EMPTY_LIST;
    INPUT_FILE:                 NAME_STRING := CURRENT_INPUT;
    OUTPUT_FILE:                NAME_STRING := CURRENT_OUTPUT;
    ERROR_FILE:                 NAME_STRING := CURRENT_ERROR;
    ENVIRONMENT_NODE:           NAME_STRING := CURRENT_NODE);

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

procedure AWAIT_PROCESS_COMPLETION
    (NODE:                       NODE_TYPE;
     RESULTS_RETURNED: in out RESULTS_LIST;
     STATUS:             out PROCESS_STATUS;
     TIME_LIMIT:         DURATION := DURATION'LAST);

procedure INVOKE_PROCESS
    (NODE: in out NODE_TYPE;
     FILE_NODE: NODE_TYPE;
     RESULTS_RETURNED: in out RESULTS_LIST;
     STATUS: out PROCESS_STATUS;
     INPUT_PARAMETERS: PARAMETER_LIST;
     KEY: RELATIONSHIP_KEY := LATEST_KEY;
     RELATION: RELATION_NAME := DEFAULT_RELATION;
     ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
     LEVEL: LIST_TYPE := EMPTY_LIST;
     ATTRIBUTES: LIST_TYPE := EMPTY_LIST;
     INPUT_FILE: NAME_STRING := CURRENT_INPUT;
     OUTPUT_FILE: NAME_STRING := CURRENT_OUTPUT;
     ERROR_FILE: NAME_STRING := CURRENT_ERROR;
     ENVIRONMENT_NODE: NAME_STRING := CURRENT_NODE;
     TIME_LIMIT: DURATION := DURATION'LAST);

procedure CREATE_JOB
    (FILE_NODE: NODE_TYPE;
     INPUT_PARAMETERS: PARAMETER_LIST := EMPTY_LIST;
     KEY: RELATIONSHIP_KEY := LATEST_KEY;
     ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
     LEVEL: LIST_TYPE := EMPTY_LIST;
     ATTRIBUTES: LIST_TYPE := EMPTY_LIST;
     INPUT_FILE: NAME_STRING := CURRENT_INPUT;
     OUTPUT_FILE: NAME_STRING := CURRENT_OUTPUT;
     ERROR_FILE: NAME_STRING := CURRENT_ERROR;
     ENVIRONMENT_NODE: NAME_STRING := CURRENT_USER);

procedure APPEND_RESULTS(RESULTS: RESULTS_STRING);
procedure WRITE_RESULTS(RESULTS: RESULTS_STRING);
procedure GET_RESULTS(NODE: NODE_TYPE;
                     RESULTS: in out RESULTS_LIST);
procedure GET_RESULTS(NODE: NODE_TYPE;
                     RESULTS: in out RESULTS_LIST;
                     STATUS: out PROCESS_STATUS);
procedure GET_RESULTS(NAME: NAME_STRING;
                     RESULTS: in out RESULTS_LIST;
                     STATUS: out PROCESS_STATUS);
procedure GET_RESULTS(NAME: NAME_STRING;
                     RESULTS: in out RESULTS_LIST);
procedure GET_PARAMETERS(PARAMETERS: in out PARAMETER_LIST);
procedure ABORT_PROCESS(NODE: NODE_TYPE;
                     RESULTS: RESULTS_STRING);
procedure ABORT_PROCESS(NAME: NAME_STRING;
                     RESULTS: RESULTS_STRING);
procedure ABORT_PROCESS(NODE: NODE_TYPE);
procedure ABORT_PROCESS(NAME: NAME_STRING);
procedure SUSPEND_PROCESS(NODE: NODE_TYPE);
procedure SUSPEND_PROCESS(NAME: NAME_STRING);
procedure RESUME_PROCESS(NODE: NODE_TYPE);
procedure RESUME_PROCESS(NAME: NAME_STRING);
function STATUS_OF_PROCESS(NODE: NODE_TYPE)

```

```

        return PROCESS_STATUS;
function STATUS_OF_PROCESS(NAME: NAME_STRING)
    return PROCESS_STATUS;
function HANDLES_OPEN(NODE: NODE_TYPE)
    return NATURAL;
function HANDLES_OPEN(NAME: NAME_STRING)
    return NATURAL;
function IO_UNITS(NODE: NODE_TYPE)
    return NATURAL;
function IO_UNITS(NAME: NAME_STRING)
    return NATURAL;
function START_TIME(NODE: NODE_TYPE)
    return TIME;
function START_TIME(NAME: NAME_STRING)
    return TIME;
function FINISH_TIME(NODE: NODE_TYPE)
    return TIME;
function FINISH_TIME(NAME: NAME_STRING)
    return TIME;
function MACHINE_TIME(NODE: NODE_TYPE)
    return DURATION;
function MACHINE_TIME(NAME: NAME_STRING)
    return DURATION;

end PROCESS_CONTROL;

package IO_DEFINITIONS is
    use NODE_DEFINITIONS;
    use LIST_UTILITIES;

    type FILE_TYPE is limited private;
    type FILE_MODE is
        (IN_FILE, INOUT_FILE, OUT_FILE,
         APPEND_FILE);
    type CHARACTER_ARRAY is array (CHARACTER) of BOOLEAN;
    type FUNCTION_KEY_DESCRIPTOR(LENGTH: POSITIVE) is private;
    type TAB_ENUMERATION is (HORIZONTAL, VERTICAL);
    type POSITION_TYPE is
        record
            ROW: NATURAL;
            COLUMN: NATURAL;
        end record;
    private
        type FILE_TYPE is (IMPLEMENTATION_DEFINED);
        -- should be defined by implementor
        type FUNCTION_KEY_DESCRIPTOR(LINK: POSITIVE) is
            record
                null: -- defined by implementor
            end record;

end IO_DEFINITIONS;

package IO_CONTROL is
    use IO_DEFINITIONS;
    use NODE_DEFINITIONS;
    use LIST_UTILITIES;

    procedure OPEN_FILE_MODE
        (FILE: FILE_TYPE;
         NODE: in out NODE_TYPE;
         INTENT: in INTENTION;
         TIME_LIMIT: in DURATION := NO_DELAY);
    procedure SYNCHRONIZE(FILE: FILE_TYPE);

```

```

procedure SET_LOG(FILE: FILE_TYPE;
                  LOG_FILE: FILE_TYPE);
procedure CLEAR_LOG(FILE: FILE_TYPE);
function LOGGING(FILE: FILE_TYPE)
    return BOOLEAN;
function GET_LOG(FILE: FILE_TYPE)
    return FILE_TYPE;
function NUMBER_OF_ELEMENTS(FILE: FILE_TYPE)
    return NATURAL;
procedure SET_PROMPT(TERMINAL: FILE_TYPE;
                    PROMPT: STRING);
function GET_PROMPT(TERMINAL: FILE_TYPE)
    return STRING;
function INTERCEPTED_CHARACTERS(TERMINAL: FILE_TYPE)
    return CHARACTER_ARRAY;
procedure ENABLE_FUNCTION_KEYS(TERMINAL: FILE_TYPE;
                              ENABLE: BOOLEAN);
function FUNCTION_KEYS_ENABLED(TERMINAL: FILE_TYPE)
    return BOOLEAN;
procedure COUPLE(Queue_BASE: NODE_TYPE;
                Queue_KEY: RELATIONSHIP_KEY := LATEST_KEY;
                Queue_RELATION: RELATION_NAME := DEFAULT_RELATION;
                FILE_NODE: NODE_TYPE;
                FORM: LIST_TYPE := EMPTY_LIST;
                ATTRIBUTES: LIST_TYPE;
                -- intentionally no default
                ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                LEVEL: LIST_TYPE := EMPTY_LIST);
procedure COUPLE(Queue_NAME: NAME_STRING;
                FILE_NODE: NODE_TYPE;
                FORM: LIST_TYPE := EMPTY_LIST;
                ATTRIBUTES: LIST_TYPE;
                ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                LEVEL: LIST_TYPE := EMPTY_LIST);
procedure COUPLE(Queue_BASE: NODE_TYPE;
                Queue_KEY: RELATIONSHIP_KEY := LATEST_KEY;
                Queue_RELATION: RELATION_NAME := DEFAULT_RELATION;
                FILE_NAME: NAME_STRING;
                FORM: LIST_TYPE := EMPTY_LIST;
                ATTRIBUTES: LIST_TYPE;
                ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                LEVEL: LIST_TYPE := EMPTY_LIST);
procedure COUPLE(Queue_NAME: NAME_STRING;
                FILE_NAME: NAME_STRING;
                FORM: LIST_TYPE := EMPTY_LIST;
                ATTRIBUTES: LIST_TYPE;
                ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                LEVEL: LIST_TYPE := EMPTY_LIST);

end IO_CONTROL;

generic
    type ELEMENT_TYPE is private;
package DIRECT_IO is
    use NODE_DEFINITIONS;
    use LIST_UTILITIES;
    use IO_DEFINITIONS;
    subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;
    subtype FILE_NODE is CAIS.IO_DEFINITIONS.FILE_NODE;

    type COUNT is range 0 .. INTEGER'LAST;
    subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

```

```
-- File management

procedure CREATE(FILE:          in out FILE_TYPE;
                 BASE:          MODE_TYPE;
                 KEY:           RELATIONSHIP_KEY := LATEST_KEY;
                 RELATION:      RELATION_NAME :=
                                DEFAULT_RELATION;
                 MODE:          FILE_MODE := INOUT_FILE;
                 FORM:          LIST_TYPE := EMPTY_LIST;
                 ATTRIBUTES:    LIST_TYPE := EMPTY_LIST;
                 ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                 LEVEL:         LIST_TYPE := EMPTY_LIST);

procedure CREATE(FILE:          in out FILE_TYPE;
                 NAME:          NAME_STRING;
                 MODE:          FILE_MODE := INOUT_FILE;
                 FORM:          LIST_TYPE := EMPTY_LIST;
                 ATTRIBUTES:    LIST_TYPE := EMPTY_LIST;
                 ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                 LEVEL:         LIST_TYPE := EMPTY_LIST);

procedure OPEN(FILE: in out FILE_TYPE;
               MODE:  FILE_MODE);

procedure OPEN(FILE: in out FILE_TYPE;
               NAME:  NAME_STRING;
               MODE:  FILE_MODE);

procedure CLOSE(FILE: in out FILE_TYPE);
procedure DELETE(FILE: in out FILE_TYPE);
procedure RESET(FILE: in out FILE_TYPE;
               MODE:  FILE_MODE);
procedure RESET(FILE: in out FILE_TYPE);

function MODE(FILE: FILE_TYPE) return FILE_MODE;
function NAME(FILE: FILE_TYPE) return STRING;
function FORM(FILE: FILE_TYPE) return STRING;

function IS_OPEN(FILE: FILE_TYPE) return BOOLEAN;
-- Input and output operations

procedure READ(FILE: FILE_TYPE;
               ITEM: out ELEMENT_TYPE;
               FROM: POSITIVE_COUNT);
procedure READ(FILE: FILE_TYPE;
               ITEM: out ELEMENT_TYPE);

procedure WRITE(FILE: FILE_TYPE;
               ITEM: ELEMENT_TYPE;
               TO: POSITIVE_COUNT);
procedure WRITE(FILE: FILE_TYPE;
               ITEM: ELEMENT_TYPE);

procedure SET_INDEX(FILE: FILE_TYPE;
                   TO: POSITIVE_COUNT);

function INDEX(FILE: FILE_TYPE) return POSITIVE_COUNT;
function SIZE(FILE: FILE_TYPE) return COUNT;

function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN;

end DIRECT_IO;

generic
  type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is
```

31 JANUARY 1985

```

use NODE_DEFINITIONS;
use LIST_UTILITIES;
use IO_DEFINITIONS;
subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;
subtype FILE_MODE is CAIS.IO_DEFINITIONS.FILE_MODE;

-- File management

procedure CREATE(FILE:          in out FILE_TYPE;
                 BASE:          NODE_TYPE;
                 KEY:            RELATIONSHIP_KEY := LATEST_KEY;
                 RELATION:       RELATION_NAME := DEFAULT_RELATION;
                 MODE:           FILE_MODE := INOUT_FILE;
                 FORM:           LIST_TYPE := EMPTY_LIST;
                 ATTRIBUTES:     LIST_TYPE := EMPTY_LIST;
                 ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                 LEVEL:          LIST_TYPE := EMPTY_LIST);

procedure CREATE(FILE:          in out FILE_TYPE;
                 NAME:          NAME_STRING;
                 MODE:           FILE_MODE := INOUT_FILE;
                 FORM:           LIST_TYPE := EMPTY_LIST;
                 ATTRIBUTES:     LIST_TYPE := EMPTY_LIST;
                 ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                 LEVEL:          LIST_TYPE := EMPTY_LIST);

procedure OPEN(FILE: in out FILE_TYPE;
               NODE:  NODE_TYPE;
               MODE:  FILE_MODE);

procedure OPEN(FILE: in out FILE_TYPE;
               NAME:  NAME_STRING;
               MODE:  FILE_MODE);

procedure CLOSE(FILE: in out FILE_TYPE);
procedure DELETE(FILE: in out FILE_TYPE);
procedure RESET(FILE: in out FILE_TYPE;
               MODE:  FILE_MODE);
procedure RESET(FILE: in out FILE_TYPE);

function MODE(FILE: FILE_TYPE) return FILE_MODE;
function NAME(FILE: FILE_TYPE) return STRING;
function FORM(FILE: FILE_TYPE) return STRING;

function IS_OPEN(FILE: FILE_TYPE) return BOOLEAN;

-- Input and output operations

procedure READ(FILE:  FILE_TYPE;
               ITEM:  out ELEMENT_TYPE);
procedure WRITE(FILE: FILE_TYPE;
               ITEM:  ELEMENT_TYPE);

function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN;
end SEQUENTIAL_IO;

package TEXT_IO is
use NODE_DEFINITIONS;
use LIST_UTILITIES;
use IO_DEFINITIONS;
subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;
subtype FILE_MODE is CAIS.IO_DEFINITIONS.FILE_MODE;

type COUNT is range 0 .. INTEGER'LAST;

```

```

subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

UNBOUNDED : constant COUNT := 0; -- line and page length

subtype FIELD      is INTEGER range 0 .. INTEGER'LAST;
subtype NUMBER_BASE is INTEGER range 2 .. 16;

type TYPE_SET is (LOWER_CASE, UPPER_CASE);

-- File Management

procedure CREATE(FILE      in out FILE_TYPE;
                  BASE:     NODE_TYPE;
                  KEY:       RELATIONSHIP_KEY := LATEST_KEY;
                  RELATION:  RELATION_NAME := DEFAULT_RELATION;
                  MODE:      FILE_MODE := INOUT_FILE;
                  FORM:      LIST_TYPE := EMPTY_LIST;
                  ATTRIBUTES: LIST_TYPE := EMPTY_LIST;
                  ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                  LEVEL:     LIST_TYPE := EMPTY_LIST);

procedure CREATE(FILE:      in out FILE_TYPE;
                  NAME:      NAME_STRING;
                  MODE:      FILE_MODE := INOUT_FILE;
                  FORM:      LIST_TYPE := EMPTY_LIST;
                  ATTRIBUTES: LIST_TYPE := EMPTY_LIST;
                  ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                  LEVEL:     LIST_TYPE := EMPTY_LIST);

procedure OPEN(FILE: in out FILE_TYPE;
                MODE:  FILE_MODE);

procedure OPEN(FILE: in out FILE_TYPE;
                NAME:  NAME_STRING;
                MODE:  FILE_MODE);

procedure CLOSE(FILE: in out FILE_TYPE);
procedure DELETE(FILE: in out FILE_TYPE);
procedure RESET(FILE: in out FILE_TYPE;
                MODE:  FILE_MODE);
procedure RESET(FILE: in out FILE_TYPE);

function MODE(FILE: FILE_TYPE) return FILE_MODE;
function NAME(FILE: FILE_TYPE) return STRING;
function FORM(FILE: FILE_TYPE) return STRING;

function IS_OPEN(FILE: FILE_TYPE) return BOOLEAN;

-- Control of default input and output files

procedure SET_INPUT(FILE: FILE_TYPE);
procedure SET_OUTPUT(FILE: FILE_TYPE);
procedure SET_ERROR(FILE: FILE_TYPE);

function STANDARD_INPUT return FILE_TYPE;
function STANDARD_OUTPUT return FILE_TYPE;
function STANDARD_ERROR return FILE_TYPE;

function CURRENT_INPUT return FILE_TYPE;
function CURRENT_OUTPUT return FILE_TYPE;
function CURRENT_ERROR return FILE_TYPE;

```


-- Specification of line and page lengths

```
procedure SET_LINE_LENGTH(FILE: FILE_TYPE;  
                           TO: COUNT);  
procedure SET_LINE_LENGTH(TO: COUNT);  
  
procedure SET_PAGE_LENGTH(FILE: FILE_TYPE;  
                           TO: COUNT);  
procedure SET_PAGE_LENGTH(TO: COUNT);  
  
function LINE_LENGTH(FILE: FILE_TYPE) return COUNT;  
function LINE_LENGTH return COUNT;  
  
function PAGE_LENGTH(FILE: FILE_TYPE) return COUNT;  
function PAGE_LENGTH return COUNT;
```

-- Column, Line and Page Control

```
procedure NEW_LINE(FILE: FILE_TYPE;  
                   SPACING: POSITIVE_COUNT := 1);  
procedure NEW_LINE(SPACING: POSITIVE_COUNT := 1);  
  
procedure SKIP_LINE(FILE: FILE_TYPE;  
                   SPACING: POSITIVE_COUNT := 1);  
procedure SKIP_LINE(SPACING: POSITIVE_COUNT := 1);  
  
function END_OF_LINE(FILE: FILE_TYPE) return BOOLEAN;  
function END_OF_LINE return BOOLEAN;  
  
procedure NEW_PAGE(FILE: FILE_TYPE);  
procedure NEW_PAGE;  
  
procedure SKIP_PAGE(FILE: FILE_TYPE);  
procedure SKIP_PAGE;  
  
function END_OF_PAGE(FILE: FILE_TYPE) return BOOLEAN;  
function END_OF_PAGE return BOOLEAN;  
  
function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN;  
function END_OF_FILE return BOOLEAN;  
  
  
procedure SET_COL(FILE: FILE_TYPE;  
                  TO: POSITIVE_COUNT);  
procedure SET_COL(TO: POSITIVE_COUNT);  
  
procedure SET_LINE(FILE: FILE_TYPE;  
                   TO: POSITIVE_COUNT);  
procedure SET_LINE(TO: POSITIVE_COUNT);  
  
  
function COL(FILE: FILE_TYPE) return POSITIVE_COUNT;  
function COL return POSITIVE_COUNT;  
  
function LINE(FILE: FILE_TYPE) return POSITIVE_COUNT;  
function LINE return POSITIVE_COUNT;  
  
function PAGE(FILE: FILE_TYPE) return POSITIVE_COUNT;  
function PAGE return POSITIVE_COUNT;  
  
  
-- Character Input-Output  
  
procedure GET(FILE: FILE_TYPE;  
             ITEM: out CHARACTER);
```

```

procedure GET(ITEM: out CHARACTER);
procedure PUT(FILE: FILE_TYPE; ITEM : CHARACTER);
procedure PUT(ITEM: CHARACTER);

-- String Input-Output

procedure GET(FILE: FILE_TYPE; ITEM : out STRING);
procedure GET(ITEM: out STRING);
procedure PUT(FILE: FILE_TYPE; ITEM : STRING);
procedure PUT(ITEM: STRING);

procedure GET_LINE(FILE: FILE_TYPE;
                    ITEM: out STRING;
                    LAST: out NATURAL);
procedure GET_LINE(ITEM: out STRING;
                    LAST: out NATURAL);

procedure PUT_LINE(FILE: FILE_TYPE; ITEM: STRING);
procedure PUT_LINE(ITEM: STRING);

-- generic package for Input-Output of Integer Types

generic
  type NUM is range <>;
package INTEGER_IO is

  DEFAULT_WIDTH: FIELD := NUM'WIDTH;
  DEFAULT_BASE:  NUMBER_BASE := 10;

  procedure GET(FILE: FILE_TYPE;
                ITEM: out NUM;
                WIDTH: FIELD := 0);
  procedure GET(ITEM: out NUM;
                WIDTH: FIELD := 0);

  procedure PUT(FILE: FILE_TYPE;
                ITEM: NUM;
                WIDTH: FIELD := DEFAULT_WIDTH;
                BASE: NUMBER_BASE := DEFAULT_BASE);
  procedure PUT(ITEM: NUM;
                WIDTH: FIELD := DEFAULT_WIDTH;
                BASE: NUMBER_BASE := DEFAULT_BASE);

  procedure GET(FROM: STRING;
                ITEM: out NUM;
                LAST: out POSITIVE);
  procedure PUT(TO: out STRING;
                ITEM: NUM;
                BASE: NUMBER_BASE := DEFAULT_BASE);

end INTEGER_IO;

