AD-A160 355   KAPSE (KERNEL ADA PROGRAMMING SUPPORT ENVIRONMENT)   1/4
              INTERFACE TEAM PUBLIC REPORT VOLUME 5(U) NAVAL OCEAN
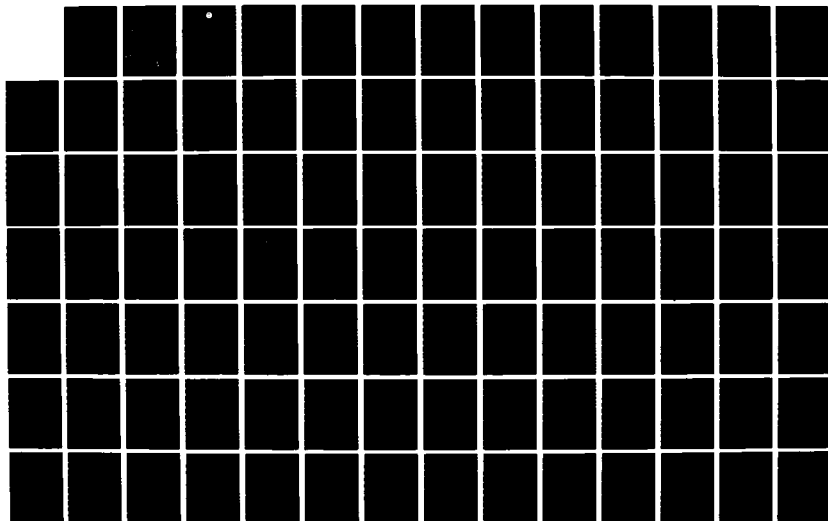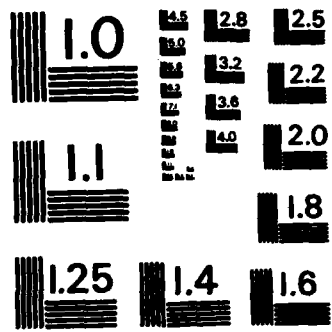   SYSTEMS CENTER SAN DIEGO CA P A OBERNDORF AUG 85

UNCLASSIFIED NOSC/TD-552-VOL-5   F/G 9/2   NL

MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

⑩

AD-A160 355

Technical Document 552

August 1985

KAPSE INTERFACE TEAM PUBLIC REPORT

Volume V

Patricia A. Oberndorf

Prepared for
ADA Joint Program Office

Naval Ocean Systems Center

San Diego, California 92152-5000

DTIC
ELECTE
S
OCT 1 6 1985
D
E

DTIC FILE COPY

85 10 15 062

ADMINISTRATIVE INFORMATION

This report was compiled by the Software Engineering Technology Branch (Code 423) of the NAVOCEANSYSCEN for the ADA Joint Program Office.

PK

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
UNCLASSIFIED	

2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	Approved for public release; distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NOSC TD 552	

6a. NAME OF PERFORMING ORGANIZATION	6b. OFFICE SYMBOL (if applicable)	7a. NAME OF MONITORING ORGANIZATION
Naval Ocean Systems Center	Code 423	

6c. ADDRESS (City, State and ZIP Code)	7b. ADDRESS (City, State and ZIP Code)
San Diego, CA 92152-5000	

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADA Joint Program Office		

8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	Agency Accession
The Pentagon, 3D139 Washington, DC 20301-3081	63226F	CS22	0	DN288 534

11. TITLE (Include Security Classification)

KAPSE INTERFACE TEAM PUBLIC REPORT, Volume V

12. PERSONAL AUTHOR(S)

Patricia A. Oberndorf

13a. TYPE OF REPORT	13b. TIME COVERED		14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Jul 84	TO Aug 85	August 1985	334

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	CAIS
			ADA Programming Support Environment (APSE)

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

This report gives the minutes of the April 1984, the July 1984, and the October 1984 KAPSE Interface Team. The report also includes the details of the Second CAIS Review Meeting, August 1984.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
[X] UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL
Patricia A. Oberndorf	(619) 225-6682	Code 423

DD FORM 1473, 84 JAN

83 APR EDITION MAY BE USED UNTIL EXHAUSTED
ALL OTHER EDITIONS ARE OBSOLETE

CONTENTS

SECTION I

INTRODUCTION

INTRODUCTION

This report is the fifth in a series that is being published by the KAPSE Interface Team (KIT). The first was published as a Naval Ocean Systems Center (NOSC) Report, TD-209, dated April 1982, and is now available through the National Technical Information Service (NTIS) for $19.50 hardcopy or $4.00 microfiche; ask for order number AD A115 590. All subsequent issues of the public reports are being issued as volumes of NOSC TD-552. The second public report is dated October 1982 and is now availabe through NTIS for $44.50 hardcopy; ask for order number AD A123 136. The third is dated October 1983 and is available through NTIS; ask for order number AD A141 576. The fourth is dated 30 April 1984 and will be available through NTIS. This series of reports serves to record the activities which have taken place to date and to submit for public review the products that have resulted. The reports are issued approximately every six months. They should be viewed as snapshots of the progress of the KIT and its companion team, the KAPSE Interface Team from Industry and Academia (KITIA); everything that is ready for public review at a given time is included. These reports represent evolving ideas, so the contents should not be taken as fixed or final.

MEETINGS

During this reporting period the two teams met jointly in July 1984 in Toronto, Canada, and in October 1984 in Merrimack, New Hampshire. The approved minutes from the April, July and October 1984 meetings are included in this report. As usual, some of the working groups have also held individual meetings between regular KIT/KITIA meetings.

COMMON APSE INTERFACE SET (CAIS)

Since one of the frequently-heard comments on the CAIS was that there was insufficient public involvement in its review, it was decided to hold two more public reviews before the scheduled delivery of a CAIS Version 1 document to the AJPO. Version 1.3 of the CAIS was drafted by the CAISWG and distributed in August in conjunction with a second Public Review meeting on 2 August 1984 in Hyannis, Massachusetts, that was held immediately following the AdaTEC meeting there. About 100 people participated and provided the CAISWG with many useful comments. A summary of this meeting and the comments made is included in this report.

The following day the key CAIS designers met with Dr. Robert Mathis, director of the AJPO. It was decided that, based on the results of the review the previous day, the work would proceed on schedule with a January 1985 delivery to the AJPO of a document which would be put forward for standardization by the three services. It was also decided to hold an additional public review meeting in November in conjunction with the AdaJUG/SIGAda meeting scheduled for Washington, D.C. In response to this, the CAISWG held a special meeting in October 1984 to produce an additional revision, CAIS 1.4, of the interface set. That version is included in this report.

It was also decided in this time period that the CAIS Version 2 which is due for delivery to the AJPO in January 1987 should be developed by a contractor and that the contractor would be selected competitively, although

it was determined that parallel competitive contracts such as were used to develop Ada would not be pursued. A Request for Procurement was prepared by NOSC and announced in the Commerce Business Daily in September. Award of that contract is expected in 1985.

REQUIREMENTS AND DESIGN CRITERIA (RAC)

Work on the RAC has now produced a version which has been accepted by the KIT and KITIA for baselining; it now includes all sections anticipated for the document. That means that the groups have agreed to the contents in concept. The October version of the complete RAC document is included in this report. It includes slight revisions of some previously baselined sections, a significant re-write (more in terms of style than content) of the previously baselined section on processes and newly baselined versions of the sections on the database and on the input and output.

This document is considered to be quite important to the future work of the KIT and KITIA as it encompasses the requirements to which Version 2 of the CAIS is to be designed. It is important that this receive considerable public review before work on Version 2 proceeds too far. All comments on that part of this report are especially welcome.

STANDARDS (STANDWG)

Because of the small number of members involved in the Standards Working Group (STANDWG), it was decided to incorporate its charter with that of the Compliance Working Group (COMPWG). Reviews of standards that may be of interest, such as the one included in this report, will continue to be produced, however.

POLICY DISCUSSIONS

Policy issues have continued to be a subject of extreme concern to the members of the KIT and KITIA. These concerns evoked serious discussions at the July meeting regarding CAIS standardization and how the DoD should proceed regarding derivation of CAIS Version 2. These discussions culminated in an invitation to Dr. Mathis and the three military service program managers to join the teams for a discuon at the October meeting. Some of these concerns and KITIA recommendations with regard to them are reflected in the KITIA proposal for a second CAIS Version 2 contract, which is included in this report.

I&T TOOLS

The work to implement the APSE Interactive Monitor (AIM) has continued with the acquisition of a Data General (DG) system running the Ada Development Environment (ADE). The completion of this work will result in an AIM implementation on the DG and is expected in June 1985.

KIT/KITIA PAPERS

Only one paper is included in this report which was generated by an individual member of the KIT or KITIA. This paper, by Dr. Chris Napjus of the KIT, articulates some of the ideas which have been discussed concerning the concept of a standard MAPSE (i.e., one minimal set of tools which are ALWAYS available identically on ALL APSEs).

OTHER KIT/KITIA ACTIVITIES

The planning activities for the Ada Run-Time Environment Working Group (ARTEWG) have continued and the attendance has been increasing. There is growing interest in the community in issues which this group is addressing.

The CAIS Implementor's Group (CIG) held its first meeting in June in San Diego. Rebecca Bowerman (MITRE) was elected chairperson, and the group made various decisions regarding how it would consider the addition of members and what its goals and mode of business would be. A subsequent meeting was held in Hyannis in conjunction with the AdaTEC meeting there in August and another meeting was planned for the November AdaTEC meeting in Washington, D.C.

CONCLUSION

This Public Report is provided by the KIT and KITIA to solicit comments and feedback from those who do not regularly participate on either of the teams. Comments on this and all previous reports are encouraged. They should be addressed to:

Patricia Oberndorf
Code 423
Naval Ocean Systems Center
San Diego, CA 92152-5000

or sent via ARPANET/MILNET to POBERNDORF@ECLB.

SECTION II

TEAM PROCEEDINGS

KIT/KITIA MINUTES
9-12 APRIL 1984
SEATTLE, WA

ATTENDEES: SEE APPENDIX A
HANDOUTS: SEE APPENDIX B
WORKING GROUP MEMBERS: SEE APPENDIX C
PLANNING GROUP MEMBERS: SEE APPENDIX D

9 APRIL 1984 - KITIA MEETING

1. OPENING REMARKS

- Herm Fischer, KITIA chairperson, brought the meeting to order. All
  were welcomed.

2. GENERAL BUSINESS

- Herm Fischer gave a presentaion of an Onion Skin Model to add to
  discussion of a layering to the KITIA. A CODSIA Report was made
  available to explain the model.

- Discussion of the layered KITIA took place, and it was decided that
  the two of Judy Kerner and Eli Lamb would be formalized and
  presented later in the meeting.

- The question of a Run Time working group made up of an independent
  team was brought up. This team's objective would be to look into
  the issues of Run Time standards, not necessarily taking as a premise
  that we have a Run Time Standard.

- Tricia announced to the guests the named and number working groups
  and encouraged the guests to join whichever group they seemed most
  interested in.

3. KITIA MEETING ADJOURNED

10 APRIL 1984 - JOINT KIT/KITIA MEETING

1. OPENING REMARKS

- Tricia Oberndorf, KIT chairperson, brought the meeting to order.

- New replacement members and visitors were introduced. Bill Barry of
  FCDSSA, San Diego is here to replace George Robertson. Tom Conrad
  from NUSC is Ron House's replacement. Cheng-Chi Huang of Hughes
  is here instead of Jim Ruby. Geoff Fitch from Intermetrics is here
  instead of Jim Moloney. New members are Tony Steadman of ESD, his
  alternate Bob Munck from MITRE and Susan Good from NOSC with Tricia.
  Visitors/Guests are Ben Brosgol of Alsys, Mike Dolbec of Rational,
  Larry Druffel of Rational, Gerry Fisher of CSC, Sylvester Fernandez
  of Sperry, Dean Herington of Data General and Charles McKay of NASA/
  University of Houston at Clear Lake.

2. GENERAL BUSINESS

- Status of the TI, TRW contracts and E&V task were presented. Herm took the floor to summarize the KITIA meeting of Monday, 9 April 1984. A discussion of the Onion Model and a layered KIT/KITIA were presented. Anthony Gargaro was also appointed temporary chair of the Run Time Planning Group.

3. NAMED WORKING GROUP REPORTS

- Jack Kramer of the CAISWG discussed that CAIS version 1.2 would be delayed. Accomplishments are work done on mandatory and discretionary security and distribution/work stations. Projected work includes changes due to comments. CAIS Version 1.2 will be due next quarter and will include changes from 1.1 to present. At next meeting plans are to discuss remaining items for 1.3. A CAISWG forum/workshop is being planned for 2 August at the Hyannis AdaTEC.

- Hal Hart of the RACWG discussed the RAC's approval as an action item for the meeting. Accomplishments are the rewrite of the RAC(17 Feb) based on the January meeting, the development, distribution, collection and analysis of the 17 responses, and the revision of the document 23 March 1984. Unresolved problems are sections 4 and 6 Projected work is the approval of section 4 and 6, to begin the rationale for the RAC, and begin evaluation of the CAIS against the RAC. Item due next quarter is a complete approved RAC document. Presentation planned for next meeting is status of the document. Discussion of procedures for approving/disapproving the RAC then took place. Tricia also expressed her disappointment in only 17 responses to the earlier draft.

- Ann Reedy of STONEWG discussed items due this quarter. She reported that a decision was made to make a preliminary analysis of the STONEMAN document. There is currently a draft of these analyses. The unresolved problem is primarily a long term work schedule. Projected work includes a cross reference for the RAC and a presentation by Frank Belz of TRW's PA-APSE work at the next meeting. Everything else is TBD.

- Ron Johnson and members of the GACWG are currently working on a draft of 4 chapters of the document. They have come to find out that after looking at the users guide to transportability there is more than the CAIS to making a transportable tool set. The more they investigate the more they find out they need to look into. The group is in hopes of having a complete draft of a Guidelines And Criteria document by December, entitled "A User's Guide to Ada Transportability."

- Rudy Krutar of the DEFWG stated the combined glossary is due this quarter. Drafts are available. Accomplishments are an established definitions pool, a draft of the CAIS glossary, and a combined glossary for the RAC and CAIS. Unresolved problems are moving the definitions pool to ECLB, and the KIT/KITIA review of the combined glossary. Projected work for next quarter is an additon of the STONEMAN terms to the glossary and the open definitions pool for the KIT/KITIA. Items due next quarter are TBD. Presentations planned for next meeting are how to use the definitions pool. Dission of a glossary for every document and format for definitions followed.

- Bernie Abrams of the STANDWG stated that a draft charter is due this quarter. Accomplishments this quarter are review of the Meta Specs, DOD 4120.3 and MIL STD 962. Unresolved problems are CAIS compliance to MIL STD 962 and personnel. Clarification was made as to the role of the STANDWG: to make an "offical" document and not to make sure everyone builds a standard CAIS. Work for next quarter is to review the CAIS for compliance and review existing standards for conflict.

- Tim Linquist of COMPWG stated that this group is still in the formulation stage. There are no real items due this quarter. Activities are the relationship between RAC and the CAIS and the semantics of the CAIS. Unresolved problems are taking the charter of the COMPWG and defining a set of deliverables, deciding what standards relate and configuration management of the COMPWG activities/tracking different versions. Projected work is a deliverables list.

4. ANNOUNCEMENTS

- CM workshop last June - participants are asked to review their sections.

- Public Report IV closes 30 April. People with individual papers should give them to their working group chairs.

- CAIS comments and responses will be available the week of 16 April in KIT-INFORMATION.

- RADC is another source for the Public Report III, cost is 35 dollars.

- A proposal for the CAIS Implementors Group is in the works. A meeting is scheduled for early June. Tricia does not plan to head it. She just plans to have a kick-off meeting in hopes the group will organize and start itself.

- A Human Factors workshop is being hosted by IDA in early May by invitation only. Talk to Jack Kramer for more information.

- IEEE announced a conference in Ada Applications in Environments for mid October. Topics include education, programming techniques, design methodologies, environmental structures, and distributed environments. Papers due 15 May.

- Rocky Mountain Institute of Software Engineering is hosting a conference for 16-31 July. A complete brochure is available through Tricia.

- The CAIS made the Government Computer News, April 1984.

- The KITIA policy statement made the Language Control Facility Newsletter for the Ada Jovial Working Group.

- There will be a CAIS session at AdaEurope, Brussels this June.

- The question was asked if anyone would be interested in X3H1-the group working on the OSCRL document.

- An invitation to join under the CODSIA/OSCRL committee to work on model screen driven operation systems was announced.

5. MEETING SCHEDULE

1984
July 16-19   Toronto
October 1-4   New Hampshire/ San Francisco

1985
January   San Diego
April   Washington D.C.
July   San Francisco
September   Connecticut/Texas/United Kingdom (?)

1986
January   San Diego

6. NEW BUSINESS

- Tricia expressed her thanks from the group to Ron Johnson, the meeting host, on the nice arrangements.

- There was a proposal from Tim Lyons for a birds of a feather session on formal definitions of the CAIS. A group was formed and a meeting planned.

- Numbered Working Group Chairs were informed to contact Debbie Barba for mail forwarding directories.

7. ADDRESS FROM BRAIN SCHAAR

- Brian commended the group on its accomplishments and informed everyone of his moving more towards education and training and less involvement with the KIT/KITIA.

8. MORNING BREAK

9. REQUIREMENTS AND CRITERIA VOTE

- Discussion of approval of the document and format of the vote took place. The vote was taken in written ballot to the following statement.

    Resolved: This section of the RAC document substantially
    represents the correct set of requirements for
    the CAIS, and shall be placed under configuration
    management and in Public Report IV.

  The vote was taken in silent ballot for section 2,3,5 and 7 (4 and 6 were optional).

10. LUNCH

11. NAMED WORKING GROUP MEETINGS

WEDNESDAY, 11 APRIL 1984

12. NUMBERED WORKING GROUP MEETINGS

13. BREAK

14. NUMBERED WORKING GROUP MEETINGS

15. JOINT KIT/KITIA RAC DISCUSSION

● The RAC votes were presented, and the RAC document sections 2,3,5 and 7
will be placed under configuration management and in Public Report IV.
The sections 2,3,5 and 7 were approved with comments. Tricia stated
that she sees a conflict in complete and 90% I&T. The RAC right now reads
for complete I&T. She also strongly suggests that WG3 put up on the net
the differences between operating systems and interfaces.

● Section 7 was presented by Nick Baker, and it was decided that the
entire section disappear and be incorporated in other existing
sections.

● Section 5's name is changed to Execution Facilities. Presentation
and discussion of the initiation, termination, communication and
sychronization issues of this section took place. Also discussion of
the vote awhere that puts the document occured.

● Section 6 was also presented and discussed. Changes made and ideas to
add in the future to the document were discussed in each part of this
section.

● Section 4 was presented by Tim Lyons with discussion. Decisions made
by the group are on the following topics: basic levels of dynamic database
abstraction mechanisms, affirmation of object/attribute/relationship
approach, uni-and-bi-directional relationships, objects never die
(they just become inaccessible and relationships to these objects
become unaffected), detailing of relationship type, many-to-many
discussion, move security, processes in 5 should have relations to 4,
and other dbms's (as a measure of capability not as a requirement it
be used as such.) Issues to address are object/relationship/attribute
identity/select/operations, detailing of object, relationship and
attribute operations, transactions and dynamic access synchronization,
and integrity, back-up, archiving, and history.

THURSDAY, 12 APRIL 1984

16. UNIX AND THE CAIS: ISSUES AND DISCUSSION

● Eli Lamb first presented a Unix tutorial then went on to describe the
motivation for a Unix-based approach (timely, credible, and economical).
Rich Thall then spoke of the problems with using Unix as a base. He
brought up the issues as technical vs. administrative. Issues include
what definition is to be used and who supports the product.

● Eli went on to clarify the Unix System I/O and tasking and to describe
the problem of Unix's System Support for tasking.

- In summary the CAIS approach based on existing systems is attractive. Concerns are control and technical. The technical concerns are surmountable. The ALS and AIE are both reasonably compatible with the Unix-based approach. The issues are a schedule and the economics of the CAIS.

17. NAMED WORKING GROUP MEETINGS

18. KIT/KITIA SEPARATE MEETINGS

- At the KITIA meeting a discussion and vote took place as to the idea of a layered KITIA. Two view points were presented by Eli and Judy. The vote was taken, and it was decided to have NO change to the KITIA. It is assumed that the KITIA chair retains the power to invite guests on a limited basis.

- There was a review of the Onion Model to clarify the organization and role of the KITIA members. Herm asked for input on guests and will try to keep the number of guest between 4 and 6.

19. LUNCH

20. NUMBERED WORKING GROUP MEETINGS

21. KIT/KITIA WRAP UP AND CAIS STATUS

- The CAIS schedule was discussed and an updated version will be available on the ARPANET. Public Report IV papers are Fitch's papers on mandatory and discretionary access controls and Larry Yelowitz's paper on formal semantics.

- DEFWG stated that they are nearly at a consensus on Definitions for the CAIS to be in Public Report IV and the RAC document.

- STONEWG stated the draft of the definition for the term APSE for DEFWG and the status report of STONEWG will be put in Public Report IV.

- GACWG will have a GAC document for review in Toronto.

- STANDWG's report was more of their original report with the announcement of a new member, Tony Steadman, and discussion of their relation to the COMPWG.

- COMPWG accomplished their deliverables list. They plan to have a draft of objectives for the COMPWG in Toronto and one for the KIT/KITIA at the October KIT/KITIA meeting.

- RACWG discussed the appearance of the RAC in Public Report IV. A consensus was arrived at to let Tricia and Hal make public the RAC document with an explanation as to expected changes. The RAC schedule was then presented.

22. MEETING ADJOURNED

APPENDIX A
ATTENDEES
KIT/KITIA Meeting
9-12 April 1984

KIT Attendees:

BARBA, Debbie	TRW
BARRY, Bill	FCDSSA-SD
BELZ, Frank	TRW
CASTOR, Jinny	AFWAL/AAAF
CONRAD, Tom	NUSC
FERGUSON, Jay	DoD
FITCH, Geoff	Intermetrics
FROMHOLD, Barbara	CECOM
GOOD, Susan	NOSC
HARRISON, Tim	Texas Instruments
HART, Hal	TRW
JOHNSTON, Larry	NADC
KEAN, Elizabeth	RADC/COES
KRAMER, Jack	IDA
KRUTAR, Rudy	NRL
LINDLEY, Larry	NAC
MILLER, Jo	NWC
MUNCK, Bob	MITRE
MYERS, Gil	NOSC
MYERS, Philip	NAVELEX
OBERNDORF, Tricia	NOSC
PEELE, Shirley	FCDSSA-DN
SCHAAR, Brian	AJPO
STEADMAN, Tony	ESD/ALEE

TAFT, Tucker   Intermetrics

TAYLOR, Guy   FCDSSA-DN

THALL, Rich   SofTech

WILDER, Bill   PMS-408

KITIA Attendees:

ABRAMS, Bernard	Grumman Aerospace Corp.
BAKER, Nick	McDonnell Douglas Astronautics
BRYAN, Doug	Lockheed Missiles & Space Company
COX, Fred	Georgia Institute of Technology
DRAKE, Dick	IBM
FELLOWS, Jon	System Development Corp.
FISCHER, Herman	Litton Data Systems
FREEDMAN, Roy	Hazeltine Corp.
GALLAHER, Larry	ESS/SEL/DSD Georgia Tech
GARGARO, Anthony	CSC
GLASEMAN, Steve	Aerospace Corp.
HUANG, Cheng-chi	Hughes Aircraft Co.
HUMPHREY, Dianna	Control Data Corp.
JOHNSON, Ron	Boeing Aerospace Co.
KERNER, Judy	Norden Systems
LAHTINEN, Pekka	Oy Softplan AB
LAMB, J. Eli	Bell Labs
LINDQUIST, Tim	Virginia Institute of Technology
LYONS, Tim	Software Sciences Ltd.
PLOEDEREDER, Erhard	IABG West Germany
REEDY, Ann	PRC
RUDMIK, Andy	GTE
WESTERMANN, Rob	TNO-IBBC The Netherlands
WILLMAN, Herb	Raytheon Company
WREGE, Doug	Control Data Corp.
YELOWITZ, Larry	Ford Aerospace & Communications Corp.

KIT/KITIA Guests:

BROSGOL, Ben   Alsys

DOLBEC, Mike   Rational

DRUFFEL, Larry   Rational

FERNANDEZ, Sylvester   Sperry

FISHER, Gerry   CSC

HERINGTON, Dean   Data General

McKAY, Charles   NASA, University of Houston at Clear Lake

APPENDIX B - MEETING HANDOUTS

1. "DOD Requirements and Design Criteria for the Common APSE Interface Set", Draft 23 March 1984.

2. "Draft Specification of the Common APSE Interface Set(CAIS) Version 1.1.1", 9 April 1984.

3. "CODSIA, Council of Defense and Space Industry Associations Report 13-82, Volume II, Background and Issues, DOD Management of Mission-Critical Computer Resources", March 1984.

APPENDIX C - NAMED WORKING GROUP MEMBERS

CAISWG Members:

BARBA, Debbie	TRW
FISCHER, Herm	Litton Data Systems
FITCH, Geoff	Intermetrics
FROMHOLD, Barbara	CECOM
GOOD, Susan	NOSC
GOUW, Bob	TRW
GRANT, Brenda	IDA
HARRISON, Tim	Texas Instruments
HOLLISTER, John	CECOM
KEAN, Elizabeth	RADC/COES
KRAMER, Jack	IDA
LAMB, J. Eli	Bell Labs
LYONS, Tim	Software Sciences Ltd.
McGONAGLE, Dave	GE CR&D
MORSE, H.R.	Frey Federal Systems
OBERNDORF, Tricia	NOSC
PLOEDEREDER, Erhard	IABG West Germany
SCHAAR, Brian	AJPO
TAFT, Tucker	Intermetrics
THALL, Rich	SofTech
WILDER, Bill	PMS-408
WILLMAN, Herb	Raytheon Company
YELOWITZ, Larry	Ford Aerospace & Communcations Corp.

COMFWG Members:

BARRY, Bill   FCDSSA-SD

CASTOR, Jimmy   AFWAL/AAAF

COX, Fred   Georgia Institute of Technology

DRAKE, Dick   IBM

FREEDMAN, Roy   Hazeltine Corp.

FOIDL, Jack   TRW

LINDQUIST, Tim   Virginia Institute of Technology

PEELE, Shirley   FCDSSA-DN

DEFWG Members:

BAKER, Nick   McDonnell Douglas Astronautics

KERNER, Judy   Norden Systems

KRUTAR, Rudy   NRL

GACWG Members:

DUDASH, Ed   NSWC/DL

FRENCH, Stewart   Texas Instruments

JOHNSON, Ron   Boeing Aerospace Co.

LINDLEY, Larry   NAC

WALTRIP, Chuck   John Hopkins Univ.

MAGLIERI, Lucas M.   National Defense Hqds.

RUDMIK, Andy   GTE Network Systems R&D

RACWG Members:

CORNHILL, Dennis   Honeywell/SRC

FELLOWS, Jon   Systems Development Corp.

GARGARO, Anthony   CSC

HART, Hal   TRW

HUANG, Cheng-chi   Hughes Aircraft Co.

KOTLER, Reed   Lockheed Missiles & Space

MILLER, Jo   NWC

MUNCK, Bob   MITRE

MYERS, Philip   NAVELEX

OBERNDORF, Tricia   NOSC

SIBLEY, Edgar   Alpha Omega Group, Inc.

WESTERMANN, Rob   TNO-IBBC
                      The Netherlands

WREGE, Doug   Control Data Corp.

STANDWG Members:

ABRAMS, Bernie   Grumman Aerospace Corp.

LOPER, Warren   NOSC

STEADMAN, Tony   ESD/ALEE

STONEWG Members:

BELZ, Frank   TRW

CONRAD, Tom   NUSC

FERGUSON, Jay   DoD

GLASEMAN, Steve   Aerospace Corp.