-- generic package for Input-Output of Floating Point
-- Types

generic
  type NUM is digits <>;
package FLOAT_IO is

  DEFAULT_FORE: FIELD := 2;
  DEFAULT_AFT:  FIELD := NUM'DIGITS - 1;

```

```

DEFAULT_EXP:  FIELD := 3;

procedure GET(FILE:  FILE_TYPE;
              ITEM:  out NUM;
              WIDTH:  FIELD := 0);
procedure GET(ITEM:  out NUM;
              WIDTH:  FIELD := 0);

procedure PUT(FILE:  FILE_TYPE;
              ITEM:  NUM;
              FORE:  FIELD := DEFAULT_FORE;
              AFT:  FIELD := DEFAULT_AFT;
              EXP:  FIELD := DEFAULT_EXP);
procedure PUT(ITEM:  NUM;
              FORE:  FIELD := DEFAULT_FORE;
              AFT:  FIELD := DEFAULT_AFT;
              EXP:  FIELD := DEFAULT_EXP);

procedure GET(FROM:  STRING;
              ITEM:  out NUM;
              LAST:  out POSITIVE);
procedure PUT(TO:  out STRING;
              ITEM:  NUM;
              AFT:  FIELD := DEFAULT_AFT;
              EXP:  FIELD := DEFAULT_EXP);

end FLOAT_IO;

-- generic package for Input-Output of Fixed Point Types
generic
  type NUM is delta <>;
package FIXED_IO is

  DEFAULT_FORE:  FIELD := NUM_FORE;
  DEFAULT_AFT:  FIELD := NUM_AFT;
  DEFAULT_EXP:  FIELD := 0;

  procedure GET(FILE:  FILE_TYPE;
                ITEM:  out NUM;
                WIDTH:  FIELD := 0);
  procedure GET(ITEM:  out NUM;
                WIDTH:  FIELD := 0);

  procedure PUT(FILE:  FILE_TYPE;
                ITEM:  NUM;
                FORE:  FIELD := DEFAULT_FORE;
                AFT:  FIELD := DEFAULT_AFT;
                EXP:  FIELD := DEFAULT_EXP);
  procedure PUT(ITEM:  NUM;
                FORE:  FIELD := DEFAULT_FORE;
                AFT:  FIELD := DEFAULT_AFT;
                EXP:  FIELD := DEFAULT_EXP);

  procedure GET(FROM:  STRING;
                ITEM:  out NUM;
                LAST:  out POSITIVE);
  procedure PUT(TO:  out STRING;
                ITEM:  NUM;
                AFT:  FIELD := DEFAULT_AFT;
                EXP:  FIELD := DEFAULT_EXP);

end FIXED_IO;

```

```
-- generic package for Input-Output of Enumeration Types

generic
  type ENUM is (<>);
package ENUMERATION_IO is

  DEFAULT_WIDTH:  FIELD := 0;
  DEFAULT_SETTING: TYPE_SET := UPPER_CASE;

  procedure GET(FILE:  FILE_TYPE; ITEM : out ENUM);
  procedure GET(ITEM:  out ENUM);

  procedure PUT(FILE:  FILE_TYPE;
                ITEM:  ENUM;
                WIDTH:  FIELD := DEFAULT_WIDTH;
                SET:    TYPE_SET := DEFAULT_SETTING);
  procedure PUT(ITEM:  ENUM;
                WIDTH:  FIELD := DEFAULT_WIDTH;
                SET:    TYPE_SET := DEFAULT_SETTING);

  procedure GET(FROM:  STRING;
                ITEM:  out ENUM;
                LAST:  out POSITIVE);
  procedure PUT(TO:    out STRING;
                ITEM:  ENUM;
                SET:    TYPE_SET := DEFAULT_SETTING);
end ENUMERATION_IO;

end TEXT_IO;

package SCROLL_TERMINAL is
  use MODE_DEFINITIONS;
  use IO_DEFINITIONS;
  use IO_CONTROL;
  subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;
  subtype FUNCTION_KEY_DESCRIPTOR is
    CAIS.IO_DEFINITIONS.FUNCTION_KEY_DESCRIPTOR;
  subtype POSITION_TYPE is CAIS.IO_DEFINITIONS.POSITION_TYPE;
  subtype TAB_ENUMERATION is CAIS.IO_DEFINITIONS.TAB_ENUMERATION;

  procedure SET_POSITION(TERMINAL:  FILE_TYPE;
                        POSITION:  POSITION_TYPE);
  procedure SET_POSITION(POSITION:  POSITION_TYPE);
  function GET_POSITION(TERMINAL:  FILE_TYPE)
    return POSITION_TYPE;
  function GET_POSITION return POSITION_TYPE;
  function TERMINAL_SIZE(TERMINAL:  FILE_TYPE)
    return POSITION_TYPE;
  function TERMINAL_SIZE return POSITION_TYPE;
  procedure SET_TAB(TERMINAL:  FILE_TYPE;
                    KIND:  TAB_ENUMERATION := HORIZONTAL);
  procedure SET_TAB(KIND:  TAB_ENUMERATION := HORIZONTAL);
  procedure CLEAR_TAB(TERMINAL:  FILE_TYPE;
                      KIND:  TAB_ENUMERATION := HORIZONTAL);
  procedure CLEAR_TAB(KIND:  TAB_ENUMERATION := HORIZONTAL);
  procedure TAB(TERMINAL:  FILE_TYPE;
                KIND:  TAB_ENUMERATION := HORIZONTAL;
                COUNT:  POSITIVE := 1);
  procedure TAB(KIND:  TAB_ENUMERATION := HORIZONTAL;
                COUNT:  POSITIVE := 1);
  procedure BELL(TERMINAL:  FILE_TYPE);
  procedure BELL;
  procedure PUT(TERMINAL:  FILE_TYPE;
                ITEM:  CHARACTER);
```

```

procedure PUT(ITEM: CHARACTER);
procedure PUT(TERMINAL: FILE_TYPE;
              ITEM: STRING);
procedure PUT(ITEM: STRING);
procedure SET_ECHO(TERMINAL: FILE_TYPE;
                  TO: BOOLEAN := TRUE);
procedure SET_ECHO(TO: BOOLEAN := TRUE);
function ECHO(TERMINAL: FILE_TYPE)
              return BOOLEAN;
function ECHO return BOOLEAN;
function MAXIMUM_FUNCTION_KEY(TERMINAL: FILE_TYPE)
              return NATURAL;
function MAXIMUM_FUNCTION_KEY return NATURAL;
procedure GET(TERMINAL: FILE_TYPE;
              ITEM: out CHARACTER;
              KEYS: out FUNCTION_KEY_DESCRIPTOR);
procedure GET(ITEM: out CHARACTER;
              KEYS: out FUNCTION_KEY_DESCRIPTOR);
procedure GET(TERMINAL: FILE_TYPE;
              ITEM: out STRING;
              LAST: out NATURAL;
              KEYS: out FUNCTION_KEY_DESCRIPTOR);
procedure GET(ITEM: out STRING;
              LAST: out NATURAL;
              KEYS: out FUNCTION_KEY_DESCRIPTOR);
function FUNCTION_KEY_COUNT(KEYS: FUNCTION_KEY_DESCRIPTOR)
              return NATURAL;
procedure FUNCTION_KEY(KEYS: FUNCTION_KEY_DESCRIPTOR;
                      INDEX: POSITIVE;
                      KEY_IDENTIFIER: out POSITIVE;
                      POSITION: out NATURAL);
procedure FUNCTION_KEY_NAME(TERMINAL: FILE_TYPE;
                           KEY_IDENTIFIER: POSITIVE;
                           KEY_NAME: out STRING;
                           LAST: out POSITIVE);
procedure FUNCTION_KEY_NAME(KEY_IDENTIFIER: POSITIVE;
                           KEY_NAME: out STRING;
                           LAST: out POSITIVE);
procedure NEW_LINE(TERMINAL: FILE_TYPE;
                  COUNT: POSITIVE := 1);
procedure NEW_LINE(COUNT: POSITIVE := 1);
procedure NEW_PAGE(TERMINAL: FILE_TYPE);
procedure NEW_PAGE;

end SCROLL_TERMINAL;

package PAGE_TERMINAL is
  use NODE_DEFINITIONS;
  use IO_DEFINITIONS;
  use IO_CONTROL;
  subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;
  subtype FUNCTION_KEY_DESCRIPTOR is
    CAIS.IO_DEFINITIONS.FUNCTION_KEY_DESCRIPTOR;
  subtype POSITION_TYPE is
    CAIS.IO_DEFINITIONS.POSITION_TYPE;
  subtype TAB_ENUMERATION is
    CAIS.IO_DEFINITIONS.TAB_ENUMERATION;

  type SELECT_ENUMERATION is
    (FROM_ACTIVE_POSITION TO END,
     FROM_START_TO_ACTIVE_POSITION,
     ALL_POSITIONS);
  type GRAPHIC_RENDITION_ENUMERATION is
    (PRIMARY_RENDITION, BOLD,

```

```

    FAINT,                UNDERSCORE,
    SLOW_BLINK,           RAPID_BLINK,
    REVERSE_IMAGE);
type GRAPHIC_RENDITION_ARRAY is array
    (GRAPHIC_RENDITION_ENUMERATION) of BOOLEAN;

DEFAULT_GRAPHIC_RENDITION: constant GRAPHIC_RENDITION_ARRAY :=
    (PRIMARY_RENDITION =>
        TRUE,
        BOLD .. REVERSE_IMAGE =>
        FALSE);

procedure SET_POSITION(TERMINAL: FILE_TYPE;
    POSITION: POSITION_TYPE);
procedure SET_POSITION(POSITION: POSITION_TYPE);
function GET_POSITION
    (TERMINAL: FILE_TYPE) return POSITION_TYPE;
function GET_POSITION return POSITION_TYPE;
function TERMINAL_SIZE
    (TERMINAL: FILE_TYPE) return POSITION_TYPE;
function TERMINAL_SIZE return POSITION_TYPE;
procedure SET_TAB
    (TERMINAL: FILE_TYPE;
    KIND: TAB_ENUMERATION := HORIZONTAL);
procedure SET_TAB
    (KIND: TAB_ENUMERATION := HORIZONTAL);
procedure CLEAR_TAB
    (TERMINAL: FILE_TYPE;
    KIND: TAB_ENUMERATION := HORIZONTAL);
procedure CLEAR_TAB
    (KIND: TAB_ENUMERATION := HORIZONTAL);
procedure TAB
    (TERMINAL: FILE_TYPE;
    KIND: TAB_ENUMERATION := HORIZONTAL;
    COUNT: POSITIVE := 1);
procedure TAB
    (KIND: TAB_ENUMERATION := HORIZONTAL;
    COUNT: POSITIVE := 1);
procedure BELL
    (TERMINAL: FILE_TYPE);
procedure BELL;
procedure PUT(TERMINAL: FILE_TYPE;
    ITEM: CHARACTER);
procedure PUT(ITEM: CHARACTER);
procedure PUT(TERMINAL: FILE_TYPE;
    ITEM: STRING);
procedure PUT(ITEM: STRING);
procedure SET_ECHO(TERMINAL: FILE_TYPE;
    TO: BOOLEAN := TRUE);
procedure SET_ECHO(TO: BOOLEAN := TRUE);
function ECHO(TERMINAL: FILE_TYPE) return BOOLEAN;
function ECHO return BOOLEAN;
function MAXIMUM_FUNCTION_KEYS
    (TERMINAL: FILE_TYPE) return NATURAL;
function MAXIMUM_FUNCTION_KEYS return NATURAL;
procedure GET(TERMINAL: FILE_TYPE;
    ITEM: out CHARACTER;
    KEYS: out FUNCTION_KEY_DESCRIPTOR);
procedure GET(ITEM: out CHARACTER;
    KEYS: out FUNCTION_KEY_DESCRIPTOR);
procedure GET(TERMINAL: FILE_TYPE;
    ITEM: out STRING;
    LAST: out NATURAL;
    KEYS: out FUNCTION_KEY_DESCRIPTOR);

```

```

CLOSE (NODE);
raise;
end NODE_ATTRIBUTE_ITERATE;

```

```

procedure PATH_ATTRIBUTE_ITERATE
  (ITERATOR: out ATTRIBUTE_ITERATOR;
   BASE:      NODE_TYPE;
   KEY:        RELATIONSHIP_KEY;
   RELATION:   RELATION_NAME := DEFAULT_RELATION;
   PATTERN:   ATTRIBUTE_PATTERN := "**") is separate;

```

```

procedure PATH_ATTRIBUTE_ITERATE
  (ITERATOR: out ATTRIBUTE_ITERATOR;
   NAME:      NAME_STRING;
   PATTERN:   ATTRIBUTE_PATTERN := "**")
is
  BASE : NODE_TYPE;
begin
  OPEN (BASE, BASE_PATH (NAME), (1 => READ_RELATIONSHIPS));
  PATH_ATTRIBUTE_ITERATE
    (ITERATOR, BASE, LAST_KEY (NAME), LAST_RELATION (NAME), PATTERN);
  CLOSE (BASE);
exception
  when others =>
    CLOSE (BASE);
    raise;
end PATH_ATTRIBUTE_ITERATE;

```

```

function MORE(ITERATOR: ATTRIBUTE_ITERATOR)
  return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end MORE;

```

```

procedure GET_NEXT(ITERATOR: in out ATTRIBUTE_ITERATOR;
  ATTRIBUTE: out ATTRIBUTE_NAME;
  VALUE: in out LIST_TYPE)
is
begin
  null; -- should be defined by implementor
end GET_NEXT;

```

end ATTRIBUTES;

```

separate (CAIS)
package body ACCESS_CONTROL is
  use NODE_DEFINITIONS;
  use NODE_MANAGEMENT;

```

```

procedure SET_ACCESS_CONTROL(NODE:      NODE_TYPE;
  ROLE_NODE: NODE_TYPE;
  GRANT:      GRANT_VALUE) is separate;

```

```

procedure SET_ACCESS_CONTROL(NAME:      NAME_STRING;
  ROLE_NAME: NAME_STRING;
  GRANT:      GRANT_VALUE)

```

```

is
  NODE, ROLE_NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (1 => CONTROL));

```

```

procedure GET_NODE_ATTRIBUTE
  (NODE:      NODE_TYPE;
   ATTRIBUTE: ATTRIBUTE_NAME;
   VALUE:     in out LIST_TYPE) is separate;

procedure GET_NODE_ATTRIBUTE
  NAME:      NAME_STRING;
  ATTRIBUTE: ATTRIBUTE_NAME;
  VALUE:     in out LIST_TYPE)
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));
  GET_NODE_ATTRIBUTE (NODE, ATTRIBUTE, VALUE);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (NODE);
    raise;
end GET_NODE_ATTRIBUTE;

procedure GET_PATH_ATTRIBUTE
  BASE:      NODE_TYPE;
  KEY:       RELATIONSHIP_KEY;
  RELATION:  RELATION_NAME := DEFAULT_RELATION;
  ATTRIBUTE: ATTRIBUTE_NAME;
  VALUE:     in out LIST_TYPE) is separate;

procedure GET_PATH_ATTRIBUTE
  NAME:      NAME_STRING;
  ATTRIBUTE: ATTRIBUTE_NAME;
  VALUE:     in out LIST_TYPE)
is
  BASE : NODE_TYPE;
begin
  OPEN (BASE, BASE_PATH (NAME), (1 => READ_RELATIONSHIPS));
  GET_PATH_ATTRIBUTE
    (BASE, LAST_KEY (NAME), LAST_RELATION (NAME),
     ATTRIBUTE, VALUE);
  CLOSE (BASE);
exception
  when others =>
    CLOSE (BASE);
    raise;
end GET_PATH_ATTRIBUTE;

procedure NODE_ATTRIBUTE_ITERATE
  (ITERATOR: out ATTRIBUTE_ITERATOR;
   NODE:     NODE_TYPE;
   PATTERN:  ATTRIBUTE_PATTERN := "**") is separate;

procedure NODE_ATTRIBUTE_ITERATE
  ITERATOR: out ATTRIBUTE_ITERATOR;
  NAME:     NAME_STRING;
  PATTERN:  ATTRIBUTE_PATTERN := "**")
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));
  NODE_ATTRIBUTE_ITERATE (ITERATOR, NODE, PATTERN);
  CLOSE (NODE);
exception
  when others =>

```



```

procedure DELETE_PATH_ATTRIBUTE(BASE:      NODE_TYPE;
                                KEY:        RELATIONSHIP_KEY;
                                RELATION:    RELATION_NAME := DEFAULT_RELATION;
                                ATTRIBUTE:   ATTRIBUTE_NAME) is separate;

```

```

procedure DELETE_PATH_ATTRIBUTE(NAME:      NAME_STRING;
                                ATTRIBUTE:  ATTRIBUTE_NAME)

```

```

is
BASE : NODE_TYPE;
begin
OPEN (BASE, BASE_PATH (NAME), (1 => WRITE_RELATIONSHIPS));
DELETE_PATH_ATTRIBUTE
  (BASE, LAST_KEY (NAME), LAST_RELATION (NAME), ATTRIBUTE);
CLOSE (BASE);
exception
when others =>
  CLOSE (BASE);
  raise;
end DELETE_PATH_ATTRIBUTE;

```

```

procedure SET_NODE_ATTRIBUTE(NODE:      NODE_TYPE;
                              ATTRIBUTE:  ATTRIBUTE_NAME;
                              VALUE:      LIST_TYPE) is separate;

```

```

procedure SET_NODE_ATTRIBUTE(NAME:      NAME_STRING;
                              ATTRIBUTE:  ATTRIBUTE_NAME;
                              VALUE:      LIST_TYPE)

```

```

is
NODE : NODE_TYPE;
begin
OPEN (NODE NAME, (1 => WRITE_ATTRIBUTES));
SET_NODE_ATTRIBUTE (NODE, ATTRIBUTE, VALUE);
CLOSE (NODE);
exception
when others =>
  CLOSE (NODE);
  raise;
end SET_NODE_ATTRIBUTE;

```

```

procedure SET_PATH_ATTRIBUTE
(BASE:      NODE_TYPE;
 KEY:        RELATIONSHIP_KEY;
 RELATION:    RELATION_NAME := DEFAULT_RELATION;
 ATTRIBUTE:   ATTRIBUTE_NAME;
 VALUE:      LIST_TYPE) is separate;

```

```

procedure SET_PATH_ATTRIBUTE(NAME:      NAME_STRING;
                              ATTRIBUTE:  ATTRIBUTE_NAME;
                              VALUE:      LIST_TYPE)

```

```

is
BASE : NODE_TYPE;
begin
OPEN (BASE, BASE_PATH (NAME), (1 => WRITE_RELATIONSHIPS));
SET_PATH_ATTRIBUTE
  (BASE, LAST_KEY (NAME), LAST_RELATION (NAME), ATTRIBUTE, VALUE);
CLOSE (BASE);
exception
when others =>
  CLOSE (BASE);
  raise;
end SET_PATH_ATTRIBUTE;

```

```
use NODE_MANAGEMENT;

procedure CREATE_NODE_ATTRIBUTE
  (NODE:      NODE_TYPE;
   ATTRIBUTE: ATTRIBUTE_NAME;
   VALUE:     LIST_TYPE) is separate;

procedure CREATE_NODE_ATTRIBUTE
  (NAME:      NAME_STRING;
   ATTRIBUTE: ATTRIBUTE_NAME;
   VALUE:     LIST_TYPE)
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (1 => APPEND_ATTRIBUTES));
  CREATE_NODE_ATTRIBUTE (NODE, ATTRIBUTE, VALUE);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (NODE);
    raise;
end CREATE_NODE_ATTRIBUTE;

procedure CREATE_PATH_ATTRIBUTE(BASE:      NODE_TYPE;
                                KEY:        RELATIONSHIP_KEY;
                                RELATION:    RELATION_NAME := DEFAULT_RELATION;
                                ATTRIBUTE:   ATTRIBUTE_NAME;
                                VALUE:       LIST_TYPE) is separate;

procedure CREATE_PATH_ATTRIBUTE(NAME:      NAME_STRING;
                                ATTRIBUTE:  ATTRIBUTE_NAME;
                                VALUE:      LIST_TYPE)
is
  BASE : NODE_TYPE;
begin
  OPEN (BASE, BASE_PATH (NAME), (1 => WRITE_ATTRIBUTES));
  CREATE_PATH_ATTRIBUTE
    (BASE, LAST_KEY (NAME), LAST_RELATION (NAME), ATTRIBUTE, VALUE);
  CLOSE (BASE);
exception
  when others =>
    CLOSE (BASE);
    raise;
end CREATE_PATH_ATTRIBUTE;

procedure DELETE_NODE_ATTRIBUTE(NODE:      NODE_TYPE;
                                ATTRIBUTE:  ATTRIBUTE_NAME) is separate;

procedure DELETE_NODE_ATTRIBUTE(NAME:      NAME_STRING;
                                ATTRIBUTE:  ATTRIBUTE_NAME)
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (1 => WRITE_ATTRIBUTES));
  DELETE_NODE_ATTRIBUTE (NODE, ATTRIBUTE);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (NODE);
    raise;
end DELETE_NODE_ATTRIBUTE;
```

```

procedure ITERATE
  (ITERATOR: out NODE_ITERATOR;
   NAME: NAME_STRING;
   KIND: NODE_KIND;
   KEY: RELATIONSHIP_KEY_PATTERN := "";
   RELATION: RELATION_NAME_PATTERN :=
             DEFAULT_RELATION;
   PRIMARY_ONLY: BOOLEAN := TRUE)
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (1 => READ_RELATIONSHIPS));
  ITERATE (ITERATOR, NODE, KIND, KEY, RELATION,
           PRIMARY_ONLY);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (NODE);
    raise;
end ITERATE;

function MORE (ITERATOR: NODE_ITERATOR) return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end MORE;

procedure GET_NEXT
  (ITERATOR: in out NODE_ITERATOR;
   NEXT_NODE: in out NODE_TYPE;
   INTENT: INTENTION := (1 => EXISTENCE);
   TIME_LIMIT: DURATION := NO_DELAY)
is
begin
  null; -- should be defined by implementor
end GET_NEXT;

procedure SET_CURRENT_NODE (NODE: NODE_TYPE) is separate;

procedure SET_CURRENT_NODE (NAME: NAME_STRING)
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (1 => EXISTENCE));
  SET_CURRENT_NODE (NODE);
exception
  when others =>
    CLOSE (NODE);
    raise;
end SET_CURRENT_NODE;

procedure GET_CURRENT_NODE
  (NODE: in out NODE_TYPE;
   INTENT: INTENTION := (1 => EXISTENCE);
   TIME_LIMIT: DURATION := NO_DELAY) is separate;

end NODE_MANAGEMENT;

separate (CAIS)
package body ATTRIBUTES is
  use NODE_DEFINITIONS;

```

```

procedure DELETE_TREE(NAME: NAME_STRING)
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (EXCLUSIVE_WRITE, READ_RELATIONSHIPS));
  DELETE_TREE (NODE);
exception
  when others =>
    CLOSE (NODE);
    raise;
end DELETE_TREE;

procedure LINK(NODE:      NODE_TYPE;
               NEW_BASE:   NODE_TYPE;
               NEW_KEY:    RELATIONSHIP_KEY;
               NEW_RELATION: RELATION_NAME :=
                           DEFAULT_RELATION) is separate;

procedure LINK(NODE: NODE_TYPE;
               NEW_NAME: NAME_STRING)
is
  NEW_BASE : NODE_TYPE;
begin
  OPEN (NEW_BASE, BASE_PATH (NEW_NAME), (1 => APPEND_RELATIONSHIPS));
  LINK (NODE, NEW_BASE, LAST_KEY (NEW_NAME),
        LAST_RELATION (NEW_NAME));
  CLOSE (NEW_BASE);
exception
  when others =>
    CLOSE (NEW_BASE);
    raise;
end LINK;

procedure UNLINK(BASE:      NODE_TYPE;
                 KEY:        RELATIONSHIP_KEY;
                 RELATION:   RELATION_NAME :=
                             DEFAULT_RELATION) is separate;

procedure UNLINK(NAME: NAME_STRING)
is
  BASE : NODE_TYPE;
begin
  OPEN (BASE, BASE_PATH (NAME),
        (1 => WRITE_RELATIONSHIPS));
  UNLINK (BASE, LAST_KEY (NAME), LAST_RELATION (NAME));
  CLOSE (BASE);
exception
  when others =>
    CLOSE (BASE);
    raise;
end UNLINK;

procedure ITERATE
  (ITERATOR:      out NODE_ITERATOR;
   NODE:          NODE_TYPE;
   KIND:          NODE_KIND;
   KEY:           RELATIONSHIP_KEY_PATTERN := "";
   RELATION:      RELATION_NAME_PATTERN :=
                   DEFAULT_RELATION;
   PRIMARY_ONLY:  BOOLEAN := TRUE) is separate;

```

end COPY_NODE;

```

procedure COPY_TREE(FROM:      NODE_TYPE;
                    TO_BASE:    NODE_TYPE;
                    TO_KEY:     RELATIONSHIP_KEY;
                    TO_RELATION: RELATION_NAME :=
                                DEFAULT_RELATION) is separate;

```

```

procedure COPY_TREE(FROM: NODE_TYPE;
                    TO:     NAME_STRING)
is
    TO_BASE : NODE_TYPE;
begin
    OPEN (TO_BASE, BASE_PATH (TO), (1 => APPEND_RELATIONSHIPS));
    COPY_TREE
        (FROM, TO_BASE, LAST_KEY (TO), LAST_RELATION (TO));
    CLOSE (TO_BASE);
exception
    when others =>
        CLOSE (TO_BASE);
        raise;
end COPY_TREE;

```

```

procedure RENAME(NODE:      NODE_TYPE;
                 NEW_BASE:   NODE_TYPE;
                 NEW_KEY:    RELATIONSHIP_KEY;
                 NEW_RELATION: RELATION_NAME :=
                             DEFAULT_RELATION) is separate;

```

```

procedure RENAME(NODE:      NODE_TYPE;
                 NEW_NAME:   NAME_STRING)
is
    NEW_BASE : NODE_TYPE;
begin
    OPEN (NEW_BASE, BASE_PATH (NEW_NAME), (1 => APPEND_RELATIONSHIPS));
    RENAME
        (NODE, NEW_BASE, LAST_KEY (NEW_NAME),
         LAST_RELATION (NEW_NAME));
    CLOSE (NEW_BASE);
exception
    when others =>
        CLOSE (NEW_BASE);
        raise;
end RENAME;

```

```

procedure DELETE_NODE(NODE: in out NODE_TYPE) is separate;

```

```

procedure DELETE_NODE(NAME: NAME_STRING)
is

```

```

    NODE : NODE_TYPE;
begin
    OPEN (NODE, NAME, (EXCLUSIVE_WRITE, READ_RELATIONSHIPS));
    DELETE_NODE (NODE);
exception
    when others =>
        CLOSE (NODE);
        raise;
end DELETE_NODE;

```

```

procedure DELETE_TREE(NODE: in out NODE_TYPE) is separate;

```

31 JANUARY 1985

```

        return BOOLEAN
is
    NODE : NODE_TYPE;
    RESULT : BOOLEAN;
begin
    OPEN (NODE, BASE, KEY, RELATION, (1 => EXISTENCE));
    RESULT := IS_OBTAINABLE (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others => return FALSE;
end IS_OBTAINABLE;

function IS_SAME(NODE1:  NODE_TYPE;
                  NODE2:  NODE_TYPE) return BOOLEAN is separate;

function IS_SAME(NAME1:  NAME_STRING;
                  NAME2:  NAME_STRING) return BOOLEAN
is
    NODE1, NODE2 : NODE_TYPE;
    RESULT : BOOLEAN;
begin
    OPEN (NODE1, NAME1, (1 => EXISTENCE));

begin
    OPEN (NODE2, NAME2, (1 => EXISTENCE));
exception
    when others =>
        CLOSE (NODE1);
        raise;
end;

    RESULT := IS_SAME(NODE1, NODE2);
    CLOSE (NODE1);
    CLOSE (NODE2);
    return RESULT;
end IS_SAME;

procedure GET_PARENT
(PARENT:      in out NODE_TYPE;
 NODE:        NODE_TYPE;
 INTENT:      INTENTION := (1 => READ);
 TIME_LIMIT:  DURATION := NO_DELAY) is separate;

procedure COPY_NODE
(FROM:        NODE_TYPE;
 TO_BASE:     NODE_TYPE;
 TO_KEY:      RELATIONSHIP_KEY;
 TO_RELATION: RELATION_NAME := DEFAULT_RELATION) is separate;

procedure COPY_NODE(FROM:  NODE_TYPE;
                    TO:     NAME_STRING)
is
    TO_BASE : NODE_TYPE;
begin
    OPEN (TO_BASE, BASE_PATH (TO), (1 => APPEND_RELATIONSHIPS));
    COPY_NODE
        (FROM, TO_BASE, LAST_KEY (TO), LAST_RELATION (TO));
    CLOSE (TO_BASE);
exception
    when others =>
        CLOSE (TO_BASE);
        raise;

```

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

procedure CHANGE_INTENT
  (NODE: in out NODE_TYPE;
   INTENT: INTENTION;
   TIME_LIMIT: DURATION := NO_DELAY) is separate;

function IS_OPEN(NODE: NODE_TYPE) return BOOLEAN
is
  RESULT: BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end IS_OPEN;

function INTENT_OF(NODE: NODE_TYPE)
  return INTENTION is separate;
function KIND(NODE: NODE_TYPE) return NODE_KIND is separate;

function PRIMARY_NAME(NODE: NODE_TYPE)
  return NAME_STRING is separate;

function PRIMARY_KEY(NODE: NODE_TYPE)
  return RELATIONSHIP_KEY is separate;

function PRIMARY_RELATION(NODE: NODE_TYPE)
  return RELATION_NAME is separate;

function PATH_KEY(NODE: NODE_TYPE)
  return RELATIONSHIP_KEY is separate;

function PATH_RELATION(NODE: NODE_TYPE)
  return RELATION_NAME is separate;

function BASE_PATH(NAME: NAME_STRING)
  return NAME_STRING is separate;

function LAST_RELATION(NAME: NAME_STRING)
  return RELATION_NAME is separate;

function LAST_KEY(NAME: NAME_STRING)
  return RELATIONSHIP_KEY is separate;

function IS_OBTAINABLE(NODE: NODE_TYPE)
  return BOOLEAN is separate;

function IS_OBTAINABLE(NAME: NAME_STRING) return BOOLEAN
is
  NODE: NODE_TYPE;
  RESULT: BOOLEAN;
begin
  OPEN (NODE, NAME, (1 => EXISTENCE));
  RESULT := IS_OBTAINABLE (NODE);
  CLOSE (NODE);
  return RESULT;
exception
  when others => return FALSE;
end IS_OBTAINABLE;

function IS_OBTAINABLE
  (BASE: NODE_TYPE;
   KEY: RELATIONSHIP_KEY;
   RELATION: RELATION_NAME := DEFAULT_RELATION)
```

Appendix C CAIS Body

```
with CALENDAR;
package body CAIS is

    package body NODE_MANAGEMENT is separate;
    package body ATTRIBUTES is separate;
    package body ACCESS_CONTROL is separate;
    package body STRUCTURAL_NODES is separate;
    package body PROCESS_CONTROL is separate;
    package body DIRECT_IO is separate;
    package body SEQUENTIAL_IO is separate;
    package body TEXT_IO is separate;
    package body IO_CONTROL is separate;
    package body SCROLL_TERMINAL is separate;
    package body PAGE_TERMINAL is separate;
    package body FORM_TERMINAL is separate;
    package body MAGNETIC_TAPE is separate;
    package body FILE_IMPORT_EXPORT is separate;
    package body LIST_UTILITIES is separate;
end CAIS;

with CALENDAR;
separate (CAIS)

package body NODE_MANAGEMENT is
    use NODE_DEFINITIONS;
    use CALENDAR;

    procedure OPEN(NODE:          in out NODE_TYPE;
                   NAME:          NAME_STRING;
                   INTENT:        INTENTION := (1 => READ)
                   TIME_LIMIT:    DURATION := NO_DELAY)
    is
    begin
        null; -- should be defined by implementor
    end OPEN;

    procedure OPEN(NODE:          in out NODE_TYPE;
                   BASE:          NODE_TYPE;
                   KEY:            RELATIONSHIP_KEY;
                   RELATION:       RELATION_NAME := DEFAULT_RELATION;
                   INTENT:        INTENTION := (1 => READ)
                   TIME_LIMIT:    DURATION := NO_DELAY)
    is
    begin
        null; -- should be defined by implementor
    end OPEN;
```


PROPOSED MIL-STD-CAIS
31 JANUARY 1985

```
                                HOST_FILE_NAME: STRING);  
procedure EXPORT(NODE:         NODE_TYPE;  
                                HOST_FILE_NAME: STRING);  
procedure EXPORT(NAME:         NAME_STRING;  
                                HOST_FILE_NAME: STRING);  
  
    end FILE_IMPORT_EXPORT;  
  
end CAIS;
```

```

use IO_DEFINITIONS;

type TAPE_POSITION is
  (BEGINNING_OF_TAPE,    PHYSICAL_END_OF_TAPE,
   TAPE_MARK,            OTHER);

subtype VOLUME_STRING is STRING (1 .. 6);
subtype FILE_STRING   is STRING (1 .. 17);
subtype REEL_NAME     is STRING;
subtype FILE_TYPE     is CAIS.IO_DEFINITIONS.FILE_TYPE;
subtype LABEL_STRING  is STRING (1...80);

procedure MOUNT(TAPE_DRIVE: FILE_TYPE;
                TAPE_NAME: REEL_NAME;
                DENSITY: POSITIVE);
procedure LOAD_UNLABELED(TAPE_DRIVE: FILE_TYPE;
                        DENSITY: POSITIVE;
                        BLOCK_SIZE: POSITIVE);
procedure INITIALIZE_UNLABELED(TAPE_DRIVE: FILE_TYPE;
                              DENSITY: POSITIVE;
                              BLOCKSIZE: POSITIVE);
procedure LOAD_LABELED(TAPE_DRIVE: FILE_TYPE;
                      VOLUME_IDENTIFIER: VOLUME_STRING;
                      DENSITY: POSITIVE;
                      BLOCK_SIZE: POSITIVE);
procedure INITIALIZE_LABELED(TAPE_DRIVE: FILE_TYPE;
                            VOLUME_IDENTIFIER: VOLUME_STRING;
                            DENSITY: POSITIVE;
                            BLOCK_SIZE: POSITIVE;
                            ACCESSIBILITY: CHARACTER := ' ');
procedure UNLOAD(TAPE_DRIVE: FILE_TYPE);
procedure DISMOUNT(TAPE_DRIVE: FILE_TYPE);
function IS_LOADED(TAPE_DRIVE: FILE_TYPE)
  return BOOLEAN;
function IS_MOUNTED(TAPE_DRIVE: FILE_TYPE)
  return BOOLEAN;
function TAPE_STATUS(TAPE_DRIVE: FILE_TYPE)
  return TAPE_POSITION;
procedure REWIND_TAPE(TAPE_DRIVE: FILE_TYPE);
procedure SKIP_TAPE_MARKS(TAPE_DRIVE: FILE_TYPE;
                          NUMBER: INTEGER := 1;
                          TAPE_STATE: out TAPE_POSITION);
procedure WRITE_TAPE_MARK(TAPE_DRIVE: FILE_TYPE;
                          NUMBER: POSITIVE := 1;
                          TAPE_STATE: out TAPE_POSITION);
procedure VOLUME_HEADER(TAPE_DRIVE: FILE_TYPE;
                        VOLUME_IDENTIFIER: VOLUME_STRING;
                        ACCESSIBILITY: CHARACTER := ' ');
procedure FILE_HEADER(TAPE_DRIVE: FILE_TYPE;
                     FILE_IDENTIFIER: FILE_STRING;
                     EXPIRATION_DATE: STRING := " 99366";
                     ACCESSIBILITY: CHARACTER := ' ');
procedure END_FILE_LABEL(TAPE_DRIVE: FILE_TYPE);
procedure READ_LABEL(TAPE_DRIVE: FILE_TYPE;
                    LABEL: out LABEL_STRING);

end MAGNETIC_TAPE;

package FILE_IMPORT_EXPORT is
  use NODE_DEFINITIONS;
  procedure IMPORT(NODE: NODE_TYPE;
                  HOST_FILE_NAME: STRING);
  procedure IMPORT(NAME: NAME_STRING);

```

```

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

type FORM_TYPE (ROW:                POSITIVE;
                COLUMN:             POSITIVE;
                AREA_QUALIFIER_REQUIRES_SPACE: BOOLEAN) is private;

subtype PRINTABLE_CHARACTER is CHARACTER range ' ' .. ' ';

function MAXIMUM_FUNCTION_KEY
    (TERMINAL: FILE_TYPE) return NATURAL;
function MAXIMUM_FUNCTION_KEY return NATURAL;
procedure DEFINE_QUALIFIED_AREA
    (FORM:      in out FORM_TYPE;
     INTENSITY: AREA_INTENSITY := NORMAL;
     PROTECTION: AREA_PROTECTION := PROTECTED;
     INPUT:      AREA_INPUT := GRAPHIC_CHARACTERS;
     VALUE:      AREA_VALUE := NO_FILL);
procedure REMOVE_AREA_QUALIFIER (FORM: in out FORM_TYPE);
procedure SET_POSITION (FORM:      in out FORM_TYPE;
                        POSITION:     POSITION_TYPE);
procedure NEXT_QUALIFIED_AREA (FORM: in out FORM_TYPE;
                              COUNT:  POSITIVE := 1);
procedure PUT (FORM: in out FORM_TYPE;
              ITEM:   PRINTABLE_CHARACTER);
procedure PUT (FORM: in out FORM_TYPE;
              ITEM:   STRING);
procedure ERASE_AREA (FORM: in out FORM_TYPE);
procedure ERASE_FORM (FORM: in out FORM_TYPE);
procedure ACTIVATE (TERMINAL: FILE_TYPE;
                   FORM:      in out FORM_TYPE);
procedure GET (FORM: in out FORM_TYPE;
              ITEM:   out PRINTABLE_CHARACTER);
procedure GET (FORM: in out FORM_TYPE;
              ITEM:   out STRING);
function IS_FORM_UPDATED (FORM: FORM_TYPE) return BOOLEAN;
function TERMINATION_KEY (FORM: FORM_TYPE) return NATURAL;
function FORM_SIZE (FORM: FORM_TYPE) return POSITION_TYPE;
function TERMINAL_SIZE (TERMINAL: FILE_TYPE) return POSITION_TYPE;
function TERMINAL_SIZE return POSITION_TYPE;
function AREA_QUALIFIER_REQUIRES_SPACE
    (FORM: FORM_TYPE) return BOOLEAN;
function AREA_QUALIFIER_REQUIRES_SPACE
    (TERMINAL: FILE_TYPE) return BOOLEAN;
function AREA_QUALIFIER_REQUIRES_SPACE return BOOLEAN;

private

type FORM_TYPE (ROW:                POSITIVE;
                COLUMN:             POSITIVE;
                AREA_QUALIFIER_REQUIRES_SPACE: BOOLEAN) is
    record
        null; -- should be defined by implementor
    end record;

end FORM_TERMINAL;

package MAGNETIC_TAPE is
    use MODE_DEFINITIONS;