HART, Hal   TRW

JOHNSON, Doug   SoftWrights Inc.

JOHNSTON, Larry   NADC

MYERS, Philip   NAVELEX

REEDY, Ann   PRC

SAIB, Sabina   General Research Corp.

APPENDIX D - PLANNING GROUP MEMBERS

RUNPG Members:

ABRAMS, Bernard	Grumman Aerospace Corp.
BAKER, Nick	McDonnell Douglas Astronautics
BELZ, Frank	TRW
BROSGOL, Ben	Alsys
DRAKE, Dick	IBM
DRUFFEL, Larry	Rational
FELLOWS, Jon	System Development Corp.
FERNANDEZ, Sylvester	Sperry
FISCHER, Herman	Litton Data Systems
FISHER, Gerry	CSC
FREEDMAN, Roy	Hazeltine Corp.
GARGARO, Anthony	CSC
HART, Hal	TRW
HERINGTON, Dean	Data General
HUANG, Cheng-chi	Hughes Aircraft Co.
JOHNSON, Ron	Boeing Aerospace Corp.
JOHNSTON, Larry	NADC
KERNER, Judy	Norden Systems
MILLER, Jo	NWC
MUENNICHOW, Isabelle	TRW
MYERS, Philip	NAVELEX
WILDER, Bill	PMS-408

Minutes
of the
KIT/KITIA Meeting
16-19 July 1984
Toronto, Canada

ATTENDEES: See Appendix A
HANDOUTS: See Appendix B

16 July 1984

1. Numbered Working Group Meetings 0800-1200

2. Named Working Group Meetings 1300-1700

16 July 1984 - KITIA MEETING

1. Herm Fischer, KITIA chairperson, brought the meeting to order.

2. Herm reviewed the status of the KITIA which he believed was about half complete regarding its original charter. The KITIA may want to identify new goals for future activity.

3. A discussion of Gerry Fisher's comments in Ada LETTERS followed. Although a time warp was evident between when the article was written and when it was published, the KITIA felt the concerns regarding the ALS were still valid. The CAIS has made steady progress but the focus is still in the paper realm.

4. GENERAL BUSINESS

   ● There has been no KITIA objection to the thirty member limit.

   ● The KITIA chair requests feedback on the need for an implementors board.

   ● Olivier Roubine will be joining the KITIA as a representative of the EEC. Teledyne has resigned their seat on the KITIA. Mike Kamrad is the Honeywell representative and Cheng-Chi Huang is present for Hughes Aircraft.

   ● Anthony Gargaro summarized the background and status of the Run-Time Planning Group (RLNPG) which will meet the day after the KIT/KITIA meeting. They expect to have a preliminary report for the next SIGAda meeting. The emphasis of this meeting is to establish a charter and goals.

   ● A CAIS Implementor's Group held their initial meeting in San Diego. The group is composed of companies and organizations that may implement the CAIS. Rebecca Bowerman was elected as the Chairperson of the group at this first meeting

5. KITIA CAIS Policy and Direction

   ● Eli Lamb offered some considerations regarding the CAIS. With the Second Public Review of the CAIS scheduled for Hyannis the KITIA may want to examine the CAIS from the perspective of the AIE, the ALS, or some compilers for a different perspective. Fred Cox expressed concern that the DoD will not listen to offered criticism/opinions and will go ahead with MIL-STD 1 and 2 in direct evolution from Versions 1.1,1.2 but without significant changes. The KITIA should consider drafting a recommended policy for consideration by the AJPO.

● The present policy is vague in certain areas and totally lacking in others. For example, a CAIS that is "upwards compatible" would be great but do we know enough now to define a reasonable base for the next ten years? How do we go from our old systems to Ada? Are subsets allowed and how can compliance be established for subsets? The lack of working prototypes is a significant issue: we need operational experience. Additional concerns within the Ada community are the level to which the interfaces are defined, the impact of the CAIS on the overall Ada Program (will it slow down the utilization of Ada), how to get industry to implement the CAIS, and if in fact a CAIS is technically and politically viable.

● Edgar Sibley stated if we do not have a Standard soon we will have many problems in Ada with interoperability.

● Tim Lyons presented the STONEMAN concept of building the environment around the database. Without a database concept the CAIS is fairly conventional. Since we understand conventional systems it is feasible to standardize the CAIS. But since we have no experience on environments built around a database we really don't know how it should look or work in practice and we desparately need experimentation in this area. Standardization normally builds on existing implementations of which we have none so the CAIS becomes a design exercise, not just a standardization effort. Since the DoD is going to standardize the CAIS the KITIA should try to make it as good as it can possibly be.

● Tricia Oberndorf reviewed the DoD development schedules of the AIE, the ALS, and the ALS/N, the STARS concern for near-term interfaces and VHSIC commitments to a full CAIS. The CAIS development and standardization effort becomes a focal point for these various programs.

● Eli Lamb presented some alternatives for consideration including:
  - definition of an evolutionary path to a standard CAIS
  - wait for more prototypes for evaluation
  - start over at a lower level or use an existing base such as UNIX
  - define subsets for piggy-back implementations

6. Herm Fischer prepared a KITIA ballot for the following issues. KITIA results are included.

● Should the CAIS become a Military Standard in January 1985?
  Yes - 9 No - 3 Later - 9

● Should the KITIA offer to sponsor the ARTEWG/RUNPG?
  Yes - 13 No - 7 Abstain - 2

● Should the KITIA lobby for a "CAIS'ed" UNIX in quasi-public domain (a la Berkley)?
  Yes - 18 NO - 3 Abstain - 1

● Should the CAIS become a Military Standard in January 1985?
  Yes - 5 No - 17 Abstain - 1

● Should the CAIS become a Draft MIL-STD in January 1985?
  Yes - 17 No - 6 Abstain - 0

7. MEETING ADJOURNED

17 July 1984 - Joint KIT/KITIA Meeting

1. Tricia Oberndorf, KIT chairperson, brought the meeting to order.

2. GENERAL BUSINESS

• The following new members of the KITIA were welcomed: Michael Horton (SDC), Mike Kamrad (Honeywell), and Charlie Pow (Lockheed). Paul Riley (Data General) and Kathy Gilroy (Harris) were invited guests.

• Texas Instruments is responding to an RFQ for extension to allow the AIM implementation to proceed using the Ada Development Environment. CAIS type interfaces are also being analyzed. A contract mod is in progress to TRW to continue support until the RFP for the NOSC support contract is available. The CAIS Version 2 Design RFP is in contracts for release for a potential January award.

• Herm Fischer summarized the results of the preceding night's KITIA meeting including the results of the KITIA ballot (presented above).

• Jinny Castor reported on the Evaluation and Validation program (see handout). The E&V Quarterly report is now available. A procurement for CAIS Validation work is expected in August.

• Ada Europe is producing a MAPSE selection guide following the format of the compiler selection guide. The MAPSE selection guide may be available by the end of the year.

3. WORKING GROUP REPORTS

• RACWG
  - Sections 2&3 received abot 90% approval and sections 5&7 about 70% approval at the April meeting. Sections 4&6 are expected to be baselined this quarter. Draft rationales for Sections 4&5 are in progress. Expect a votable RAC for the next meeting.

• CAISWG
  - Expect a Version 1.3 to be completed in this quarter. Mandatory and discretionary access control have been added. The CAIS has been reformatted to comply with MIL-STD document format. Better semantics have been provided with additional clarification of the node and process models. Responses to previous CAIS comments are being formulated. A second Public Review of the CAIS is scheduled for Hyannis at the SIGAda meeting. A CAIS Panel was conducted at the Ada Europe meeting during which valuable feedback was received.

• DEFWG
  - Drafted and submitted for review a draft Glossary for use in KIT/KITIA documents which will be published in the Public Report. The definitions pool will be moved to ECLB for KIT/KITIA access. Additions to the pool to include STONEMAN cited terms will be added in the next quarter.

• STONEWG
  - A STONEMAN Analysis Report and a Status Report were provided for the Public Review. The Status Report contains an outline for a revised STONEMAN document. A copy resides on the KIT-INFORMATION account and comments are requested.

• STANDWG
  - Reviewed existing standards for comparison to the CAIS. Expect future work to be merged with the COMPWG efforts.

● COMPWG

    - Preparing a draft CAIS versus RAC analysis. Plan to define a CAIS semantics
      description technique, develop a Guidelines and Conventions for implementors, and
      complete a STONEMAN versus RAC analysis.

● GACWG

    - Drafted "A User's Guide to Ada Transportability" outline. Expect to have a draft for
      review for the October meeting. Continuing problem of availability of personnel.

4. GENERAL ANNOUNCEMENTS

● Judy Kerner advised the KIT/KITIA that IEEE has a working group on Ada as a PDL. A draft of
their work is expected to be available at the August SIGAda meeting.

● Hal Hart reminded the members about the Future APSE Workshop scheduled for Santa Barbara
in September and that AdaTEC has been elevated to a Special Interest Group and is now SIGAda.

● Tricia Oberndorf reported on the following topics:

    - The latest Public Report is in reproduction for distribution.

    - As a result of the May Tri-Service review the STARS program is looking to strengthen
      the area of software reliability and may have FY-85 funding to support this effort.

    - A CAIS Implementors Group (CIG) has been formed and held its first meeting in San
      Diego (hosted by Data General). Rebecca Bowerman (MITRE) was elected
      chairperson and plans future meetings in Hyannis and Washington D.C. This group
      expects to generate some point papers for AJPO based on the results of their
      implementation efforts. This group is independent of the KIT/KITIA and is basically
      composed of CAIS implementors. Contact R. Bowerman for additional details.

    - Hank Steubing of the JSSEE will make available their "Operational Concept Document"
      for review and feedback.

    - A second Public Review of the CAIS will be held at Hyannis, MA with emphasis on
      feedback from the audience on the major features of the document such as the node
      model, process, I/O, and non-technical issues. Working groups in each of these areas
      will be formed to address specific topics and formulate recommendations. The emphasis
      will focus on the concepts rather than specific details.

    - There will be a meeting of the Runtime Planning Group Thursday evening and Friday;
      contact A. Gargaro for details as attendence is limited.

    - Bob Mathis and the AJPO are interested in comparing the CAIS and different operating
      systems; any analysis previously performed is welcomed.

    - The future meeting schedule stands as:
      - 1984 Oct. 1-4 in Merrimac NH
      - 1985 Jan. 14-17 in San Diego
              Apr. 15-18 in Washington D.C.
              Jul. 8-11 in San Francisco area
              Sep. 23-26 in Rhode Island

- 1986 Jan. 13-16 in San Diego
    Apr. 14-17 in Atlanta
    Jul. 7-10 to be determined (Schenectady, NY)
    Sep. 22-25 to be determined (Twin Cities, MN)
- 1987 Jan. 19-22 San Diego

4. BREAK

5. TECHNICAL REPORTS

● Tom Conrad (NUSC) reported on the status of the Joint Service Software Engineering Environment plans for development of a Plan of Action and Milestones (POAM) and an Operational Concept Document (OCD) as the high level planning documents for the Software Engineering Institute (SEI). The SEI, which is expected to be announced in the fall, will be responsible for implementing the requirements formulated by the JSSEE.

● Dennis Cornhill (Honeywell) presented results of their experiences with Ada for distributed targets. Their goal was a realization of an "Ada machine" through the integration of a number of heterogeneous processors. The methodology was to complete a detailed partitioning of software prior to the detailed design phase. The partitioning was based on packages, subprograms, named blocks, tasks, objects (variables, constants, etc.) and instantiations of generic units. It was found that a preliminary functional partitioning was required before the detailed design phase. Configuration control during the functional allocation phase was maintained by basing the configuration change on the accepted partitioning change. The partitioning was made on application functional boundaries.

● Bernie Abrams (Grumman) reported on a survey conducted for STANDWG to see if the CAIS is conflicting or redundant with existing standards. The results were that although there are a number of organizations that issue standards (ANSI, ANS, DoD, FIPS, IEEE, MIL-STD) there is presently no standard comparable to the CAIS. The closest document may be the ANSI 3XH1 Operating System Command & Response Language effort that is ongoing.

● Rudy Krutar reported on the construction of a schema for maintenance of definitions in a pool for the DEFWG. Queries can be obtained for sources which will list terms or for the glossary which lists the glossary. A standard data entry format has been identified.

6. BREAK FOR LUNCH.

7. RAC DISCUSSION.

● Tim Lyons (Software Sciences Ltd.) gave presentations on the level of the CAIS interfaces and the merger of database and filing system concepts. A transcript of these presentations is to be made for KIT/KITIA availability.

● Hal Hart (TRW) summarized the Requirements and Criteria (RAC) efforts and distributed a copy of the latest RAC Section 4 contents. Issues for consideration include exact names versus exact identifiers, data integrity area definition and consistent terminology.

8. REORGANIZE INTO WORKING GROUPS.

Wednesday, 18 July

9. CONTINUE WORKING GROUPS

10. CONTINUE RAC DISCUSSIONS

- Frank Belz (TRW) summarized current requirements for data management. These include
    - a means of retaining data
    - a means of creating and operating on data
    - a description of data (which may be operated upon)
    - a separation of relationships and properties from both their existence and the tools that operate on them
    - a way to develop new data by inheriting properties of existing data

11. ADJOURN FOR DAY

Thursday, 19 July

12. CONTINUE WORKING GROUP MEETINGS.

13. TECHNICAL PRESENTATIONS

- Andy Rudmik (GTE) presented the status of their work on a Distributed Software Engineering Control Process. An integral part of this effort required definition of an interface set comparable to the CAIS. Analysis results thus far indicate the CAIS interface level is the most appropriate for implementation. This project now has a working prototype with approximately 300 packages.

- Frank Belz (TRW) presented the status of the Prototype Advanced APSE task being performed for the Naval Ocean Systems Center (NOSC). This task calls for a prototype implementation of CAIS interfaces and integration of selected tools utilizing these interfaces. The host environment is a VAX/UNIX system. Results of this effort will be provided to NOSC for consideration by the CAISWG.

- Jack Kramer (Institute for Defense Analyses) chaired a summary of current changes to the CAIS and future directions for expansion.

14. KITIA MEETING.

- Herm Fischer (Litton), KITIA chair, conducted a KITIA meeting. KIT members were invited to attend. Herm reviewed the previously voted issues and discussed the benefits of a "CAIS'd UNIX" implementation which could be accomplished in various ways including AJPO contracts or grants to universities. Herm suggested the KITIA should look to future directions as a group possibly considering the areas of methodology, risk reduction or prototype development. Subsequent discussion by the KITIA recommended formulation of a risk reduction proposal for submission to the AJPO.

15. BREAK FOR LUNCH.

16. REORGANIZE INTO WORKING GROUPS.

17. WORKING GROUP REPORTS.

- STONEWG - Herb Willman (Raytheon) will be joining this group which will continue its work on formulation of a STONEMAN II document.

- GACWG - plans to expand its outline of a Guidelines and Conventions document.

• RACWG - expects to obtain consensus on RAC Sections 4-6 in August.

• DEFWG - solicited KIT/KITIA support for commonality of terminology among KIT/KITIA documents and requested member input via NET for expansion of definition pool.

• COMPWG - will continue analysis of the CAIS versus the RAC documents to help identify any inconsistencies between these documents. Also expects to initiate work in conformance areas and to identify related policy and technical issues.

• STANDWG - will continue work in specification analysis.

18. CLOSING REMARKS.

• Herm Fischer (Litton) indicated the results of the KITIA vote for future direction supported the risk reduction proposal.

• No major problems were identified with the April meeting minutes.

• KIT/KITIA gratitude was expressed to Lucas Maglieri (National Defense Headquarters, Canada) for a superb job as meeting host.

19. MEETING ADJOURNED.

APPENDIX A
ATTENDEES
KIT/KITIA MEETING

KIT Attendees:

BARRY, Bill	FCDSSA-SD
BELZ, Frank	TRW
CASTOR, Jinny	AFWAL/AAAF
CONRAD, Tom	NUSC
FERGUSON, Jay	DoD
FITCH, Geoff	Intermetrics
HARRISON, Tim	Texas Instruments
HART, Hal	TRW
JOHNSON, Doug	SoftWrights
JOHNSTON, Larry	NADC
KEAN, Elizabeth	RADC/COES
KRAMER, Jack	IDA
KRUTAR, Rudy	NRL
LOPER, Warren	NOSC
MAGLIERI, Lucas	National Defense Hq., Canada
MILLER, Jo	NWC
MUNCK, Bob	MITRE
MYERS, Philip	NAVELEX
OBERNDORF, Tricia	NOSC
PEELE, Shirley	FCDSSA-DN
TAYLOR, Guy	FCDSSA-DN
THALL, Rich	SofTech
WILDER, Bill	PMS-408

KITIA Attendees:

ABRAMS, Bernie	Grumman
BAKER, Nick	McDonnell Douglas
CORNHILL, Dennis	Honeywell/SRC
DRAKE, Dick	IBM
FISCHER, Herm	Litton
FREEEDMAN, Roy	Hazeltine
GARGARO, Anthony	CSC
GLASEMAN, Steve	Aerospace
HUANG Cheng-Chi	Hughes
HORTON, Michael	SDC
JOHNSON, Ron	Boeing
KAMRAD, Mike	Honeywell/SRC
KERNER, Judy	Norden
LAMB, Eli	Bell Labs
LINDQUIST, Tim	Virginia Tech
LYONS, Tim	Software Sciences Ltd.
McGONAGLE, Dave	General Electric
MORSE, H. R.	Frey Federal Systems
PLOEDEREDER, Erhard	Tartan Labatories
POW, Charley	Lockheed
REEDY, Ann	PRC
RUDMIK, Andy	GTE
SIBLEY, Edgar	AOG Systems Corp.
WREGE, Doug	Control Data Corp.

KITIA Guest:

GILROY, Kathy	Harris Corp.
RILEY, Paul	Data General

APPENDIX B - MEETING HANDOUTS

1. CAIS Standards Sub-group Coordination Report, 9 July 1984.

2. E&V Status Report, not dated.

KIT/KITIA MINUTES
1-4 OCTOBER 1984
Merrimack, New Hampshire

ATTENDEES: See APPENDIX A

2 OCTOBER 1984 - JOINT KIT/KITIA MEETING

1. GENERAL BUSINESS

- Introduction of visitors and new representatives were made.

- Observers from Los Alamos were welcomed. Los Alamos is pursuing
  a CAIS implementation on Unix.

- Evaluations of the bids for the KIT Support contract have been
  completed.

- The AIM contract has been extended for eight to nine months.

- An E&V procurement for support of the team is imminent.

- DIANA maintenance has been contracted to Intermetrics.

2. NAMED WORKING GROUP REPORTS

- CAISWG
  Jack Kramer announced John Long of TRW is now supporting the CAIS
  effort at I.D.A. Completed projects this quarter were the CAIS
  version 1.3 and the AdaTEC CAIS review in Hyannis. Revisions have
  begun on CAIS 1.3 to complete version 1.4 for next quarter. CAIS
  version 1.4 is due for typeset in January and will be delivered to
  Bob Mathis. Presentations for next quarter are a status of the CAIS
  document as a military standard and progress on the accompanying
  rationale. Jack also requested additional help to work with Tim
  Harrison on the I/O section of the CAIS.

- RACWG
  Sections 4,5, and 6 of the RAC document were scheduled for a vote,
  but section 4 was deferred until next quarter. Also an action item
  from the July meeting was raised, stating that the group was
  dissatisfied with the treatment of security. The group extracted
  functional requirements from the "orange book" and stated that the
  RAC nomenclature was inconsistent with definition of terms in the
  "orange book". Projected work includes obtaining approved sections of

all requirements chapters, which will be attached to the CAIS Version 2
RFP. Also planned is a more formal procedure for configuration management
and the development of a rationale for the RAC.

- **DEFWG**
  Due this quarter was a combined glossary and an inclusion of STONEMAN
  terms. Completed work includes a definition tool on ECLB and a review of
  the glossary over the ARPANET. Projected work includes adding STONEMAN
  terms to the glossary when a revised version of STONEMAN is complete,
  opening a definition tool on the ARPANET for use by other team members,
  and a requirements and criteria glossary update. A presentation is
  scheduled for next quarter on using the definition tool. Other actions
  include a decision that "orange book" definitions take precedence over
  all other definitions.

- **STONEWG**
  Ann Reedy discussed the items due this quarter which included a revision
  of a very rough draft of STONEMAN II and to review the tool set section.

- **GACWG**
  Ron Johnson expressed difficulty with availability of personnel, which
  has been slowing the progess on drafts for al. chapters of "A User's Guide
  to Ada Transportability". The group expanded resources to include other
  guides and conventions and are currently working on detailed outlines of
  these. Projected work for next quarter is editing and completing all
  chapters of the guide and a description of the user's guide. Tricia stated
  the possibility that some CAISWG members might support the GACWG during
  the next quarter.

- **COMPWG**
  Tim Linquist focused on a draft on standards related to the CAIS and
  discussed CAIS conformance and semantics.

3. GENERAL ANNOUNCEMENTS

- Tricia noted the absence of San Diego TRW representatives, considering
  that Jack Foidl's daughter, Andrea, had recently undergone heart surgery,
  and suggested sending a post card to her.

- Please check master KIT/KITIA address listing and make any necessary
  corrections.

- Susan Good is now Susan Ferdman and is working in Philadelphia as a
  consultant for the CAIS until the middle of this month.

- There have been obvious problems with ECLB, due to moving the machine,
  which forced the utilization of a much slower circuit. When the MILNET
  is up again, the new host address will be 26.7.0.65.

- SIGAda elections results indicate Anthony Gargaro is the new SIGAda chairperson and Hal Hart is the vice-chair for Liaison.

- Our comments on the JSSEE OCD were received. The chairperson, Hank Stuebing, emphasized that the KIT/KITIA should be aware of two important points:

    a. The OCD is a user's view of SEE, and any reference to implementation is only for illustration.

    b. Also, the OCD is not a follow-on to the STONEMAN, since the OCD is a user's view of SEE and STONEMAN is an implementor's view of APSE.

- A Future APSE Workshop was held in Santa Barbara and, although the CAIS was discussed, it was not the focal point. Emphasis was more on a parallel with JSSEE and a user's view of advanced capabilities. All working group chairs are coordinating reports of working groups' discussions and a summary should be contained in a special Ada Letters sometime in the spring.

4. MEETING SCHEDULE

1985
January	14-17	San Diego
April	15-18	Washington D.C.
July	8-11	San Francisco
September	23-26	Rhode Island

1986
January	13-16	San Diego
April	14-17	Atlanta
July	7-10	Schenectady, NY
September	22-25	Minneapolis/St. Paul

1987
| January | 19-22 | San Diego |

5. DEFINITIONS AND DISCUSSIONS

- "Orange Book" Glossary:
  It was determined by WG2 that the definition of "object" should be changed to "entity" throughout the RAC, considering the fact that the "orange book" terms have precedence over all other definitions and its definition of "object" did not mesh with the group's intended use of the word.

- DEFWG Terminology Issue Resolution:
  Jack Kramer reviewed the problems with terms and definitions in the

glossary. The definition of "process" was discussed in reference to the "orange book" and in the RAC. Several opinions and responses about conflicts in definitions were expressed and an effort was made to determine exactly what was needed. Tricia asked DEFWG to obtain information from the people at the meeting to distinguish the terms and definitions and also to design a procedure for collecting and reviewing these terms.

6. SEPARATE MEETINGS ON RAC SECTIONS

3 OCTOBER 1984

7. CAIS HYANNIS REVIEW REPORT

● Node model discussions concentrated on standard attributes, distributed systems, and access control.

● Process model discussions emphasized the blocking/nonblocking issue.

● Security discussions centered on the need for a consolidated approach.

● Input/Output discussions included magnetic tape support, pragmatics, standards, and queue nodes.

● Non-technical discussions centered on the pros and cons of military standardization according to the current schedule.

8. STRATEGY DISCUSSIONS WITH Ø6'S AND BOB MATHIS

● Tricia welcomed Bill Wilder of the Navy representing Captain Boslaugh, Bob Mathis (the director of AJPO and STARS), Brian Schaar (the Navy deputy to the AJPO), Col. Nidiffer of the Air Force, and Jim Hess of the Army. Discussion began with the CAIS introduction and general strategies for furthering the Ada program.

● From the Hyannis Review, concerning the original plan of submitting a military standard in January 1985, it was decided to submit to the AJPO a candidate for standardization instead.

● Herm Fisher reviewed the discussion of the July meeting and the major ideas he and Eli Lamb expressed: (1) the CAIS draft due in January should not be standardized, (2) prototyping should be the next object of contracts, (3) Eli's alternative look at a risk reduction contract.

● Key issues of the risk reduction contract were to capitalize on industry's current ideas and to create some type of CAIS subset to be compatible with industry and allow tool transportability.

- Bob Mathis expressed the STARS objective to develop the best Software Engineering Environment (SEE) by the end of the decade in two phases and have it installed on a service project.

- The majority agreed on the need for an interface standard, however disagreements arose concerning the strategy necessary to achieve this, the time constraints involved considering other environments being developed, and how to keep industry's support and interest by clarifying the market.

- Rich Thall of SofTech suggested a proposal for advancing the Ada environment standard by developing two parallel standards: (1) the first step was to put an Ada environment on top of an existing operating system and achieve transportability this way, (2) the next issue was a common APSE operating system design to achieve the long term goal of transportability and interoperability. Rich also discussed the pros and cons of using the already existing UNIX system.

- Many diverse views and opinions were expressed regarding alternatives for possible conformance policies, including CAIS subsets and/or supersets.

9. RAC SECTIONS 4 & 5 Presentations/Discussions

Tim Harrison reviewed RAC section 6 revisions. A vote of approval was deferred until the end of the session, along with votes on sections 4 and 5. Frank Belz discussed RAC section 4, involving data management, typing, identification, operations, transactions, and history.

Results of balloting:

Section 4	yes(21), qualified yes(15), no(1), abstain(0)
Section 5	yes(26), qualified yes(8), no(1), abstain(1)
Section 6	yes(24), qualified yes(11), no(1), abstain(2)

10. JSSEE review by Herman Fischer.

4 OCTOBER 1984

11. STONEWG Presentation - Steve Glaseman

12. ALS/CAIS Study Preliminary Report - SofTech

Rich Thall introduced Rich Simpson, Nancy Yost, and Carl Hitchon as the other members of the SofTech team that is focusing on resolving the differences between the ALS and CAIS, and designing

a strategy for an ALS-to-CAIS transition. The general strategy is
to build a CAIS KAPSE on top of an ALS interface and, alternatively,
build an ALS KAPSE on top of a CAIS interface. The group has completed
a draft document studying these ideas and is approximately halfway
through with the project. Rich Simpson discussed a database comparison
for mapping, followed by a process comparison by Rich Thall.

13. SEPARATE KIT/KITIA MEETINGS

14. NUMBERED WORKING GROUPS MEETINGS

15. KIT/KITIA WRAP UP

16. MEETING ADJOURNED

KIT Attendees:

BELZ, Frank   TRW

CASTOR, Jinny   AFWAL/AAAF

FERGUSON, Jay   DoD

FITCH, Geoff   Intermetrics

HARRISON, Tim   Texas Instruments

HART, Hal   TRW

JOHNSTON, Larry   NADC

KRAMER, Jack   I.D.A.

KRUTAR, Rudy   NRL

LINDLEY, Larry   NAC

LOPER, Warren   NOSC

MAGLIERI, Lucas   National Defense Hq.

MILLER, Jo   NWC

MUNCK, Bob   MITRE

MYERS, Gil   NOSC

MYERS, Philip   NAVELEX

OBERNDORF, Tricia   NOSC

PEELE, Shirley   FCDSSA-DN

SCHAAR, Brian   AJPO

TAYLOR, Guy   FCDSSA-DN

THALL, Rich   SofTech

WILDER, Bill   PMS-408

KIT Guests:

TALLMAN, David   LANL

TOMLINSON, Bob   LANL

KITIA Attendees:

ABRAMS, Bernie   Grumman Aerospace

BAKER, Nicholas   McDonnell Douglas Astronautics

CORNHILL, Dennis   Honeywell/SRC

DRAKE, Dick   IBM

FISCHER, Herman   Litton Data Systems

FREEDMAN, Roy   Hazeltine Corp.