```

```

procedure GET(ITEM: out STRING;
              LAST: out NATURAL;
              KEYS: out FUNCTION_KEY_DESCRIPTOR);
function FUNCTION_KEY_COUNT(KEYS: FUNCTION_KEY_DESCRIPTOR)
                           return NATURAL;
procedure FUNCTION_KEY(KEYS: FUNCTION_KEY_DESCRIPTOR;
                       INDEX: POSITIVE;
                       KEY_IDENTIFIER: out POSITIVE;
                       POSITION: out NATURAL);
procedure FUNCTION_KEY_NAME(TERMINAL: FILE_TYPE;
                            KEY_IDENTIFIER: POSITIVE;
                            KEY_NAME: out STRING;
                            LAST: out POSITIVE);
procedure FUNCTION_KEY_NAME(KEY_IDENTIFIER: POSITIVE;
                            KEY_NAME: out STRING;
                            LAST: out POSITIVE);
procedure DELETE_CHARACTER(TERMINAL: FILE_TYPE;
                           COUNT: POSITIVE := 1);
procedure DELETE_CHARACTER(COUNT: POSITIVE := 1);
procedure DELETE_LINE(TERMINAL: FILE_TYPE;
                       COUNT: POSITIVE := 1);
procedure DELETE_LINE(COUNT: POSITIVE := 1);
procedure ERASE_CHARACTER(TERMINAL: FILE_TYPE;
                           COUNT: POSITIVE := 1);
procedure ERASE_CHARACTER(COUNT: POSITIVE := 1);
procedure ERASE_IN_DISPLAY(TERMINAL: FILE_TYPE;
                            SELECTION: SELECT_ENUMERATION);
procedure ERASE_IN_DISPLAY(SELECTION: SELECT_ENUMERATION);
procedure ERASE_IN_LINE(TERMINAL: FILE_TYPE;
                         SELECTION: SELECT_ENUMERATION);
procedure ERASE_IN_LINE(SELECTION: SELECT_ENUMERATION);
procedure INSERT_SPACE(TERMINAL: FILE_TYPE;
                       COUNT: POSITIVE := 1);
procedure INSERT_SPACE(COUNT: POSITIVE := 1);
procedure INSERT_LINE(TERMINAL: FILE_TYPE;
                      COUNT: POSITIVE := 1);
procedure INSERT_LINE(COUNT: POSITIVE := 1);
function GRAPHIC_RENDITION_SUPPORT
  (TERMINAL: FILE_TYPE;
   RENDITION: GRAPHIC_RENDITION_ARRAY)
  return BOOLEAN;
function GRAPHIC_RENDITION_SUPPORT
  (RENDITION: GRAPHIC_RENDITION_ARRAY)
  return BOOLEAN;
procedure SELECT_GRAPHIC_RENDITION
  (TERMINAL: FILE_TYPE;
   RENDITION: GRAPHIC_RENDITION_ARRAY :=
    DEFAULT_GRAPHIC_RENDITION);
procedure SELECT_GRAPHIC_RENDITION
  (RENDITION: GRAPHIC_RENDITION_ARRAY :=
    DEFAULT_GRAPHIC_RENDITION);

end PAGE_TERMINAL;

package FORM_TERMINAL is
  use MODE_DEFINITIONS;
  use IO_DEFINITIONS;
  use IO_CONTROL;
  subtype FILE_TYPE is CAIS.IO_DEFINITIONS.FILE_TYPE;

  type AREA_INTENSITY is (NONE, NORMAL, HIGH);
  type AREA_PROTECTION is (UNPROTECTED, PROTECTED);
  type AREA_INPUT is
    (GRAPHIC_CHARACTERS, NUMERICS,

```

```

OPEN (OLE_NODE, ROLE_NAME, (1 => EXISTENCE));
SET ACCESS_CONTROL (NODE, ROLE_NODE, GRANT);
CLOSE NODE;
CLOSE ROLE_NODE);
except on
  when others =>
    CLOSE (NODE);
    CLOSE (ROLE_NODE);
    raise;
end SET_ACCESS_CONTROL;

function IS_GRANTED (OBJECT_NODE:  NODE_TYPE;
                     ACCESS_RIGHT:  NAME_STRING)
  return BOOLEAN is separate;

function IS_GRANTED (OBJECT_NAME:  NAME_STRING;
                     ACCESS_RIGHT:  NAME_STRING)
  return BOOLEAN
is
  OBJECT_NODE : NODE_TYPE;
  RESULT      : BOOLEAN;
begin
  OPEN (OBJECT_NODE, OBJECT_NAME, (1 => READ_RELATIONSHIPS));
  RESULT := IS_GRANTED (OBJECT_NODE, ACCESS_RIGHT);
  CLOSE (OBJECT_NODE);
  return RESULT;
exception
  when others =>
    CLOSE (OBJECT_NODE);
    raise;
end IS_GRANTED;

procedure ADOPT (ROLE_NODE:  NODE_TYPE;
                 ROLE_KEY:  RELATIONSHIP_KEY := LATEST_KEY) is separate;

procedure UNADOPT (ROLE_KEY:  RELATIONSHIP_KEY) is separate;
end ACCESS_CONTROL;

separate (CAIS)
package body STRUCTURAL_NODES is
  use NODE_DEFINITIONS;
  use NODE_MANAGEMENT;

  procedure CREATE_NODE
    (NODE:          in out NODE_TYPE;
     BASE:          NODE_TYPE;
     KEY:           RELATIONSHIP_KEY := LATEST_KEY;
     RELATION:      RELATION_NAME := DEFAULT_RELATION;
     ATTRIBUTES:    LIST_TYPE := EMPTY_LIST;
     ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
     LEVEL:         LIST_TYPE := EMPTY_LIST) is separate;

  procedure CREATE_NODE
    (NODE:          in out NODE_TYPE;
     NAME:          NAME_STRING;
     ATTRIBUTES:    LIST_TYPE := EMPTY_LIST;
     ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
     LEVEL:         LIST_TYPE := EMPTY_LIST)
  is
    BASE : NODE_TYPE;
  begin
    OPEN (BASE, BASE_PATH (NAME),
          (1 => APPEND_RELATIONSHIPS));

```

```

CREATE_NODE
(NODE, BASE, LAST_KEY (NAME), LAST_RELATION (NAME),
ATTRIBUTES, ACCESS_CONTROL, LEVEL);
CLOSE (BASE);
exception
when others =>
CLOSE (NODE);
CLOSE (BASE);
raise;
end CREATE_NODE;

procedure CREATE_NODE
(BASE:          NODE_TYPE;
KEY:            RELATIONSHIP_KEY := LATEST_KEY;
RELATION:       RELATION_NAME := DEFAULT_RELATION;
ATTRIBUTES:     LIST_TYPE := EMPTY_LIST;
ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
LEVEL:          LIST_TYPE := EMPTY_LIST)
is
NODE : NODE_TYPE;
begin
CREATE_NODE
(NODE, KEY, RELATION, ATTRIBUTES, ACCESS_CONTROL,
LEVEL);
CLOSE (NODE);
end CREATE_NODE;

procedure CREATE_NODE
(NAME:          NAME_STRING;
ATTRIBUTES:     LIST_TYPE := EMPTY_LIST;
ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
LEVEL:          LIST_TYPE := EMPTY_LIST)
is
NODE : NODE_TYPE;
begin
CREATE_NODE
(NODE, NAME, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
CLOSE (NODE);
end CREATE_NODE;
end STRUCTURAL_NODES;

separate (CAIS)
package body PROCESS_CONTROL is
use NODE_DEFINITIONS;
use PROCESS_DEFINITIONS;
use NODE_MANAGEMENT;

procedure SPAWN_PROCESS
(NODE:          in out NODE_TYPE;
FILE_NODE:     NODE_TYPE;
INPUT_PARAMETERS: PARAMETER_LIST := EMPTY_LIST;
KEY:            RELATIONSHIP_KEY := LATEST_KEY;
RELATION:       RELATION_NAME := DEFAULT_RELATION;
ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
LEVEL:          LIST_TYPE := EMPTY_LIST;
ATTRIBUTES:     LIST_TYPE := EMPTY_LIST;
INPUT_FILE:     NAME_STRING := CURRENT_INPUT;
OUTPUT_FILE:    NAME_STRING := CURRENT_OUTPUT;
ERROR_FILE:     NAME_STRING := CURRENT_ERROR;
ENVIRONMENT_NODE: NAME_STRING := CURRENT_NODE) is separate;

procedure AWAIT_PROCESS_COMPLETION
(NODE:          NODE_TYPE;

```

```

        TIME_LIMIT:      DURATION := DURATION'LAST)
                           is separate;

procedure AWAIT_PROCESS_COMPLETION
  (NODE:      NODE_TYPE;
   RESULTS_RETURNED: in out RESULTS_LIST;
   STATUS:      out PROCESS_STATUS;
   TIME_LIMIT:      DURATION := DURATION'LAST)
is
begin
  AWAIT_PROCESS_COMPLETION (NODE, TIME_LIMIT);
  GET_RESULTS (NODE, RESULTS_RETURNED);
  STATUS := STATUS_OF_PROCESS (NODE);
end AWAIT_PROCESS_COMPLETION;

procedure INVOKE_PROCESS
  (NODE:      in out NODE_TYPE;
   FILE_NODE:      NODE_TYPE;
   RESULTS_RETURNED: in out RESULTS_LIST;
   STATUS:      out PROCESS_STATUS;
   INPUT_PARAMETERS:      PARAMETER_LIST;
   KEY:      RELATIONSHIP_KEY := LATEST_KEY;
   RELATION:      RELATION_NAME := DEFAULT_RELATION;
   ACCESS_CONTROL:      LIST_TYPE := EMPTY_LIST;
   LEVEL:      LIST_TYPE := EMPTY_LIST;
   ATTRIBUTES:      LIST_TYPE := EMPTY_LIST;
   INPUT_FILE:      NAME_STRING := CURRENT_INPUT;
   OUTPUT_FILE:      NAME_STRING := CURRENT_OUTPUT;
   ERROR_FILE:      NAME_STRING := CURRENT_ERROR;
   ENVIRONMENT_NODE:      NAME_STRING := CURRENT_NODE;
   TIME_LIMIT:      DURATION :=
                           DURATION'LAST) is separate;

procedure CREATE_JOB
  (FILE_NODE:      NODE_TYPE;
   INPUT_PARAMETERS:      PARAMETER_LIST := EMPTY_LIST;
   KEY:      RELATIONSHIP_KEY := LATEST_KEY;
   ACCESS_CONTROL:      LIST_TYPE := EMPTY_LIST;
   LEVEL:      LIST_TYPE := EMPTY_LIST;
   ATTRIBUTES:      LIST_TYPE := EMPTY_LIST;
   INPUT_FILE:      NAME_STRING := CURRENT_INPUT;
   OUTPUT_FILE:      NAME_STRING := CURRENT_OUTPUT;
   ERROR_FILE:      NAME_STRING := CURRENT_ERROR;
   ENVIRONMENT_NODE:      NAME_STRING := CURRENT_USER)
                           is separate;

procedure APPEND_RESULTS(RESULTS: RESULTS_STRING)
                           is separate;

procedure WRITE_RESULTS(RESULTS: RESULTS_STRING) is separate;

procedure GET_RESULTS(NODE:      NODE_TYPE;
                     RESULTS: in out RESULTS_LIST)
                           is separate;

procedure GET_RESULTS(NODE:      NODE_TYPE;
                     RESULTS: in out RESULTS_LIST;
                     STATUS:      out PROCESS_STATUS)
is
begin
  GET_RESULTS (NODE, RESULTS);
  STATUS := STATUS_OF_PROCESS (NODE);
end GET_RESULTS;

```

```

procedure GET_RESULTS(NAME:          NAME_STRING;
                      RESULTS: in out RESULTS_LIST;
                      STATUS:      out PROCESS_STATUS)
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));
  GET_RESULTS (NODE, RESULTS);
  STATUS := STATUS_OF_PROCESS (NODE);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (NODE);
    raise;
end GET_RESULTS;

procedure GET_RESULTS(NAME:          NAME_STRING;
                      RESULTS: in out RESULTS_LIST)
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));
  GET_RESULTS (NODE, RESULTS);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (NODE);
    raise;
end GET_RESULTS;

procedure GET_PARAMETERS
  (PARAMETERS: in out PARAMETER_LIST) is separate;

procedure ABORT_PROCESS(NODE:      NODE_TYPE;
                       RESULTS: RESULTS_STRING) is separate;

procedure ABORT_PROCESS(NAME:      NAME_STRING;
                       RESULTS: RESULTS_STRING)
is
  NODE : NODE_TYPE;
begin
  OPEN (NODE, NAME, (READ_RELATIONSHIPS, WRITE_CONTENTS, WRITE_ATTRIBUTES));
  ABORT_PROCESS (NODE, RESULTS);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (NODE);
    raise;
end ABORT_PROCESS;

procedure ABORT_PROCESS(NODE: NODE_TYPE)
is
begin
  ABORT_PROCESS (NODE, 'ABORTED');
end ABORT_PROCESS;

procedure ABORT_PROCESS(NAME: NAME_STRING)
is
  NODE : NODE_TYPE;
begin

```



```
OPEN (NODE, NAME, (READ_RELATIONSHIPS,WRITE_CONTENTS,WRITE_ATTRIBUTES));  
ABORT_PROCESS (NODE, "ABORTED");  
CLOSE (NODE);  
exception  
  when others =>  
    CLOSE (NODE);  
    raise;  
end ABORT_PROCESS;
```

procedure SUSPEND_PROCESS(NODE: NODE_TYPE) is separate;

```
procedure SUSPEND_PROCESS(NAME: NAME_STRING)  
is  
  NODE : NODE_TYPE;  
begin  
  OPEN (NODE, NAME, (READ_RELATIONSHIPS,WRITE_CONTENTS,WRITE_ATTRIBUTES));  
  SUSPEND_PROCESS (NODE);  
  CLOSE (NODE);  
exception  
  when others =>  
    CLOSE (NODE);  
    raise;  
end SUSPEND_PROCESS;
```

procedure RESUME_PROCESS(NODE: NODE_TYPE) is separate;

```
procedure RESUME_PROCESS(NAME: NAME_STRING)  
is  
  NODE : NODE_TYPE;  
begin  
  OPEN (NODE, NAME, (READ_RELATIONSHIPS,WRITE_CONTENTS,WRITE_ATTRIBUTES));  
  RESUME_PROCESS (NODE);  
  CLOSE (NODE);  
exception  
  when others =>  
    CLOSE (NODE);  
    raise;  
end RESUME_PROCESS;
```

function STATUS_OF_PROCESS(NODE: NODE_TYPE)
 return PROCESS_STATUS is separate;

function STATUS_OF_PROCESS(NAME: NAME_STRING)
 return PROCESS_STATUS

```
is  
  NODE : NODE_TYPE;  
  RESULT : PROCESS_STATUS;  
begin  
  OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));  
  (RESULT := STATUS_OF_PROCESS (NODE);  
  CLOSE (NODE);  
  return RESULT;  
exception  
  when others =>  
    CLOSE (NODE);  
    raise;  
end STATUS_OF_PROCESS;
```

function HANDLES_OPEN(NODE: NODE_TYPE) return NATURAL
 is separate;

```
function HANDLES_OPEN(NAME: NAME_STRING) return NATURAL
is
  NODE   : NODE_TYPE;
  RESULT : NATURAL;
begin
  OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));
  RESULT := HANDLES_OPEN (NODE);
  CLOSE (NODE);
  return RESULT;
exception
  when others =>
    CLOSE (NODE);
    raise;
end HANDLES_OPEN;

function IO_UNITS(NODE: NODE_TYPE) return NATURAL is separate;

function IO_UNITS(NAME: NAME_STRING) return NATURAL
is
  NODE   : NODE_TYPE;
  RESULT : NATURAL;
begin
  OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));
  RESULT := IO_UNITS (NODE);
  CLOSE (NODE);
  return RESULT;
exception
  when others =>
    CLOSE (NODE);
    raise;
end IO_UNITS;

function START_TIME(NODE: NODE_TYPE)
  return TIME is separate;

function START_TIME(NAME: NAME_STRING)
  return TIME
is
  NODE   : NODE_TYPE;
  RESULT : TIME;
begin
  OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));
  RESULT := START_TIME (NODE);
  CLOSE (NODE);
  return RESULT;
exception
  when others =>
    CLOSE (NODE);
    raise;
end START_TIME;

function FINISH_TIME(NODE: NODE_TYPE)
  return TIME is separate;

function FINISH_TIME(NAME: NAME_STRING)
  return TIME
is
  NODE   : NODE_TYPE;
  RESULT : TIME;
begin
  OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));
```

```
RESULT := FINISH_TIME (NODE);  
CLOSE (NODE);  
return RESULT;  
exception  
when others =>  
    CLOSE (NODE);  
    raise;  
end FINISH_TIME;
```

```
function MACHINE_TIME(NODE:  NODE_TYPE) return DURATION  
is separate;
```

```
function MACHINE_TIME(NAME:  NAME_STRING) return DURATION  
is  
    NODE :  NODE_TYPE;  
    RESULT :  DURATION;  
begin  
    OPEN (NODE, NAME, (1 => READ_ATTRIBUTES));  
    RESULT := MACHINE_TIME (NODE);  
    CLOSE (NODE);  
    return RESULT;  
exception  
when others =>  
    CLOSE (NODE);  
    raise;  
end MACHINE_TIME;
```

```
end PROCESS_CONTROL;
```

```
separate (CAIS)  
package body IO_CONTROL is  
    use NODE_DEFINITIONS;  
    use NODE_MANAGEMENT;  
    use IO_DEFINITIONS;  
    use LIST_UTILITIES;
```

```
procedure OPEN_FILE_NODE(FILE:  FILE_TYPE;  
                           NODE:  in out NODE_TYPE;  
                           INTENT:  INTENTION;  
                           TIME_LIMIT:  DURATION := NO_DELAY)  
is separate;
```

```
procedure SYNCHRONIZE(FILE:  FILE_TYPE) is separate;
```

```
procedure SET_LOG(FILE:  FILE_TYPE;  
                  LOG_FILE:  FILE_TYPE) is separate;
```

```
procedure CLEAR_LOG(FILE:  FILE_TYPE) is separate;
```

```
function LOGGING(FILE:  FILE_TYPE) return BOOLEAN is separate;
```

```
function GET_LOG(FILE:  FILE_TYPE) return FILE_TYPE is separate;
```

```
function NUMBER_OF_ELEMENTS(FILE:  FILE_TYPE) return NATURAL  
is  
    RESULT : NATURAL;  
begin  
    -- sould be defined by implementor;  
    return RESULT;  
end NUMBER_OF_ELEMENTS;
```

```
procedure SET_PROMPT(TERMINAL:  FILE_TYPE;  
                     PROMPT:  STRING) is separate;
```

```

function GET_PROMPT(TERMINAL: FILE_TYPE) return STRING
    is separate;

function INTERCEPTED_CHARACTERS(TERMINAL: FILE_TYPE)
    return CHARACTER_ARRAY is separate;

procedure ENABLE_FUNCTION_KEYS(TERMINAL: FILE_TYPE;
                                ENABLE: BOOLEAN)
    is separate;

function FUNCTION_KEYS_ENABLED(TERMINAL: FILE_TYPE)
    return BOOLEAN is separate;

procedure COUPLE(Queue_Base:      Node_Type;
                 Queue_Key:      Relationship_Key := Latest_Key;
                 Queue_Relation: Relation_Name := Default_Relation;
                 File_Node:      Node_Type;
                 Form:           List_Type := Empty_List;
                 Attributes:      List_Type; -- intentionally
                                   -- not defaulted
                 Access_Control: List_Type := Empty_List;
                 Level:          List_Type := Empty_List) is separate;

procedure COUPLE(Queue_Name:      Name_String;
                 File_Node:      Node_Type;
                 Form:           List_Type := Empty_List;
                 Attributes:      List_Type;
                 Access_Control: List_Type := Empty_List;
                 Level:          List_Type := Empty_List)
is
    Base : Node_Type;
begin
    OPEN (Base, Base_Path (Queue_Name), (1 => Append_Relationships));
    COUPLE
        (Base, Last_Key (Queue_Name),
         Last_Relation (Queue_Name), File_Node, Form,
         Attributes, Access_Control, Level);
    CLOSE (Base);
exception
    when others =>
        CLOSE (Base);
        raise;
end COUPLE;

procedure COUPLE(Queue_Base:      Node_Type;
                 Queue_Key:      Relationship_Key := Latest_Key;
                 Queue_Relation: Relation_Name := Default_Relation;
                 File_Name:      Name_String;
                 Form:           List_Type := Empty_List;
                 Attributes:      List_Type;
                 Access_Control: List_Type := Empty_List;
                 Level:          List_Type := Empty_List)
is
    File_Node : Node_Type;
begin
    OPEN (File_Node, File_Name, (Read_Attributes, Read_Contents));
    COUPLE
        (Queue_Base, Queue_Key, Queue_Relation, File_Node,
         Form, Attributes, Access_Control, Level);
    CLOSE (File_Node);
exception
    when others =>
        CLOSE (File_Node);
        raise;

```

```

end COUPLE;

procedure COUPLE(Queue_Name:      NAME_STRING;
                 File_Name:      NAME_STRING;
                 Form:            LIST_TYPE := EMPTY_LIST;
                 Attributes:      LIST_TYPE;
                 Access_Control:  LIST_TYPE := EMPTY_LIST;
                 Level:           LIST_TYPE := EMPTY_LIST)
is
  File_Node : Node_Type;
  Queue_Base : Node_Type;
begin
  OPEN (Queue_Base, Base_Path (Queue_Name), (1 => APPEND_RELATIONSHIPS));
  OPEN (File_Node, File_Name, (READ_ATTRIBUTES, READ_CONTENTS));
  COUPLE
    (Queue_Base, Last_Key (Queue_Name),
     Last_Relation (Queue_Name), File_Node, Form,
     Attributes, Access_Control, Level);
  CLOSE (Queue_Base);
  CLOSE (File_Node);
exception
  when others =>
    CLOSE (Queue_Base);
    CLOSE (File_Node);
raise;
end COUPLE;

end IO_Control;

separate (CAIS)
package body DIRECT_IO is
  use NODE_DEFINITIONS;
  use IO_DEFINITIONS;
  use NODE_MANAGEMENT;

  -- File management

  procedure CREATE(File:           in out FILE_Type;
                   Base:           Node_Type;
                   Key:             Relationship_Key := Latest_Key;
                   Relation:        Relation_Name := Default_Relation;
                   Mode:            File_Mode := INOUT_File;
                   Form:            List_Type := EMPTY_LIST;
                   Attributes:      List_Type := EMPTY_LIST;
                   Access_Control:  List_Type := EMPTY_LIST;
                   Level:           List_Type := EMPTY_LIST)
  is
  begin
    null; -- should be defined by implementor
  end CREATE;

  procedure CREATE(File:           in out FILE_Type;
                   Name:            Name_String;
                   Mode:            File_Mode := INOUT_File;
                   Form:            List_Type := EMPTY_LIST;
                   Attributes:      List_Type := EMPTY_LIST;
                   Access_Control:  List_Type := EMPTY_LIST;
                   Level:           List_Type := EMPTY_LIST)
  is
    Base : Node_Type;

```

```
begin
  OPEN (BASE, BASE_PATH (NAME), (1 => APPEND_RELATIONSHIPS));
  CREATE (FILE, BASE, LAST_KEY (NAME),
    LAST_RELATION (NAME), MODE, FORM, ATTRIBUTES,
    ACCESS_CONTROL, LEVEL);
  CLOSE (BASE);
exception
  when others =>
    CLOSE (FILE);
    CLOSE (BASE);
    raise;
end CREATE;

procedure OPEN(FILE: in out FILE_TYPE;
  MODE:      NODE_TYPE;
  MODE:      FILE_MODE)
is
begin
  null; -- should be defined by implementor
end OPEN;

procedure OPEN(FILE: in out FILE_TYPE;
  NAME:      NAME_STRING;
  MODE:      FILE_MODE)
is
  NODE : NODE_TYPE;
begin
  case MODE is
    when IN_FILE      => OPEN (NODE, NAME, (1 => READ_CONTENTS));
    when OUT_FILE     => OPEN (NODE, NAME, (1 => WRITE_CONTENTS));
    when INOUT_FILE   => OPEN (NODE, NAME, (READ_CONTENTS, WRITE_CONTENTS));
    when APPEND_FILE  => raise USE_ERROR;
  end case;

  OPEN (FILE, NODE, MODE);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (FILE);
    CLOSE (NODE);
    raise;
end OPEN;

procedure CLOSE(FILE: in out FILE_TYPE)
is
begin
  null; -- should be defined by implementor
end CLOSE;

procedure DELETE(FILE: in out FILE_TYPE)
is
begin
  null; -- should be defined by implementor
end DELETE;

procedure RESET(FILE: in out FILE_TYPE;
  MODE:      FILE_MODE)
is
begin
  null; -- should be defined by implementor
end RESET;

procedure RESET(FILE: in out FILE_TYPE)
```

```
is
begin
  null; -- should be defined by implementor
end RESET;

function MODE(FILE: FILE_TYPE) return FILE_MODE
is
  RESULT : FILE_MODE;
begin
  -- should be defined by implementor
  return RESULT;
end MODE;

function NAME(FILE: FILE_TYPE) return STRING is separate;

function FORM(FILE: FILE_TYPE) return STRING
is
  RESULT : STRING( 1 .. 10);
begin
  -- should be defined by implementor
  return RESULT;
end FORM;

function IS_OPEN(FILE: FILE_TYPE) return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end IS_OPEN;

-- Input and output operations

procedure READ(FILE: FILE_TYPE;
               ITEM: out ELEMENT_TYPE;
               FROM: POSITIVE_COUNT)
is
begin
  null; -- should be defined by implementor
end READ;

procedure READ(FILE: FILE_TYPE;
               ITEM: out ELEMENT_TYPE)
is
begin
  null; -- should be defined by implementor
end READ;

procedure WRITE(FILE: FILE_TYPE;
                ITEM: ELEMENT_TYPE;
                TO: POSITIVE_COUNT)
is
begin
  null; -- should be defined by implementor
end WRITE;

procedure WRITE(FILE: FILE_TYPE;
                ITEM: ELEMENT_TYPE)
is
begin
  null; -- should be defined by implementor
end WRITE;

procedure SET_INDEX(FILE: FILE_TYPE;
```

```

                                TO:    POSITIVE_COUNT) is separate;

function INDEX(FILE:  FILE_TYPE) return POSITIVE_COUNT is separate;

function SIZE(FILE:  FILE_TYPE) return COUNT
                                is separate;

function END_OF_FILE(FILE:  FILE_TYPE) return BOOLEAN
is
    RESULT : BOOLEAN;
begin
    -- should be defined by implementor
    return RESULT;
end END_OF_FILE;
end DIRECT_IO;

separate (CAIS)
package body SEQUENTIAL_IO is
    use NODE_DEFINITIONS;
    use NODE_MANAGEMENT;
    use IO_DEFINITIONS;

    -- File management

    procedure CREATE(FILE:          in out FILE_TYPE;
                     BASE:          NODE_TYPE;
                     KEY:           RELATIONSHIP_KEY := LATEST_KEY;
                     RELATION:      RELATION_NAME := DEFAULT_RELATION;
                     MODE:          FILE_MODE := INOUT_FILE;
                     FORM:          LIST_TYPE := EMPTY_LIST;
                     ATTRIBUTES:    LIST_TYPE := EMPTY_LIST;
                     ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                     LEVEL:         LIST_TYPE := EMPTY_LIST)
    is
    begin
        null; -- should be defined by implementor
    end CREATE;

    procedure CREATE(FILE:          in out FILE_TYPE;
                     NAME:          NAME_STRING;
                     MODE:          FILE_MODE := INOUT_FILE;
                     FORM:          LIST_TYPE := EMPTY_LIST;
                     ATTRIBUTES:    LIST_TYPE := EMPTY_LIST;
                     ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                     LEVEL:         LIST_TYPE := EMPTY_LIST)
    is
        BASE : NODE_TYPE;
    begin
        OPEN (BASE, BASE_PATH (NAME), (1 => APPEND_RELATIONSHIPS));
        CREATE (FILE, BASE, LAST_KEY (NAME),
                LAST_RELATION (NAME), MODE, FORM, ATTRIBUTES,
                ACCESS_CONTROL, LEVEL);
        CLOSE (BASE);
    exception
        when others =>
            CLOSE (FILE);
            CLOSE (BASE);
            raise;
    end CREATE;

    procedure OPEN(FILE:  in out FILE_TYPE;

```



```

                                NODE:      NODE_TYPE;
                                MODE:      FILE_MODE)
is
begin
  null; -- should be defined by implementor
end OPEN;

procedure OPEN(FILE: in out FILE_TYPE;
               NAME:      NAME_STRING;
               MODE:      FILE_MODE)
is
  NODE : NODE_TYPE
begin
  case MODE is
    when IN_FILE => OPEN (NODE, NAME, (1 => READ_CONTENTS));
    when OUT_FILE => OPEN (NODE, NAME, (1 => WRITE_CONTENTS));
    when INOUT_FILE => OPEN (NODE, NAME, (READ_CONTENTS, WRITE_CONTENTS));
    when APPEND_FILE => OPEN (NODE, NAME, (1=> APPEND_CONTENTS));
  end case;

  OPEN (FILE, NODE, MODE);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (FILE);
    CLOSE (NODE);
    raise;
end OPEN;

procedure CLOSE(FILE: in out FILE_TYPE)
is
begin
  null; -- should be defined by implementor
end CLOSE;

procedure DELETE(FILE: in out FILE_TYPE)
is
begin
  null; -- should be defined by implementor
end DELETE;

procedure RESET(FILE: in out FILE_TYPE;
               MODE: FILE_MODE)
is
begin
  null; -- should be defined by implementor
end RESET;

procedure REPLACE(FILE: in out FILE_TYPE)
is
begin
  null; -- should be defined by implementor
end REPLACE;

function MODE(FILE: FILE_TYPE) return FILE_MODE
is
  RESULT : FILE_MODE;
begin
  -- should be defined by implementor
  return RESULT;
end MODE;

function NAME(FILE: FILE_TYPE) return STRING
```

31 JANUARY 1985

```

is
    RESULT : STRING(1..10);
begin
    -- should be defined by implementor
    return RESULT;
end NAME;

function FORM(FILE: FILE_TYPE) return STRING
is
    RESULT : STRING(1..10);
begin
    -- should be defined by implementor
    return RESULT;
end FORM;

function IS_OPEN(FILE: FILE_TYPE) return BOOLEAN
is
    RESULT : BOOLEAN;
begin
    -- should be defined by implementor
    return RESULT;
end IS_OPEN;

-- Input and output operations

procedure READ(FILE: FILE_TYPE;
               ITEM: out ELEMENT_TYPE) is separate;

procedure WRITE(FILE: FILE_TYPE; ITEM : ELEMENT_TYPE) is separate;

function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN
is
    RESULT : BOOLEAN;
begin
    -- should be defined by implementor
    return RESULT;
end END_OF_FILE;
end SEQUENTIAL_IO;

separate (CAIS)
package body TEXT_IO is
    use MODE_DEFINITIONS;
    use NODE_MANAGEMENT;
    use IO_DEFINITIONS;

    -- File Management

    procedure CREATE(FILE: in out FILE_TYPE;
                     BASE:  NODE_TYPE;
                     KEY:    RELATIONSHIP_KEY := LATEST_KEY;
                     RELATION: RELATION_NAME := DEFAULT_RELATION;
                     MODE:   FILE_MODE := INOUT_FILE;
                     FORM:   LIST_TYPE := EMPTY_LIST;
                     ATTRIBUTES: LIST_TYPE := EMPTY_LIST;
                     ACCESS_CONTROL: LIST_TYPE := EMPTY_LIST;
                     LEVEL:   LIST_TYPE := EMPTY_LIST)
                     is separate;

    procedure CREATE(FILE: in out FILE_TYPE;
                     NAME:  NAME_STRING;
                     MODE:  FILE_MODE := INOUT_FILE;
                     FORM:  LIST_TYPE := EMPTY_LIST;
                     ATTRIBUTES: LIST_TYPE := EMPTY_LIST;

```

AD-A157-589

MILITARY STANDARD COMMON APSE (ADA PROGRAMMING SUPPORT
ENVIRONMENT) INTERFACE SET (CAIS) (U) ADA JOINT PROGRAM
OFFICE ARLINGTON VA JAN 89

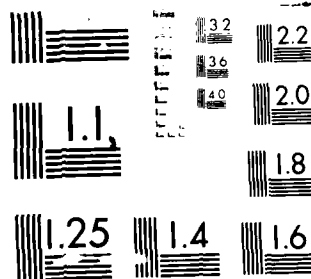
4/4

UNCLASSIFIED

F/G 9/2

ML

END
DATE
JUN 87
587



MICROCOPY RESOLUTION TEST CHART
 NATIONAL BUREAU OF STANDARDS-1963-A

31 JANUARY 1985

```

ACCESS_CONTROL:    LIST_TYPE := EMPTY_LIST;
LEVEL:             LIST_TYPE := EMPTY_LIST)

is
  BASE : NODE_TYPE;
begin
  OPEN (BASE, BASE_PATH (NAME), (1 => APPEND_RELATIONSHIPS));
  CREATE (FILE, BASE, LAST_KEY (NAME),
    LAST_RELATION (NAME), MODE, FORM, ATTRIBUTES,
    ACCESS_CONTROL, LEVEL);
  CLOSE (BASE);
exception
  when others =>
    CLOSE (FILE);
    CLOSE (BASE);
raise;
end CREATE;

procedure OPEN(FILE: in out FILE_TYPE;
  NODE:          NODE_TYPE;
  MODE:          FILE_MODE) is separate;

procedure OPEN(FILE: in out FILE_TYPE;
  NAME:          NAME_STRING;
  MODE:          FILE_MODE)
is
  NODE : NODE_TYPE;
begin
  case MODE is
    when IN_FILE => OPEN (NODE, NAME, (1 => READ_CONTENTS));

    when OUT_FILE => OPEN (NODE, NAME, (1 => WRITE_CONTENTS));

    when INOUT_FILE =>
      OPEN (NODE, NAME, (READ_CONTENTS, WRITE_CONTENTS));

    when APPEND_FILE => OPEN (NODE, NAME, (1 => APPEND_CONTENTS));
  end case;

  OPEN (FILE, NODE, MODE);
  CLOSE (NODE);
exception
  when others =>
    CLOSE (FILE);
    CLOSE (NODE);
raise;
end OPEN;

procedure CLOSE(FILE: in out FILE_TYPE)
is
begin
  null; -- should be defined by implementor
end CLOSE;

procedure DELETE(FILE: in out FILE_TYPE) is separate;

procedure RESET(FILE: in out FILE_TYPE;
  MODE:          FILE_MODE)
is
begin
  null; -- should be defined by implementor
end RESET;

procedure RESET(FILE: in out FILE_TYPE)
is

```

```
begin
  null; -- should be defined by implementor
end RESET;

function MODE(FILE: FILE_TYPE) return FILE_MODE is separate;

function NAME(FILE: FILE_TYPE) return STRING
is
  RESULT: STRING(1..10);
begin
  --should be defined by implementor
  return RESULT;
end NAME;

function FORM(FILE: FILE_TYPE) return STRING is separate;

function IS_OPEN(FILE: FILE_TYPE) return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end IS_OPEN;

-- Control of default input and output files

procedure SET_INPUT(FILE: FILE_TYPE) is separate;
procedure SET_OUTPUT(FILE: FILE_TYPE) is separate;
procedure SET_ERROR(FILE: FILE_TYPE) is separate;
function STANDARD_INPUT return FILE_TYPE is separate;
function STANDARD_OUTPUT return FILE_TYPE is separate;
function STANDARD_ERROR return FILE_TYPE is separate;
function CURRENT_INPUT return FILE_TYPE is separate;
function CURRENT_OUTPUT return FILE_TYPE is separate;
function CURRENT_ERROR return FILE_TYPE is separate;

-- Specification of line and page lengths

procedure SET_LINE_LENGTH(FILE: FILE_TYPE;
                           TO: COUNT)
is
begin
  null; -- should be defined by implementor
end SET_LINE_LENGTH;

procedure SET_LINE_LENGTH(TO: COUNT)
is
begin
  null; -- should be defined by implementor
end SET_LINE_LENGTH;

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

```
is
begin
  null; -- should be defined by implementor
end SET_PAGE_LENGTH;

procedure SET_PAGE_LENGTH(TO: COUNT)
is
begin
  null; -- should be defined by implementor
end SET_PAGE_LENGTH;

function LINE_LENGTH(FILE: FILE_TYPE) return COUNT
is
  RESULT : COUNT;
begin
  -- should be defined by implementor
  return LINE_LENGTH;
end LINE_LENGTH;

function LINE_LENGTH return COUNT
is
  RESULT : COUNT;
begin
  -- should be defined by implementor
  return RESULT;
end LINE_LENGTH;

function PAGE_LENGTH (FILE: FILE_TYPE) return COUNT
is
  RESULT : COUNT;
begin
  -- should be defined by implementor
  return RESULT;
end PAGE_LENGTH;

function PAGE_LENGTH return COUNT
is
  RESULT : COUNT;
begin
  -- should be defined by implementor
  return RESULT;
end PAGE_LENGTH;

-- Column, Line and Page Control

procedure NEW_LINE(FILE: FILE_TYPE;
                   SPACING: POSITIVE_COUNT := 1)
is
begin
  null; -- should be defined by implementor
end NEW_LINE;

procedure NEW_LINE(SPACING: POSITIVE_COUNT := 1)
is
begin
  null; -- should be defined by implementor
end NEW_LINE;

procedure SKIP_LINE(FILE: FILE_TYPE;
                   SPACING: POSITIVE_COUNT := 1)
is
begin
  null; -- should be defined by implementor
```

```
end SKIP_LINE;

procedure SKIP_LINE(SPACING: POSITIVE_COUNT := 1)
is
begin
  null; -- should be defined by implementor
end SKIP_LINE;

function END_OF_LINE(FILE: FILE_TYPE) return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end END_OF_LINE;

function END_OF_LINE return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end END_OF_LINE;

procedure NEW_PAGE(FILE: FILE_TYPE)
is
begin
  null; -- should be defined by implementor
end NEW_PAGE;

procedure NEW_PAGE
is
begin
  null; -- should be defined by implementor
end NEW_PAGE;

procedure SKIP_PAGE(FILE: FILE_TYPE)
is
begin
  null; -- should be defined by implementor
end SKIP_PAGE;

procedure SKIP_PAGE
is
begin
  null; -- should be defined by implementor
end SKIP_PAGE;

function END_OF_PAGE(FILE: FILE_TYPE) return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end END_OF_PAGE;

function END_OF_PAGE return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end END_OF_PAGE;
```



```
function END_OF_FILE(FILE: FILE_TYPE) return BOOLEAN
is
    RESULT : BOOLEAN;
begin
    -- should be defined by implementor
    return RESULT;
end END_OF_FILE;
```

```
function END_OF_FILE return BOOLEAN
is
    RESULT : BOOLEAN;
begin
    -- should be defined by implementor
    return END_OF_FILE;
end END_OF_FILE;
```

```
procedure SET_COL(FILE: FILE_TYPE;
                  TO: POSITIVE_COUNT)
is
begin
    null; -- should be defined by implementor
end SET_COL;
```

```
procedure SET_COL(TO: POSITIVE_COUNT)
is
begin
    null; -- should be defined by implementor
end SET_COL;
```

```
procedure SET_LINE(FILE: FILE_TYPE;
                  TO: POSITIVE_COUNT)
is
begin
    null; -- should be defined by implementor
end SET_LINE;
```

```
procedure SET_LINE(TO: POSITIVE_COUNT)
is
begin
    null; -- should be defined by implementor
end SET_LINE;
```

```
function COL(FILE: FILE_TYPE) return POSITIVE_COUNT
is
    RESULT : POSITIVE_COUNT;
begin
    -- should be defined by implementor
    return RESULT;
end COL;
```

```
function COL return POSITIVE_COUNT
is
    RESULT : POSITIVE_COUNT;
begin
    -- should be defined by implementor
    return RESULT;
end COL;
```

```
function LINE (FILE: FILE_TYPE) return POSITIVE_COUNT
is
    RESULT : POSITIVE_COUNT;
begin
```

```
-- should be defined by implementor
return RESULT;
end LINE;

function LINE return POSITIVE_COUNT
is
    RESULT : POSITIVE_COUNT;
begin
    -- should be defined by implementor
    return RESULT;
end LINE;

function PAGE(FILE: FILE_TYPE) return POSITIVE_COUNT
is
    RESULT : POSITIVE_COUNT;
begin
    -- should be defined by implementor
    return RESULT;
end PAGE;

function PAGE return POSITIVE_COUNT
is
    RESULT : POSITIVE_COUNT;
begin
    -- should be defined by implementor
    return RESULT;
    PAGE;
end PAGE;
```

-- Character Input-Output

```
procedure GET(FILE: FILE_TYPE;
              ITEM: out CHARACTER)
is
begin
    null; -- should be defined by implementor
end GET;

procedure GET(ITEM: out CHARACTER)
is
begin
    null; -- should be defined by implementor
end GET;

procedure PUT(FILE: FILE_TYPE;
              ITEM: CHARACTER)
is
begin
    null; -- should be defined by implementor
end PUT;

procedure PUT(ITEM: CHARACTER)
is
begin
    null; -- should be defined by implementor
end PUT;
```

-- String Input-Output

```
procedure GET(FILE: FILE_TYPE;
```

```
ITEM: out STRING)
is
begin
  null; -- should be defined by implementor
end GET;

procedure GET_ITEM: out STRING)
is
begin
  null; -- should be defined by implementor
end GET;

procedure PUT(FILE: FILE_TYPE;
ITEM: STRING)
is
begin
  null; -- should be defined by implementor
end PUT;

procedure PUT_ITEM: STRING)
is
begin
  null; -- should be defined by implementor
end PUT;

procedure GET_LINE(FILE: FILE_TYPE;
ITEM: out STRING;
LAST: out NATURAL)
is
begin
  null; -- should be defined by implementor
end GET_LINE;

procedure GET_LINE_ITEM: out STRING;
LAST: out NATURAL)
is
begin
  null; -- should be defined by implementor
end GET_LINE;

procedure PUT_LINE(FILE: FILE_TYPE; ITEM: STRING)
is
begin
  null; -- should be defined by implementor
end PUT_LINE;

procedure PUT_LINE_ITEM: STRING)
is
begin
  null; -- should be defined by implementor
end PUT_LINE;

-- generic package for Input-Output of Integer Types
package body INTEGER_IO is separate;
-- generic package for Input-Output of Floating Point Types
package body FLOAT_IO is separate;
-- generic package for Input-Output of Fixed Point Types
package body FIXED_IO is separate;
-- generic package for Input-Output of Enumeration Types
```