GARGARO, Anthony   CSC

HORTON, Michael   SDC

JOHNSON, Ron   Boeing Aerospace Co.

KERNER, Judy   Norden Systems

LINDQUIST, Tim   Virginia Tech

LYONS, Tim   Software Sciences Ltd.

MARTIN, Ed   Lockheed Missiles & Space Div.

McGONAGLE, Dave   General Electric

MORSE, H.R.   Frey Federal Systems

PLOEDEREDER, Erhard   Tartan Laboratories

REEDY, Ann   PRC

ROUBINE, Olivier   Informatique Internationale

RUDMIK, Andy   GTE

SIBLEY, Edgar   AOG Systems Corp.

WILLMAN, Herb   Raytheon Company

WREGE, Doug   Control Data Corp.

YELOWITZ, Larry   Ford Aerospace & Communications

KITIA Guest:

GILROY, Kathy Harris

SECTION III

KIT/KITIA DOCUMENTATION

THE SECOND CAIS REVIEW MEETING
HYANNIS, MASSACHUSETTS
2 AUGUST 1984

T. Oberndorf
Chairman
KAPSE Interface Team

INTRODUCTION

The second Common Ada Programming Support Environment (APSE) Interface Set (CAIS) Public Review was held in Hyannis, MA on 2 August 1984 in conjunction with the AdaTEC meeting at that time. Over one hundred participants from industry, academia and government attended. The subject of the review was Version 1.3 of the CAIS document.

The CAIS has been developed by the CAIS Working Group (CAISWG) of the Kernel APSE (KAPSE) Interface Team (KIT, a Navy-lead DoD team) and the KAPSE Interface Team from Industry and Academia (KITIA, it's industry/academia counterpart) for the Ada Joint Program Office (AJPO). These teams have been meeting since early 1982 in an attempt to define a set of interfaces which could be implemented in all APSEs in order to make it possible for APSEs to share tools and databases. The current version of the CAIS primarily addresses only those interfaces needed to share tools.

The review was preceded the previous night by a two-hour discussion of the changes that have been made to the CAIS interfaces since the last Public Review (held September 1983 in Washington, D.C.). Consequently, on the morning of the review, only a short presentation was given on this topic. The participants then organized into discussion groups which focused on five areas: the node model, the process model, the input/output, the security model and non-technical issues. The summaries of the discussions held by these five groups are found in each of the following five sections. Late in the afternoon everyone met back together to hear and discuss the reports from the separate groups.

General comments reflected that there has been considerable improvement in the document since the Version (1.0) which was reviewed in September. The semantics have been addressed more completely and consistently. Some deficiencies in some sections still exist and were pointed out by the Hyannis reviewers. All of the models have matured; the node model has become very stable (aside from the introduction of security and access control mechanisms) and the process model seemed to meet with general acceptance. Much work remains to be done on the input/output section, but many helpful suggestions for directions to pursue were provided by the reviewers, and the next version should show maturity in this area as well. Very few of the comments received indicated that significant interfaces were missing, and, of those mentioned, most cannot be easily provided in a portable manner.

Overall, the session was very well-attended with many helpful comments and much useful feedback received. The participation of all those present was very impressive, especially considering that the document had only been made available to most attendees the day before the review meeting. The suggestions presented were sound and showed a great deal of insight into the objectives of the task in general and the design goals of the CAIS in particular. The attendees are all to be highly praised for this one day of work, and their contributions are all greatly appreciated by the AJPO, the KIT, the KITIA and the CAISWG.

The CAISWG is now diligently working its way through the many comments and discussions and is drafting a new version of the document. This version will be presented to the KIT and KITIA at their October meeting. The comments received there will be the major inputs to an intensive one-week CAISWG meeting two weeks later. That meeting will result in a new draft 1.4 of the CAIS which will be mailed early in November to all participants in the Hyannis review and anyone else requesting a copy. That version will be the subject of review meetings to be held in Washington, D.C. in conjunction with the SIGAda meeting 26-30 November. Final comments must be received by 7 December, as the CAISWG is scheduled to deliver the draft CAIS to the Ada Joint Program Office (AJPO) in mid-January 1985. This draft will be the one proposed for military standardization, and its delivery to the AJPO in January will signal the initiation of a full tri-service coordination for standardization.

Many comments about the CAIS have been made, both publicly in journals such as Ada Letters and privately. The KIT, the KITIA, the CAISWG and the AJPO constantly invite all those who hold opinions on the CAIS to make those opinions and the reasons for them known to us. Anyone wishing to participate in the November review who did not receive a copy of CAIS 1.3 should make sure they will receive Version 1.4 by contacting:

Patricia Oberndorf
Code 423
NOSC
San Diego, CA 92152
(619)225-6682
POBERNDORF@ECLB

Even if you are unable to attend the November review, be sure that your comments (including the rationale behind them) are sent either to the above address, to the address in the back of the document or to CAIS-COMMENT@ECLB on the ARPANET. The final cutoff date for all comments regarding CAIS Version 1 is 7 December 1984. We appreciate the participation of each and every one of you, and we look forward to hearing from you in the future.

SECTION 1: NODE MODEL DISCUSSION GROUP

The main topics addressed by the discussion group on the node model were:

a) support for distributed implementations
b) concerns of intra- and interoperability of tools
c) the interrelation of KAPSE/CAIS/Run-time Environment

Several other issues regarding specific concepts of the CAIS were discussed, and recommendations for further refinements of the CAIS were made.

The general atmosphere of the discussion was one of contentment with the concepts of the current node model, mixed with concerns about implementation and efficiency aspects on distributed systems. There was a uniform expression of the need to expand the CAIS in the area of predefined attributes and relationships.

1. **The CAIS must provide support for distributed environments.**

   Given the clear trend toward distributed systems of workstations as the system configuration for program development, there is much concern about the implementability and the appropriateness of the CAIS for such distributed systems.

   Two situations must be distinguished:

   - a single CAIS implemented on top of a distributed system, and
   - communication between CAIS implementations on different components of a distributed system.

   With regard to the former, the issues of communication among the system components are largely hidden within the CAIS, which is free to implement whichever communication mechanism appears most appropriate. Likewise, the distribution of the CAIS nodes among various servers or their administration at a central server are problems that need to be solved by the CAIS implementor, rather than at the CAIS interface level. Currently, the CAIS has no known deficiencies that would preclude either implementation. There is, however, a need for resource control mechanisms within the CAIS to allow the user to exercise control over which system components are tasked to perform activities on his/her behalf. These interfaces fall in the category of deferred CAIS issues, mainly because the nature of such interfaces is as yet uncertain. Nevertheless, their importance has to be acknowledged and work to define such interfaces should be started as soon as feasible.

With regard to communication between different CAIS implementations, two levels need to be distinguished:

- the "physical" mechanisms for communication
- the interoperability between tools residing on different CAIS implementations.

These issues are not unique to systems consisting of distributed workstations, but also arise in general, when data and tools are to be exchanged between different CAIS implementations.

With regard to the communication mechanism, two proposals were made:

a. The ISO Model should be examined as a possible basis for common communication interfaces between CAIS implementations. If this examination determines that the model is adequate, CAIS interfaces should be added that reflect the various levels of the ISO Model.

b. There is an urgent need for the specification of an external form for nodes, attributes, and relationships, very much like the specification of an external form for DIANA, so that node structures can be exchanged between different CAIS implementations, given CAIS-specific reader and writer programs for this external representation.

2. There were concerns regarding inter- and intraoperability.

There was a general opinion in the working group that the CAIS, as currently defined, provides little support that would enhance the interoperability or intraoperability of tools, since very few of the attributes and relationships important for communication among tools are predefined by the CAIS. The CAIS, in order to be true to its objectives, must include such commonly defined attributes and relationships. As a concrete step toward this goal, members of the working group agreed to send recommendations for such attributes and relationships to the MILNET account CAIS-COMMENT@ECLB; the public at large is invited to do so as well. While CAIS 1.3 provides a framework for the terminology used, CAIS 2.0 should define a substantial set of attributes and relationships with predefined meanings. As an interim solution, agreement on common attributes and relationships could also be reached among the members of the CAIS Implementors' Group, which should be a prime source of

valuable contributions.

It was pointed out that the objective of CAIS 1.3, apart from
providing a general framework, was mainly to support
portability of closed tool-sets, i.e., sets of tools with
coordinated information interfaces between themselves, but with
little, if any, dependency on data interfaces with other tools
present in the APSE. Predefining certain attributes and
relationships often infringes on issues of the methodology of
the tool support for a given activity. It was therefore felt
by the CAISWG that such attributes and relationships can be
predefined only with the widest possible participation of
current and future CAIS- and APSE-implementors.

3. **What is the interrelation of CAIS/KAPSE/Run-Time Environment?**

A lengthy discussion dealt with the interrelation of the CAIS
with the STONEMAN KAPSE model and with the Run-Time Environment
of Ada programs. This discussion showed the diversity of
perceptions of what constitutes a STONEMAN KAPSE and how the
CAIS relates to this model (a phenomenon that has already been
observed in discussions of KIT/KITIA, which eventually led the
CAISWG to an approach that avoided mentioning the concept of a
KAPSE at all). It was recommended that a list of STONEMAN
KAPSE features not supported by the CAIS be provided, in an
attempt to delineate the distinction between the CAIS and the
STONEMAN KAPSE.

It was then observed that many of the interfaces of the CAIS
bear a strong resemblance to interfaces required for the
run-time support of Ada programs, which brought up the question
of whether the CAIS should recognize and incorporate these
interfaces as well. Such an approach would enable implementors
to integrate their implementation of the run-time environment
with the CAIS. It was pointed out that the counter-argument to
such an inclusion in the CAIS is three-fold. Firstly, the
inclusion would expand the CAIS beyond its scope of providing a
basis for porting program development tools among APSEs.
Secondly, the implementors today can already take advantage of
the commonalities and base both the CAIS and the Ada run-time
environment on the same underpinnings; there is no immediate
need for a CAIS to prescribe such implementation commonality.
Thirdly, an encroachment of the CAIS into the area of run-time
environments would almost certainly make existing compilers and
their implementation of the run-time environment incompatible
with the CAIS. (For the latter reason, the CAISWG has
studiously tried to avoid imposing requirements on the run-time
environment that could not be expected to be an immediate
consequence of Ada semantics or satisfiable in any reasonable
implementation of the Ada run-time environments.) One argument
that was also discussed was that, if that interconnection was

made explicit in the CAIS, the CAIS could be much more "imperative" about run-time issues (as opposed to the current approach of "don't hurt existing implementations" by putting requirements on them). The results were inconclusive; opinion was divided as to whether or not this should be done.

4. **There is concern over the potential efficiency of piggy-backed CAIS implementations.**

    Some concern was expressed over whether CAIS implementations on top of existing operating systems would be efficient enough to meet the needs of tools. While there cannot be a conclusive answer to this question before pilot implementations have provided us with empirical data, it is to be expected that some overhead cost is to be paid for the transformation through multiple layers. On the other hand, if the added support provided by the node model is appropriate for many tools, then one can surmise that existing tools of this flavor are already paying the same overhead for administrating their internal data representations. Hence, if these tools were to use the CAIS interfaces, instead of having their own but similar data administration, their overall performance is likely to improve, since the CAIS implementor can exploit underlying system features generally not available to, or not used by, existing tools. Efficiency of the CAIS should not be measured by its own efficiency, but rather by the efficiency of the tools based on the CAIS compared against similar tools without CAIS support.

    One area in which the CAIS may cause undue loss of efficiency is the area of revision control. Currently, the CAIS has no built-in mechanisms for revision control. Instead, it provides the primitives required to implement any number of different revision control schemes. This implies that tools must first go through the revision control layer before reaching the CAIS layer and the operating system below it. A CAIS implementation following this layered scheme cannot employ special mechanisms for speeding up this particular aspect of file handling. It was therefore recommended that the inclusion of a revision model in the general node model should be examined.

5. **The CAIS needs to provide a transaction concept.**

    The CAIS currently does not provide a transaction scheme that allows transaction control beyond the life-time of a single process. It was recommended that transaction control mechanisms be included that allow transactions to span over the consecutive execution of multiple processes.

6. The "current_user" and "user" concepts should be renamed.

   The CAIS is currently predicated on an administrative organization that considers the individual user as a focal point, as demonstrated by the concepts of "current_user" and "user" relations. However, several existing operating systems used for program development are more oriented towards "projects" as the main administrative focus, with users connecting or logging into particular projects. Although "projects" can be supported in a CAIS implementation by considering a "project" as still another "user", it was recommended that the terminology of "user" be changed to a terminology less biased towards a particular administrative view.

SECTION 2: PROCESS MODEL DISCUSSION GROUP

The main topics addressed by the discussion group on the process model were:

   a) parallel execution of tasks
   b) process states
   c) interrupts.

The general tone of the discussion was that the CAIS process model is workable and not far from its goals. The comments were of the nature of specific points for improvement, with the underlying assumption that the larger issues are already well solved.

1. The ability of tasks to execute in parallel is important.

   Several comments concerned the ability of tasks to execute in parallel. The Ada Reference Manual does not require that parallel tasks actually execute simultaneously, only that they proceed independently, except at points where they synchronize. The comments expressed the concern that code written with the assumption that one task's execution will NOT block the execution of another task would not be transportable to a CAIS implementation where one task's execution DOES block the execution of another.

   The participants agreed on the following statement as a compromise between a desire to have non-blocking of parallel tasks and the reality that compilers at present are blocking. "The goal is to have non-blocking compilers, but the assumption should not be made that the compiler is non-blocking, because in early implementations compilers will be blocking."

2. **More states are needed for reporting process status.**

The process states transition table given in Table II of CAIS 1.3 (page 70) does not show how a process is created. A new row needs to be added for the operation of creating a process. A new column needs to be added for processes which do not exist, in order to make the state transition table complete.

Some comments concerned a wish to be able to distinguish between a process which is in the READY state, but which is waiting for resources or rendezvous, and a process which is in the READY state and is actually executing. This desire stems from a need to monitor the progress of a process.

Some comments concerned the completion status enumeration values. The addition of the value COMPLETED, distinct from ABORTED and TERMINATED, was suggested. The Ada Reference Manual describes a task as "completed but not terminated" if it has completed the execution of its sequence of statements but is waiting for the termination of a child. It was pointed out, however, that CAIS processes are closely analogous to Ada programs, which may include many tasks, so this distinction is not appropriate.

3. **Some terminology is ambiguous.**

In the description of the semantics of ABORT_PROCESS as well as other places, the term "descendant" is used. There is ambiguity about the meaning of the term. Does it mean "a process I created" or "a process I am the parent of"?

4. **Interrupts should be integrated into the node model.**

The suggestion was made that the receipt of a signal should be able to change the priority of the receiver. However, the benefit gained was judged not to warrant the cost.

The suggestion was made that processes might need an analogy to the terminate alternative for Ada tasks. But the user can design it using existing facilities, so it is not needed in the CAIS.

The suggestion was made that the CAIS could be made more consistent if the interrupts were nodes themselves, just as queues are, because an interrupt is a named entity that is not currently integrated into the node model.

5. **More explicit interaction between the CAIS and its hardware (both host and target) was called for.**

There is a concern that the underlying hardware may fail and destroy the tree structure of the CAIS, indicating a need for a mechanism for reporting hardware problems. However, this seemed to some participants to be outside the scope of the CAIS.

There is a desire for explicit support for targets and distribution, so that the user or tool designer can have some control over the assignment of processes to logical processors.

SECTION 3: INPUT/OUTPUT DISCUSSION GROUP

The main topics addressed by the discussion group on Input/Output (I/O) were:

    a) the tape standard used
    b) the terminal capabilities
    c) status information.

The general atmosphere of the discussion indicates that the I/O section needs substantial modification before it is appropriate to establish it as part of a MIL-STD. There were a large number of comments dealing with lack of clarity, missing interfaces, inadequacy of interfaces, inconsistencies and other deficiencies.

1. **The use of a different magnetic tape standard for the CAIS model should be considered.**

Several people in the group indicated that there are problems in assuming that the ANSI tape standards can be used to define a mechanism for transporting source code from one APSE to another. In particular, they sited that tape drives by different manufacturers deviate slightly from the standards, thus making them incompatible. It was also noted that the ANSI standards cause a large amount of tape space to be wasted.

It was recommended by the group that TCP be used to provide for the transporting of textual data. This method eliminates the problem of defining the "exact" positioning of data on magnetic media (inter-record gaps).

2. **There are problems with inconsistencies.**

It was noted by the group that there are inconsistencies among the interfaces. This is a problem that deserves much attention before the I/O section is proposed as a draft. (They noted that the different sections are inconsistent with each other as well.) The interplay between CAIS_TEXT_IO and the other I/O packages is poorly defined.

3. **Queue nodes were well-received.**

The concept of queue nodes was favorably received. In fact, this was one of the major deficiencies that people had come prepared to complain about.

4. **Additional interfaces were proposed to address current deficiencies.**

Additional interfaces were presented as neccessary for a useful I/O model. Among them were file node attributes (file/queue size, device type, terminal class, terminal capabilities) and host buffer control (flushing, forcing to disk).

5. **Additional terminal capabilities were suggested.**

It was suggested that interfaces be provided that enable a user to determine (1) which function key has been pressed, (2) which class of terminal is associated with a particular node, and (3) which interfaces are "directly" supported by the terminal being used.

Get/Put should be more carefully defined.

Form terminals should contain a "form" data type that is manipulated by interfaces. This permits several "forms" to be in existence at once.

6. **There is a desire for better status information in the event of failures.**

There was considerable discussion about the problems of trying to recover from I/O operations that fail. Exceptions do not provide enough information for a programmer to inform the user of his/her program what actually caused the I/O operation to fail. There is a need to provide an interface that can obtain information from the host OS about the reason for the failure. There was no resolution of this matter as no "transportable"

interface could be determined.

7. The following miscellaneous topics were discussed only briefly.

* Industry standards (e.g., GKS, TCP-IP, ISO communications) should be considered in the design of the CAIS.

* The byte stream used in TEXT_IO should be recognizable. That is, the CAIS should define the meanings of all byte sequences and how they are to be interpreted.

* A single file should be defined upon which all I/O operations are performed.

* Is windowing (like on a LISA) possible with the current design? (It was determined by the group that it could be done, but would be hidden within the implementation.)

* There is a need to query the capabilities of a terminal. In particular, a programmer needs to be able to determine which of the "terminal I/O" interfaces are efficiently supported and which are inefficiently supported.

SECTION 4: SECURITY DISCUSSION GROUP

The main topics addressed by the discussion group on security were:

    a) CAIS dictation of security policy
    b) mandatory versus discretionary security.

Other issues were discussed, and a set of recommendations was made. These recommendations appear at the end of this section.

The general atmosphere of the discussion was one of concern over the tight coupling of security policy with the current CAIS interfaces.

    1. Some were concerned that the CAIS dictates a certain security policy, as well as a model for implementing that policy.

        It was felt that there are several problems with this approach. The CAISWG has used the DoD Trusted Computer System Evaluation Criteria as the basis for the mandatory and discretionary security aspects found in CAIS 1.3. This guide is published by the Department of Defense Computer Security Center. It provides "a uniform set of basic requirements and evaluation classes for assessing the effectiveness of security controls built into Automated Data Processing (ADP) systems." This guide itself does not require a specific policy; it only establishes

criteria for the achievement of various levels of security. The current CAIS approach that dictates a certain security policy makes it difficult or impossible to implement the CAIS on top of security kernels that embody different policies and use different mechanisms. It was felt that the CAIS should define a simple interface to a security mechanism, rather than the mechanism itself and a security policy implemented by the mechanism.

Even if this approach was desirable, it would not be wise for the CAIS to simply define some new security mechanism. Security mechanisms require thorough analysis to determine their validity, and the CAIS should not be tied to a mechanism that has not undergone this analysis. Extensive research is still being invested in discovering good security models. The CAIS interface should therefore allow many security models and should, in general, be flexible. It was pointed out that the security community itself does not have standards for security policies and mechanisms, so the CAIS probably should not try to define one.

It was also mentioned by one person in the group that the approach does not allow for Multics-style ring protection, although other members of the group expressed doubt that ring protection pertained to Programming-Support Environment tools.

Group opinion was divided as to what should happen regarding this question for CAIS Version 1. Although it was generally agreed that the security mechanisms should be separated at least into one chapter, possibly into an appendix, there were two views as to what should happen for CAIS Version 1:

a. Keep the current security in the CAIS as a basis for further work and commercial implementations; make Version 2 a secure system, but change Version 1 to allow different mechanisms also.

b. If you can't prove that it's secure, leave out the current security mechanisms; define generic interfaces (what exceptions will be raised), allow for lack of visibility (see next topic) and possibly leave security mechanisms in an appendix.

2. Should the CAIS security policy and mechanisms be visible to the user or tools?

It was felt that the CAIS should allow the implementation to choose to make some nodes not 'visible' to some subject (the CAIS already allows this) and that it should let the implementation provide a policy for governing when this happens

and a mechanism for doing it. Some also felt it is not necessary to have the security information visible to the user (i.e., classification labels in CAIS 1.3 are node attributes). It was noted that a tool writer and user should not have to worry about the security policy and mechanism; it should be transparent to the tool/user. However, opinion was divided, as some felt that separating out security mechanisms from the CAIS is not in support of portability, while some felt that portable programs should not rely on a specific security policy or mechanism.

3. Should mandatory security be separated from discretionary?

Some felt that the CAIS should treat DoD security (mandatory security) separately from commercial security (discretionary security). Most of the discussion centered around mandatory security. Some felt that discretionary access control should be left in the CAIS because it is useful.

4. The CAIS should not make claims about the level of security that the mechanism provides.

As noted above, sufficient analysis of the proposed CAIS mechanism has not been performed, so no claims about the level of security provided are warranted.

5. It was felt that there should be more interaction between the CAISWG and the security community in general and the DoD Computer Security Center in particular.

6. A set of statements was developed by the group to form an 'official recommendation to the CAISWG'.

   a. The security policy model should be separated from the CAIS model.

   b. The CAIS document should not state that the CAIS supplies a "security mechanism", as the validity of such a system requires thorough analysis and proof. The term "security system" should be changed to "protective controls".

   c. Conditions under which the exceptions Access_Violation, Security_Violation, and Name_Error are raised should be specified as recommendations; final specification should be left to the implementor of the system.

d. In order to isolate security mechanisms in the CAIS,
    security information should be stored as attributes with
    limited functionality (not accessible by users).

e. The CAIS should allow different security policies; it
    should not mandate a specific one.

f. An implementor of the security kernel should be required to
    specify functionality and use of the kernel and to specify
    the exceptions that are raised at the CAIS interface and
    under what conditions they are raised.

SECTION 5: NON-TECHNICAL ISSUES DISCUSSION GROUP

The main topics addressed by the discussion group on non-technical
issues were:

a) the push for a military standard (MIL-STD) in January 1985
b) the need to establish a standard at all
c) the expected cost/benefit of such a standard.

The general atmosphere of the discussion was that such a common set of
interfaces would be desirable, but that more time is required to
properly address all of the issues involved.

1. The CAIS is ambitious with respect to the current state of the
    art, and it is complex.

    This statement was being quoted from one of the KITIA members.
    Afternoon contributors who had been in other groups in the
    morning reported that the node model group had not considered
    the node model to be pushing the state of the art, and the
    process group said that parts of the CAIS were not yet up to
    the state of the art. It was suggested that this was a
    reaction based on the Ada experience in which the aspects of
    each component are known, but unexpected complications arise
    when all these are put together. It was acknowledged that the
    impact of the security mechanisms is less well-known.

2. There is insufficient calendar time for public review between
    now and January 1985.

    It was felt that viable public reviews could not be achieved if
    the CAIS was to be made a MIL-STD so quickly.

3. **There is insufficient time to produce a credible document.**

   The sheer effort required to take care of all the missing details and to correct all typos and other minor errors was felt to be too great for the time available between now and January.

4. **There is no technical requirements statement associated with the CAIS proposal.**

   It appears that the CAIS is a "solution" document, but the technical requirements document to which this interface set is responding is missing. It was pointed out that the interface set now called the CAIS had evolved from a study of the AIE and ALS and that it took its requirements largely from agreement between those two systems about what a "KAPSE" was, both in level and content. A requirements document is being generated now which will guide the development of CAIS Version 2, and its development, although in parallel with the production of CAIS Version 1, has had some impact already.

5. **Why is the CAIS being made a MIL-STD? What does the DoD expect to accomplish by making it one?**

   Most of the group discussion was spent on this set of concerns. The discussions centered around a few main issues.

   a. A CAIS would be good in the long run: Two votes were taken in the morning session. In these votes, a clear majority (about 30 to 5) felt that the CAIS (i.e., some set of common interfaces at about this level) would be useful in the long run and of benefit to the whole community. However, the vote was unanimous that January 1985 was too early for a military standard; the vote was on the question "is a MIL-STD in January 1985 premature?"

   b. This approach does not mesh well with the current commercial marketplace: It seems that there is not a lot of industry impetus right now for sharing tools. Most contractors currently have one or more host systems in-house which they use for development and maintenance of the systems they produce. The requirement to conform to the CAIS would mean a considerable delay and expense for everyone; it is easier to transport or even re-implement one tool than it is to re-do one's whole tool-suite/host-system to conform to the CAIS.

   The primary service concern is for simplifying life-cycle support. It was suggested that the services should accept

a demonstration of a system's maintainability in a service environment, rather than insist on use of a particular set of tools. It is argued that, especially in the short term, it will be more cost-effective to use what everyone already has in-house than to impose a new set of interfaces.

c. No cost/benefit analysis exists to support the DoD's perspective on potential payoff from this approach: The DoD has based much of the Ada program on the philosophy that the more that is common, the more will be shared and the more we will be able to cut costs. But we currently have no real answers to such questions as what it will cost to build a CAIS implementation or what it will take to effectively move tools between CAIS implementations.

The real question was whether or not the expected payoff was there. The suggestion was made that the DoD needs to do a cost analysis of the expense of implementing the CAIS versus the expense of writing all the intended tools in Ada for all the different operating systems currently in use for such work. It was suggested that this analysis should separate out the concept of writing tools in Ada from the further complication of writing them in Ada for the CAIS, since Ada itself is supposed to be transportable. The analysis should also include the cost of running the tools on a CAIS implementation which is piggy-backed on another operating system. Another aspect to be considered is the effect that the CAIS might have on how new tools are approached and written.

It must be kept in mind, however, that the expected benefits of the CAIS exceed the vision of a single contractor; they must be "amortized" over many projects conducted by many contractors for the DoD as a whole.

d. The means of transitioning to the CAIS is a key element in its success: The key point in this discussion was that, just as with the language, there must be a transition plan for the CAIS. This plan must handle the near-term problem of starting to get the CAIS in use while not requiring that existing complete, integrated environments be overhauled to include the CAIS. It was noted that we do not want to lose the opportunity to capture some new projects which are just now putting new development environments together, but we do not want to disrupt work that is currently being done successfully in existing environments. That is, the CAIS should be applied to new projects, and existing projects should not be expected to retrofit their existing facilities.

e. The real fear about making the CAIS a standard is that it will be recklessly or inappropriately applied: Given the vote in favor of a common set of interfaces but the

sentiment against a near-term MIL-STD, the question was
asked "If not a MIL-STD, then what? Would it be better as
an IEEE or ANSI standard, for example?" The consensus was
that the declaration of ANY standard - not just a military
one - at this point would cause everyone great concern.
The concern focuses on the expectation that a military
standard (or any standard, for that matter) would have an
aura of authority that would lead people to misapply it to
contract situations where it is not appropriate.

The conclusion was that the policy used in applying this
standard must be perceptive enough to apply it discriminately.
Particularly in the first year or two, it would be much more
appropriately applied to contracts for prototypes of tools and
CAIS implementations than to any contract for an applications
system in which the delivery of tools is more or less
incidental. A lot of these concerns might be alleviated if the
issuance of the CAIS standard were accompanied by guidelines
for its imposition on contracts.

6. **How does the CAIS relate to a "standard DoD operating system"?**

This question was not discussed much. It was pointed out that
the CAIS was being developed in response to a tri-service
memorandum of agreement calling for a set of interfaces to
support the sharing of tools. Any resemblance it may bear to a
"standard DoD operating system" is just the result of the fact
that the interfaces required to support transportability tend
to look a lot like a virtual operating system. There is no
known charter currently to develop a standard DoD operating
system nor any known interest in establishing such a charter.

7. **What are the real prospects for achieving CAIS validation?**

The prospect of CAIS validation appeared frightening to several
discussants for two main reasons: (1) the perceived complexity
of the CAIS and (2) the perceived lack of technology to define
the tests and experiments to achieve it. The basic consensus
was that we do not know enough yet. During the conversation it
became apparent that people were variously talking about one or
more of at least three things when they spoke of "CAIS
validation":

- validating that an implementation meets the specification
  (this is what the KIT, KITIA and E&V teams mean by "CAIS
  validation")

- "validating" the CAIS concepts, i.e., prototyping

- "validating" that two real CAIS implementations really
  achieve transportability; i.e., showing that the fact that
  two implementations conform to the CAIS means that they
  really can share tools.

8. There should be a demonstration of the distributivity of the
   CAIS.

   The general question was when and how would this and other
   deferred issues in the current CAIS be addressed.
   Distributivity was just one example. In particular, it was
   stated by one participant that the transparency to distribution
   which was pursued for CAIS Version 1 would not be adequate and
   that explicit control over such aspects would be required.
   These will all be topics for the Version 2 contractor.

9. The criteria for conformance of an implementation to the CAIS
   are not adequate.

   There was special concern expressed over what was called the
   "monolithicness" of the CAIS; i.e., the words concerning
   conformance in the current draft imply that an implementation
   must have all the interfaces exactly as shown, with a few minor
   exceptions in detail. It is recognized that it is not
   practical to require absolutely every package of absolutely
   every implementation, yet it is desirable for all things
   calling themselves "CAIS implementations" to have a substantial
   number of interfaces in common.

NOTE: This draft, dated 31 Oct 1984, prepared by KIT/KITIA
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)

MILITARY STANDARD

COMMON APSE INTERFACE SET (CAIS)

VERSION 1.4

Prepared by

KIT/KITIA

CAIS Working Group

for the

Ada Joint Program Office

(Ada is a Registered Trademark of the Department of Defense,

Ada Joint Program Office)

AREA ECRS

DEPARTMENT of DEFENSE

Washingtion, DC 20302

Common APSE Interface Set

PROPOSED MIL-STD-CAIS

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, by using the self addressed Standardization Document Improvement Proposal (DD Form 1426) 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 Environments (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, changing 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 Analysis), P. Oberndorf (Naval Ocean Systems Center), T. Taft (Intermetrics), R. Thall (SofTech) and W. Wilder (NAVSEA PMS-408). In February 1983 the design team was expanded to include LCDR. B. Schaar (Ada Joint Program Office) and KAPSE Interface Team from Industry and Academia (KITIA) members: H. Fischer (Litton Data Systems), T. Harrison (Texas Instruments), E. Lamb (Bell Labs), T. Lyons (Software Sciences Ltd., U.K.), D. McGonagle (General Electric), H. Morse (Frey Federal Systems), E. Ploedereder (Tartan Laboratories), H. Willman (Raytheon), and L. Yelowitz (Ford Aerospace). During 1984, the following people assisted in preparation of this document: K. Connolly (TRW), S. Ferdman (Data General), G. Fitch (Intermetrics), R. Gouw (TRW), B. Grant (Intermetrics), N. Lee (IDA), J. Long (TRW), and R. Robinson (IDA). Additional constructive criticism and direction was provided by G. Myers (Naval Ocean Systems Center), R. Olivier (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 those KITIA members and their companies for providing the time and resources that significantly contributed to this document.)

CONTENTS

v

vi

viii

ix

x

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. This version of the CAIS is intended to provide the basis for evolution of the CAIS as APSEs are implemented, as tools are transported, and as tool interoperability issues are encountered. Tools written in Ada, using only the packages described herein, should be transportable 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.

The CAIS applies to Ada Programming Support Environments , which are to become the basic software life-cycle environments for 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 inter-related 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]

[1] Requirements for Ada Programming Support Environments, "Stoneman"; Department of Defense; February 1980.
[2] KAPSE Interface Team: Public Report, Volume I, 1 April 1982; p. C1.

1.2 Content

This version of the CAIS establishes interface requirements for the transportability of Ada tool sets to be utilized in Department of Defense (DoD) APSEs. Strict adherence to this interface set will ensure that the Ada tool sets will possess the highest degree of portability 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. This version of 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.

b. Process Nodes. This area covers program invocation and control.

c. Input/Ouput. This area covers file input/output, basic device input/output support, special device control facilities, and interprocess communication.

d. Utilities. This area covers list operations useful for parameter and attribute value manipulation.

1.3 Excluded and deferred topics

During the design of this version 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 multi-lingual 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.

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 to have the CAIS enforce a particular configuration management approach and if so what particular methodology should

2

be chosen.

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 ANSI/ISO interfaces to adopt,
such as the ISO/DIS 7942 Graphical Kernel System (GKS).

Distributed environments - The existing CAIS packages
are intended to be implementable on a distributed set
of processors, but this would be 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.

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 to provide
process execution information.

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 host and
target.

Typing methodology - The current CAIS provides
attributes and relations which can be 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

3

made available to support it.

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

4

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.

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

(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 Institue, "Magnetic Tape Labels and File Structure for Information Interchange (ANSI Standard x3.27-1978)"; (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.

[IEEE] - IEEE Standard Glossary of Software Engineering Terminology, ANSI/IEEE Std 729-1983.

[TCSEC] - Department of Defense Trusted Computer System Evaluation Criteria, Department of Defense Computer Security Center, CSC-STD-001-83, 15 August 1983.

[UK Ada Study] - United Kingdom Ada Study Final Technical Report, Volume I, London, Department of Industry, 1981.

5

3. DEFINITIONS

The following are terms which are used in the description of the CAIS. Words followed by * are defined by the CAIS document in terms of the CAIS design. 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 with intents according to the access control rules and either permitting or denying the intended operation.

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

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

access control rules - The rules describing the correlations between access rights and intents.

access rights - Descriptions of the type of operations which can be performed (See Section 4.4.1.2).

accessible * - The subject has (adopted a role which has) been granted 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

6

life cycle, with particular emphasis on software for embedded computer applications. The principal features are the database, the interfaces and the toolset.

adopt a role * - The action of a process to acquire the access rights which have been or will be granted by an object to adopters of that role; in the CAIS this is accomplished by establishing a secondary ADOPTED_ROLE relationship from the process node to the role.

adopted role of a process * - The role or ancestor of the role that is the target of an ADOPTED_ROLE relationship from the process node.

advance (of an active postion) - Takes place on a scroll or page terminal whenever (1) the row number of a new position is greater than the row number of the old or, (2) the row number of the new position is the same but the column number of the new postion is greater than that of the old; takes place on a form terminal whenever the indices of its position are incremented.

ancestor of a role * - Any group reachable from the given role via PARENT relationships.

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

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

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

base * - The given starting node of a path.

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

current node * - The node that is currently the focus or context for the activities of the current process.

current process * - The currently executing process making the call to a CAIS operation. It defines the context in which the parameters of the call are to be interpreted.

current user * - The user's top-level node.

dependent process node * - A process node which is not a root process node.

descendant of a group * - Any role reachable from the given group via primary PERMANENT_MEMBER relationships.

7

3-35

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

device name * - The key of a primary DEVICE relationship emanating from the system node.

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

group *, group node * - A collection of roles represented by a structural node with emanating relationships identifying each of the group's members. A group member may be a user top-level node, a program node, or another group.

identification - A reference to a node provided by a pathname or a base/key/relation name.

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

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, required resources are allocated, and execution is started.

initiated process * - The process whose program is placed into execution.

initiating process * - The process placing a program into execution.

interface - [DACS] A shared boundary.

iterator * - A variable which facilitates 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.

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 items.

8

list item * - A data item in a list.

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

named * - A list item which has a name associated with it or a list all of whose items are named.

node * - Representation within the CAIS of an entity relevant to the APSE.

node handle * - A value that represents a reference (to a CAIS node) that 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.

open node handle * - A node handle that has been assigned to a particular node by means of an OPEN procedure.

parent * - The source node of a primary relationship.

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/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 encountered.

permanent member * - A group member which is related to the group via a primary PERMANENT_MEMBER relationship.

position - 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 role that is the target of a POTENTIAL_MEMBER relationship emanating from that group or from any of that group's descendants.

pragmatics - [- implementation pragmatics] Constraints imposed by an implementation or use 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

9

existence of a node is controlled by the existence of the primary relationship of which it is the target.

privilege specification - A list of access rights in accordance with the syntax given in Table III.

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, which may invoke subprograms declared in the other compilation units of the program.

program node * - A short-hand reference of the file node representing the file which contains the executable image of a program.

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

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

relation * - 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.

role * - A user top-level node, program node, or group node that is the target of a secondary ROLE relationship.

root process node * - The initial node created when a user enters 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.

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

10

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 node.·

system node * - The root of the entire CAIS node structure.

target node * - Node at which a relationship terminates.

task - [LRM] An entity whose execution proceeds in parallel with other parts of the program.

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

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 * - The root of the tree that includes all of the structural, file and process nodes created on the user's behalf; from it the user can access these and others. Each user has a top-level node.

track * - Open node handles are said to track the nodes to which they refer. This means that 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.

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

traversal of a relationship * - Following a relationship from its source node to its target node in the process of accessing a node along a path.

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

unique primary path * - Every accessible node can be traced back to a top-level node by recursively following PARENT relationships; the path obtained by inverting this chain is the unique primary path to the node.

unnamed * - A list item which has no name associated with it or a list all of whose items are unnamed.

unobtainable * - A node is unobtainable if the primary relationship of which it is the target has been deleted.

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

11

user name * -. The key of a primary USER relationship.

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 provides a single model for understanding the CAIS concepts; it provides a means for making facilities available equally to 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 also 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 the input/output model used in the CAIS, Section 4.6 describes practical limits and constraints not defined by the interfaces. Section 5 provides detailed descriptions of the interfaces.

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

Appendix B will provide a set of the Ada package specifications of the CAIS interfaces which compiles correctly. This set of interfaces will be extracted from the final construction of the CAIS design and included in the Military Standard 1 document.

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

  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

13

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. The actual Ada package specifications may differ from the canonical specifications in the following ways (which are indistinguishable to programs calling these interfaces):

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

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

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

    d. Packages may be instantiations of generic packages.

The following differences in CAIS package implementation from the specifications in this document are NOT permitted:

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

    b. Parameter mode IN being changed to IN OUT, as this prevents passing of expressions.

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

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

The interface semantics are described in most cases through narrative. These narratives are divided into up to 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 are describable in terms of the basic form of the interface and/or other CAIS interfaces, these versions are described in a paragraph, called Additional Interfaces, using Ada.

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,
       [Editor's Note: Typeset document only]

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.
[Editor's Note: Typeset version only; this version utilizes quotation marks in lieu of italics]

4.3 CAIS node model

The CAIS provides interfaces for administering entities relevant during the software lifecycle, such as files, directories, processes and devices. These entities have properties and may be interrelated in many ways.

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.

15

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 are an Ada external file. 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 are a representation of the execution of an Ada program. If a node is a "structural node", there are no contents and the node is used strictly as a holder of relationships and attributes.

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.

16

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 a CAIS interface 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 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 CAIS_STRUCTURAL_NODES, Section 5.2.2 CAIS_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. These 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. In this document, a relation name is often referred to simply as a relation and a relationship key is often referred to simply as a key.

Nodes in the environment are attainable by navigating along the relationships. Operations are provided to move from one node (along one of its relationships) to a connected node (see Section 4.3.4).

17

4.3.3.1 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. This initial relationship is marked as 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 the predefined PARENT relation. Primary relationships form a strictly hierarchical tree. There is no requirement that all primary relationships emanating from a node have the same relation name. The primary relationship is broken by DELETE_NODE or DELETE_TREE operations (see Section 5.1.2). After such deletion the node is said to be "unobtainable". A "non-existing node" is one which has never been created. RENAME operations (see Section 5.1.2.19) may be used to make the primary relationship emanate from a different parent. The operations DELETE_NODE, DELETE_TREE, RENAME, and the operations creating nodes are the only ones that manipulate primary relationships. They maintain a state in which each node has exactly one parent and a unique primary name (see Section 4.3.4). "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 (see Section 5.1.2.22) and broken with the UNLINK procedure. Secondary relationships may exist to unobtainable nodes.

4.3.3.2 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, 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 CAIS identifies the following basic predefined relations: PARENT, USER, DEVICE, JOB, CURRENT_JOB, CURRENT_USER, CURRENT_NODE.

The CAIS node model incorporates the notion of a user. Each user has one "top-level node" (similar to a file-system directory). This top-level node is an entry point to the CAIS directed node graph and from it the user can access other structural, file and process nodes.

The CAIS node model incorporates the notion of a "system" node which acts as the root of the entire CAIS node structure. Each top-level node is reachable from the system node along a primary relationship of the predefined relation USER emanating from the system node. The key of this relationship is the "user name". Each user name has a top-level node associated with it. The system node is not manipulable via the CAIS interfaces. It may only be manipulated by interfaces outside the CAIS, e.g., to add new USER relationships emanating from the system node.

18

A "user" may be an individual, project, or other organizational entity; this notion is not equated with an individual person. It is left to each CAIS implementation to set up a methodology for users to enter the APSE and for enforcing any constraints that limit the top-level nodes at which users may enter the APSE. After entering the APSE, the user will be regarded by the CAIS as the user associated with the top-level node at which he entered the APSE.

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 node along a primary relationship of the predefined relation DEVICE emanating from the system 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.

When a user enters the APSE, a "root process" node is created which often may represent a command interpreter or other user-communication process. A process node tree, spanned by primary relationships, develops from this root node as other processes (called "dependent" processes) are initiated for the user. A particular user may have entered the APSE several times 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 JOB relationship emanates from each user's top-level node to the root process node of each of the user's jobs.

The CAIS does not specify an interface for creating the initial root process node when a user enters the APSE, but the CAIS node model requires that secondary USER and DEVICE relationships with the appropriate user and device names as keys emanate from the root process node to all top-level nodes to which this user has access rights (see Section 4.4) and that these USER and DEVICE relationships are inherited by the rest of the process nodes in the job. Top-level nodes of other users and devices can therefore be reached from a process node using the relation USER or DEVICE and a relationship key which is interpreted as the respective user or device name.

Any process node in a job has associated with it at least three additional predefined secondary relationships. The predefined CURRENT_JOB relationship always points to the root node for a process node's job. The predefined CURRENT_USER relationship always points to the user's top-level node. The predefined CURRENT_NODE relationship always points 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 paths (see Section 4.3.4). All three of these relations (CURRENT_JOB, CURRENT_USER, and CURRENT_NODE) provide a convenient means for identifying (see Section 4.3.4) other CAIS nodes. The CAIS requires that the root process node created when the user enters the APSE has a CURRENT_NODE relationship pointing to the top-level node for the user.

The node model also uses the concept of "current process". This 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, paths are determined in the context of the current process.

4.3.4 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 starting node is called the "base" node. Every accessible node can be traced back to a top-level node by following PARENT relationships; the path obtained by inverting this chain is the "unique primary path" to the 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 encountered. The pathname associated with the unique primary path is called the "unique primary name" of the node. The base node of a path may be identified explicitly as an additional argument, the BASE, to many of the CAIS operations, signaling the starting point for interpretation of a pathname. Otherwise, the current process node is used as the starting point for interpretation of the pathname. The unique primary name of a node is syntactically identical to, and therefore can be used as, a pathname whose interpretation starts at the current process node.

"Identification" of a node is provided by a pathname or a base/key/relation name. The phrase "toidentify" means to provide an identification for a node. A node identification is considered "illegal" if either the pathname or the relationship key or relation name are syntactically illegal with respect to the syntax defined in Table I below. An illegal identification is treated as an identification for a non-existing node.

A pathname implies "traversalofanode" 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.

Many CAIS operations allow the omission of the relation name when referring to a relationship, defaulting it to 'DOT'. Relationship keys of DOT relationships may not be the empty string. Instances of the DOT relation are fully manipulable by the user within access right constraints. DOT relationships are not restricted to be primary relationships and are not associated with any other CAIS-specific semantics.

20

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. If the relationship key is the empty string, the parentheses may be omitted. Thus, 'PARENT and 'PARENT() refers to the same node. If the relation is DOT, then the path element may be represented simply by a dot ('.') followed by the key for the DOT relation. Thus, 'DOT(CONTROLLER) is the same as .CONTROLLER .

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 of the CURRENT_NODE with the relation name DOT and with the given key. Thus AIRPORT is the same as 'CURRENT_NODE.AIRPORT .

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

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

a. Landing_System'With_unit(Radar)

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

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

By convention, a relationship key ending in ' ' is taken to represent the LATEST_KEY (lexicographically last). When creating a node or relationship, use of ' ' to end the final key of a pathname will cause a key to be automatically assigned, lexicographically following all previous keys for the same relation and initial relationship key character sequence.

TABLE I. Pathname BNF

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


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


relation_name::=identifier
relationship_key::=identifier | identifier #
```

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

4.3.5 Attributes -

Both nodes and relationships may have attributes which provide information about the node or relationship. Attributes are identified by an attribute name. Each attribute (see Section 5.1.3 CAIS_ATTRIBUTES) has a list of the values assigned to it, represented using the CAIS_LIST_UTILITIES (see Section 5.4.1) type called LIST_TYPE.

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.

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.

4.4 Discretionary and mandatory access control

The CAIS specifies mechanisms for discretionary and mandatory access control. In the CAIS, the following operations constitute "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, querying the kind of a node, and traversing a node as implied by a pathname. The phrase 'reading relationships' is a convenient short-hand for 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.

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 their attributes, and querying the status of node handles to a node.

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

1. "access rights": descriptions of the types of operations which
        can be performed (See Section 4.4.1.2).

2. "access control rules": the rules describing the correlations
        between access rights and intents.

3. "access checking": the operation of checking access rights
        with intents 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."

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 being accessed.

These concepts and the CAIS mechanisms which support them are discussed in the following sections.

23
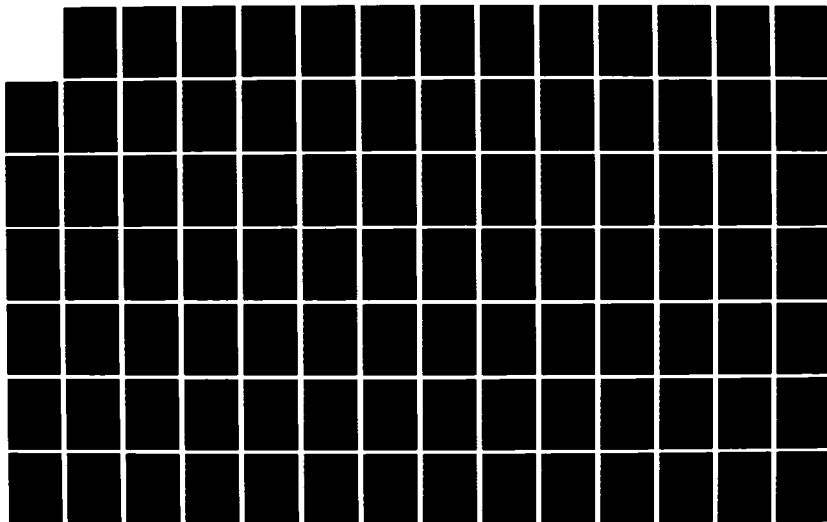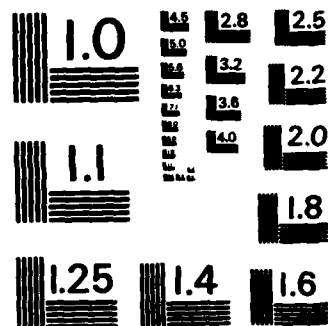
MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

4.4.1 Discretionary access control

Discretionary access control limits authorized access to nodes to named users or groups of users. The establishment of access rights to an object is performed by an authorized user; typically the creator of the object.

In the CAIS, an "object" is any node to be accessed and a "subject" is any process node (acting on the behalf of a given user) performing an operation requiring access to an object node.

Normally, a user may adopt one or more roles in order to affect some purpose. In these roles, the user may be acting as himself, acting on behalf of another person, acting as a member of some group, or acting with some rights granted him as a result of the function being performed. The roles a user has adopted are one factor used in determining access rights. In the CAIS, a subject (process node) may act in the capacity of one or more roles. Each role identifies a CAIS user, a program being executed, or a particular group of users, programs, or sub-groups.

In the CAIS, a "role" may be a user top-level node, a program node, or a group node. A "program node" is the file node containing the executable image of the program. Roles can be grouped by a "group node" which is a structural node with emanating relationships identifying each of the group's members; a group member may be either an individual user node, a program node, or another group node. For example, a group node may have as its members all nodes representing compatible versions of the compiler or a collection of tools designed to operate on a given type of data, or a group node may have as its members all nodes representing individuals on a particular project or all subunits of a particular organization.

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 primary PERMANENT_MEMBER relationship is used to identify those members that are considered "permanent members" of a given group. The secondary relationship POTENTIAL_MEMBER is used to identify those members that may dynamically acquire membership in the group. The phrase "potential member" of a group refers to any role that is the target of a POTENTIAL_MEMBER relationship from that group or from any of that group's descendants.

The primary relation PERMANENT_MEMBER may be used to create a hierarchy of roles by defining members of a group that are themselves groups. A user top-level node may not be the target of a primary PERMANENT_MEMBER relationship emanating from a group node, due to the restriction that user top-level nodes must have the system node as their parent. The phrase "descendant of a group" refers to any role reachable from the given group via primary PREMANENT_MEMBER relationships. The phrase "ancestor of a role" refers to any group reachable from the given role via PARENT relationships. Hence, only nodes that represent permanent

members of a group have ancestors.

The CAIS discretionary access control model requires that, upon creation
of a root process node, secondary relationships of the predefined
relation ROLE emanate from the created root process node to an
implementation-defined set of roles; at least the secondary ROLE
relationship with the user name as key must be established from the root
process node to the user top-level node. These ROLE relationships are
inherited by the process nodes initiated on behalf of the user (similar
to USER and DEVICE relationships). Roles can therefore be reached from
a process node with the relation ROLE and a relationship key interpreted
to be the role name.

4.4.1.1 Adopting a role

When a process "adopts" a particular role, a secondary relationship of
the predefined relation ADOPTED_ROLE is created from the process node to
the role. There may be multiple ADOPTED_ROLE relationships emanating
from a process node. The phrase "adopted role of a process" refers to
the role , or ancestor of the role, that is the target of an
ADOPTED_ROLE relationship from the process node. Roles are adopted
either implicitly or explicitly. When a process is created, it
implicitly adopts the program node for the program it is executing.
When a root process node is created, it implicitly adopts its current
user node. When any process node is created, it implicitly inherits the
ADOPTED_ROLE relationships of the node of its creating process. A
process may explicitly adopt a group using the ADOPT procedure (Section
5.1.4.4). For a process to adopt a given group, some other role the
process has already adopted must be a potential member of the group to
be adopted.

4.4.1.2 The access relationship and the privilege attribute

An access relationship is any secondary relationship of the predefined
relation ACCESS from an object to a role. Any object may have zero or
more access relationships. Each access relationship has a predefined
privilege attribute, called GRANT, which specifies what access rights to
the object are granted to adopters of the role.

The privilege attribute value consists of a list of "privilege
specifications". Each privilege specification consists of a necessary
privileges list, followed by a right-arrow (=>), followed by a resulting
privileges list. A list with one element may be replaced by the element
alone. If the necessary privileges list is empty, the privilege
specification may be replaced by the resulting privilege list alone.
The for the BNF privilege specifications is given in Table II.

25

```
                    TABLE II. Privilege Specification BNF


    privilege_specification::= ( [ necessary_privileges => ]
                                  resulting_privileges )

    necessary_privileges::= privilege_list
    resulting_privileges::= privilege_list

    privilege_list      ::= identifier |
                            ( identifier { , identifier} )




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

The necessary and resulting privileges lists are lists of privilege
names. Privilege names imply the granting of certain access rights. A
privilege name has the syntax of an Ada identifier. Privilege names may
be user-defined, but certain privilege names have special significance
to CAIS operations. In particular, the CAIS recognizes the privilege
names given in Table III and the access rights for which they are
necessary or sufficient.

TABLE III. Privilege Names and Access Rights

EXISTENCE	Without this access right, the object is inaccessible to the subject. Without additional access rights, subject may neither read nor write attributes, relationships or contents of the object.
READ_RELATIONSHIPS	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. Necessary to open the object with intent READ_RELATIONSHIPS.

26

WRITE_RELATIONSHIPS	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. Necessary to open the object with intent WRITE_RELATIONSHIPS.
READ_ATTRIBUTES	subject may read attributes of the object; the access right EXISTENCE is implicitly granted Necessary to open the object with intent READ_ATTRIBUTES.
WRITE_ATTRIBUTES	subject may create, write, or delete attributes of the object; the access right EXISTENCE is implicitly granted. Necessary to open the object with intent WRITE_ATTRIBUTES.
READ_CONTENTS	subject may read contents of the object; the access right EXISTENCE is implicitly granted. Necessary to open the object with intent READ_CONTENTS.
WRITE_CONTENTS	subject may write contents of the object; the access right EXISTENCE is implicitly granted. Necessary to open the object with intent WRITE_CONTENTS.
READ	union of READ_RELATIONSHIPS, READ_ATTRIBUTES, READ_CONTENTS, and EXISTENCE access rights. Necessary to open the object with intent READ. Sufficient to open the object with intent READ_RELATIONSHIPS, READ_ATTRIBUTES, or READ_CONTENTS.
WRITE	union of WRITE_RELATIONSHIPS, WRITE_ATTRIBUTES, WRITE_CONTENTS, and EXISTENCE access rights. Necessary to open the object with intent WRITE. Sufficient to open the object with intent WRITE_RELATIONSHIPS, WRITE_ATTRIBUTES, or WRITE_CONTENTS.
EXECUTE	subject may create a process that takes the object as its executable image; the access right EXISTENCE is implicitly granted. Necessary to open the object with intent EXECUTE.
CONTROL	subject may modify access control information of the object; the access right EXISTENCE is implicitly granted. Necessary to open the object with intent CONTROL.

The following are examples of privilege specifications:

    (READ, WRITE, APPEND_MAIL)
    (COMPILE, CONTROL)
    ((EDIT, COMPILE)=>(READ, WRITE_CONTENTS))
    (READMAIL=>(READ, WRITE), SENDMAIL=>APPEND)

Access relationships and privilege attributes are established for
objects using the interfaces provided in the package CAIS_ACCESS_CONTROL
or may be established at node creation. When a node is created, the
initial access control information may be supplied by the ACCESS_CONTROL
parameter. If non-null, this parameter specifies the initial access
control information to be established for the created node, using named
Ada aggregate syntax. Each named choice given in the ACCESS_CONTROL
parameter identifies a ROLE relationship key. Each selected expression
identifies a privilege specification. For each of the component
associations, an access relationship is created from the created node to
a role identified by the pathname built from the relation ROLE and the
relationship key given by the choice. The privilege specification is
the initial value of the GRANT attribute of the access relationship.

4.4.1.3 Discretionary access checking

When access control is enforced for a given operation, each
discretionary access right required for each object involved in the
operation is compared to the process's access rights as defined by the
object access relationships. If the object has an access relationship
to a role that is an adopted role of the subject and the relationship
allows the access right being checked, then the operation is allowed.
Otherwise the operation is not allowed, and the operation is terminated
by raising an exception.

For an access relationship to grant an access right, the access right
must appear in a resulting privilege list in a GRANT attribute of the
relationship, and the access rights in the associated required privilege
list must have been granted.