```
package body ENUMERATION_IO is separate;

end TEXT_IO;

separate (CAIS)
package body SCROLL_TERMINAL is
  use NODE_DEFINITIONS;
  use NODE_MANAGEMENT;
  use IO_DEFINITIONS;
  use TEXT_IO;

  procedure SET_POSITION(TERMINAL: FILE_TYPE;
                        POSITION: POSITION_TYPE)
  is
  begin
    null; -- should be defined by implementor
  end SET_POSITION;

  procedure SET_POSITION(POSITION: POSITION_TYPE)
  is
  begin
    SET_POSITION (CURRENT_OUTPUT, POSITION);
  end SET_POSITION;

  function GET_POSITION (TERMINAL: FILE_TYPE)
    return POSITION_TYPE
  is
    RESULT : POSITION_TYPE;
  begin
    -- should be defined by implementor
    return RESULT;
  end GET_POSITION;

  function GET_POSITION return POSITION_TYPE
  is
  begin
    return GET_POSITION (CURRENT_OUTPUT);
  end GET_POSITION;

  function TERMINAL_SIZE(TERMINAL: FILE_TYPE)
    return POSITION_TYPE
  is
    RESULT : POSITION_TYPE;
  begin
    -- should be defined by implementor
    return RESULT;
  end TERMINAL_SIZE;

  function TERMINAL_SIZE return POSITION_TYPE
  is
  begin
    return TERMINAL_SIZE (CURRENT_OUTPUT);
  end TERMINAL_SIZE;

  procedure SET_TAB(TERMINAL: FILE_TYPE;
                  KIND: TAB_ENUMERATION := HORIZONTAL)
  is
  begin
    -- should be defined by implementor
    null;
  end SET_TAB;
```

```
end SET_TAB;

procedure SET_TAB(KIND: TAB_ENUMERATION := HORIZONTAL)
is
begin
  SET_TAB (CURRENT_OUTPUT, KIND);
end SET_TAB;

procedure CLEAR_TAB(TERMINAL: FILE_TYPE;
                    KIND: TAB_ENUMERATION := HORIZONTAL)
is
begin
  -- should be defined by implementor
  null;
end CLEAR_TAB;

procedure CLEAR_TAB(KIND: TAB_ENUMERATION := HORIZONTAL)
is
begin
  CLEAR_TAB (CURRENT_OUTPUT, KIND);
end CLEAR_TAB;

procedure TAB(TERMINAL: FILE_TYPE;
              KIND: TAB_ENUMERATION := HORIZONTAL;
              COUNT: POSITIVE := 1)
is
begin
  -- should be defined by implementor
  null;
end TAB;

procedure TAB(KIND: TAB_ENUMERATION := HORIZONTAL;
              COUNT: POSITIVE := 1)
is
begin
  TAB (CURRENT_OUTPUT, KIND, COUNT);
end TAB;

procedure BELL(TERMINAL: FILE_TYPE)
is
begin
  -- should be defined by implementor
  null;
end BELL;

procedure BELL
is
begin
  BELL (CURRENT_OUTPUT);
end BELL;

procedure PUT(TERMINAL: FILE_TYPE;
              ITEM: CHARACTER)
is
begin
  -- should be defined by implementor
  null;
end PUT;

procedure PUT(ITEM: CHARACTER)
is
```

```
begin
  PUT (CURRENT_OUTPUT, ITEM);
end PUT;

procedure PUT(TERMINAL: FILE_TYPE; ITEM : STRING)
is
begin
  for INDEX in ITEM'FIRST .. ITEM'LAST loop
    PUT (TERMINAL, ITEM (INDEX));
  end loop;
end PUT;

procedure PUT(ITEM: STRING)
is
begin
  PUT (CURRENT_OUTPUT, ITEM);
end PUT;

procedure SET_ECHO(TERMINAL: FILE_TYPE;
                  TO: BOOLEAN := TRUE)
is
begin
  -- should be defined by implementor
  null;
end SET_ECHO;

procedure SET_ECHO(TO: BOOLEAN := TRUE)
is
begin
  SET_ECHO (CURRENT_INPUT, TO);
end SET_ECHO;

function ECHO (TERMINAL: FILE_TYPE) return BOOLEAN
is separate;

function ECHO return BOOLEAN
is
begin
  return ECHO (CURRENT_INPUT);
end ECHO;

function MAXIMUM_FUNCTION_KEY(TERMINAL: FILE_TYPE)
return NATURAL
is
  RESULT : NATURAL;
begin
  -- should be defined by implementor
  return RESULT;
end MAXIMUM_FUNCTION_KEY;

function MAXIMUM_FUNCTION_KEY return NATURAL
is
begin
  return MAXIMUM_FUNCTION_KEY (CURRENT_INPUT);
end MAXIMUM_FUNCTION_KEY;

procedure SET(TERMINAL: FILE_TYPE;
             ITEM: out CHARACTER;
```

```

                                KEYS:    out FUNCTION_KEY_DESCRIPTOR)
is
begin
  null; -- should be defined by implementor
end GET;

procedure GET(ITEM:    out CHARACTER;
              KEYS:    out FUNCTION_KEY_DESCRIPTOR)
is
begin
  GET (CURRENT_OUTPUT, ITEM, KEYS);
end GET;

procedure GET(TERMINAL:    FILE_TYPE;
              ITEM:        out STRING;
              LAST:        out NATURAL;
              KEYS:        out FUNCTION_KEY_DESCRIPTOR)
is
begin
  null; -- should be defined by implementor
end GET;

procedure GET(ITEM:    out STRING;
              LAST:    out NATURAL;
              KEYS:    out FUNCTION_KEY_DESCRIPTOR)
is
begin
  GET (CURRENT_INPUT, ITEM, LAST, KEYS);
end GET;

function FUNCTION_KEY_COUNT(KEYS: FUNCTION_KEY_DESCRIPTOR)
  return NATURAL
is
  RESULT : NATURAL;
begin
  -- should be defined by implementor
  return RESULT;
end FUNCTION_KEY_COUNT;

procedure FUNCTION_KEY(KEYS:    FUNCTION_KEY_DESCRIPTOR;
                      INDEX:    POSITIVE;
                      KEY_IDENTIFIER: out POSITIVE;
                      POSITION:    out NATURAL)
is
begin
  -- should be defined by implementor
  null;
end FUNCTION_KEY;

procedure FUNCTION_KEY_NAME(TERMINAL:    FILE_TYPE;
                           KEY_IDENTIFIER: POSITIVE;
                           KEY_NAME:    out STRING;
                           LAST:        out POSITIVE)
is
begin
  -- should be defined by implementor
  null;
end FUNCTION_KEY_NAME;

procedure FUNCTION_KEY_NAME(KEY_IDENTIFIER: POSITIVE;
                           KEY_NAME:    out STRING;
                           LAST:        out POSITIVE)
```

```
is
begin
  FUNCTION KEY_NAME
    (CURRENT_INPUT, KEY_IDENTIFIER, KEY_NAME, LAST);
end FUNCTION KEY_NAME;

procedure NEW_LINE(TERMINAL: FILE_TYPE;
                   COUNT: POSITIVE := 1) is separate;

procedure NEW_LINE(COUNT: POSITIVE := 1)
is
begin
  NEW_LINE (CURRENT_OUTPUT, COUNT);
end NEW_LINE;

procedure NEW_PAGE(TERMINAL: FILE_TYPE) is separate;

procedure NEW_PAGE
is
begin
  NEW_PAGE (CURRENT_OUTPUT);
end NEW_PAGE;

end SCROLL_TERMINAL;

separate (CAIS)
package body PAGE_TERMINAL is
  use NODE_DEFINITIONS;
  use NODE_MANAGEMENT;
  use IO_DEFINITIONS;
  use TEXT_IO;

  procedure SET_POSITION(TERMINAL: FILE_TYPE;
                        POSITION: POSITION_TYPE)
  is
  begin
    null; -- should be defined by implementor
  end SET_POSITION;

  procedure SET_POSITION(POSITION: POSITION_TYPE)
  is
  begin
    SET_POSITION (CURRENT_OUTPUT, POSITION);
  end SET_POSITION;

  function GET_POSITION(TERMINAL: FILE_TYPE)
    return POSITION_TYPE
  is
  begin
    RESULT : POSITION_TYPE;
    -- should be defined by implementor
    return RESULT;
  end GET_POSITION;

  function GET_POSITION return POSITION_TYPE
  is
  begin
    return GET_POSITION (CURRENT_OUTPUT);
  end GET_POSITION;
```


Appendix D

PACKAGE LISTING OF CAIS PROCEDURES AND FUNCTIONS

This appendix lists the CAIS procedures and functions in the context of their associated packages. This appendix is intended to provide a simple reference to the CAIS procedures and functions in package order.

Operation	Description and Interfaces
	Package NODE_MANAGEMENT
Manipulation of node handles	The following interfaces are used for manipulating node handles and determining node handle status and node handle intent. procedure OPEN procedure CLOSE procedure CHANGE_INTENT function IS_OPEN function INTENT_OF
Querying node kind and name	The following interfaces are used to determine the kind of a node (file, process, or structural) and the primary name of a node. function KIND function PRIMARY_NAME
Pathname queries	The following interfaces allow queries about pathnames. None of these interfaces perform accesses to nodes; they perform pathname manipulations at the syntactic level only. function PRIMARY_NAME function PRIMARY_KEY function PRIMARY_RELATION function PATH_KEY function PATH_RELATION function BASE_PATH function LAST_RELATION function LAST_KEY
Node queries	The following interfaces allow queries about nodes. function IS_OBTAINABLE function IS_SAME procedure GET_PARENT
Node duplication interfaces	The following interfaces are used to duplicate single nodes or trees of nodes spanned by primary relationships. procedure COPY_NODE procedure COPY_TREE

```
function TEXT_LENGTH(LIST: LIST_TYPE;  
                     NAMED: NAME_STRING)  
    return POSITIVE  
is  
    RESULT: POSITIVE;  
begin  
    -- should be defined by implementor  
    return RESULT;  
end TEXT_LENGTH;  
function TEXT_LENGTH(LIST: LIST_TYPE;  
                     NAMED: TOKEN_TYPE)  
    return POSITIVE  
is  
    RESULT: POSITIVE;  
begin  
    -- should be defined by implementor  
    return RESULT;  
end TEXT_LENGTH;  
package body IDENTIFIER_ITEM is separate  
package INTEGER_ITEM is separate  
package FLOAT_ITEM is separate  
package STRING_ITEM is separate  
  
end LIST_UTILITIES;
```

```

                                POSITION:  POSITION_COUNT)
    return ITEM_KIND
is
    RESULT: ITEM_KIND;
begin
    -- should be defined by implementor
    return RESULT;
end GET_ITEM_KIND;
function GET_ITEM_KIND(LIST:  LIST_TYPE;
                        NAMED:  NAME_STRING)
    return ITEM_KIND
is
    RESULT: ITEM_KIND;
begin
    -- should be defined by implementor
    return RESULT;
end GET_ITEM_KIND;
procedure MERGE(FRONT:  LIST_TYPE;
                BACK:  LIST_TYPE;
                RESULT:  in out LIST_TYPE)
    is separate;
function LENGTH(LIST:  LIST_TYPE) return COUNT
    is separate;
procedure ITEM_NAME(LIST:  LIST_TYPE;
                    POSITION:  POSITION_COUNT;
                    NAME:  out TOKEN_TYPE)
    is separate;
function POSITION_BY_NAME(LIST:  LIST_TYPE;
                        NAMED:  NAME_STRING)
    return POSITION_COUNT
is
    RESULT: POSITION_COUNT;
begin
    -- should be defined by implementor
    return RESULT;
end POSITION_BY_NAME;
function POSITION_BY_NAME(LIST:  LIST_TYPE;
                        NAMED:  TOKEN_TYPE)
    return POSITION_COUNT
is
    RESULT: POSITION_COUNT;
begin
    -- should be defined by implementor
    return RESULT;
end POSITION_BY_NAME;

function TEXT_LENGTH(LIST:  LIST_TYPE)
    return NATURAL
is
    RESULT: NATURAL;
begin
    -- should be defined by implementor
    return RESULT;
end TEXT_LENGTH;

function TEXT_LENGTH(LIST:  LIST_TYPE;
                    POSITION:  POSITION_COUNT)
    return POSITIVE
is
    RESULT: POSITIVE;
begin
    -- should be defined by implementor
    return RESULT;
end TEXT_LENGTH;

```

```

return RESULT;
end SET_EXTRACT;

procedure SPLICE(LIST:      in out LIST_TYPE;
                  POSITION:   POSITION_COUNT;
                  SUB_LIST:  LIST_TEXT)
is
    RESULT: LIST_TEXT(1..10);
begin
    null; -- should be defined by implementor
end SPLICE;

procedure SPLICE(LIST:      in out LIST_TYPE;
                  POSITION:   POSITION_COUNT;
                  SUB_LIST:  LIST_TYPE)
is
    RESULT: LIST_TEXT(1..10);
begin
    null; -- should be defined by implementor
end SPLICE;

procedure DELETE(LIST:      in out LIST_TYPE;
                  POSITION:   POSITION_COUNT)
is
    RESULT: LIST_TEXT(1..10);
begin
    null; -- should be defined by implementor
end DELETE;

procedure DELETE(LIST:      in out LIST_TYPE;
                  NAMED:     NAME_STRING)
is
    RESULT: LIST_TEXT(1..10);
begin
    null; -- should be defined by implementor
end DELETE;

procedure DELETE(LIST: in out LIST_TYPE;
                  NAMED: TOKEN_TYPE)
is
    RESULT: LIST_TEXT(1..10);
begin
    null; -- should be defined by implementor
end DELETE;

function GET_LIST_KIND(LIST: LIST_TYPE)
    return LIST_KIND;
is
    RESULT: LIST_KIND;
begin
    -- should be defined by implementor
    return RESULT;
end GET_LIST_KIND;

function GET_ITEM_KIND(LIST: LIST_TYPE;
                       NAMED: TOKEN_TYPE)
    return ITEM_KIND
is
    RESULT: ITEM_KIND;
begin
    -- should be defined by implementor
    return RESULT;
end GET_ITEM_KIND;

function GET_ITEM_KIND(LIST:      LIST_TYPE;

```

```

    null; -- should be defined by implementor
end EXTRACT;

procedure EXTRACT(LIST:          LIST_TYPE;
                  NAMED:         TOKEN_TYPE)
    LIST_ITEM: out LIST_TYPE);
begin
    null; -- should be defined by implementor
end EXTRACT;

procedure REPLACE(LIST:          in out LIST_TYPE;
                  LIST_ITEM:      LIST_TYPE;
                  POSITION:         POSITION_COUNT)
begin
    null; -- should be defined by implementor
end REPLACE;

procedure REPLACE(LIST:          in out LIST_TYPE;
                  LIST_ITEM:      LIST_TYPE;
                  NAMED:         NAME_STRING)
begin
    null; -- should be defined by implementor
end REPLACE;

procedure REPLACE(LIST:          in out LIST_TYPE;
                  LIST_ITEM:      LIST_TYPE;
                  NAMED:         TOKEN_TYPE)
begin
    null; -- should be defined by implementor
end REPLACE;

procedure INSERT(LIST:          in out LIST_TYPE;
                  LIST_ITEM:      LIST_TYPE;
                  POSITION:         COUNT)
begin
    null; -- should be defined by implementor
end INSERT;

procedure INSERT(LIST:          in out LIST_TYPE;
                  LIST_ITEM:      LIST_TYPE;
                  NAMED:         NAME_STRING;
                  POSITION:         COUNT)
begin
    null; -- should be defined by implementor
end INSERT;

procedure INSERT(LIST:          in out LIST_TYPE;
                  LIST_ITEM:      LIST_TYPE;
                  NAMED:         TOKEN_TYPE;
                  POSITION:         COUNT)
begin
    null; -- should be defined by implementor
end INSERT;

function POSITION_BY_VALUE(LIST:          LIST_TYPE;
                          VALUE:          LIST_TYPE;
                          START_POSITION: POSITION_COUNT
                              := POSITION_COUNT'FIRST;
                          END_POSITION:   POSITION_COUNT
                              := POSITION_COUNT'LAST)
    return POSITION_COUNT is separate;

function SET_EXTRACT(LIST:          LIST_TYPE;
                     POSITION:        POSITION_COUNT;
                     LENGTH:         POSITIVE := POSITIVE'LAST)
    return LIST_TEXT
is
    RESULT: LIST_TEXT(1..10);
begin
    -- should be defined by implementor

```

```

procedure IMPORT(NODE:      NODE_TYPE;
                  HOST_FILE_NAME: STRING) is separate;
procedure IMPORT(NAME:      NAME_STRING;
                  HOST_FILE_NAME: STRING)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (1=> WRITE_CONTENTS));
  IMPORT(NODE, HOST_FILE_NAME);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end IMPORT;
procedure EXPORT(NODE:      NODE_TYPE;
                  HOST_FILE_NAME: STRING) is separate;
procedure EXPORT(NAME:      NAME_STRING;
                  HOST_FILE_NAME: STRING)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (1=> READ_CONTENTS));
  EXPORT(NODE, HOST_FILE_NAME);
  CLOSE(NODE);
exception
  when others =>
    CLOSE(NODE);
    raise;
end EXPORT;
end FILE_IMPORT_EXPORT;

```

```

separate (CAIS)
package body LIST_UTILITIES is
  use NODE_DEFINITIONS;
  use NODE_MANAGEMENT;
  procedure COPY(TO_LIST: out LIST_TYPE;
                 FROM_LIST: LIST_TYPE) is separate;

  function TO_LIST(LIST_STRING: STRING)
    return LIST_TYPE is separate;
  function TO_TEXT(LIST_ITEM: LIST_TYPE)
    return LIST_TEXT
  is
    RESULT: LIST_TEXT(1..10);
  begin
    -- should be defined by implementor
    return RESULT;
  end TO_TEXT;
  function IS_EQUAL(LIST1: LIST_TYPE;
                    LIST2: LIST_TYPE)
    return BOOLEAN is separate;

  procedure EXTRACT(LIST: LIST_TYPE;
                    POSITION: POSITION_COUNT;
                    LIST_ITEM: out LIST_TYPE);

  begin
    null; -- should be defined by implementor
  end EXTRACT;
  procedure EXTRACT(LIST: LIST_TYPE;
                    NAMED: NAME_STRING;
                    LIST_ITEM: out LIST_TYPE);

  begin

```

```

end AREA_QUALIFIER_REQUIRES_SPACE;

end FORM_TERMINAL;

separate (CAIS)
package body MAGNETIC_TAPE is
  use NODE_DEFINITIONS;
  use NODE_MANAGEMENT;

  procedure MOUNT(TAPE_DRIVE: FILE_TYPE;
                  TAPE_NAME: REEL_NAME;
                  DENSITY: POSITIVE) is separate;
  procedure LOAD_UNLABELED(TAPE_DRIVE: FILE_TYPE;
                          DENSITY: POSITIVE;
                          BLOCK_SIZE: POSITIVE)
    is separate;
  procedure INITIALIZE_UNLABELED(TAPE_DRIVE: FILE_TYPE;
                                DENSITY: POSITIVE;
                                BLOCK_SIZE: POSITIVE)
    is separate;
  procedure LOAD_LABELED(TAPE_DRIVE: FILE_TYPE;
                        VOLUME_IDENTIFIER: VOLUME_STRING;
                        DENSITY: POSITIVE;
                        BLOCK_SIZE: POSITIVE) is separate;
  procedure INITIALIZE_LABELED(TAPE_DRIVE: FILE_TYPE;
                              VOLUME_IDENTIFIER: VOLUME_STRING;
                              DENSITY: POSITIVE;
                              BLOCK_SIZE: POSITIVE;
                              ACCESSIBILITY: CHARACTER := ' ')
    is separate;
  procedure UNLOAD(TAPE_DRIVE: FILE_TYPE) is separate;
  procedure DISMOUNT(TAPE_DRIVE: FILE_TYPE) is separate;
  function IS_LOADED(TAPE_DRIVE: FILE_TYPE)
    return BOOLEAN is separate;
  function IS_MOUNTED(TAPE_DRIVE: FILE_TYPE)
    return BOOLEAN is separate;
  function TAPE_STATUS(TAPE_DRIVE: FILE_TYPE)
    return TAPE_POSITION is separate;
  procedure REWIND_TAPE(TAPE_DRIVE: FILE_TYPE) is separate;
  procedure SKIP_TAPE_MARKS(TAPE_DRIVE: FILE_TYPE;
                           NUMBER: INTEGER := 1;
                           TAPE_STATE: out TAPE_POSITION)
    is separate;
  procedure WRITE_TAPE_MARK(TAPE_DRIVE: FILE_TYPE;
                           NUMBER: POSITIVE := 1;
                           TAPE_STATE: out TAPE_POSITION)
    is separate;
  procedure VOLUME_HEADER(TAPE_DRIVE: FILE_TYPE;
                          VOLUME_IDENTIFIER: VOLUME_STRING;
                          ACCESSIBILITY: CHARACTER := ' ')
    is separate;
  procedure FILE_HEADER(TAPE_DRIVE: FILE_TYPE;
                       FILE_IDENTIFIER: FILE_STRING;
                       EXPIRATION_DATE: STRING := " 99366";
                       ACCESSIBILITY: CHARACTER := ' ') is separate;
  procedure END_FILE_LABEL(TAPE_DRIVE: FILE_TYPE) is separate;
  procedure READ_LABEL(TAPE_DRIVE: FILE_TYPE;
                      LABEL: out LABEL_STRING) is separate;

end MAGNETIC_TAPE;

package FILE_IMPORT_EXPORT is
  use NODE_DEFINITIONS;
  use NODE_MANAGEMENT;

```