4.4.2 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.

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 constraints 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.

4.4.2.1 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 node's 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 limited functionality 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. 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 participant classification attribute is given in Table IV.

TABLE IV. Participant 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  |     - spearates alternatives
```

The hierarchical classification level set and the non-hierarchical
category set are implementation-defined.

4.4.2.2 Labeling of subject 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.

4.4.2.3 Labeling of object 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.

4.4.2.4 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 process opening one of these nodes is assigned to the file node while it is open.

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 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 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.

4.4.2.5 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 Input and output

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

31

4.5.1 CAIS file nodes

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 this section provide facilities that operate on CAIS external files.

There are four types of CAIS supported Ada external files: secondary storage, queue, terminal, and magnetic tape.

4.6 Pragmatics

4.6.1 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. Nevertheless, it is important to establish certain minimums for each type to enhance portability.

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

b. RELATIONSHIP_KEY   At least 20 leading characters must be
                           significant in a (relationship) key.

c. ATTRIB_NAME   At least 20 leading characters must be
   RELATION_NAME   significant in attribute and relation names.

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

e. Record size number   At least 32767 bits per record must be
                           supported.

f. Open node count   Each process must be able to have at least
                           15 nodes open simultaneously.

4.6.2 Pragmatics for CAIS_SEQUENTIAL_IO

A conforming 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 32767. A conforming implementation must also support instantiation with unconstrained record types which have default constraints and a maximum size in bits of at least 32767, and may (but need not) use variable length elements to conserve space in the external file.

4.6.3 Pragmatics for CAIS_DIRECT_IO

Each element of a direct-access file is selected by an integer index of type COUNT. A conforming implementation must at least support a range of indices from one to 32767 (215-1).

A conforming 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 32767. A conforming implementation must also support instatiation with unconstrained record types which have default constraints and a maximum size in bits of at least 32767, and may (but need not) use variable length elements to conserve space in the external file.

4.6.4 Pragmatics for CAIS_TEXT_IO

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

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 manipulation of nodes in general, of relationships and of attributes. These interfaces are defined in three packages: CAIS_NODE_DEFINTIONS defines types, subtypes, exceptions, and constants used throughout the CAIS; CAIS_NODE_MANAGEMENT defines interfaces for the manipulation of nodes in general, of relationships and of attributes; and CAIS_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 an OPEN procedure, causing an "open node handle" to be assigned to the object. Most procedures expect either a parameter of type NODE_TYPE, or 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.

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.

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.

The key of a node is the relationship key of the last element of its pathname.

34

5.1.1 Package CAIS_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_CONTENT, WRITE_CONTENT,
     APPEND_CONTENT, 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_CONTENT,
     EXCLUSIVE_WRITE_CONTENT, EXCLUSIVE_APPEND_CONTENT,
     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;
```

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 5.1.2.1.
INTENTION is the type of parameter INTENT of CAIS subprograms OPEN and
CHANGE_INTENT, as further explained in Section 5.1.2.1.

NAME_STRING,   RELATIONSHIP_KEY,   RELATION_NAME, and FORM_STRING are
subtypes for pathnames, relationship keys, and relation names, as well
as for form strings used for the notation of aggregates of attribute
values (c.f. [LRM] 14). The value of such strings is subject to
certain syntactic restrictions whose violation causes exceptions to be
raised.

```
TOP_LEVEL        : 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";
```

TOP_LEVEL, 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.

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

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

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, 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.

LAYOUT_ERROR is raised whenever an error is encountered with regard to layouts.

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

ACCESS_VIOLATION is raised whenever an operation is attempted which violates access right constraints other than knowledge of existence of the node.

INTENT_VIOLATION is raised whenever an operation is attempted on an open node handle which is in violation of the intent specified when the node handle was opened. SECURITY_VIOLATION is raised whenever an operation is attempted which violates mandatory access controls for 'write' operations.

5.1.2 Package CAIS_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 CAIS_STRUCTURAL_NODES (Section 5.1.5), the creation of nodes for processes is performed by INVOKE_PROCESS and SPAWN_PROCESS of CAIS_PROCESS_CONTROL (Section 5.2.2), and the creation of nodes for files is performed by the CREATE procedures of the input/output packages

(Section 5.3).

There are three CAIS interfaces for manipulating node handles; OPEN opens a node handle, CLOSE closes the node handle, and CHANGE_INTENT alters the specification of the intention of node handle usage. These interfaces perform access synchronization in accordance with an intent specified by the parameter INTENT.

These interfaces are central to the general node administration, since 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 defaulted TIME_LIMIT parameter.

One or more of the intentions 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 node only. Locks on the node have no delaying effect.

READ, EXCLUSIVE_READ:
   The OPEN operation is 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 all opens with write intent. In addition, the OPEN operation is delayed if there are open node handles to the node with write intent.

WRITE, EXCLUSIVE_WRITE:
   The OPEN operation is delayed if the node, its contents, attributes 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 all opens with read, write or append intent. In addition, the OPEN operation is delayed if there are open node handles to the node with read, write or append intent.

READ_CONTENTS, EXCLUSIVE_READ_CONTENTS:
   The OPEN operation is 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. In addition, the OPEN operation is delayed if there are open node handles to the node with intent to write its contents.

37

WRITE_CONTENTS, EXCLUSIVE_WRITE_CONTENTS:
The OPEN operation is 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 all opens with read, write or append intent. In addition, the OPEN operation is delayed if there are open node handles to the node with intent to read, write or append its contents.

APPEND_CONTENTS, EXCLUSIVE_APPEND_CONTENTS:
The OPEN operation is 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 all opens with append or write intent. In addition, the OPEN operation is delayed if there are open node handles to the node with intent to append or write its contents.

READ_ATTRIBUTES, EXCLUSIVE_READ_ATTRIBUTES:
The OPEN operation is 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 all opens with intent to write attributes. In addition, the OPEN operation is delayed if there are open node handles to the node with intent to write attributes.

WRITE_ATTRIBUTES, EXCLUSIVE_WRITE_ATTRIBUTES:
The OPEN operation is 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. For EXCLUSIVE_WRITE_ATTRIBUTES, the node is locked against all opens with intent to read, write or append attributes. In addition, the OPEN operation is delayed if there are open node handles to the node with intent to read, write or append attributes.

APPEND_ATTRIBUTES, EXCLUSIVE_APPEND_ATTRIBUTES:
The OPEN operation is 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 all opens with intent to write or append attributes. In addition, the OPEN operation is delayed if there are open node handles to the node with intent to write or append attributes.

READ_RELATIONSHIPS, EXCLUSIVE_READ_RELATIONSHIPS:
The OPEN operation is delayed if the node or its relationships are locked against read operations. The established access right for subsequent operations is to read node relationships. For EXCLUSIVE_READ_RELATIONSHIPS, the node is locked against all opens with intent to write relationships. In addition, the OPEN

operation is delayed if there are open node handles to the node
with intent to write relationships.

WRITE_RELATIONSHIPS, EXCLUSIVE_WRITE_RELATIONSHIPS:
The OPEN operation is 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. For EXCLUSIVE_WRITE_RELATIONSHIPS, the node is
locked against all opens with intent to read, write or append
relationships. In addition, the OPEN operation is delayed if
there are open node handles to the node with intent to read,
write, or append relationships.

APPEND_RELATIONSHIPS, EXCLUSIVE_APPEND_RELATIONSHIPS:
The OPEN operation is delayed if the node or its relationships
are locked against append operations. The established access
right for subsequent operations is to create node relationships.
For EXCLUSIVE_APPEND_RELATIONSHIPS, the node is locked against
all opens with intent to write or append relationships. In
addition, the OPEN operation is delayed if there are open node
handles to the node with intent to write or append relationships.

CONTROL, EXCLUSIVE_CONTROL:
The OPEN operation is delayed if the node or its relationships
are locked against write or access-control operations. The
established access right for subsequent operations is to read and
alter access control information. For EXCLUSIVE_CONTROL, the node
is locked against all opens to write node contents, attributes or
relationships, or to modify access control information. In
addition, the OPEN operation is delayed if there are open node
handles to the node with intent to write node contents,
attributes or relationships, or to read or modify access control
information.

EXECUTE:
The OPEN operation is 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.

3-67

Table VI. presents an overview of interfaces to query nodes and manipulate primary and secondary relationships.

Table VI. Query/Manipulation Interfaces

Pathname queries	The following interfaces allow certain queries about pathnames. None of these interfaces performs accesses to nodes; they perform pathname manipulations at the syntactic level only.
	These interfaces can also be used to establish the syntactic legality of a pathname.
	function BASE_PATH function LAST_RELATION function LAST_KEY
Node queries	The following interfaces allow certain queries about nodes.
	function OBTAINABLE function IS_SAME procedure GET_PARENT
Node duplication interfaces	The following two interfaces can be used to duplicate single nodes or trees of nodes spanned by primary relationships.
	procedure COPY_NODE procedure COPY_TREE
Alteration of primary relationships	The following interface can be 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. Removal of the primary relationship to a node makes the node unobtainable. The semantics of the CAIS allow, but do not force, individual implementations of the CAIS to delete the physical representation of unobtainable nodes.

procedure DELETE_NODE
procedure DELETE_TREE

Creation and deletion of secondary relationships

The following two interfaces allow the creation and deletion of user-defined secondary relationships.

procedure LINK
procedure UNLINK

Node iterators

The following definitions and interfaces allow allow the iteration over nodes reachable from a given node via its emanating relationships of the specified relation name and relationship key patterns.

procedure ITERATE
function MORE
procedure GET_NEXT

Manipulation of the CURRENT_NODE relationship

The following two interfaces allow changes of the CURRENT_NODE relationship emanating from the current process node and obtaining an open node handle on the that is the target of the CURRENT_NODE relationship.

procedure SET_CURRENT_NODE
procedure GET_CURRENT_NODE

5.1.2.1 Opening a node handle —

```
procedure OPEN (NODE:        in out NODE_TYPE;
                NAME:        in     NAME_STRING;
                INTENT:      in     INTENTION := (READ);
                TIME_LIMIT:  in     DURATION := DURATION'FIRST);

procedure OPEN (NODE:        in out NODE_TYPE;
                BASE:        in     NODE_TYPE;
                KEY:         in     RELATIONSHIP_KEY;
                RELATION:    in     RELATION_NAME :=
DEFAULT_RELATION;
                INTENT:      in     INTENTION := (READ);
                TIME_LIMIT:  in     DURATION := DURATION'FIRST);
```

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 given 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

41

3-69

node. A delayed OPEN call completes when the node is unlocked or
the specified time limit has elapsed.

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 any traversed node in the path specified by NAME is syntactically illegal or the node specified by BASE is unobtainable, inaccessible or non-existent, existing, or if the relationship specified by RELATION and KEY or by the last path element implied by NAME does not exist. NAME_ERROR is also raised if the node to be opened is inaccessible or unobtainable and the given INTENT is not (EXISTENCE).

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

STATUS_ERROR
: is raised if the NODE is already open prior to the call on OPEN or if the 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 BASE, preventing the reading of relationships emanating from these nodes.

ACCESS_VIOLATION
: is raised if the current process's discretionary access control rights are insufficient to traverse the path specified by NAME or BASE/KEY, 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 specified by 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 opened node pointed to is renamed (potentially by another process), the operations on the opened node handle track the renaming.

42

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 corresponding OPEN or CHANGE_INTENT operation.
  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 actual parameter to the
  formal NODE parameter of OPEN.

5.1.2.3 Changing the specified intent of node handle usage -

    procedure CHANGE_INTENT (NODE: in out NODE_TYPE;
                             INTENT: in   INTENTION;
                             TIME_LIMIT:in   DURATION := DURATION'FIRST);

Purpose:
  This procedure changes the specified intent of usage 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 a node handle referring to the
  same node as referred to prior to the call (see
  the issue explained in the note below).

Parameter:
  NODE   is an open node handle
  INTENT   is a specification of the usage intent as for OPEN
  TIME_LIMIT is a duration for the maximum delay of the operation
             caused by locks, as for OPEN

Exceptions:
  NAME_ERROR is raised if the node to be opened is unobtainable
              and INTENT is not (EXISTENCE).
  STATUS_ERROR is raised if NODE is not an open node handle.

43

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 the current process's
                discretionary access control rights are insufficient
                to obtain access to the node consistent with the
                specified by 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 to the
                node specified by 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:

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 open status of node handle -

    function IS_OPEN (NODE: in NODE_TYPE) return BOOLEAN;

Purpose:
  This function returns TRUE or FALSE according to the open status
  of the node handle NODE.

Parameter:
  NODE   is a node handle.

Exceptions: none

5.1.2.5 Examining kind of node -

    function KIND   (NODE: in NODE_TYPE) return NODE_KIND;

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

Parameter:
  NODE   is an open node handle.

Exceptions:
  STATUS_ERROR   is raised if the node handle NODE is not open.
  INTENT_VIOLATION is raised if the node was not opened with an
                   intent establishing the right to read attributes.

44

5.1.2.6 Obtaining unique primary name -

 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 or unobtainable.
 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's 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.

5.1.2.7 Obtaining 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 path to 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's discretionary
                 access control rights are insufficient to obtain
                 access to the node's parent consistent with intent to

45

3-73

READ_RELATIONSHIP. ACCESS_VIOLATION is raised only if the conditions for NAME_ERROR are not present.

5.1.2.8 Obtaining 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 path to 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's 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.

5.1.2.9 Obtaining relationship key of last relation traversed -

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

Purpose:
  This function returns the relationship key of the last path element of the path used in opening this node handle.

Parameter:
  NODE   is an open node handle.

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

46

5.1.2.10 Obtaining relation name of last relation traversed -

    function PATH_RELATION (NODE: in NODE_TYPE) return RELATION_NAME;

Purpose:
  This function returns the relation name of the last path element of
  the path used in opening this node handle.

Parameter:
  NODE   is an open node handle.

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

47

5.1.2.11 Obtaining a partial pathname

    function BASE_PATH   (NAME: NAME_STRING) return NAME_STRING;

Purpose:
  This function checks the syntactic legality of the pathname NAME.
  It 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.

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

Exceptions:
  NAME_ERROR is raised if NAME is a syntactically illegal pathname.

5.1.2.12 Obtaining the name of the last relationship in a pathname

    function LAST_RELATION (NAME: NAME_STRING) return RELATION_NAME;

Purpose:
  This function checks the syntactic legality of the pathname NAME.
  It 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.

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 key of the last relationship in a pathname

    function LAST_KEY   (NAME: NAME_STRING)   return RELATIONSHIP_KEY;

Purpose:
  This function checks the syntactic legality of the pathname NAME.
  It 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.

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

Exceptions:
  NAME_ERROR is raised if NAME is a syntactically illegal pathname.

48

5.1.2.14 Querying existence of node

    function OBTAINABLE   (NODE: NODE_TYPE)   return BOOLEAN;

Purpose:
 This function returns TRUE if the node identified by NODE is not
 unobtainable and not inaccessible. It returns FALSE 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 OBTAINABLE   (NAME: NAME_STRING) return BOOLEAN
    is
        NODE: NODE_TYPE;
        RESULT: BOOLEAN;
    begin
        OPEN(NODE, NAME, (EXISTENCE));
        RESULT := OBTAINABLE(NODE);
        CLOSE(NODE);
        return RESULT;
    exception
        when others => return FALSE;
    end OBTAINABLE;

    function 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, (EXISTENCE));
        RESULT := OBTAINABLE(NODE);
        CLOSE(NODE);
        return RESULT;
    exception
        when others => return FALSE;
    end 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 5.1.2.1.1).

49

5.1.2.15 Querying sameness

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

Purpose:
  This function returns true or false, depending on whether the nodes
  identified by its arguments are the same node.

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 the node handles NODE1 or NODE2 are 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, (EXISTENCE));
    begin
        OPEN(NODE2, NAME2, (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.

5.1.2.16 Obtaining open node handle to parent node

```
procedure GET_PARENT (PARENT:    in      NODE_TYPE;
                      NODE:      in out  NODE_TYPE;
                      INTENT:    in      INTENTION := (READ);
                      TIME_LIMIT: in     DURATION := DURATION'FIRST);
```

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)).

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 maximum delay on waiting for the unlocking of
               the node in accordance with the specified intent.

Exceptions:
 NAME_ERROR   is raised if the node identified by NODE is a
               top-level node or if its parent node is inaccessible.
 STATUS_ERROR is raised if the node handle PARENT is open prior to
               the call or if the node handle NODE is not open.
 USE_ERROR   is raised if INTENT is an empty intent
               specification.
 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's discretionary
               access control rights are insufficient to obtain
               access to the parent node with the specified INTENT.
 SECURITY_VIOLATION is raised if the attempt to gain EXISTENCE
               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.

5.1.2.17 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 into 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 PARENT relationship (which is appropriately adjusted),
are recreated in the copy. If the target of the original node's
relationship is the node itself, then the copied relationship still
refers to the same target node. If the target is the node itself,
then the copy 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.
  STATUS_ERROR is raised if the node handles FROM and TO_BASE are
                not 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

52

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), (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;
```

5.1.2.18 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 nodes formed by primary
relationships emanating from the node identified by the 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 old (uncopied) node. Secondary relationships
emanating from a node which is not copied, but which refer to nodes
inside the tree being copied, are unaffected.

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

53

3-81

                            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
                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
                open.
    USE_ERROR   is raised if the original node is not a file or
                structural node.
    LOCK_ERROR is raised if any node to be copied 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's discretionary
                access control rights are insufficient to obtain access
                to each node to be copied with intent READ.
    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), (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;

54

5.1.2.19 Renaming 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 effectively changes the unique primary name 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 the new primary relationship to the renamed node emanates.
NEW_KEY   is a relationship key for the new primary relationship.
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).
STATUS_ERROR is raised if the node handles NODE and NEW_BASE are not open.
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), (APPEND_RELATIONSHIPS));
   COPY_TREE(FROM, NEW_BASE, LAST_KEY(NEW_NAME),
                                     LAST_RELATION(NEW_NAME));
   CLOSE(NEW_BASE);
exception
   when others =>
      CLOSE(NEW_BASE);
      raise;
end RENAME;
```

Notes:
   Open node handles from existing processes track the renamed node.

5.1.2.20 Deleting 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 it is not yet TERMINATED
   (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 was not opened with an
               intent including EXCLUSIVE_WRITE.
   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
               for NAME_ERROR are not present.

56

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
    NODE: NODE_TYPE;
begin
    OPEN(NODE, NAME, (EXCLUSIVE_WRITE));
    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
node in a single operation. It is left to an implementation
decision, whether and when nodes whose primary relationships have
been broken are deleted. However, secondary relationships to such
nodes must remain until they are explicitly deleted using the UNLINK
procedures.

5.1.2.21 Deleting 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
whose parent is the the designated node. The order in which the
deletions of primary relationships is performed is not specified.
If the operations raise 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 nodes to be deleted are
              inaccessible.
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 specified by NODE cannot be

57

obtained or if access, with intent EXCLUSIVE_WRITE,
cannot be obtained to any node whose unique primary
path traverses the node identified by NODE, due to an
existing lock on the node.

INTENT_VIOLATION is raised if the NODE was not opened with an
intent including EXCLUSIVE_WRITE intent.

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 node 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));
    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.

5.1.2.22 Creating user-defined 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.

58

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 a relationship key for the new secondary relationship.

NEW_RELATION   is a relation name for the new secondary relationship.

Exceptions:

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.

STATUS_ERROR   is raised if the node handles NODE or 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    (OLD_NAME:     in NAME_STRING;
                   NEW_NAME:     in NAME_STRING)
   is
       NODE,
       NEW_BASE: NODE_TYPE;
   begin
       OPEN(NODE, OLD_NAME, (EXISTENCE));
       OPEN(NEW_BASE, BASE_PATH(NEW_NAME), (APPEND_RELATIONSHIPS));
       LINK(NODE, NEW_BASE, LAST_KEY(NEW_NAME),
                                        LAST_RELATION(NEW_NAME));
       CLOSE(NEW_BASE);
       CLOSE(NODE);
   exception
       when others =>
           CLOSE(NEW_BASE);
           CLOSE(NODE);
           raise;
   end LINK;
```

59

5.1.2.23 Deleting user-defined 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.
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.

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), (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.

5.1.2.24 Iteration types and subtypes

```
type NODE_ITERATOR is 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/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.

5.1.2.25 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. The nodes are returned in ASCII lexicographical order by relation name and then by relationship key. Nodes that are of a different kind than the KIND specified are omitted. If PRIMARY_ONLY is true, then only primary relationships are considered when creating the iterator.

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 selected by ITERATE.
  KEY   is the pattern for the relationship keys of nodes 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, only primary relationships will be used in constructing the iterator; if FALSE, all relationships satisfying the patterns will be used.

Exceptions:
  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.
  SECURITY_VIOLATION is raised if the operation represents a

61

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

Additional Interface:

```
procedure ITERATE(ITERATOR:    out NODE_ITERATOR;
                  NAME:        in  NAME_STRING;
                  KIND:        in  NODE_KIND;
                  KEY:         in  RELATIONSHIP_KEY_PATTERN := "*";
                  RELATION:    in  RELATION_NAME_PATTERN :=
                                        DEFAULT_RELATION;
                  PRIMARY_ONLY:in  BOOLEAN := TRUE)
is
    NODE: NODE_TYPE;
begin
    OPEN(NODE, NAME, (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 (and
subsequently delete) relationships to inaccessible or unobtainable
nodes.

5.1.2.26 Determining iteration status

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

Purpose:
The function MORE returns TRUE or FALSE, depending on whether all
nodes contained in the node iterator have been retrieved with the
GET_NEXT procedure.

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.

62

5.1.2.27 Getting the next node in an iteration

```
procedure GET_NEXT(ITERATOR:   in out   NODE_ITERATOR;
                   NEXT_NODE:   in out   NODE_TYPE;
                   INTENT:      in       INTENTION := (EXISTENCE);
                   TIME_LIMIT:  in       DURATION := DURATION'FIRST);
```

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 of 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.
  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 maximum delay on waiting for the unlocking of
               the node in accordance with the specified intent.

Exceptions:
  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's discretionary
               access control privliges are insufficient to obtain
               access to the parent node with the specified INTENT.

5.1.2.28 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 CURRENT_NODE relationship 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:

63

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.
ACCESS_VIOLATION is raised if the current process does not have
sufficient discretionary access control rights to obtain
access to the current process node 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 SET_CURRENT_NODE(NAME: in     NAME_STRING)
is
    NODE: NODE_TYPE;
begin
    OPEN(NODE, NAME, (EXISTENCE));
    SET_CURRENT_NODE(NODE);
exception
    when others =>
        CLOSE(NODE);
        raise;
end SET_CURRENT_NODE;
```

5.1.2.29 Getting an open node handle to the CURRENT_NODE

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

Purpose:
This procedure returns in NODE an open node handle to the current
node of the current process; the node handle is opened with intent
EXISTENCE.

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

Exceptions:
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.
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:

64

The call on GET_CURRENT_NODE is equivalent to
OPEN(NODE, "'CURRENT_NODE", (EXISTENCE)).

5.1.3 Package CAIS_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 of the format
defined by the package CAIS_LIST_UTILITIES (see Section 5.4).
Upper/lower case distinctions are significant within the value of
attributes, but not within the attribute name.

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 identify a relationship implicitly by the last path
element of a pathname or explicitly by base node, key and relation name
identification.

Any syntactically illegal attribute name is treated as the name of a
non-existing attribute.

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 name given is
                syntactically illegal.
  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, (APPEND_RELATIONSHIPS));
   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 set 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.
  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/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.
  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.

66

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), (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.

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.
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;
                                      VALUE:     in LIST_TYPE)
    is
        NODE: NODE_TYPE;
    begin
        OPEN(NODE, NAME, (WRITE_RELATIONSHIPS));
        DELETE_NODE_ATTRIBUTE(NODE, ATTRIBUTE, VALUE);
        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.
  KEY   is the relationship key of the affected relationship.
  RELATION   is the relation name of the affected relationship.
  ATTRIBUTE   is the attribute name of hte attribute to be deleted.

Exceptions:
  NAME_ERROR   is raised if the relationship identified by the
                BASE/KEY/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 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 DELETE_PATH_ATTRIBUTE(NAME:      in NAME_STRING;
                                    ATTRIBUTE: in ATTRIBUTE_NAME;
                                    VALUE:     in LIST_TYPE)
```

68

```
    is
        BASE: NODE_TYPE;
    begin
        OPEN(BASE, BASE_PATH(NAME), (WRITE_RELATIONSHIPS));
        DELETE_PATH_ATTRIBUTE(BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
                              ATTRIBUTE, VALUE);
        CLOSE(BASE);
    exception
        when others =>
            CLOSE(BASE);
            raise;
    end DELETE_PATH_ATTRIBUTE;
```

5.1.3.5 Setting node attributes

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

Purpose:
This procedure sets the value of the node attribute named by
ATTRIBUTE to the value given by VALUE. The node is identified by an
open node handle NODE.

Parameters:
NODE   is an open node handle to a node 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.
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, (WRITE_ATTRIBUTES));
        SET_NODE_ATTRIBUTE(NODE, ATTRIBUTE, VALUE);
        CLOSE(NODE);
```

69

```
    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 a relationship attribute named by ATTRIBUTE to
  the value specified by VALUE. The relationship is identified
  explicitly by the BASE node, 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/RELATION parameters does not exist.
  USE_ERROR is raised if the node 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 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), (WRITE_RELATIONSHIPS));
        SET_PATH_ATTRIBUTE(BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
```

70

3-98

```
                        ATTRIBUTE, VALUE);
        CLOSE(BASE);
    exception
        when others =>
            CLOSE(BASE);
            raise;
    end SET_PATH_ATTRIBUTE;
```

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 assigns the value of the node attribute named by
  ATTRIBUTE to the parameter VALUE. The node is identified by open
  node handle NODE.

Parameters:
  NODE   is an open node handle to a node 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 LIST_TYPE)
    is
        NODE: NODE_TYPE;
    begin
        OPEN(NODE, NAME, (READ_RELATIONSHIPS));
        GET_NODE_ATTRIBUTE(NODE, ATTRIBUTE, VALUE);
        CLOSE(NODE);
    exception
        when others =>
            CLOSE(NODE);
            raise;
    end GET_NODE_ATTRIBUTE;
```

71

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, 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/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 LIST_TYPE)
is
    BASE: NODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(NAME), (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 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.

5.1.3.10 Creating iterators over node attributes

```
procedure NODE_ATTRIBUTE_ITERATE(ITERATOR:in 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.

Parameters:
  ITERATOR   is a result parameter containing the iterator to be
                  constructed.
  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 and 5.1.3.9.

Exceptions:
  STATUS_ERROR   is raised if NODE is not an open node handle.
  USE_ERROR   is raised if the PATTERN is syntactically illegal.
  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:in out ATTRIBUTE_ITERATOR;
                                 NAME:    in  NAME_STRING;
                                 PATTERN:.in  ATTRIBUTE_PATTERN:="*")
  is
     NODE: NODE_TYPE;
  begin
     OPEN(NODE, NAME, (READ_ATTRIBUTES));
     NODE_ATTRIBUTE_ITERATE(ITERATOR, NODE, PATTERN);
     CLOSE(NODE);
  exception
     when others =>
```

73

3-101

```
        CLOSE(NODE);
        raise;
  end NODE_ATTRIBUTE_ITERATE;
```

Notes:

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

5.1.3.11 Determining iteration status

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

Purpose:
The function MORE returns true or false depending on whether all attributes contained in the attribute iterator have been retrieved with the procedure GET_NEXT.

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.

5.1.3.12 Getting the next node 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.

74

5.1.3.13 Obtaining an iterator over relationship attributes

```
procedure PATH_ATTRIBUTE_ITERATE(ITERATOR: in out ATTRIBUTE_ITERATOR;
                                 BASE:     in NODE_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, 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 a result parameter containing the iterator to be
                constructed.
   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 and 5.1.3.9.

Exceptions:
   NAME_ERROR   is raised if the relationship identified by the
                BASE/KEY/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: in out ATTRIBUTE_ITERATOR;
                                 NAME:     in NAME_STRING;
                                 PATTERN:  in ATTRIBUTE_PATTERN:="*");
   is
      BASE: NODE_TYPE;
   begin
      OPEN(BASE, BASE_PATH(NAME), (WRITE_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;
```

75

5.1.4 Package CAIS_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.

5.1.4.1 Types, subtypes, constants, and exceptions

    subtype PRIVILEGE_SPECIFICATION is STRING;

A privilege specification is a character string in the syntax described in Table V.

5.1.4.2 Setting access control

    procedure SET_ACCESS_CONTROL(NODE: in NODE_TYPE;

                                 ROLE_NODE: in NODE_TYPE;
                                 GRANT: in PRIVILEGE_SPECIFICATION);

Purpose:
  This procedure sets access control information for a given node. If one does not exist, a secondary ACCESS relationship is created from the node specified by NODE to the node specified by ROLE_NODE. If necessary, the attribute GRANT is created on this relationship. The value of the GRANT attribute is set to the proper form (see Table V) of the privilege specification GRANT. The effect is to grant the access specified by GRANT to processes that have adopted the role ROLE_NODE.

Parameters:
  NODE   is a node handle to the node whose access control information is to be set.
  ROLE_NODE   is a node handle to the role.
  GRANT   is a privilege specification describing what privileges are to be granted.

Exceptions:
  USE_ERROR   is raised if GRANT is not in valid syntax.
  STATUS_ERROR is raised if NODE or ROLE_NODE is not open.
  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;

76

```
                                    GRANT:      in PRIVILEGE_SPECIFICATION);
is
     NODE, ROLE_NODE: NODE_TYPE;
begin
     OPEN(NODE, NAME(EXISTENCE));
     OPEN(ROLE_NODE, ROLE_NAME, (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 Indicating access mode

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

Purpose:
  This function indicates whether or not a specific right to a node is
  granted for the caller. If the caller has adopted a role that is
  the target of an access relationship emanating from the object node
  and if that relationship's GRANT attribute allows the access right
  specified by PRIVILEGE_NAME, the function returns TRUE.

Parameters:
     OBJECT_NODE   is a node handle to the object node.
     PRIVILEGE_NAME   is a privilege name

Exceptions:
     USE_ERROR   is raised if PRIVILEGE_NAME is not in valid syntax.
     STATUS_ERROR   is raised if OBJECT_NODE is not open.
     INTENT_VIOLATION is raised if NODE was not opened with an intent
                       establishing the right to read 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

```
   function IS_GRANTED (OBJECT_NAME: in NAME_STRING;
                        PRIVILEGE:   in NAME_STRING) return BOOLEAN
   is
       OBJECT_NODE: NODE_TYPE;
       RESULT:      BOOLEAN;
   begin
       OPEN(OBJECT_NODE, OBJECT_NAME, (READ_RELATIONSHIPS));
       RESULT := IS_GRANTED(OBJECT_NODE, PRIVILEGE);
       CLOSE(OBJECT_NODE);
```

77

3-105

```
    exception
       when others =>
           CLOSE(OBJECT_NODE);
           raise;
    end IS_GRANTED;
```

5.1.4.4 Adopting a group

```
    procedure ADOPT (ROLE_NODE:  in NODE_TYPE);
```

Purpose:
   This procedure causes the calling process to adopt the group
   specified by the ROLE_NODE.   A relationship of the predefined
   relation ADOPTED_ROLE is created from the calling process node to
   the given group. In order for the current process to adopt the
   group, some other role the current process has already adopted must
   be a potential member of the group to be adopted.

Parameters:
   ROLE_NODE   is an open node handle to a group node.

Exceptions:
   USE_ERROR   is raised if their is no adopted role of the current
                    process that is a potential member of the group
                    ROLE_NODE.
   STATUS_ERROR   is raised if ROLE_NODE is not an open node handle.
   INTENT_VIOLATION is raised if ROLE_NODE was not opened with an
                    intent establishing the right to read 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.

5.1.5 Package CAIS 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 CAIS_STRUCTURAL_NODES defines the primitive operations for
creating structural nodes.

78

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     FORM_STRING := "";
                      LEVEL:     in     FORM_STRING :=
```

Purpose:
  This procedure creates a structural node and installs the primary
  relationship to it. The relation name and 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.

  The ATTRIBUTES parameter defines and provides initial values for
  attributes of the node (for the use of values of type LIST_TYPE, see
  Section 5.4 CAIS_LIST_UTILITIES). 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 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 initial values for attributes of the newly created
              node.
  ACCESS_CONTROL is the initial access control information associated
              with the created node.
  LEVEL   is the classification label for the created node.

Exceptions:
  NAME_ERROR is raised if a node already exists for the node
              identification given, if the node identification is
              illegal, or if any group node specified in the given
              ACCESS_CONTROL parameter is unobtainable.
  USE_ERROR is raised if the ACCESS_CONTROL or LEVEL parameters do
              does not adhere to the required syntax or if the
              ATTRIBUTES parameter contains references to predefined
              attributes not modifiable 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.

79

3-107

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    FORM_STRING := "";
                      LEVEL:   in     FORM_STRING := "");
is
    BASE: NODE_TYPE;

begin
    OPEN(BASE, BASE_PATH(NAME), (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);
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    FORM_STRING := "";
                      LEVEL:   in     FORM_STRING := "");
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    FORM_STRING := "";
                      LEVEL:   in     FORM_STRING := "");
is
    NODE: NODE_TYPE;
begin
    CREATE_NODE(NODE, NAME, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
    CLOSE(NODE);
end CREATE_NODE;
```

80

Notes:
> Use of the sequence of a CREATE_NODE call followed by a call on OPEN for the created node, using the node identification of the created node, cannot guarantee that the node just created is opened, since 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 events in a process's life are:

a. Initiation
b. Running, which may include suspension or resumption
c. Termination or abortion

This section of the CAIS defines facilities to control and coordinate the initiation, suspension, resumption, and termination or abortion of processes.

A process is said to be "terminated" when the subprogram which is 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. Aborting a process can be considered pre-emptive termination and may be initiated by the process itself or by another process with sufficient access rights.

The following rules apply to process nodes for which the process has terminated or aborted:

a. The process node remains in existence until explicitly deleted.

b. Any processes in the process tree emanating from the terminated/aborted process have terminated or aborted.

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

a. Invoke - the procedure INVOKE_PROCESS does not return control to the calling task until 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 execute in parallel with the initiated process and its tasks. Execution of the initiating task is synchronized with the initiated process, which has no implicit effect on other tasks of the initiating process. This kind of process initiation is analogous to calling the specified program as a procedure.

82

b. 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 these, their tasks execute in parallel. The processes run asynchronously, except as they are synchronized by the use of other CAIS facilities. This kind of process initiation is analogous to activation of the specified program as a task.

Every process node has three predefined attributes: RESULTS_LIST, which can be used to store user-defined strings giving intermediate results of the process; PARAMETER_LIST, which contains the parameters with which the process was called; and CURRENT_STATUS, which gives the current status of the process. In addition, every process node has several predefined attributes which provide information for standardized debugging and performance measurement for processes within the CAIS implementation. One of these predefined attributes has an implementation-independent value. This attribute is HANDLES_OPEN, which gives the number of open node handles the process has. 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 times of activation and 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 I/O units used. SIZE gives a measure of the size of the process.

For purposes of input and output, every process node has several predefined relationships. These predefined relationships are named STANDARD_INPUT, STANDARD_OUTPUT, STANDARD_ERROR, CURRENT_INPUT, CURRENT_OUTPUT, and CURRENT_ERROR. STANDARD_INPUT, STANDARD_OUTPUT and STANDARD_ERROR are 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 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 semantics of [LRM] 14.3.2. Interfaces are provided in the CAIS Input/Output packages *(Section 5.3) to read these predefined relationships and to change the value of CURRENT_INPUT, CURRENT_OUTPUT, and CURRENT_ERROR.

5.2.1 Package CAIS_PROCESS_DEFINITIONS

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

    type PROCESS_STATUS is
                (READY, SUSPENDED, ABORTED, TERMINATED);
The PROCESS_STATUS is the state of a process. Table VII indicates the states and the events which will cause transition from one state to another.

84

TABLE VII. Process state transition table

state: 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 state specified.
  —: marks events that have no effect on the state.

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

85

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. Any open node handles emanating from a process are closed after the process is terminated or aborted.

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 PROCESS_STATUS is immediately available.

PROCESS_STATUS may also be examined by the CAIS procedures STATE_OF_PROCESS and GET_RESULTS.

```
ROOT_PROCESS   :  constant NAME_STRING := "'CURRENT_JOB";
CURRENT_NODE   :  constant NAME_STRING := "'CURRENT_NODE";
CURRENT_INPUT  :  constant NAME_STRING := "'CURRENT_INPUT";
CURRENT_OUTPUT:  constant NAME_STRING := "'CURRENT_OUTPUT";
CURRENT_ERROR  :  constant NAME_STRING := "'CURRENT_ERROR";
CURRENT_PROCESS:STRING renames CAIS_NODE_DEFINITIONS.CURRENT_PROCESS;
```

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

Table VIII presents an overview of interfaces to change the status, review results, or determine predefined attribute values in a process.

TABLE VIII. Process Interfaces

Changing the status of a process	These three procedures change the process status of a process. Because of timing circumstances in a distributed environment, a change to the process status may not take effect immediately. In particular, a process may terminate before ABORT_PROCESS or SUSPEND_PROCESS is enacted.
	procedure ABORT_PROCESS procedure SUSPEND_PROCESS procedure RESUME_PROCESS
Determining the value of a predefined attribute	These procedures read the predefined attributes on process nodes.
	function HANDLES_OPEN function IO_UNITS function START_TIME function FINISH_TIME

86

function MACHINE_TIME

Results from tasks   These two procedures provide the techniques
in a process   for a process to examine and modify a
                            results list. The results list is returned
                            to a task which named that process in a call
                            to INVOKE_PROCESS or AWAIT_PROCESS_COMPLETION.

                            The results list can be examined before or
                            after process termination or abortion by any
                            process with the appropriate access rights.

                                  procedure WRITE_RESULTS
                                  procedure APPEND_RESULTS

5.2.2   Package CAIS_PROCESS_CONTROL

5.2.2.1   Types, subtypes, constants, and exceptions

```
   subtype LIST_TYPE       is CAIS_LIST_UTILITIES.LIST_TYPE;
   subtype RESULTS_LIST    is CAIS_LIST_UTILITIES.LIST_TYPE;
   subtype RESULTS_STRING  is STRING;
   subtype PARAMETER_LIST  is CAIS_LIST_UTILITIES.LIST_TYPE;
   subtype NAME_STRING     is CAIS_NODE_DEFINITIONS.NAME_STRING;
   subtype RELATION_NAME   is CAIS_NODE_DEFINITIONS.RELATION_NAME;
   subtype RELATIONSHIP_KEY   is CAIS_NODE_DEFINITIONS.RELATIONSHIP_KEY;
   subtype NODE_TYPE       is CAIS_NODE_DEFINITIONS.NODE_TYPE;
   subtype PROCESS_STATUS  is CAIS_PROCESS_DEFINITIONS.PROCESS_STATUS;

   EMPTY_LIST:   constant LIST_TYPE renames
                             CAIS_LIST_UTILITIES.EMPTY_LIST;

   LATEST_KEY:   constant RELATIONSHIP_KEY renames
                             CAIS_NODE_DEFINITIONS.LATEST_KEY;

   DEFAULT_RELATION: constant RELATION_NAME renames
                             CAIS_NODE_DEFINITIONS.DEFAULT_RELATION;

   NAME_ERROR:   exception renames CAIS_NODE_DEFINITIONS.NAME_ERROR;

   USE_ERROR:    exception renames CAIS_NODE_DEFINITIONS.USE_ERROR;

   INTENT_VIOLATION: exception renames
                             CAIS_NODE_DEFINITIONS.INTENT_VIOLATION;
   ACCESS_VIOLATION: exception renames
                             CAIS_NODE_DEFINITIONS.ACCESS_VIOLATION;
```

87

SECURITY_VIOLATION: exception renames
                    CAIS_NODE_DEFINITIONS.SECURITY_VIOLATION;

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, is unobtainable, discretionary
                    access controls for knowledge of existence of a node
                    are violated, or mandatory access controls for read
                    operations are violated.

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

INTENT_VIOLATION is raised whenever an operation is attempted on an
                    open node handle which is in violation of the access
                    intent specified when the node handle was opened.

ACCESS_VIOLATION is raised whenever an operation is attempted which
                    violates access right constraints other than
                    knowledge of existence of the node.

SECURITY_VIOLATION is raised whenever an operation is attempted which
                    violates mandatory access controls.

5.2.2.2 Spawning a process

procedure SPAWN_PROCESS
            (NODE: in out NODE_TYPE;
             FILE_NODE: in   NODE_TYPE;
             INPUT_PARAMETERS: in   PARAMETER_LIST;
             KEY: in   RELATIONSHIP_KEY := LATEST_KEY;
             RELATION: in   RELATION_NAME := DEFAULT_RELATION;
             ACCESS_CONTROL: in   FORM_STRING := "";
             LEVEL: in   FORM_STRING := "";
             FORM: 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.
  SPAWN_PROCESS accepts a list of parameters and makes it available to
  the new process via the procedure GET_PARAMETERS. 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. The created process node has secondary relationships to the
  input, output, and error files. An open node handle on the new node
  is returned, with an intent establishing all access rights.

88

The ACCESS_CONTROL parameter specifies the initial access control information to be established for the created node. The current user must have all access rights to the created node.

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

Parameters:

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

FILE_NODE   is a 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(Section 5.4). INPUT_PARAMETERS is stored in a predefined attribute PARAMETER_LIST 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 name of the primary relation from the current process 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. The current user must have all access rights to the created node.

LEVEL   is a string defining the classification label for the created node.

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

INPUT_FILE,
OUTPUT_FILE, and
ERROR_FILE   are path names to file nodes for the new process node.

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

Exceptions:

89

NAME_ERROR   is raised if the node indicated by FILE_NODE is
                          inaccessible or unobtainable or if a node already
                          exists for the relationship specified by KEY and
                          RELATION.

USE_ERROR   is raised if it can determined that the node
                          indicated by FILE_NODE does not contain an
                          executable image.

LOCK_ERROR   is raised if the node designated by FILE_NODE is
                          locked against execution.

INTENT_VIOLATION is raised if the node designated by FILE_NODE was
                          not opened with an intent establishing the right
                          to execute 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/Output *(Section 5.3) must be used. When the parent process
  terminates or aborts, the child process will be aborted.

5.2.2.3 Awaiting termination or abortion of another process

     procedure AWAIT_PROCESS_COMPLETION
                    (NODE: in   NODE_TYPE;
                     RESULTS_RETURNED: in out RESULTS_LIST;
                     STATUS: out PROCESS_STATUS;
                     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.

     AWAIT_PROCESS_COMPLETION returns the results list and process
  completion status to the calling task, even if the process has
  already terminated or aborted when the call is made, so long as the
  process node exists.

   Parameters:

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

  RESULTS_RETURNED is a list of results, which are represented by
                strings, from the process. The individual results may
                be extracted from the list using the tools provided
                in CAIS_LIST_UTILITIES.

90

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.

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 identified node is inaccessible or
                unobtainable.

LOCK_ERROR   is raised if the identified node is locked against
                reading attributes.

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

SECURITY_VIOLATION   is raised if the attempt to access the
                identified node represents a violation of mandatory
                access controls. SECURITY_VIOLATION is raised only
                if the conditions for the other exceptions are not
                satisfied.

5.2.2.4 Invoking a new process

```
procedure INVOKE_PROCESS
            (FILE_NODE:        in    NODE_TYPE;
             INPUT_PARAMETERS: in    PARAMETER_LIST;
             RESULTS_RETURNED: in out RESULTS_LIST;
             STATUS:                 out PROCESS_STATUS;
             KEY:              in    RELATIONSHIP_KEY := LATEST_KEY;
             RELATION:         in    RELATION_NAME:= DEFAULT_RELATION;
             ACCESS_CONTROL:   in    FORM_STRING := "";
             LEVEL:            in    FORM_STRING := "";
             FORM:             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 of a call to SPAWN_PROCESS
  followed by a call to AWAIT_PROCESS_COMPLETION, as an indivisible

91

3-119

operation.

INVOKE_PROCESS creates a new process node whose contents represent the execution of the program contained in the specified file node. INVOKE_PROCESS accepts a list of parameters and makes it available to the new process via the procedure GET_PARAMETERS. If termination or abortion of the identified process can be reported within the specified time limit, control is returned with the process status. Otherwise control is returned when the specified time limit is exceeded with a status of READY or SUSPENDED. The process node containing the calling task must have execution rights for the file node. The created process node has secondary relationships to the input, output, and error files.

The ACCESS_CONTROL parameter specifies the initial access control information to be established for the created node, as described in *(Section 5.1.4). The current user must have all access rights to the created node. The LEVEL parameter specifies the security level at which the node is to be created, as described in *(Section 5.1.4).

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 list handling tools of CAIS_LIST_UTILITIES. INPUT-PARAMETER is stored in the predefined attribute PARAMETER_LIST of the new node.

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.

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 LATEST_KEY function.

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

92

ACCESS_CONTROL is a string defining the initial access control information associated with the created node. The current user must have all access rights to the created node.

LEVEL is a string defining the classification label for the created node.

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

INPUT_FILE,
OUTPUT_FILE, and
ERROR_FILE are path names to file nodes for the new process node.

ENVIRONMENT_NODE is the node the new process will have as its current node. The default is CURRENT_NODE of the invoking process.

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 the node indicated by FILE_NODE is inaccessible or unobtainable or if a node already exists for the relationship specified by KEY and RELATION.

USE_ERROR is raised if it can be determined that the node indicated by FILE_NODE does not contain an executable image.

LOCK_ERROR is raised if the node designated by FILE_NODE is locked against execution.

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.

93

3-121

5.2.2.5 Creating a new job

procedure CREATE_JOB
        (FILE_NODE: in   NODE_TYPE;
         INPUT_PARAMETERS: in   PARAMETER_LIST;
         KEY: in   RELATIONSHIP_KEY := LATEST_KEY;
         ACCESS_CONTROL: in   FORM_STRING := "";
         LEVEL: in   FORM_STRING := "";
         FORM: 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. CREATE_JOB establishes the USER, ROLE, and DEVICE
    relationships as described in *(Section 4), General Requirements.
    CREATE_JOB accepts a list of parameters and makes it available to
    the new process via the procedure GET_PARAMETERS. 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 append_relationship rights to CURRENT_USER. The new job
    has secondary relationships to the input, output, and error files.
    A new primary JOB relationship is established from CURRENT_USER to
    the new job.

    The ACCESS_CONTROL parameter specifies the initial access control
    information to be established for the created node. The current
    user must have all access rights to the created node.

    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 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.

    KEY   is the relationship key of the primary JOB
                       relationship from the current user node to the new
                       process node. The default is supplied by the
                       LATEST_KEY function.

    ACCESS_CONTROL   is a string defining the initial access control
                       information associated with the created node. The
                       current user must have all access rights to the

94

created node.

LEVEL   is a string defining the classification label for the created node.

FORM   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, and
ERROR_FILE   are path names to file nodes for the new process node.

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

Exceptions:
  NAME_ERROR   is raised if the node indicated by FILE_NODE is inaccessible or unobtainable or if a node already exists for the relationship specified by KEY and RELATION.

  USE_ERROR   is raised if it can determined that the node indicated by FILE_NODE does not contain an executable image.

  LOCK_ERROR   is raised if the node designated by FILE_NODE is locked against execution.

  INTENT_VIOLATON 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 NAME_ERROR are not satisfied.

  SECURITY_VIOLATION is raised if the attempt to obtain access to CURRENT_USER or the file node 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.

Notes:
  CREATE_JOB does not return results or process status to the calling

95

program unit. If coordination between any program unit and the new process is desired, AWAIT_PROCESS_COMPLETION or the techniques provided in CAIS Input/Output *(Section 5.3) must be used.

The relation name for the primary relationship to the new node is JOB.

5.2.2.6 Appending results

procedure APPEND_RESULTS(RESULTS: in RESULTS_STRING);

Purpose:
This procedure adds its specified results parameter to the list of results from other calls to APPEND_RESULTS in the current process. The procedure appends results to the list in the order in which they are received. Upon termination or abortion of the current process, the results list is returned to any task which named this process in a call to AWAIT_PROCESS_COMPLETION or INVOKE_PROCESS.

Parameters:

RESULTS   is a string to be stored in the RESULTS_LIST attribute of the current process node and ultimately returned to the awaiting or invoking task.

Exceptions:
None.

Notes:
Until the process node is deleted or the results are overwritten, the results are stored in a results list which is th evalue of the CAIS predefined attribute RESULTS_LIST.

5.2.2.7 Overwriting results

procedure WRITE_RESULTS (RESULTS : in RESULTS_STRING);

Purpose:
This procedure replaces the current list of results with the specified results.

Parameters:

RESULTS   is a string to be stored in the RESULTS_LIST and ultimately returned to the awaiting or invoking task.

Exceptions:
None.

96

5.2.2.8 Getting results from a process

```
procedure GET_RESULTS (NODE:          in     NODE_TYPE;
                       RESULTS:        in out RESULTS_LIST;
                       STATUS:         out PROCESS_STATUS);
```

Purpose:
  This procedure reads the attributes RESULTS_LIST and CURRENT_STATUS
  associated with a process 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 in the process
  contained in NODE.

Parameters:
  NODE   is an open node handle on a process node.

  RESULTS   is a list resulting from calls made by program
                  units in the process node to the procedures
                  WRITE_RESULTS and APPEND_RESULTS. The individual
                  results may be extracted from the list using the
                  tools of CAIS_LIST_UTILITIES *(Section 5.4).

  STATUS   is the process status of the process.

Exceptions:
  NAME_ERROR   is raised if the node identified by NODE is
                  inaccessible or unobtainable or is not a process
                  node.

  LOCK_ERROR   is raised if the node identified by NODE is locked
                  against reading attributes.

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

  SECURITY_VIOLATION is raised if the attempt to obtain access to the
                  node identified by NODE 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.

Additional Interfaces:

```
procedure GET_RESULTS (NODE:          in     NODE_TYPE;
                       RESULTS:        in out RESULTS_LIST)
is
   STATUS: PROCESS_STATUS;
begin
   GET_RESULTS(NODE, RESULTS, STATUS);
end GET_RESULTS;

procedure GET_RESULTS (NAME:          in     NAME_STRING;
```

97

```
                              RESULTS:      in out RESULTS_LIST;
                              STATUS:          out PROCESS_STATUS);
   is
      NODE: NODE_TYPE;
   begin
      OPEN(NODE, NAME, (READ_ATTRIBUTES));
      GET_RESULTS(NODE, RESULTS, STATUS);
      CLOSE(NODE);
   exception
      when others =>
         CLOSE(NODE);
         raise;
   end GET_RESULTS;

   procedure GET_RESULTS (NAME:        in      NAME_STRING;
                          RESULTS:     in out RESULTS_LIST)
   is
      NODE: NODE_TYPE;
      STATUS: PROCESS_STATUS;
   begin
      OPEN(NODE, NAME, (READ_ATTRIBUTES));
      GET_RESULTS(NODE, RESULTS, STATUS);
      CLOSE(NODE);
   exception
      when others =>
         CLOSE(NODE);
         raise;
   end GET_RESULTS;
```

5.2.2.9 Getting the parameter list

```
   procedure GET_PARAMETERS(PARAMETERS: in out PARAMETER_LIST);
```

Purpose:
This procedure retrieves the list of parameters passed to the
current process node by the task which created it. The list is
stored in the predefined attribute PARAMETER_LIST of the current
process node.

Parameters:
   PARAMETERS   is a list containing parameter information. The
                     list is constructed and can be manipulated using
                     the tools provided in CAIS_LIST_UTILITIES.

Exceptions:
   None.

98

5.2.2.10   Aborting a process

```
procedure ABORT_PROCESS(NODE:    in NODE_TYPE;
                        RESULTS: in RESULTS_STRING);
```

Purpose:
This procedure aborts the process contained in 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 aborted process is examined, it will be ABORTed
provided that the process existed and was not terminated at the time
of the call to ABORT_PROCESS. The node associated with the aborted
process remains until explicitly deleted.

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_LIST
                attribute of the node of the aborted process.

Exceptions:
NAME_ERROR   is raised if the process node is inaccessible or
                unobtainable.

INTENT_VIOLATION is raised if the identified process node was not
                opened with an intent establishing exclusive write
                access rights.

SECURITY_VIOLATION is raised if the attempt to obtain access to
                the node identified by NODE represents a
                violation of mandatory access controls in the CAIS.
                SECURITY_VIOLATION is raised only if the conditions
                for raising the othr 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, (EXCLUSIVE_WRITE));
   ABORT_PROCESS(NODE, RESULTS);
   CLOSE(NODE);
exception
   when others =>
        CLOSE(NODE);
        raise;
end ABORT_PROCESS;

procedure ABORT_PROCESS(NODE: in NODE_TYPE)
is
```

99

```
begin
  ABORT_PROCESS(NODE, "");
end ABORT_PROCESS;

procedure ABORT_PROCESS(NAME: in NAME_STRING)
is
  NODE: NODE_TYPE;
begin
  OPEN(NODE, NAME, (EXCLUSIVE_WRITE));
  ABORT_PROCESS(NODE, "");
  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. Any open node handles emanating from NODE are closed
after the process is aborted.

5.2.2.11 Suspending a process

procedure SUSPEND_PROCESS(NODE: in NODE_TYPE);

Purpose:
This procedure suspends the process contained in NODE. After
SUSPEND_PROCESS is called, the PROCESS_STATUS of the identified
process is SUSPENDED, provided that the process was in the READY
state at the time that the suspension took effect. SUSPEND_PROCESS
has no effect if the process is not in the READY state.

Parameters:
NODE   is an open node handle for the node of the process
                       to be suspended.
Exceptions:
NAME_ERROR   is raised if the node is inaccessible or
                       unobtainable or is not a process node.

  INTENT_VIOLATION is raised if the identified process node was not
                       opened with an intent establishing write access
                       rights.

  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 Interfaces:

100
```

```
procedure SUSPEND_PROCESS(NAME: in NAME_STRING)
is
   NODE: NODE_TYPE;
begin
   OPEN(NODE, NAME, (WRITE));
   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.

5.2.2.12 Resuming a process

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

Purpose:
   This procedure causes the process contained in NODE to resume
   execution. RESUME_PROCESS has no effect 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 state at the time that the resumption took effect.

Parameters:
   NODE   is an open node handle for the node of the process
                       to be resumed.
Exceptions:
   NAME_ERROR   is raised if the node is inaccessible or
                       unobtainable.

   INTENT_VIOLATION is raised if the identified process node was not
                       opened with an intent establishing write access
                       rights.

   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 Interfaces:

```
procedure RESUME_PROCESS (NAME: in NAME_STRING)
is
   NODE: NODE_TYPE;
begin
```

101

3-129

```
    OPEN(NODE, NAME, (WRITE));
    RESUME_PROCESS(NODE);
    CLOSE(NODE);
exception
  when others =>
     CLOSE(NODE);
     raise;
end RESUME_PROCESS;
```

5.2.2.13 Determining the state of a process

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

Purpose:
  This function returns the current state of the process contained
  in NODE.

Parameters:
  NODE   is an open node handle for the process whose status
                  is to be queried.

Exceptions:
  NAME_ERROR   is raised if the node is inaccessible or
                  unobtainable.

  LOCK_ERROR   is raised if the node is locked against reading
                  attributes.

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

  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 Interfaces:

```
function STATE_OF_PROCESS(NAME: in NAME_STRING)
        return PROCESS_STATUS
is
   NODE: NODE_TYPE;
   RESULT: PROCESS_STATUS;
begin
   OPEN(NODE, NAME, (WRITE));
   RESULT := STATE_OF_PROCESS(NODE);
   CLOSE(NODE);
   return RESULT;
exception
```

102