```
ITEM : out PRINTABLE_CHARACTER)
is
begin
  -- should be defined by implementor
  null;
end GET;

procedure GET(FORM: in out FORM_TYPE;
ITEM: out STRING)
is
begin
  for INDEX in ITEM'FIRST .. ITEM'LAST loop
    GET (FORM, ITEM (INDEX)); -- Read a single character
  end loop;
end GET;

function IS_FORM_UPDATED(FORM: FORM_TYPE) return BOOLEAN
is separate;

function TERMINATION_KEY(FORM: FORM_TYPE) return NATURAL
is separate;

function FORM_SIZE(FORM: FORM_TYPE) return POSITION_TYPE
is separate;

function TERMINAL_SIZE(TERMINAL: FILE_TYPE)
return POSITION_TYPE
is
  RESULT: POSITION_TYPE;
begin
  -- should be defined by implementor
  return RESULT;
end TERMINAL_SIZE;

function TERMINAL_SIZE return POSITION_TYPE
is
begin
  return TERMINAL_SIZE (CURRENT_OUTPUT);
end TERMINAL_SIZE;

function AREA_QUALIFIER_REQUIRES_SPACE(FORM: FORM_TYPE)
return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end AREA_QUALIFIER_REQUIRES_SPACE;

function AREA_QUALIFIER_REQUIRES_SPACE
(TERMINAL: FILE_TYPE) return BOOLEAN
is
  RESULT : BOOLEAN;
begin
  -- should be defined by implementor
  return RESULT;
end AREA_QUALIFIER_REQUIRES_SPACE;

function AREA_QUALIFIER_REQUIRES_SPACE return BOOLEAN
is
begin
  return AREA_QUALIFIER_REQUIRES_SPACE (CURRENT_OUTPUT);
```



```
is
begin
  SELECT_GRAPHIC_RENDITION (CURRENT_OUTPUT, RENDITION);
end SELECT_GRAPHIC_RENDITION;

end PAGE_TERMINAL;

separate (CAIS)
package body FORM_TERMINAL is
  use NODE_DEFINITIONS;
  use NODE_MANAGEMENT;
  use IO_DEFINITIONS;
  use TEXT_IO;

  function MAXIMUM_FUNCTION_KEY(TERMINAL: FILE_TYPE)
    return NATURAL is separate;

  function MAXIMUM_FUNCTION_KEY return NATURAL
  is
  begin
    return MAXIMUM_FUNCTION_KEY (CURRENT_INPUT);
  end MAXIMUM_FUNCTION_KEY;

  procedure DEFINE_QUALIFIED_AREA
    (FORM: in out FORM_TYPE;
     INTENSITY: AREA_INTENSITY := NORMAL;
     PROTECTION: AREA_PROTECTION := PROTECTED;
     INPUT: AREA_INPUT := GRAPHIC_CHARACTERS;
     VALUE: AREA_VALUE := NO_FILL) is separate;

  procedure REMOVE_AREA_QUALIFIER(FORM: in out FORM_TYPE) is separate;

  procedure GET_POSITION(FORM: in out FORM_TYPE;
    POSITION: POSITION_TYPE) is separate;

  procedure NEXT_QUALIFIED_AREA(FORM: in out FORM_TYPE;
    COUNT: POSITIVE := 1) is separate;

  procedure PUT(FORM: in out FORM_TYPE;
    ITEM: PRINTABLE_CHARACTER)
  is
  begin
    null; -- should be defined by implementor
  end PUT;

  procedure PUT(FORM: in out FORM_TYPE; ITEM : STRING)
  is
  begin
    for INDEX in ITEM'FIRST .. ITEM'LAST loop
      PUT (FORM, ITEM (INDEX)); -- Write a single character
    end loop;
  end PUT;

  procedure ERASE_AREA(FORM: in out FORM_TYPE) is separate;

  procedure ERASE_FORM(FORM: in out FORM_TYPE) is separate;

  procedure ACTIVATE(TERMINAL: FILE_TYPE;
    FORM: in out FORM_TYPE) is separate;

  procedure GET(FORM: in out FORM_TYPE;
```

```
procedure ERASE_IN_DISPLAY
  (TERMINAL: FILE_TYPE;
   SELECTION: SELECT_ENUMERATION) is separate;

procedure ERASE_IN_DISPLAY
  (SELECTION: SELECT_ENUMERATION)
is
begin
  ERASE_IN_DISPLAY (CURRENT_OUTPUT, SELECTION);
end ERASE_IN_DISPLAY;

procedure ERASE_IN_LINE (TERMINAL: FILE_TYPE;
                          SELECTION: SELECT_ENUMERATION) is separate;

procedure ERASE_IN_LINE (SELECTION: SELECT_ENUMERATION)
is
begin
  ERASE_IN_LINE (CURRENT_OUTPUT, SELECTION);
end ERASE_IN_LINE;

procedure INSERT_SPACE (TERMINAL: FILE_TYPE;
                        COUNT: POSITIVE := 1) is separate;

procedure INSERT_SPACE (COUNT: POSITIVE := 1)
is
begin
  INSERT_SPACE (CURRENT_OUTPUT, COUNT);
end INSERT_SPACE;

procedure INSERT_LINE (TERMINAL: FILE_TYPE;
                       COUNT: POSITIVE := 1) is separate;

procedure INSERT_LINE (COUNT: POSITIVE := 1)
is
begin
  INSERT_LINE (CURRENT_OUTPUT, COUNT);
end INSERT_LINE;

function GRAPHIC_RENDITION_SUPPORT
  (TERMINAL: FILE_TYPE;
   RENDITION: GRAPHIC_RENDITION_ARRAY)
  return BOOLEAN is separate;

function GRAPHIC_RENDITION_SUPPORT
  (RENDITION: GRAPHIC_RENDITION_ARRAY)
  return BOOLEAN
is
begin
  return GRAPHIC_RENDITION_SUPPORT
    (CURRENT_OUTPUT, RENDITION);
end GRAPHIC_RENDITION_SUPPORT;

procedure SELECT_GRAPHIC_RENDITION
  (TERMINAL: FILE_TYPE;
   RENDITION: GRAPHIC_RENDITION_ARRAY :=
    DEFAULT_GRAPHIC_RENDITION) is separate;

procedure SELECT_GRAPHIC_RENDITION
  (RENDITION: GRAPHIC_RENDITION_ARRAY :=
    DEFAULT_GRAPHIC_RENDITION)
```

```
    RESULT: NATURAL;
begin
    -- should be defined by implementor
    return RESULT;
end FUNCTION_KEY_COUNT;

procedure FUNCTION_KEY(KEYS:          FUNCTION_KEY_DESCRIPTOR;
                       INDEX:         POSITIVE;
                       KEY_IDENTIFIER: out POSITIVE;
                       POSITION:       out NATURAL)
is
begin
    -- should be defined by implementor
    null;
end FUNCTION_KEY;

procedure FUNCTION_KEY_NAME(TERMIAL:    FILE_TYPE;
                            KEY_IDENTIFIER: POSITIVE;
                            KEY_NAME:    out STRING;
                            LAST:        out POSITIVE)
is
begin
    -- should be defined by implementor
    null;
end FUNCTION_KEY_NAME;

procedure FUNCTION_KEY_NAME(KEY_IDENTIFIER: POSITIVE;
                            KEY_NAME:      out STRING;
                            LAST:          out POSITIVE)
is
begin
    FUNCTION_KEY_NAME
        (CURRENT_INPUT, KEY_IDENTIFIER, KEY_NAME, LAST);
end FUNCTION_KEY_NAME;

procedure DELETE_CHARACTER(TERMIAL: FILE_TYPE;
                          COUNT:    POSITIVE := 1) is separate;

procedure DELETE_CHARACTER(COUNT: POSITIVE := 1)
is
begin
    DELETE_CHARACTER (CURRENT_OUTPUT, COUNT);
end DELETE_CHARACTER;

procedure DELETE_LINE(TERMIAL: FILE_TYPE;
                     COUNT:    POSITIVE := 1) is separate;

procedure DELETE_LINE(COUNT: POSITIVE := 1)
is
begin
    DELETE_LINE (CURRENT_OUTPUT, COUNT);
end DELETE_LINE;

procedure ERASE_CHARACTER(TERMIAL: FILE_TYPE;
                         COUNT:    POSITIVE := 1) is separate;

procedure ERASE_CHARACTER(COUNT: POSITIVE := 1)
is
begin
    ERASE_CHARACTER (CURRENT_OUTPUT, COUNT);
end ERASE_CHARACTER;
```

```
    return RESULT;  
end ECHO;
```

```
function ECHO return BOOLEAN  
is  
begin  
    return ECHO (CURRENT_INPUT);  
end ECHO;
```

```
function MAXIMUM_FUNCTION_KEY(TERMINAL: FILE_TYPE)  
    return NATURAL  
is  
    RESULT : NATURAL;  
begin  
    -- should be defined by implementor  
    return RESULT;  
end MAXIMUM_FUNCTION_KEY;
```

```
function MAXIMUM_FUNCTION_KEY return NATURAL  
is  
begin  
    return MAXIMUM_FUNCTION_KEY (CURRENT_INPUT);  
end MAXIMUM_FUNCTION_KEY;
```

```
procedure GET(TERMINAL: FILE_TYPE;  
              ITEM: out CHARACTER;  
              KEYS: out FUNCTION_KEY_DESCRIPTOR)  
is  
begin  
    null; -- should be defined by implementor  
end GET;
```

```
procedure GET(ITEM: out CHARACTER;  
              KEYS: out FUNCTION_KEY_DESCRIPTOR)  
is  
begin  
    GET (CURRENT_INPUT, ITEM, KEYS);  
end GET;
```

```
procedure GET(TERMINAL: FILE_TYPE;  
              ITEM: out STRING;  
              LAST: out NATURAL;  
              KEYS: out FUNCTION_KEY_DESCRIPTOR)  
is  
begin  
    null; -- should be defined by implementor  
end GET;
```

```
procedure GET(ITEM: out STRING;  
              LAST: out NATURAL;  
              KEYS: in out FUNCTION_KEY_DESCRIPTOR)  
is  
begin  
    GET (CURRENT_INPUT, ITEM, LAST, KEYS);  
end GET;
```

```
function FUNCTION_KEY_COUNT(KEYS: FUNCTION_KEY_DESCRIPTOR)  
    return NATURAL  
is
```

```
is
begin
  -- should be defined by implementor
  null;
end BELL;

procedure BELL
is
begin
  BELL (CURRENT_OUTPUT);
end BELL;

procedure PUT(TERMINAL: FILE_TYPE;
              ITEM: CHARACTER)
is
begin
  -- should be defined by implementor
  null;
end PUT;

procedure PUT(ITEM: CHARACTER)
is
begin
  PUT (CURRENT_OUTPUT, ITEM);
end PUT;

procedure PUT(TERMINAL: FILE_TYPE; ITEM : STRING)
is
begin
  for INDEX in ITEM'FIRST .. ITEM'LAST loop
    PUT (TERMINAL, ITEM (INDEX));
  end loop;
end PUT;

procedure PUT(ITEM: STRING)
is
begin
  PUT (CURRENT_OUTPUT, ITEM);
end PUT;

procedure SET_ECHO(TERMINAL: FILE_TYPE;
                  TO: BOOLEAN := TRUE)
is
begin
  -- should be defined by implementor
  null;
end SET_ECHO;

procedure SET_ECHO(TO: BOOLEAN := TRUE)
is
begin
  SET_ECHO (CURRENT_INPUT, TO);
end SET_ECHO;

function ECHO(TERMINAL: FILE_TYPE) return BOOLEAN
is
  RESULT: BOOLEAN;
begin
  -- should be defined by implementor
```

```
function TERMINAL_SIZE(TERMINAL: FILE_TYPE)
    return POSITION_TYPE
is
    RESULT : POSITION_TYPE;
begin
    -- should be defined by implementor
    return RESULT;
end TERMINAL_SIZE;

function TERMINAL_SIZE return POSITION_TYPE
is
begin
    return TERMINAL_SIZE (CURRENT_OUTPUT);
end TERMINAL_SIZE;

procedure SET_TAB(TERMINAL: FILE_TYPE;
                  KIND:     TAB_ENUMERATION := HORIZONTAL)
is
begin
    -- should be defined by implementor
    null;
end SET_TAB;

procedure SET_TAB(KIND: TAB_ENUMERATION := HORIZONTAL)
is
begin
    SET_TAB (CURRENT_OUTPUT, KIND);
end SET_TAB;

procedure CLEAR_TAB(TERMINAL: FILE_TYPE;
                    KIND:     TAB_ENUMERATION := HORIZONTAL)
is
begin
    -- should be defined by implementor
    null;
end CLEAR_TAB;

procedure CLEAR_TAB(KIND: TAB_ENUMERATION := HORIZONTAL)
is
begin
    CLEAR_TAB (CURRENT_OUTPUT, KIND);
end CLEAR_TAB;

procedure TAB(TERMINAL: FILE_TYPE;
              KIND:     TAB_ENUMERATION := HORIZONTAL;
              COUNT:    POSITIVE := 1)
is
begin
    -- should be defined by implementor
    null;
end TAB;

procedure TAB(KIND: TAB_ENUMERATION := HORIZONTAL;
              COUNT: POSITIVE := 1)
is
begin
    TAB (CURRENT_OUTPUT, KIND, COUNT);
end TAB;

procedure BELL(TERMINAL: FILE_TYPE)
```

**Alteration of
relationships**

The following interface is used to alter the primary relationship of a node, thereby changing its unique primary name.

procedure RENAME

**Deletion of primary
relationships**

The following two interfaces allow the deletion of the primary relationship of a single node or of the primary relationships of a node and all the nodes that are contained in the tree spanned by primary relationships emanating from these nodes.

**procedure DELETE_NODE
procedure DELETE_TREE**

**Creation and
deletion of
secondary
relationships**

The following interfaces allow the creation and deletion of user-defined secondary relationships.

**procedure LINK
procedure UNLINK**

Node Iterators

The following interfaces allow the iteration over nodes reachable from a given node via its emanating relationships.

**procedure ITERATE
function MORE
procedure GET_NEXT**

**Manipulation of the
CURRENT_NODE
relationship**

The following interfaces allow changes to the relationship of the predefined relation CURRENT_NODE emanating from the current process node and open a node handle on the node that is the target of such a relationship.

**procedure SET_CURRENT_NODE
procedure GET_CURRENT_NODE**

Package ATTRIBUTES

**Manipulation of
attributes**

The following interfaces are used for defining and manipulating the attributes for nodes and relationships.

**procedure CREATE_NODE_ATTRIBUTE
procedure CREATE_PATH_ATTRIBUTE
procedure DELETE_NODE_ATTRIBUTE
procedure DELETE_PATH_ATTRIBUTE
procedure SET_NODE_ATTRIBUTE
procedure SET_PATH_ATTRIBUTE**

```
procedure GET_NODE_ATTRIBUTE
procedure GET_PATH_ATTRIBUTE
procedure NODE_ATTRIBUTE_ITERATE
procedure PATH_ATTRIBUTE_ITERATE
function MORE
procedure GET_NEXT
```

Package ACCESS_CONTROL

Manipulation of
access control

The following interfaces are used to manipulate
access control information for nodes.

```
procedure SET_ACCESS_CONTROL
function IS_GRANTED
procedure ADOPT
procedure UNADOPT
```

Package STRUCTURAL_NODES

Creation of
structural node

The following interface is used to create a
structural node and to establish the primary
relationship to it.

```
procedure CREATE_NODE
```

Package PROCESS_CONTROL

Spawning a
process

This interface creates a process node, initiates
the new process, and returns control to the calling
task upon node creation.

```
procedure SPAWN_PROCESS
```

Awaiting process
termination
or abortion

This interface suspends the calling task and
waits for the process to terminate or abort.

```
procedure AWAIT_PROCESS_COMPLETION
```

Invoking a
process

This interface is functionally the same as
performing a call to SPAWN_PROCESS followed
by a call to AWAIT_PROCESS_COMPLETION.

```
procedure INVOKE_PROCESS
```

Creating a
new job

This interface creates a new root process node.
Control is returned to the calling task after
the new job is created.

```
procedure CREATE_JOB
```

Examination and
modification of
results list

These interfaces provide the techniques for a
process to examine and modify a results list.

31 JANUARY 1985

	procedure APPEND_RESULTS procedure WRITE_RESULTS procedure GET_RESULTS
Determination of state of process and input parameters	These interfaces are used to determine the value of the predefined attributes CURRENT_STATUS and PARAMETERS. function STATUS_OF_PROCESS procedure GET_PARAMETERS
Modification of the status of a process	These interfaces change the process status of a process. procedure ABORT_PROCESS procedure SUSPEND_PROCESS procedure RESUME_PROCESS
Handling I/O and time queries	These interfaces are used to query process nodes to determine the values of the predefined attributes HANDLES_OPEN, IO-UNITS, START_TIME, FINISH_TIME, and MACHINE_TIME. function HANDLES_OPEN function IO_UNITS function START_TIME function FINISH_TIME function MACHINE_TIME
Packages CAIS.DIRECT_IO, CAIS.SEQUENTIAL_IO, CAIS.TEXT_IO	
Creating, opening, and deleting secondary storage file	These interfaces are used to create a file and its file node, to open a handle on a file, and to delete a file. These may be used with direct access, sequential access, and text files. procedure CREATE procedure OPEN procedure DELETE
	Package TEXT_IO
Reading and writing characters from/to text file	This procedure is used to read and write characters from/to a text file procedure RESET procedure GET procedure PUT
Setting predefined relations	These interfaces set the relationships of the predefined relations CURRENT_INPUT, CURRENT_OUTPUT,

and CURRENT_ERROR.

```
procedure SET_INPUT
procedure SET_OUTPUT
procedure SET_ERROR
```

Opening and
returning
handles on
error files

These interfaces are used for returning an open file handle on the error file and for returning an open file handle on the current error output file.

```
function STANDARD_ERROR
function CURRENT_ERROR
```

Package IO_CONTROL

Opening a
file node

This interface obtains an open node handle from a file handle.

```
procedure OPEN_FILE_NODE
```

Transmitting
data from
internal to
external file

This interface is used to transmit data from an internal file to its associated external file.

```
procedure SYNCHRONIZE
```

Handling log
files, prompts,
and function
keys

These interfaces are used for performing operations on log files and for handling prompt strings, character arrays, and function keys.

```
procedure SET_LOG
procedure CLEAR_LOG
procedure ENABLE_FUNCTION_KEYS
function LOGGING
function GET_LOG
function NUMBER_OF_ELEMENTS
procedure SET_PROMPT
function GET_PROMPT
function INTERCEPTED_CHARACTERS
function FUNCTION_KEYS_ENABLED
```

Creating
coupled
queue

This interface creates a queue file and its node. The initial contents of the queue file are the same as those of the file to which it is coupled. The queue file must be of kind MIMIC or COPY.