```
      when others =>
         CLOSE(NODE);
         raise;
   end STATE_OF_PROCESS;
```

Notes:
  The process state of the process containing the calling task will
  always be READY.

5.2.2.14 Determining the number of open node handles

```
   function HANDLES_OPEN (NODE : in NODE_TYPE)
                           return NATURAL;
```

Purpose:
  This function returns the value of the predefined attribute
  HANDLES_OPEN.

Parameters:
  NODE   is an open node handle to the process node of
                  interest.

Exceptions:
  NAME_ERROR   is raised if the node is inaccessible or
                  unobtainable.

  LOCK_ERROR   is raised if the node is locked against reading
                  attributes.

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

  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 Interfaces:

```
   function HANDLES_OPEN (NAME : in NODE_TYPE)
          return NATURAL
   is
      NODE: NODE_TYPE;
      RESULT: NATURAL;
   begin
      OPEN(NODE, NAME, (WRITE));
      RESULT := HANDLES_OPEN(NODE);
      CLOSE(NODE);
      return RESULT;
```

```
      exception
         when others =>
            CLOSE(NODE);
            raise;
      end HANDLES_OPEN;
```

5.2.2.15 Determining the number of I/O units used

```
      function IO_UNITS (NODE : in NODE_TYPE)
                              return NATURAL;
```

Purpose:
  This function returns the value of the predefined attribute
  IO_UNITS.

Parameters:
  NODE   is an open node handle to the process node of
                     interest.

Exceptions:
  NAME_ERROR   is raised if the node is inaccessible or
                     unobtainable.

  LOCK_ERROR   is raised if the node is locked against reading
                     attributes.

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

  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 Interfaces:

```
      function IO_UNITS (NAME : in NAME_STRING)
              return NATURAL
      is
         NODE: NODE_TYPE;
         RESULT: NATURAL;
      begin
         OPEN(NODE, NAME, (WRITE));
         RESULT := IO_UNITS(NODE);
         CLOSE(NODE);
         return RESULT;
      exception
         when others =>
            CLOSE(NODE);
```

104

```
          raise;
      end IO_UNITS;
```

5.2.2.16 Determining the time of activation

```
    function START_TIME (NODE : in NODE_TYPE)
                        return CALENDAR.TIME;
```

Purpose:
This function returns the value of the predefined attribute
START_TIME.

Parameters:
NODE   is an open node handle to the process node of
                    interest.

Exceptions:
NAME_ERROR   is raised if the node is inaccessible or
                    unobtainable.

LOCK_ERROR   is raised if the node is locked against reading
                    attributes.

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

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 Interfaces:

```
    function START_TIME (NAME : in NAME_STRING)
          return CALENDAR.TIME
    is
       NODE: NODE_TYPE;
       RESULT: CALENDAR.TIME;
    begin
       OPEN(NODE, NAME, (WRITE));
       RESULT := START_TIME(NODE);
       CLOSE(NODE);
       return RESULT;
    exception
       when others =>
          CLOSE(NODE);
          raise;
    end START_TIME;
```

105

5.2.2.17 Determining the time of termination or abortion

```
function FINISH_TIME (NODE : in NODE_TYPE)
                    return CALENDAR.TIME;
```

Purpose:
   This function returns the value of the predefined attribute
   FINISH_TIME.

Parameters:
   NODE   is an open node handle to the process node of
                   interest.

Exceptions:
   NAME_ERROR   is raised if the node is inaccessible or
                   unobtainable.

   LOCK_ERROR   is raised if the node is locked against reading
                   attributes.

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

   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 Interfaces:

```
function FINISH_TIME (NAME : in NAME_STRING)
        return CALENDAR.TIME
is
   NODE: NODE_TYPE;
   RESULT: CALENDAR.TIME;
begin
   OPEN(NODE, NAME, (WRITE));
   RESULT := FINISH_TIME(NODE);
   CLOSE(NODE);
   return RESULT;
exception
   when others =>
      CLOSE(NODE);
      raise;
end FINISH_TIME;
```

106

5.2.2.18 Determining the time a process has been active

```
function MACHINE_TIME (NODE : in NODE_TYPE)
                         return DURATION;
```

Purpose:
  This function returns the value of the predefined attribute
  MACHINE_TIME.

Parameters:
  NODE   is an open node handle to the process node of
                interest.

Exceptions:
  NAME_ERROR   is raised if the node is inaccessible or
                unobtainable.

  LOCK_ERROR   is raised if the node is locked against reading
                attributes.

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

  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 Interfaces:

```
function MACHINE_TIME (NAME : in NAME_STRING)
        return DURATION                    .
is
   NODE: NODE_TYPE;
   RESULT: DURATION;
begin
   OPEN(NODE, NAME, (WRITE));
   RESULT := MACHINE_TIME(NODE);
   CLOSE(NODE);
   return RESULT;
exception
   when others =>
      CLOSE(NODE);
      raise;
end MACHINE_TIME;
```

107

5.2.2.19 Determining the standard input file

    function JOB_INPUT_FILE return NAME_STRING;

Purpose:
  This function returns a pathname to the standard input file node
  defined at the initiation of the root process node of the job, even
  if the current default input file for this process has been set to a
  different file.

Parameters:
  None.

Exceptions:
  LOCK_ERROR   is raised if the root process node is locked against
                 reading relationships.

Notes:
  In general, this file will refer to the interactive terminal or
  batch input file.

5.2.2.20 Determining the standard output file

    function JOB_OUTPUT_FILE return NAME_STRING;

Purpose:
  This function returns a pathname to the standard output file node
  defined at the initiation of the root process node of the job, even
  if the current default output file for this process has been set to
  a different file.

Parameters:
  None.

Exceptions:
  LOCK_ERROR   is raised if the root process node is locked against
                 reading relationships.

Notes:
  In general, this file will refer to the interactive terminal or
  batch error messages file.

108

5.2.2.21 Determining the standard error messages file

function JOB_ERROR_FILE return NAME_STRING;

Purpose:
This function returns a pathname to the standard error messages file
node defined at the initiation of the root process node of the job,
even if the current default messages file for this process has been
set to a different file.

Parameters:
None.

Exceptions:
LOCK_ERROR   is raised if the root process node is locked against
                  reading relationships.

Notes:
In general, this file will refer to the interactive terminal or
batch error messages file.

109

5.3 CAIS Input/output

Support for all kinds of files are provided by the packages CAIS_IO_CONTROL and CAIS_IO_EXCEPTIONS. Additionally, each of the different file kinds is further supported by a set of packages. Secondary storage files are further supported by the packages CAIS_SEQUENTIAL_IO, CAIS_DIRECT_IO, and CAIS_TEXT_IO. Queue files are further supported by the packages CAIS_SEQUENTIAL_IO and CAIS_TEXT_IO. Terminal files are further supported by the packages CAIS_TEXT_IO, CAIS_SCROLL_TERMINAL, CAIS_PAGE_TERMINAL, and CAIS_FORM_TERMINAL. Magnetic tape drive files are further supported by the packages CAIS_SEQUENTIAL_IO, CAIS_TEXT_IO, and CAIS_MAGNETIC_TAPE. The file kinds and their associated support packages are summarized in Table IX.

TABLE IX.   Input/Output Packages for File Kinds

	Secondary Storage	Queue	Terminal	Magnetic Tape
CAIS_IO_CONTROL	X	X	X	X
CAIS_IO_EXCEPTIONS	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

Implementations of the packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO specified in [LRM] that operate upon CAIS files are to be constructed such that the functionality of each subprogram in the respective CAIS packages CAIS_SEQUENTIAL_IO, CAIS_DIRECT_IO, and CAIS_TEXT_IO is the same when the default FORM parameter is used in the corresponding CAIS CREATE procedures.

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. Queue files may be created by use of the CREATE procedures in the packages CAIS_SEQUENTIAL_IO and CAIS_TEXT_IO. Interfaces must be provided outside the CAIS for the creation of terminal files and magnetic tape drive files.

A file node has a number of predefined attributes associated with it. The attributes ACCESS_METHOD, FILE_KIND, QUEUE_TYPE, and TERMINAL_TYPE provide information about the contents of a file node and how it may be accessed.

110

The predefined values for the attribute ACCESS_METHOD are SEQUENTIAL, DIRECT, and TEXT or any list combination of these. The value of the attribute ACCESS_METHOD determines the packages that may operate upon the file. 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 and possibly other packages (depending on other attribute values) may be used. It is possible that a single file node may have more than one access method.

The attribute FILE_KIND denotes the type 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. A file node with a value of SECONDARY_STORAGE or QUEUE may be operated upon by the packages CAIS_SEQUENTIAL_IO, CAIS_DIRECT_IO, or CAIS_TEXT_IO. A value of QUEUE permits operations by the package CAIS_SEQUENTIAL_IO or CAIS_TEXT_IO. A value of TERMINAL permits operations by the package CAIS_TEXT_IO and the three terminal packages. A value of MAGNETIC_TAPE permits operations by the CAIS_MAGNETIC_TAPE package and the CAIS_TEXT_IO package.

The predefined values for the attribute QUEUE_TYPE are SOLO, MIMIC, and COPY.

The values SCROLL, PAGE, and FORM are predefined for the attribute TERMINAL_TYPE. The package CAIS_TEXT_IO may be used on any CAIS terminal. A value of SCROLL indicates that the CAIS_SCROLL_TERMINAL package may be used on the file node. A value of PAGE indicates that the CAIS_PAGE_TERMINAL package may be used on the file node. A value of FORM indicates that the CAIS_FORM_TERMINAL package may be used on the file node.

A file node with a QUEUE_TYPE of copy or mimic will have the relationship of the predefined relation ASSOCIATE to the file node with which it is associated.

A file node for a terminal will have a value of TEXT for the attribute ACCESS_METHOD and a value of TERMINAL for the attribute FILE_KIND In addition, the attribute TERMINAL_TYPE will have one (or more) of the values SCROLL, PAGE, or FORM.

A file node for a magnetic tape drive has a value of MAGNETIC_TAPE for the attribute FILE_KIND and a value of TEXT for the attribute ACCESS_METHOD. The packages CAIS_TEXT_IO and CAIS_TAPE_IO may be used for operating on a magnetic tape drive file.

5.3.1 Package CAIS_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 data types are the same.

The package specification and semantics of the CAIS_DIRECT_IO is comparable to that of the [LRM] package DIRECT_IO. The following sections demonstrate the specifications and semantics that differ.

5.3.1.1 Types, subtypes, constants, and exceptions

    subtype FILE_TYPE is CAIS_IO_CONTROL.FILE_TYPE;

    subtype FILE_MODE is CAIS_IO_CONTROL.FILE_MODE;

    IN_FILE   : constant FILE_MODE := IN_FILE;
    INOUT_FILE : constant FILE_MODE := INOUT_FILE;
    OUT_FILE   : constant FILE_MODE := OUT_FILE;

FILE_TYPE is used as a handle for all direct input and output operations. FILE_MODE indicates the intent upon accessing the direct input or output file.

5.3.1.2 Creating a direct I/O file

    procedure CREATE(FILE   : in out FILE_TYPE;
                     BASE   : in   NODE_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   FORM_STRING := "";
                     LEVEL: in   FORM_STRING := "");

112

Purpose:

This procedure creates a file and its file node; each element of the file is directly addressable by an index. The attribute ACCESS_METHOD is assigned the value "(DIRECT, SEQUENTIAL)" as part of the creation.

The contents of FORM have the syntax of a LIST_TYPE (Section 5.4). The FORM parameter is used to provide file characteristics concerning the creation of the file. The predefined file characteristic SIZE may be used to specify an approximation to the number of STORAGE_UNITs that should be writable to the file. The SIZE characteristic is specified as "(SIZE => n)", where "n" is any NATURAL number.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node (for the use of values of type LIST_TYPE, see Section 5.4.3.2 CAIS_LIST_UTILITIES). 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.

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 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 in 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:

113

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

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

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
             not modifiable by the user.

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 security rules. 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   FORM_STRING := "";
                 LEVEL:   in    FORM_STRING := "")
is
    BASE : NODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(NAME), (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);
end CREATE;
```

114

5.3.1.3 Opening a direct I/O file

```
procedure OPEN(FILE    : in out FILE_TYPE;
               NODE    : in     NODE_TYPE;
               MODE    : in     FILE_MODE := INOUT_FILE);
```

Purpose:
 This procedure opens a handle on a file; 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 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 or the element
                    type of the file does not correspond with the element
                    type of this instantiation of the CAIS_DIRECT_IO
                    package.

 STATUS_ERROR   is raised if the FILE is already open prior to the
                    call on OPEN.

 INTENT_VIOLATION is raised if NODE has not been opened with an intent
                    establishing the right associated with the MODE
                    specified, as explained in Table VIII.

Additional Interface:

```
    procedure OPEN(FILE : in out FILE_TYPE;
                   NAME : in     NAME_STRING;
                   MODE : in     FILE_MODE := INOUT_FILE)
    is
       NODE : NODE_TYPE;
    begin
       case MODE is
          IN_FILE => OPEN(NODE, NAME,(READ_CONTENTS));
          OUT_FILE => OPEN(NODE, NAME,(WRITE_CONTENTS));
          INOUT_FILE =>OPEN(NODE,NAME,(READ_CONTENTS,WRITE_CONTENTS));
          APPEND_FILE => OPEN(NODE, NAME,(APPEND_CONTENTS));
       end case;
       OPEN(FILE, NODE, MODE)
       CLOSE(NODE);
    exception
       when others =>
           CLOSE(FILE);
           CLOSE(NODE);
    end OPEN;
```

115

Notes:
  The effects of closing an open file node handle on the open file
  handle to the contents of this node are implicitly defined. In
  particular, no assumption can be made about the access
  synchronization provided by the node model.

5.3.2 Package CAIS_SEQUENTIAL_IO

This package provides facilities for sequentially accessing data
elements in CAIS files. These facilities are comparable to those
described in the SEQUENTIAL_IO package of [LRM].

The package specification and semantics of the CAIS_SEQUENTIAL_IO is
comparable to that of the [LRM] package SEQUENTIAL_IO. The following
sections demonstrate the specifications and semantics that differ.

5.3.2.1 Types, subtypes, constants, and exceptions

  subtype FILE_TYPE is CAIS_IO_CONTROL.FILE_TYPE;

  subtype FILE_MODE is CAIS_IO_CONTROL.FILE_MODE;

  IN_FILE   : constant FILE_MODE := IN_FILE;
  INOUT_FILE : constant FILE_MODE := INOUT_FILE;
  OUT_FILE   : constant FILE_MODE := OUT_FILE;
  APPEND_FILE : constant FILE_MODE := APPEND_FILE;

FILE_TYPE is used as a handle for all sequential input and output
operations. FILE_MODE indicates the intent upon accessing the
sequential input or output 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.

5.3.2.2 Creating a sequential I/O file

  procedure CREATE(FILE   : in out FILE_TYPE;
                   BASE   : in   NODE_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   FORM_STRING := "";
                   LEVEL: in   FORM_STRING := "");

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.

116

The contents of FORM have the syntax of a LIST_TYPE (Section 5.4). The FORM parameter is used to provide file characteristics concerning the creation of the file. The predefined file characteristic SIZE may be used to specify an approximation to the number of STORAGE_UNITs that should be writable to the file. The SIZE characteristic is specified as '(SIZE => n)', where 'n' is any NATURAL number.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node (for the use of values of type LIST_TYPE, see Section 5.4.3.2 CAIS_LIST_UTILITIES). 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.

The default value for the attribute FILE_KIND for the file node will be SECONDARY_STORAGE. The default value may be overridden by explicitly specifying a value of QUEUE in the FORM parameter (e.g. '(FILE_KIND => QUEUE)'). The default QUEUE_TYPE is a solo queue.

Parameters:

FILE	is a file handle, initially closed, to be opened.
BASE	is an open 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 in 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, if the node identification is syntactically illegal, or if, for the parent node of the node to be created or any group node specified in

117

the given access list, the node is unobtainable.

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

Additional Interface:

```
with CAIS_NODE_MANAGEMENT;
use CAIS_NODE_MANAGEMENT;
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     FORM_STRING := "";
                 LEVEL:    in     FORM_STRING := "")
is
    BASE : NODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(NAME), (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);
end CREATE;
```

5.3.2.3 Opening a sequential I/O file

```
procedure OPEN(FILE     : in out FILE_KIND;
               NODE     : in     NODE_TYPE;
               MODE     : in     FILE_MODE);
```

Purpose:
 This procedure opens a handle on a file; each element of the file
 is sequentially accessible.

Parameters:
 FILE   is a file handle, initially closed, to be opened.

 NODE   is an open handle to a base node for node

118

identification.

MODE   indicates the mode for accessing the file.

Exceptions:

USE_ERROR   is raised if the attribute ACCESS_METHOD of the file
                node does not have the value SEQUENTIAL or the
                element type of the file does not correspond with the
                element type of this instantiation of the
                CAIS_SEQUENTIAL_IO package.

STATUS_ERROR   is raised if the FILE is already open prior to the
                call on OPEN.

INTENT_VIOLATION is raised if NODE was not opened with an intent
                establishing the right associated with the MODE
                specified, as explained in Table VIII.

Additional Interface:

```
procedure OPEN(FILE : in out FILE_TYPE;
               NAME : in      NAME_STRING;
               MODE : in      FILE_MODE :=INOUT_FILE);
is
    NODE : NODE_TYPE;
begin
    case MODE is
        IN_FILE => OPEN(NODE, NAME,(READ_CONTENTS));
        OUT_FILE => OPEN(NODE, NAME,(WRITE_CONTENTS));
        INOUT_FILE =>OPEN(NODE,NAME,(READ_CONTENTS,WRITE_CONTENTS));
        APPEND_FILE => OPEN(NODE, NAME,(APPEND_CONTENTS));
    end case;
    OPEN(FILE, NODE, MODE)
    CLOSE(NODE);
exception
    when others =>
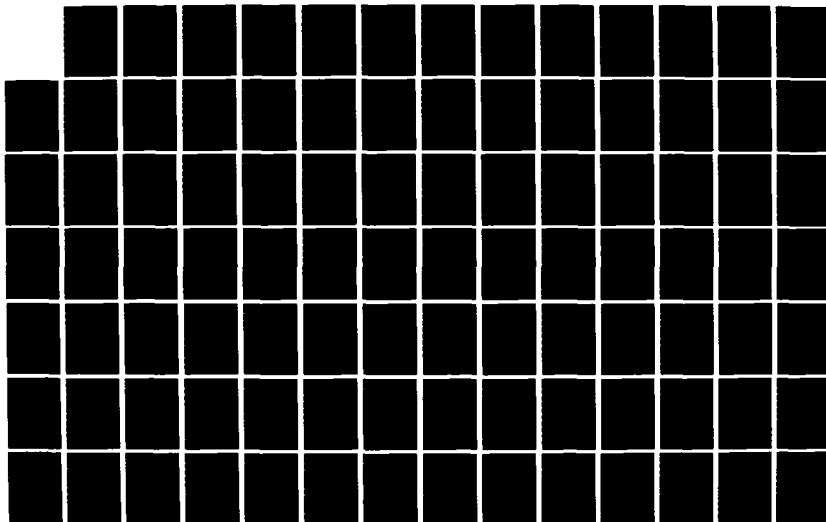        CLOSE(FILE);
        CLOSE(NODE);
end OPEN;
```

119

MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS-1963-A

5.3.3 Package CAIS_TEXT_IO

This package provides facilities for the input and output of textual data to CAIS files. These facilities are comparable to those specified in the package TEXT_IO in [LRM]. The following sections are those that differ from the specifications in [LRM].

5.3.3.1 Types, subtypes, constants, and exceptions

subtype FILE_TYPE is CAIS_IO_CONTROL.FILE_TYPE;

subtype FILE_MODE is CAIS_IO_CONTROL.FILE_MODE;

```
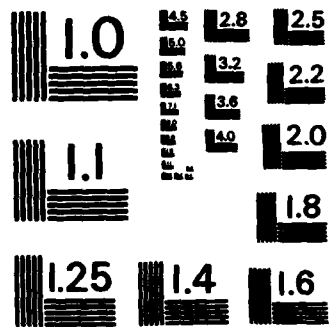IN_FILE      : constant FILE_MODE := IN_FILE;
INOUT_FILE   : constant FILE_MODE := INOUT_FILE;
OUT_FILE     : constant FILE_MODE := OUT_FILE;
APPEND_FILE  : constant FILE_MODE := APPEND_FILE;
```

FILE_TYPE is used as a handle for all text input and output operations. FILE_MODE indicates the intent upon accessing the text input or output 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.

5.3.3.2 Creating a text I/O file

```
procedure CREATE(FILE     : in out FILE_TYPE;
                 BASE     : in     NODE_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     FORM_STRING := "";
                 LEVEL:   in     FORM_STRING := "")
```

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 contents of FORM have the syntax of a LIST_TYPE (Section 5.4). The FORM parameter is used to provide file characteristics concerning the creation of the external file. The predefined file characteristic SIZE may be used to specify an approximation to the number of STORAGE_UNITs that should be writable to the file. The SIZE characteristic is specified as '(SIZE => n)', where 'n' is any NATURAL number.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node (for the use of values of type LIST_TYPE, see Section 5.4.3.2 CAIS_LIST_UTILITIES). 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.

The default value for the attribute FILE_KIND is SECONDARY_STORAGE. the default value may be overridden by explicitly specifying a value of QUEUE, TERMINAL or MAGNETIC_TAPE in the FORM parameter (e.g., "(FILE_KIND => QUEUE)'). Specifying the FILE_KIND as QUEUE creates a solo queue.

Parameters:

FILE	is a file handle, initially closed, to be opened.
BASE	is an open 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 in 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, if the node identification is syntactically illegal, or if any group node specified in the value of the ACCESS_CONTROL parameter is unobtainable.
STATUS_ERROR	is raised if BASE is not an open node handle, or if FILE is an open file handle prior to the call.
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 not modifiable by the user.

121

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 security rules. 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     FORM_STRING := "";
                 LEVEL:    in     FORM_STRING := "")
is
    BASE : NODE_TYPE;
begin
    OPEN(BASE, BASE_PATH(NAME), (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);
end CREATE;
```

5.3.3.3 Opening a text I/O file

```
procedure OPEN(FILE      : in out FILE_TYPE;
               NODE      : in     NODE_TYPE;

               MODE      : in     FILE_MODE := INOUT_FILE);
```

Purpose:
 This procedure opens a handle on a file that has textual contents.

Parameters:
 FILE   is a file handle, initially closed, to be opened.

 NODE   is an open 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

122

of the file does not correspond with the element type
of this instantiation of the CAIS_TEXT_IO package.

STATUS_ERROR   is raised if the FILE is already open prior to the
                 call on OPEN.

INTENT_VIOLATION is raised if NODE has not been opened with an intent
                 establishing the right associated with the intent
                 associated with the MODE specified, as explained in
                 Table VIII.

Additional Interface:

```
procedure OPEN(FILE : in out FILE_TYPE;
               NAME : in      NAME_STRING;
               MODE : in      FILE_MODE :=INOUT_FILE);
is
    NODE : NODE_TYPE;
begin
    case MODE is
        IN_FILE => OPEN(NODE, NAME,(READ_CONTENTS));
        OUT_FILE => OPEN(NODE, NAME,(WRITE_CONTENTS));
        INOUT_FILE =>OPEN(NODE,NAME,(READ_CONTENTS,WRITE_CONTENTS));
        APPEND_FILE => OPEN(NODE, NAME,(APPEND_CONTENTS));
    end case;
    OPEN(FILE, NODE, MODE)
    CLOSE(NODE);
exception
    when others =>
        CLOSE(FILE);
        CLOSE(NODE);
end OPEN;
```

5.3.3.4 Reading from a file

    procedure GET(...);

Purpose:
 These procedures read characters from the specified text file.

 For all values of the attribute FILE_KIND only reading of the
 printable ASCII characters plus the format effectors called
 horizontal tabulation, vertical tabulation, carriage return, line
 feed, and form feed are defined. 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

123

terminator.

When text is being read from a file node whose 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.

5.3.3.5 Setting the input file

    procedure SET_INPUT(FILE : in FILE_TYPE);

Purpose:
    In addition to the semantics specified in the [LRM], the relation
    'CURRENT_INPUT of the calling process is set to refer to the node
    associated with FILE.

Parameters:
    FILE   is an open file handle.

Exceptions:
    MODE_ERROR   is raised if the mode of FILE is OUT_FILE or
                      APPEND_FILE.

    STATUS_ERROR   is raised if FILE is not open.

5.3.3.6 Setting the output file

    procedure SET_OUTPUT(FILE : in FILE_TYPE);

Purpose:
    In addition to the semantics specified in the [LRM], the relation
    'CURRENT_OUTPUT of the calling process is set to refer to the node
    associated with FILE.

Parameters:
    FILE   is an open file handle.

Exceptions:
    MODE_ERROR   is raised if the mode of FILE is IN_FILE.
    STATUS_ERROR   is raised if FILE is not open.

124

5.3.3.7 Setting the error file

procedure SET_ERROR(FILE : in FILE_TYPE);

Purpose:
Each CAIS process has an error file upon intitiation. This procedure provides an open file handle to be used for current error output. In addition, the relation 'CURRENT_ERROR of the calling procedure is set to refer to the node associated with FILE.

Parameters:
FILE   is the desired default file.

Exceptions:
MODE_ERROR   is raised if the mode of FILE is IN_FILE.

STATUS_ERROR   is raised if FILE is not open.

5.3.3.8 Determining the standard error file

function STANDARD_ERROR return FILE_TYPE;

Purpose:
This function returns a handle on error output file that was set at the start of program execution.

Exceptions: none

5.3.3.9 Determining the current error file

function CURRENT_ERROR return FILE_KIND;

Purpose:
This function returns the current error output file, which is either the standard error file or the file specified in the most recent invocation of SET_ERROR.

Parameters: none

Exceptions: none

125

5.3.4 Package CAIS_IO_EXCEPTIONS

This package 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 [LRM].

5.3.5 Package CAIS_IO_CONTROL

This package defines facilities that may be used to modify and/or query the functionality of CAIS files.

The package 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.

5.3.5.1 Types, subtypes, constants, and exceptions

    type CHARACTER_LIST 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 is limited private;

    type POSITION_TYPE is
        record
            ROW   : NATURAL;
            COLUMN : NATURAL;
        end record;

CHARACTER_LIST 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. Analogously to [LRM] type FILE_TYPE and the CAIS type NODE_TYPE, the CAIS provides a type FILE_TYPE whose values are internal references to Ada external files. FILE_TYPE is used for controlling the operations of all files. FUNCTION_KEY_DESCRIPTOR is used to determine the function keys entered from a terminal. POSITION_TYPE is used to specify a position on a terminal.

126

3-154

5.3.5.2 Obtaining an open node handle from a file handle

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

Purpose:
  This function returns an opened node handle for the node associated
  with a file.

Parameters:
  FILE   is an open file handle.

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

Exceptions:
  STATUS_ERROR   is raised if FILE is not open or if NODE is open.

5.3.5.3 Synchronizing program IO with system IO

    procedure SYCHRONIZE(FILE : in out FILE_TYPE);

Purpose:
  This procedure forces all data that has been written to the internal
  file FILE to be transmitted to the external file with which it is
  associated.

Parameters:
  FILE   is the internal file to be sychronized.

Exceptions:
  USE_ERROR   is raised if FILE is of mode IN_FILE.

  STATUS_ERROR   is raised if FILE is not open.

5.3.5.4 Establishing a LOG file

    procedure SET_LOG (FILE   : in out FILE_TYPE;
                       LOG_FILE : in   FILE_TYPE);

Purpose:
  This procedure establishes a log file for FILE (a file of mode
  INOUT_FILE or OUT_FILE). All elements written to internal file FILE
  are also written to LOG_FILE.

Parameters:
  FILE   is the file which is to have a log file.

  LOG_FILE   is the file to which the log should be written.

Exceptions:

127

MODE_ERROR	is raised if the mode of either FILE or LOG_FILE is IN_FILE.
USE_ERROR	is raised if FILE and LOG_FILE do not have the same values for the attribute ACCESS_METHOD or do not have compatible elements (implementation-defined).
STATUS_ERROR	is raised if LOG_FILE is not open.

5.3.5.5 Removing a log file

procedure CLEAR_LOG_FILE(FILE : in FILE_TYPE);

Purpose:
This procedure removes the log file established for FILE.

Parameters:
FILE   is an open file handle that has a log file.

Exceptions:
| USE_ERROR | is raised if FILE does not have a log file. |
| STATUS_ERROR | is raised if FILE is not open. |

5.3.5.6 Determining whether logging is specified

function LOGGING (FILE : in FILE_TYPE)
    return BOOLEAN;

Purpose:
This function returns TRUE if FILE has a log file; otherwise FALSE.

Parameters:
FILE   is an open file handle.

Exceptions:
STATUS_ERROR   is raised if FILE is not OPEN.

128

5.3.5.7 Determining the log file