```
procedure COUPLE
```

Package SCROLL_TERMINAL, PAGE_TERMINAL, FORM_TERMINAL

Advancing the

This procedure advances the active position to

active position

the specified position.

procedure SET _ POSITION

Package SCROLL _ TERMINAL, PAGE _ TERMINAL

Querying terminal,
controlling tab
stop, sounding
bell and writing
a character

These interfaces are used with scroll and page terminals to determine the active position, determine terminal row and column size, manipulate tab stops, sound the bell, and write a character.

function GET _ POSITION
function TERMINAL _ SIZE
procedure SET _ TAB
procedure CLEAR _ TAB
procedure TAB
procedure BELL
procedure PUT

Contolling echo,
querying function
keys and reading
characters
function keys

These interfaces are used for echoing characters to associated output devices, determining the maximum allowable function key identification number, reading a character or characters, and determining information about function keys.

procedure SET _ ECHO
function ECHO
function MAXIMUM _ FUNCTION _ KEY
procedure GET
function FUNCTION _ KEY _ COUNT
procedure FUNCTION _ KEY
procedure FUNCTION _ KEY _ NAME

Package SCROLL _ TERMINAL

Line and page
advancement

These interfaces are used to control line and page advancement.

procedure NEW _ LINE
procedure NEW _ PAGE

Package PAGE _ TERMINAL

Performing
deletions, erasures,
and insertions on
a page

These interfaces are used for deleting characters characters and lines, for replacing characters entire displays and lines with spaces and for inserting spaces and lines.

procedure DELETE _ CHARACTER
procedure DELETE _ LINE
procedure ERASE _ CHARACTER
procedure ERASE _ IN _ DISPLAY

	<pre>procedure ERASE_IN_LINE procedure INSERT_SPACE procedure INSERT_LINE</pre>
Graphic rendition determination and selection	These interfaces are used for determining if a graphic rendition is supported and for selecting a particular graphic rendition. <pre>function GRAPHIC_RENDITION_SUPPORT procedure SELECT_GRAPHIC_RENDITION</pre>
	Package FORM_TERMINAL
Determining maximum value from TERMINATION_KEY	This interface returns the maximum value that may be returned by function TERMINATION_KEY. <pre>function MAXIMUM_FUNCTION_KEY</pre>
Opening form and defining qualified area	These interfaces open a form to the specified size, determine if the form is open, define a qualified area, and remove an area qualifier. <pre>procedure DEFINE_QUALIFIED_AREA procedure REMOVE_AREA_QUALIFIER</pre>
Qualified area advancement, writing, and erasing	These interfaces advance the active position to a subsequent qualified area, write to a form, erase a qualified area, and erase the form. <pre>procedure NEXT_QUALIFIED_AREA procedure PUT procedure ERASE_AREA procedure ERASE_FORM</pre>
Activating form, reading, and determining information about form	These interfaces activate the form on the terminal, read data from the form, determine if changes have been made to the form, determine the termination key, determine the size of the form and terminal, and determine if the area qualifier requires space. <pre>procedure ACTIVATE procedure GET function IS_FORM_UPDATED function TERMINATION_KEY function FORM_SIZE function TERMINAL_SIZE function AREA_QUALIFIER_REQUIRES_SPACE</pre>
	Package MAGNETIC_TAPE
Mounting, status checking, and	These are used to load unlabeled and labeled tapes, dismount tapes, determine if a tape is

writing tape
marks

loaded or mounted and where it is positioned,
skip tape marks, and write a tape mark.

```
procedure MOUNT
procedure LOAD_UNLABELED
procedure LOAD_LABELED
procedure UNLOAD
procedure DISMOUNT
function IS_LOADED
function IS_MOUNTED
function TAPE_STATUS
procedure REWIND_TAPE
procedure SKIP_TAPE_MARKS
procedure WRITE_TAPE_MARK
```

Initialize and
labeling tapes

These interfaces are used to initialize tapes, to
create a volume file header, end of file, read tape
label and end of volume label.

```
procedure INITIALIZE_UNLABELED
procedure INITIALIZE_LABELED
procedure VOLUME_HEADER
procedure FILE_HEADER
procedure END_FILE_LABEL
procedure READ_LABEL
```

Package FILE_IMPORT_EXPORT

Transferring
files between
CAIS and host
system

These interfaces are used to transfer files
between a CAIS implementation and the host
file system.

```
procedure IMPORT
procedure EXPORT
```

Package LIST_UTILITIES

copying and
converting lists

These interfaces perform operations on list items
that are lists. Operations performed copy
a list, convert the textual representation to an internal
list representation, and convert an internal representation
to a textual representation.

```
procedure COPY
procedure TO_LIST
function TO_TEXT
```

comparing,
deleting, and
querying lists

These list interfaces determine the equality of two
lists, delete an item from a list, determine the
kind of list, and kind of list item.

	function IS_EQUAL procedure DELETE function GET_LIST_KIND function GET_LIST_KIND function GET_ITEM_KIND
<i>List splicing, merging, and extracting</i>	<i>These list interfaces insert a sublist of items into a list, merge two lists and extract sublists of items from a list.</i>
	procedure SPLICE procedure MERGE function SET_EXTRACT
<i>Determining list lengths and names</i>	<i>These list interfaces determine the length of a list, length of a string representing text, the name of a named item and the position of a named item.</i>
	function LENGTH function TEXT_LENGTH procedure ITEM_NAME function POSITION_BY_NAME
<i>Manipulation of list items in a list</i>	<i>These list interfaces extract an item from a list, replace an item in a list, insert an item in a list, and search a list for a list value.</i>
	procedure EXTRACT procedure REPLACE procedure INSERT function POSITION_BY_VALUE
	Generic Package IDENTIFIER_ITEM
<i>Manipulation of tokens</i>	<i>These interfaces are used for manipulating list items which are tokens. Operations performed by these interfaces convert a string representation of an identifier to its token, convert a token to an identifier, determine the equality of two tokens, extract an identifier from a list, replace an identifier in a list, insert an identifier into a list, and search a list for an identifier item value.</i>
	procedure TO_TOKEN function TO_TEXT function IS_EQUAL procedure EXTRACT procedure REPLACE procedure INSERT function POSITION_BY_VALUE
	Generic Package INTEGER_ITEM

Manipulation
of integer
items in a
list

These interfaces are used for manipulating list items which are integers. Operations performed by these interfaces convert an integer item to its textual representation, extract an integer item from a list, insert an integer item into a list and search a list for an integer value.

function TO _TEXT
function EXTRACT
procedure REPLACE
procedure INSERT
function POSITION _BY _VALUE

Generic Package FLOAT _ITEM

Manipulation
of floating
point items
in a list

These interfaces are used for manipulating list items which are floating point numbers. Operations performed by these interfaces convert an floating point item to its textual representation, extract an floating point item from a list, insert an floating point item into a list and search a list for an floating point value.

function TO _TEXT
function EXTRACT
procedure REPLACE
procedure INSERT
function POSITION _BY _VALUE

Generic Package STRING _ITEM

Manipulation
of string
items in a
list

These interfaces are used for manipulating list items which are strings. Operations performed by these interfaces extract a string item from a list, replace the value of a string item in a list, insert a string item into a list, and search a list for a string value.

function EXTRACT
procedure REPLACE
procedure INSERT
function POSITION _BY _VALUE

Index

Abort 5
ABORT_PROCESS 93
Aborted 79
Access 5, 23
Access checking 5, 20
Access control 5, 20
Access control constraints 5
Access control information 5, 20
Access control mechanisms 74
Access control rights 20
Access control rules 5, 20
Access relationship 5
Access relationships 21
Access right 33
Access rights 5, 20
Access rights constraints 20
Access synchronization 33
Access to a node 5, 20, 31
ACCESS_CONTROL 31, 74
ACCESS_METHOD 101
ACCESS_VIOLATION 32
Accessible 5
ACTIVATE 170
Active position 5, 165
Ada external files 13
Ada Programming Support Environment 1, 5
ADOPT 22, 76
Adopt a role 5, 22
Adopted role of a process 6, 22
ADOPTED_ROLE 22
Adopts 22
Advance 6, 130
Append intent 33
APPEND_FILE 152
APPEND_RESULTS 90
Approved access rights 6, 23
APSE 1, 5
Area qualifier 6, 166
AREA_INPUT 166
AREA_INTENSITY 166
AREA_PROTECTION 166
AREA_QUALIFIER_REQUIRES_SPACE 173
AREA_VALUE 166
Attribute 6, 13
Attribute iteration types and subtypes 71
Attribute name 19, 62
ATTRIBUTE_ITERATOR 71
ATTRIBUTE_NAME 62, 63, 64, 65, 71
ATTRIBUTE_PATTERN 71, 72
Attributes 19, 31, 62, 114
AWAIT_PROCESS_COMPLETION 85

BASE_PATH 44
BELL 135, 150

Case distinction 62
CHANGE_INTENT 33, 40
Classification levels 28
CLEAR_LOG_FILE 123

CLEAR_TAB 148
 CLEAR_TAB 133
 CLOSE 33, 39
 Close a node handle 33
 Closed node handle 8
 Contents 8, 13
 Copy queue 101, 120
 COPY_NODE 48
 COPY_TREE 49
 Copying nodes 33
 Couple 8, 127
 CREATE 33, 105, 109, 113
 Create node attributes 33
 CREATE_JOB 88
 CREATE_NODE 33, 77, 79
 CREATE_NODE_ATTRIBUTE 62
 CREATE_PATH_ATTRIBUTE 63, 64
 Current job 8
 Current node 8
 Current process 8, 16
 Current user 8
 CURRENT_ERROR 80, 120
 CURRENT_INPUT 80, 152
 CURRENT_JOB 15
 CURRENT_NODE 15, 32
 CURRENT_OUTPUT 80, 146
 CURRENT_PROCESS 32
 CURRENT_STATUS 80, 81
 CURRENT_USER 15, 32

 DEFAULT_GRAPHIC_RENDITION 145
 DEFAULT_RELATION 32, 46
 DEFINE_QUALIFIED_AREA 167
 DELETE 108, 112, 116, 195
 DELETE_CHARACTER 158
 DELETE_LINE 159
 DELETE_NODE 53
 DELETE_NODE_ATTRIBUTE 64
 DELETE_PATH_ATTRIBUTE 65
 DELETE_TREE 34
 Deleting nodes 33
 Dependent process 8, 16
 Dependent process node 16
 Descendant (of a node) 8
 Device 8, 15
 Device name 8, 16
 DEVICE_ERROR 145
 Discretionary access checking 26
 Discretionary access control 8, 19, 21
 DISMOUNT 180
 DOT 17

 ECHO 138, 153
 Element (of a file) 8
 Empty list 191
 ENABLE_FUNCTION_KEYS 126
 End position 7, 165
 END_FILE_LABEL 167
 ERASE_AREA 170
 ERASE_CHARACTER 180
 ERASE_FORM 170
 ERASE_IN_DISPLAY 180
 ERASE_IN_LINE 181

PROPOSED MIL-STD-CAN
31 JANUARY 1985

Exceptions useful for node manipulations 31
Existence of the node 33
EXPORT 190
External file 7, 103
EXTRACT 204, 207, 210, 213

File 7
File handle 7, 103
File node 7, 13, 14
File transfer 189
FILE_HEADER 185
FILE_KIND 101
FILE_STRING 175
FILE_TYPE 145, 175
FINISH_TIME 99
Form 7, 114
Form terminal 101
FORM_SIZE 172
FORM_STRING 32, 77
FORM_TYPE 166
FUNCTION_KEY 141, 156
FUNCTION_KEY_COUNT 141, 156
FUNCTION_KEY_DESCRIPTOR 145
FUNCTION_KEY_NAME 142, 157
FUNCTION_KEYS_ENABLED 126

GET 118, 139, 140, 154, 155, 171
GET_CURRENT_NODE 61
GET_LIST_KIND 196
GET_LOG 123
GET_NEXT 59, 73
GET_NODE_ATTRIBUTE 69
GET_PARAMETERS 92
GET_PARENT 47
GET_PATH_ATTRIBUTE 69
GET_POSITION 131, 146
GET_PROMPT 125
GET_RESULTS 90
GRANT 21
GRAPHIC_RENDITION_ARRAY 145
GRAPHIC_RENDITION_ENUMERATION 145
GRAPHIC_RENDITION_SUPPORT 164
Group 7, 22

HANDLES_OPEN 96
Hierarchical classification level 26
HIGHEST_CLASSIFICATION 28

Identification 18
Illegal identification 7, 18
IMPORT 189
IN_FILE 144, 146
Inaccessible 7, 20
INITIALIZE_LABELED 178
INITIALIZE_UNLABELED 177
Initiate 7
Initiated 13
Initiated process 7, 13
Initiating process 7, 13
INOUT_FILE 144
Input and output 100
INSERT 202, 205, 208, 211, 214
INSERT_LINE 163

INSERT_SPACE 162
INTENT 21, 33
INTENT_OF 41
INTENT_SPECIFICATION 32
INTENT_VIOLATION 32
INTENTION 47
INTERCEPTED_CHARACTERS 125
Interface 7
Internal file 7, 103
Invoke 80
INVOKE_PROCESS 33, 85
IO_DEFINITIONS 104
IO_UNITS 97
IS_FORM_UPDATED 171
IS_GRANTED 75
IS_LOADED 180
IS_MOUNTED 181
IS_OPEN 41
IS_SAME 46
ITEM_NAME 199
ITERATE 58
Iteration status 72
Iterator 7

Job 7, 15, 16

Key 7, 15
Keyword member 27
Keywords 27
KIND 41, 191, 196

Label 174
Label group 7, 183
LAST_KEY 45
LAST_RELATION 45
Latest key 8, 17
LATEST_KEY 17, 77
LAYOUT_ERROR 145
LENGTH 198
LEVEL 114
LINK 55
List 8, 191
List classifications 191
List item 8, 191
LIST_TYPE 62, 63, 64, 65
LOAD_LABELED 177
LOAD_UNLABELED 176
LOCK_ERROR 32
Locked against read operations 33
Locks on the node 33
LOGGING 123
LOWEST_CLASSIFICATION 28

MACHINE_TIME 100
Magnetic tape drive file 100, 101
Mandatory access control 8, 19, 26
Manipulation of attributes 62
MAXIMUM_FUNCTION_KEY 153, 187
MAXIMUM_FUNCTION_KEYS 138
MERGE 197
Mimic queue 101, 120
MODE_ERROR 145
Modify node attributes 33

MORE 59, 72
MOUNT 175

NAME_ERROR 32
NAME_STRING 32, 44
Named Item 8, 191
Named list 8
Necessary right 24
NEW_LINE 143
NEW_PAGE 144
NEXT_QUALIFIED_AREA 109
NO_DELAY 32
Node 8, 13
Node attributes 82
Node handle 8, 31
Node management 31
NODE_ATTRIBUTE_ITERATE 71
NODE_DEFINITIONS 31
NODE_ITERATOR 57, 58
NODE_KIND 32, 58
NODE_MANAGEMENT 31, 33
NODE_TYPE 31, 45
Non-existing node 8, 15
Non-hierarchical category 26
NUMBER_OF_ELEMENTS 124

Object 8, 21, 27
Obtainable 8, 45
OPEN 33, 38, 107, 111, 115
Open a node handle 33
Open node handle 8, 31
OPEN_FILE_NODE 121
OUT_FILE 144, 152

Page terminal 101
PARAMETERS 80
Parent 8, 15
Parent node 15
Path 8, 17
Path element 8, 17
PATH_ATTRIBUTE_ITERATE 72
PATH_KEY 44
PATH_RELATION 44
Pathname 8, 17
Permanent member 8, 22
PERMANENT_MEMBER 22
Position 8, 130, 191, 209
POSITION_BY_NAME 200
POSITION_BY_VALUE 202, 206, 212, 215
POSITION_TYPE 145
Potential member 8, 22
POTENTIAL_MEMBER 22
Pragmatics 9, 29
Precede 130
Predefined attributes 82
Predefined relations 15, 80
Primary relationship 9, 15
PRIMARY_KEY 42
PRIMARY_NAME 42
PRIMARY_RELATION 43
PRINTABLE_CHARACTER 169, 171
PRINTABLE_CHARACTERS 166
PRIVILEGE_SPECIFICATION 74

Process 9, 13
Process node 9, 13, 16
Process nodes 79
Process state transitions 81
PROCESS_CONTROL 33, 82
PROCESS_DEFINITIONS 81
PROCESS_STATUS 81
Program 9
PUT 138, 151, 189

Qualified area 9
Qualified areas 166
Queue 9, 13
Queue file 100, 101
QUEUE_KIND 101

READ_LABEL 188
Reading relationships 20
REEL_NAME 175
Relation 9, 14
Relation name 9, 15, 19
RELATION_NAME 32, 45
Relationship 9, 13
Relationship key 9, 15
RELATIONSHIP_KEY 32, 45
Relevant grant items 9, 23
REMOVE_AREA_QUALIFIER 188
RENAME 51
Renaming nodes 33
REPLACE 201, 205, 208, 211, 214
Resulting right 24
RESULTS 80
RESUME_PROCESS 95
REWIND_TAPE 182
Role 9, 22
Root process 16
Root process node 9, 16

Scroll terminal 101
Secondary relationship 9, 15
Secondary storage file 100, 101
Security level 9, 28
SECURITY_VIOLATION 29, 32
SELECT_ENUMERATION 145
SELECT_GRAPHIC_RENDITION 164
SET_ACCESS_CONTROL 74
SET_CURRENT_NODE 60
SET_ECHO 137, 152
SET_ERROR 119
SET_EXTRACT 198
SET_INPUT 118
SET_LOG 122
SET_NODE_ATTRIBUTE 66
SET_OUTPUT 119
SET_PATH_ATTRIBUTE 67
SET_POSITION 130, 145, 168
SET_PROMPT 124
SET_TAB 133, 148
SKIP_TAPE_MARKS 182
Solo queue 101
Source node 9, 14
Spawn 80
SPAWN_PROCESS 33, 82

SPLICE 197
STANDARD_ERROR 80
STANDARD_ERROR 120
STANDARD_INPUT 80
STANDARD_OUTPUT 80
Start position 9, 165
START_TIME 98
STATE_OF_PROCESS 92
STATUS_ERROR 32, 145
Structural node 9, 13
Structural nodes 77
STRUCTURAL_NODES 31
Subject 10, 21, 27
SUSPEND_PROCESS 94
SYNCHRONIZE 122
System-level node 10, 15

TAB 134, 149
TAB_ENUMERATION 145
TAPE_MARK 175
TAPE_POSITION 175
TAPE_STATUS 181
Target node 10, 14
Task 10, 13, 14
Terminal file 100, 101
TERMINAL_KIND 101
TERMINAL_SIZE 132, 147, 172
Terminated 79
Termination of a process 10
TERMINATION_KEY 172
To identify 18
To obtain access 20
TO_LIST 194
TO_TEXT 195, 204
TO_TOKEN 203
Token 10
TOKEN_TYPE 192
Tokens 192
Tool 10
Top-level node 10, 15
Track 10
Tracking 31
Traversal of a node 10, 18
Traversal of a relationship 10
Traverse a relationship 15

UNADOPT 77
Un'que primary path 10, 17
Unique primary pathname 10, 18
UNLINK 56
UNLOAD 179
Unnamed item 10, 191
Unnamed list 10
Unobtainable 10, 15
USE_ERROR 32, 145
User 10, 15
User name 10, 15

VOLUME_HEADER 183
VOLUME_IDENTIFIER 178
VOLUME_STRING 175

Write Intent 33

WRITE_RESULTS 00
WRITE_TAPE_MARK 183

Custodians:

Army -

Navy -

Air Force -

Preparing activity:

(Project IPSC/ECRS 0208)

Review activities:

Army -

Navy -

Air Force -

User activities:

Army -

Navy -

Air Force -

Agent:

Postscript : Submission of Comments

For submission of comments on this proposed MIL-STD-CAIS, we would appreciate them being sent by ARPANET/MILNET to the address

CAIS-COMMENT at ECLIR

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

Patricia Oberndorf
Naval Ocean Systems Center
Code 423
San Diego, CA 92152-5000

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 MIL-STD-CAIS
!topic ...
!rationale ...

The section line includes the section number, the paragraph number enclosed in parentheses, 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 a comment from a previous version:

!section 03.02.01(12)A. Gargaro 82-04-26

The version line, for comments on the current document, should only contain "MIL-STD-CAIS". 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"

As a final example here is a complete comment:

!section 03.02.01(12)A. Gargaro 85-01-15
!version MIL-STD-CAIS
!topic FILE NODE MANAGEMENT
Change "component" to "subcomponent" in the last sentence.

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.

STANDARDIZATION DOCUMENT IMPROVEMENT PROPOSAL (See Instructions - Reverse Side)	
1. DOCUMENT NUMBER	2. DOCUMENT TITLE
3a. NAME OF SUBMITTING ORGANIZATION	4. TYPE OF ORGANIZATION (Mark one) ☐ VENDOR ☐ USER ☐ MANUFACTURER ☐ OTHER (Specify): _____
5. ADDRESS (Street, City, State, ZIP Code)	
6. PROBLEM AREAS	
a. Paragraph Number and Wording:	
b. Recommended Wording:	
c. Reason/Rationale for Recommendation:	
7. REMARKS	
7a. NAME OF SUBMITTER (Last, First, MI) - Optional	7b. WORK TELEPHONE NUMBER (Include Area Code) - Optional
8. MAILING ADDRESS (Street, City, State, ZIP Code) - Optional	9. DATE OF SUBMISSION (YYYYMM)

DD FORM 1426
20 MAR 1976

PREVIOUS EDITION IS OBSOLETE

DAT
FILM
5