```
function LOG_FILE (FILE : in FILE_TYPE)
    return FILE_KIND;
```

Purpose:
This function returns the current logging file associated with FILE.
The file handle returned is not open if not logging.

Parameters:
FILE   is an open FILE.

Exceptions:
USE_ERROR   is raised if FILE has no log file.

STATUS_ERROR   is raised if FILE is not OPEN.

5.3.5.8 Determining the file size

```
function SIZE (FILE : in FILE_TYPE)
            return NATURAL;
```

Purpose:
This function returns the number of elements contained in FILE.

Parameters:
FILE   is an open secondary storage or queue file.

Exceptions:
USE_ERROR   is raised if the file type of FILE is neither
                QUEUE nor SECONDARY_STORAGE.

STATUS_ERROR   is raised if FILE is not OPEN.

5.3.5.9 Setting the prompt string

```
procedure SET_PROMPT (TERMINAL : in FILE_TYPE;
                      PROMPT   : in STRING);
```
Purpose:
This procedure sets prompting string for TERMINAL. All future
requests for a line of input from TERMINAL will output prompt string
first. The prompting string and any echoed input are also copied to
the log file, if any.

Parameters:
TERMINAL   is an open terminal handle.

PROMPT   is the new value of the prompt.

Exceptions:
USE_ERROR   is raised if the attribute TERMINAL_TYPE does
                   not have the value SCROLL.

STATUS_ERROR   is raised if FILE is not open.

5.3.5.10 Determining the prompt string

    function PROMPT (TERMINAL : in FILE_TYPE) return STRING;

Purpose:
 This function returns the current prompt string for FILE.

Parameters:
 TERMINAL   is an open terminal handle.

Exceptions:
 USE_ERROR   is raised if the attribute TERMINAL_TYPE does
                   not have the value SCROLL.

 STATUS_ERROR   is raised if FILE is not open.

5.3.5.11 Determining intercepted characters

   function INTERCEPTED_CHARACTERS(FILE : in FILE_TYPE)
      return CHARACTER_LIST;

Purpose:
 This function returns the array CHARACTER_LIST that indicates the
 characters that cannot be read by a process.   A value of TRUE
 indicates that the character can be read (otherwise FALSE).

Parameters:
 FILE   is an open terminal handle.

Exceptions:
 USE_ERROR   is raised if the attribute FILE_KIND does not have
                   the value TERMINAL.

 STATUS_ERROR   is raised if FILE is not open.

130

5.3.5.12 Enabling/disabling function key usage

```
function ENABLE_FUNCTION_KEYS(TERMINAL : in FILE_TYPE;
                             ENABLE   : in BOOLEAN);
```

Purpose:
This procedure establishes whether function key sequences are to be
read as ASCII character sequences or as numbered function keys in a
terminal read operation. A value of TRUE for ENABLE indicates that
the function keys should be returned as a number. A value of FALSE
indicates that the function keys should be returned as an ASCII
character sequence.

Parameters:
TERMINAL   is an open terminal handle.

ENABLE   indicates how function keys are to be read.

Exceptions:
USE_ERROR   is raised if the attribute FILE_KIND does not have
                   the value TERMINAL.

STATUS_ERROR   is raised if FILE is not open.

5.3.5.13 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
(otherwise FALSE).

Parameters:
TERMINAL   is an open terminal handle.

Exceptions:
USE_ERROR   is raised if the attribute FILE_KIND does not
                   have the value TERMINAL.

STATUS_ERROR   is raised if FILE is not open.

131

5.3.5.14 Creating an associated queue

```
procedure ASSOCIATE
   (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    FORM_STRING := "";
    LEVEL:    in    FORM_STRING := "")
```

Purpose:
  This procedure creates an associated queue. The queue node to be
  created is identified by the QUEUE_BASE/QUEUE_KEY/QUEUE_RELATIONSHIP
  triad; the file node with which the queue node is to be associated,
  is identified by the open node handle FILE_NODE. The type of queue
  is specified in FORM by the value of the attribute QUEUE_TYPE. A
  value of COPY (i.e. "(QUEUE_TYPE => COPY)") specifies that a copy
  queue is to be created. A value of MIMIC (i.e. "(QUEUE_TYPE =>
  MIMIC)") specifies that a mimic queue is to be created. A
  relationship ASSOCIATE is created emanating from the queue node to
  the file node identified by the open node handle FILE_NODE. The
  queue node that is created inherits the ACCESS_METHOD of the file
  node with which it is associated. A DIRECT file (one whose
  ACCESS_METHOD is DIRECT) cannot be mimiced or copied.

  The ATTRIBUTES parameter defines and provides initial values for
  attributes of the node (for the use of values of type LIST_TYPE, see
  Section 5.4. (CAIS_LIST_UTILITIES). 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 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 handle to the file node with which
                    the queue is to associated.

  FORM   indicates file characteristics.

  ATTRIBUTES   defines initial values for attributes in the newly
                    created node.

132

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 for the queue node to be
                 created, if this node identification is syntactically
                 illegal, ACCESS_CONTROL parameter is unobtainable.

STATUS_ERROR   is raised if QUEUE_BASE is not an open node handle,
                 or if FILE is an open file handle prior to the call.

INTENT_VIOLATION is raised if QUEUE_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 security rules. SECURITY_VIOLATION is
                 raised only if the conditions for other exceptions
                 are not present.

Additional Interface:

```
procedure ASSOCIATE (QUEUE_NAME    : in NAME_STRING;
                     FILE_NODE     : in NODE_TYPE;
                     FORM          : in LIST_TYPE := EMPTY_LIST;
                     ATTRIBUTES    : in LIST_TYPE;
                     ACCESS_CONTROL : in   FORM_STRING := "";
                     LEVEL          : in   FORM_STRING := "")
is
   BASE : NODE_TYPE;
begin
   OPEN(BASE, BASE_NAME(QUEUE_NAME), (APPEND_RELATIONSHIPS));
   ASSOCIATE(BASE, LAST_KEY(QUEUE_NAME), LAST_RELATION(QUEUE_NAME),
            FILE_NODE, FORM, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
   CLOSE(BASE);
exception
   when others =>
        CLOSE(BASE);
end ASSOCIATE;

procedure ASSOCIATE (QUEUE_BASE    : in NODE_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;
```

133

```
                              ACCESS_CONTROL : in    FORM_STRING := "";
                              LEVEL          : in    FORM_STRING := "") .
   is
      FILE_NODE : NODE_TYPE;
   begin
      OPEN(FILE_NODE, FILE_NAME, (READ_ATTRIBUTES));
      ASSOCIATE(QUEUE_BASE, QUEUE_KEY, QUEUE_RELATION,
               FILE_NODE, FORM, ATTRIBUTES, ACCESS_CONTROL, LEVEL);
      CLOSE(FILE_NODE);
   exception
      when others =>
         CLOSE(FILE_NODE);
   end ASSOCIATE;

   procedure ASSOCIATE (QUEUE_NAME : in NAME_STRING;
                        FILE_NAME  : in NAME_STRING;
                        FORM       : in LIST_TYPE := EMPTY_LIST;
                        ATTRIBUTES : in LIST_TYPE;
                        ACCESS_CONTROL: in    FORM_STRING := "";
                        LEVEL      : in    FORM_STRING := "")
   is
      FILE_NODE : NODE_TYPE;
      QUEUE_BASE : NODE_TYPE;
   begin
      OPEN(QUEUE_BASE, BASE_PATH(QUEUE_NAME), (APPEND_RELATIONSHIPS));
      OPEN(FILE_NODE, FILE_NAME, (READ_ATTRIBUTES));
      ASSOCIATE(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);
   end ASSOCIATE;
```

5.3.6 Package CAIS_SCROLL_TERMINAL

This package provides the functionality of a scroll terminal. A scroll
terminal consists of two devices: an input device (keyboard) and an
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.

Each of the output devices for a scroll terminal has "positions" in
which printable ASCII characters may be graphically displayed. The

134

positions are arranged into horizontal rows and vertical columns. Each position is identifiable by the combination of a row number and a column number. 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, but the new position has a greater 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.

A printer has a fixed number of columns and might have a fixed number of rows. The rows are incrementally indexed starting with one after opening the device or performing the NEW_PAGE (Section 5.3.7.19) operation. The columns are incrementally indexed starting with one at the left side of the printer.

5.3.6.1 Types, subtypes, constants, and exceptions

```
subtype FILE_TYPE is CAIS_IO_CONTROL.FILE_TYPE;

subtype FUNCTION_KEY_DESCRIPTOR is
   CAIS_IO_CONTROL.FUNCTION_KEY_DESCRIPTOR;

subtype TAB_ENUMERATION is CAIS_IO_CONTROL.TAB_ENUMERATION;

USE_ERROR    : exception renames CAIS_IO_EXCEPTIONS.USE_ERROR;
MODE_ERROR   : exception renames CAIS_IO_EXCEPTIONS.MODE_ERROR;
STATUS_ERROR : exception renames CAIS_IO_EXCEPTIONS.STATUS_ERROR;
LAYOUT_ERROR : exception renames CAIS_IO_EXCEPTIONS.LAYOUT_ERROR;
DEVICE_ERROR : exception renames CAIS_IO_EXCEPTIONS.DEVICE_ERROR;
```

FILE_TYPE is used for file handles. FUNCTION_KEY_DESCRIPTOR is used to obtain information about function keys read from a terminal. TAB_ENUMERATION is used to specify the type of tab stop to be set. USE_ERROR is raised if an operation is attempted that is not possible for reasons that depend on the characteristics of the external file. MODE_ERROR is raised by an attempt to read from a file of mode OUT_FILE or write to a file of mode IN_FILE. STATUS_ERROR is raised if the handle on the terminal file is not open. DEVICE_ERROR is raised if an input or output operation cannot be completed because of a malfunction of the underlying system. LAYOUT_ERROR is raised by an attempt to set column or row numbers in excess of specified maximum values.

135

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 terminal file given by TERMINAL.

Parameters:
  TERMINAL   is an open handle on a terminal file.

  POSITION   is the new active position in the terminal file.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                    is not SCROLL.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  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 POSITION (TERMINAL : in FILE_TYPE)
        return POSITION_TYPE;

Purpose:
  This function returns the active position of TERMINAL.

Parameters:
  TERMINAL   is an open handle on a terminal file.

Exceptions:

136

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
               is not SCROLL.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be
               completed because of a malfunction of the underlying
               system.

Additional Interface:

```
    function POSITION
        return POSITION_TYPE
    is
    begin
        return POSITION(CURRENT_OUTPUT);
    end POSITION;
```

5.3.6.4 Determining the size of the terminal

```
    function SIZE (TERMINAL : in FILE_TYPE)
        return POSITION_TYPE;
```

Purpose:
This function returns the maximum row and maximum column of
TERMINAL. A value of zero for the row number indicates that the row
number is unlimited.

Parameters:
TERMINAL   is an open handle on a terminal file.

Exceptions:
USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
               is not SCROLL.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be
               completed because of a malfunction of the underlying
               system.

Additional Interface:

```
    function SIZE
        return POSITION_TYPE
    is
    begin
```

137

```
        return SIZE(CURRENT_OUTPUT);
    end 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/vertical tab stop at the
  column/row of the active position.

Parameters:
  TERMINAL   is an open handle on a terminal file.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                    is not SCROLL or the number of rows for the terminal
                    is unlimited.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  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_INPUT, 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/vertical tab stop from the
  column/row of the active position.

Parameters:
  TERMINAL   is an open handle on a terminal file.

Exceptions:

138

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not SCROLL or there is no tab stop of the designated type at the active position.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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 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 handle on a terminal file.

COUNT   is a positive integer indicating the number of tab stops the active position is to advance.

Exceptions:
USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not SCROLL.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

139

```
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 signals the bell (beeper) on the terminal.

Parameters:
  TERMINAL   is an open handle on a terminal file.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                    is not SCROLL.
  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  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 device and
  advances the active position by one position.

Parameter:
  TERMINAL   is an open handle on a terminal file.

  ITEM   is the character to be written.

140

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                     is not SCROLL.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  DEVICE_ERROR   is raised if an input or output operation cannot be
                     completed because of a malfunction of the underlying
                     system.

Additional Interface:

```
    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 writing the character in the rightmost position of a row, the
  active position is the first position of the next row.

5.3.6.10 Setting the ECHO on a terminal

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

Purpose:
  This procedure establishes whether characters entered at the
  terminal keyboard are echoed to its associated output device. When
  TO is given as TRUE, each character entered at the keyboard is
  echoed to the output device. When TO is given as FALSE, characters
  entered at the keyboard are not echoed to its associated output
  device.

141

Parameters:

TERMINAL   is an open handle on a terminal file.

TO   indicates the new value of ECHO.

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                  is not SCROLL.

MODE_ERROR   is raised if TERMINAL is of mode OUT_FILE or
                  APPEND_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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);
end SET_ECHO;
```

5.3.6.11 Determining the ECHO on a terminal

    function ECHO (TERMINAL : in FILE_TYPE) return BOOLEAN;

Purpose:
    This function returns whether echo is enabled (TRUE) or disabled
    (FALSE).

Parameters:

TERMINAL   is an open handle on a terminal file.

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                  is not SCROLL.

MODE_ERROR   is raised if TERMINAL is of mode OUT_FILE or
                  APPEND_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be
                  completed because of a malfunction of the underlying
                  system.

Additional Interface:

142

```
function ECHO return BOOLEAN
is
begin
    return ECHO(CURRENT_INPUT);
end ECHO;
```

5.3.6.12 Determining the number of function keys

```
function FUNCTION_KEYS(TERMINAL : in FILE_TYPE)
    return NATURAL;
```

Purpose:
This function returns the maximum function key identifier that can be returned by a GET operation in the terminal file given by TERMINAL.

Parameters:

TERMINAL   is an open handle on a terminal file.

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not SCROLL.

MODE_ERROR   is raised if TERMINAL is of mode OUT_FILE or APPEND_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function FUNCTION_KEYS return NATURAL
is
begin
    return FUNCTION_KEYS(CURRENT_INPUT);
end FUNCTION_KEYS;
```

143

5.3.6.13 Reading a character from a terminal

```
(TERMINAL : in    FILE_TYPE;
            ITEM    :    out CHARACTER;
            KEYS    : in out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:
   This procedure reads either a single character into ITEM or reads
   a single function key into KEYS.

Parameters:
   TERMINAL   is an open handle on a terminal file.

   ITEM   is the character that was read.

   KEYS   describes the function key that was read.

Exceptions:
   USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE

                     is not SCROLL.

   MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

   STATUS_ERROR   is raised if TERMINAL is not open.

   DEVICE_ERROR   is raised if an input or output operation cannot be
                     completed because of a malfunction of the underlying
                     system.

Additional Interface:

```
(ITEM     :    out CHARACTER;
            KEYS      : in out FUNCTION_KEY_DESCRIPTOR)
 is
 begin
   GET(CURRENT_OUTPUT, ITEM, KEYS);
 end GET;
```


5.3.6.14 Reading all available characters from a terminal

```
procedure GET(TERMINAL : in·   FILE_TYPE;
              ITEM    :    out STRING;
              LAST    :    out NATURAL;
              KEYS    : in out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:
   This procedure successively reads characters and function keys into
   ITEM and KEYS until either all positions of ITEM are filled or there
   are no more characters buffered for the terminal. Upon completion,
   LAST contains the index of the last position in ITEM to contain a

144

character that has been read. In addition, the function keys that were read are entered into KEYS.

Parameters:

TERMINAL is an open handle on a terminal file.

ITEM is the characters that were read.

LAST is the position of the last character read in ITEM.

KEYS describes the function keys that were read.

Exceptions:

USE_ERROR is raised if a value of the attribute TERMINAL_TYPE is not SCROLL.

MODE_ERROR is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR is raised if TERMINAL is not open.

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    : in out FUNCTION_KEY_DESCRIPTOR)
   is
   begin
      GET(CURRENT_INPUT, ITEM, LAST, KEYS);
   end GET;
```

Notes:

If there are no elements available for reading from the terminal, then LAST has a value one less than ITEM'FIRST and FUNCTION_KEY_COUNT(KEYS) 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

145

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   describes a sequence of function keys.

INDEX   is the function key sequence 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 name of the function key
 sequence designated by KEY_IDENTIFIER. It also returns the index of
 the last character of the function key name in INDEX.

Parameters:

TERMINAL   is an open handle on a terminal file.

KEY_IDENTIFIER   is the identification number of a function key for
                 the terminal associated with FILE.

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:

146

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                   is not SCROLL.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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 FUNCTION_KEYS(TERMINAL).

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 to column one, COUNT lines
   after the active position.

Parameters:
   TERMINAL   is an open handle on a terminal file.

   COUNT   is the number of lines to advance.
Exceptions:
   USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                    is not SCROLL.

   MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

   STATUS_ERROR   is raised if TERMINAL is not open.

   DEVICE_ERROR   is raised if an input or output operation cannot be
                    completed because of a malfunction of the underlying
                    system.

147

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 to the first column of the first line of a new page.

Parameters:
TERMINAL   is an open handle on a terminal file.

Exceptions:
USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                    is not SCROLL.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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;
```

148

5.3.7 Package CAIS_PAGE_TERMINAL

This package provides the functionality of a page terminal. A page terminal consists of two devices: an input device (keyboard) and an 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 printer or 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. 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, but the new position has a greater 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.

5.3.7.1 Types, subtypes, constants, and exceptions

```
subtype FILE_TYPE is CAIS_IO_CONTROL.FILE_TYPE;

subtype FUNCTION_KEY_DESCRIPTOR is
    CAIS_IO_CONTROL.FUNCTION_KEY_DESCRIPTOR;

subtype POSITION_TYPE is CAIS_IO_CONTROL.POSITION_TYPE;

subtype TAB_ENUMERATION is CAIS_IO_CONTROL.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)
```

149

of BOOLEAN;

DEFAULT_RENDITION : constant GRAPHIC_RENDITION_ARRAY
  := (PRIMARY_RENDITION => TRUE, others => FALSE);

USE_ERROR   : exception renames CAIS_IO_EXCEPTIONS.USE_ERROR;
MODE_ERROR   : exception renames CAIS_IO_EXCEPTIONS.MODE_ERROR;
STATUS_ERROR : exception renames CAIS_IO_EXCEPTIONS.STATUS_ERROR;
LAYOUT_ERROR : exception renames CAIS_IO_EXCEPTIONS.LAYOUT_ERROR;
DEVICE_ERROR : exception renames CAIS_IO_EXCEPTIONS.DEVICE_ERROR;

FILE_TYPE is used for file handles. FUNCTION_KEY_DESCRIPTOR is used to
obtain information about function keys read from a terminal.
POSITION_TYPE indicates a position on a terminal. TAB_ENUMERATION is
used to specify the type 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. USE_ERROR is
raised if an operation is attempted that is not possible for reasons
that depend on the characteristics of the external file. MODE_ERROR is
raised by an attempt to read from a file of mode OUT_FILE or write to a
file of mode IN_FILE. STATUS_ERROR is raised if the terminal is not
open. DEVICE_ERROR is raised if an input or output operation cannot be
completed because of a malfunction of the underlying system.
LAYOUT_ERROR is raised by an attempt to set column or row numbers in
excess of specified maximum values.

5.3.7.2 Setting the active position

    procedure SET_POSITION (TERMINAL : in FILE_TYPE;
                            POSITION : in POSITION_TYPE);

Purpose:
  This procedure moves the active position to the specified POSITION
  on the display of the terminal.

Parameters:
  TERMINAL   is an open handle on a terminal file.

  POSITION   is the new active position for the terminal.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                    is not PAGE.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  LAYOUT_ERROR   is raised if the position does not exist on the

150

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;
```

5.3.7.3 Determining the active position

function POSITION (TERMINAL : in FILE_TYPE)

return POSITION_TYPE;

Purpose:
  This function returns the active position of TERMINAL.

Parameters:
  TERMINAL   is an open handle on a terminal file.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                   is not PAGE.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  DEVICE_ERROR   is raised if an input or output operation cannot be
                   completed because of a malfunction of the underlying
                   system.

Additional Interface:

```
function POSITION return POSITION_TYPE
is
begin
    return POSITION(CURRENT_OUTPUT)
end POSITION;
```

151

5.3.7.4 Determining the size of the terminal

```
function SIZE (TERMINAL : in FILE_TYPE)
    return POSITION_TYPE;
```

Purpose:
This function returns the maximum row and maximum column of the terminal.

Parameters:
TERMINAL   is an open handle on a terminal file.

Exceptions:
USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not PAGE.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function SIZE
    return POSITION_TYPE
is
begin
    return SIZE(CURRENT_OUTPUT);
end 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/vertical tab stop at the column/row of the active position.

Parameters:
TERMINAL   is an open handle on a terminal file.

Exceptions:
USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not PAGE.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

152

STATUS_ERROR   is raised if TERMINAL is not open.

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_INPUT, 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/vertical tab stop from the
  column/row of the active position.

Parameters:
  TERMINAL   is an open handle on a terminal file.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                 is not PAGE or there is no tab stop of the designated
                 type at the active position.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  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;

153

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 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 handle on a terminal file.

COUNT   is a positive integer indicating the number of tab stops the active position is to advance.

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not PAGE.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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 signals the bell (beeper) on the terminal.

Parameters:

TERMINAL   is an open handle on a terminal file.

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not PAGE.

154

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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;

5.3.7.9 Writing to the terminal
    procedure PUT (TERMINAL : in out FILE_TYPE;
                   ITEM   : in   CHARACTER);

Purpose:
  This procedure writes a single character to the output device and
  advances the active position by one position.

Parameter:
  TERMINAL   is an open handle on a terminal file.

  ITEM   is the character to be written.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                 is not PAGE.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  DEVICE_ERROR   is raised if an input or output operation cannot be
                 completed because of a malfunction of the underlying
                 system.

Additional Interface:

    procedure PUT (ITEM : in CHARACTER)
    is
    begin
        PUT(CURRENT_OUTPUT, ITEM);
    end PUT;

    procedure PUT (TERMINAL : in FILE_TYPE;
                   ITEM   : in STRING)
    is
    begin

155

```
        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 writing the character in the rightmost position of a row, the
  active position is the first position of the next row.

5.3.7.10 Setting the ECHO on a terminal

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

Purpose:
  This procedure establishes whether characters entered at the
  terminal keyboard are echoed to its associated output device. When
  TO is given as TRUE, each character entered at the keyboard is
  echoed to the output device. When TO is given as FALSE, characters
  entered at the keyboard are not echoed to its associated output
  device.

Parameters:
  TERMINAL   is an open handle on a terminal file.

  TO   indicates the new value of ECHO.

Exceptions:
  USE_ERROR   is raised if a value of the attribute
  TERMINAL_TYPE

                     is not PAGE.

  MODE_ERROR   is raised if TERMINAL is of mode OUT_FILE
  or

                     APPEND_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  DEVICE_ERROR   is raised if an input or output operation cannot
  be

                     completed because of a malfunction of the
  underlying

                     system.

Additional Interface:

156

```
procedure SET_ECHO (TO : in BOOLEAN := TRUE)
is
begin
    SET_ECHO(CURRENT_INPUT, TO);
end SET_ECHO;
```

5.3.7.11 Determining the ECHO on a terminal

```
function ECHO (TERMINAL :  in FILE_TYPE return BOOLEAN;
```

Purpose:
  This function returns whether echo is enabled (TRUE) or disabled
  (FALSE).

Parameters:
  TERMINAL   is an open handle on a terminal file.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                    is not PAGE.

  MODE_ERROR   is raised if TERMINAL is of mode OUT_FILE or
                    APPEND_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  DEVICE_ERROR   is raised i. n 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 FUNCTION_KEYS(TERMINAL : in FILE_TYPE)
        return NATURAL;
```

Purpose:
  This function returns the maximum function key identifier that can
  be returned by a GET operation in the terminal file given by
  TERMINAL.

Parameters:
  TERMINAL   is an open handle on a terminal file.

157

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not PAGE.

MODE_ERROR   is raised if TERMINAL is of mode OUT_FILE or APPEND_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

```
function FUNCTION_KEYS return NATURAL
is
begin
    return FUNCTION_KEYS(CURRENT_INPUT);
end FUNCTION_KEYS;
```

5.3.7.13 Reading a character from a terminal

```
procedure GET(TERMINAL : in     FILE_TYPE;
              ITEM      :    out CHARACTER;
              KEYS      : in out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:
  This procedure reads either a single character into ITEM or reads a single function key into KEYS.

Parameters:

TERMINAL   is an open handle on a terminal file

ITEM   is the character that was read.

KEYS   describes the function key that was read.

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not PAGE.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

Additional Interface:

158

```
procedure GET(ITEM     :     out CHARACTER;
              KEYS     : in out FUNCTION_KEY_DESCRIPTOR)
is
begin
  GET(CURRENT_OUTPUT, ITEM, KEYS);
end GET;
```

5.3.7.14 Reading all available characters from a terminal

```
procedure GET(TERMINAL : in     FILE_TYPE;
              ITEM      :    out STRING;
              LAST      :    out NATURAL;
              KEYS      : in out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:
This procedure successively reads characters and function keys into ITEM and KEYS until either all positions of ITEM are filled or there are no more characters buffered for the terminal. Upon completion, LAST contains the index of the last position in ITEM to contain a character that has been read. In addition, the function keys that were read are entered into KEYS.

Parameters:

TERMINAL   is an open handle on a terminal file.

ITEM   is the string that was read.

LAST   is the position of the last character read in ITEM.

KEYS   describes the function keys that were read.

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not PAGE.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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      : in out FUNCTION_KEY_DESCRIPTOR)
is
begin
  GET(CURRENT_INPUT, ITEM, LAST, KEY)
end GET;
```

159

Notes:

   If there are no elements available for reading from the terminal, then *LAST* has a value one less than ITEM'FIRST and FUNCTION_KEY_COUNT(KEYS) 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.

Parameters:
   KEYS   describes a sequence of function keys.

   INDEX   is the function key sequence 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).

160

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 name of the function key
sequence designated by KEY_IDENTIFIER. It also returns the index of
the last character of the function key name in INDEX.

Parameters:
TERMINAL   is an open handle on a terminal file.

KEY_IDENTIFIER   is the identification number of a function key for
                  the terminal.

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 a value of the attribute TERMINAL_TYPE
                  is not PAGE.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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 FUNCTION_KEYS(TERMINAL).

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);
```

161

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 handle on a terminal file.

  COUNT   is the number of characters to be deleted.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                      is not PAGE or the value of COUNT is greater than the
                      number of positions including and following the
                      active position.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  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_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 out FILE_TYPE;
                       COUNT     : in     POSITIVE);
```

Purpose:
  This procedure eleted COUNT lines starting at the active position
  and advancing toward the end position. Adjacent lines are shifted
  from the bottom toward the active position. COUNT lines from the
  bottom of the display are erased. The active position is not
  changed.

162

Parameters:

TERMINAL   is an open handle on a terminal file.

COUNT   is the number of lines to be deleted.

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                      is not PAGE or the value of COUNT is greater than the
                      number of rows including and after the active
                      position.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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;
```

5.3.7.20 Erasing characters in a line

```
procedure ERASE_CHARACTER (TERMINAL : in out 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 handle on a terminal file.

COUNT   is the number of characters to be erased

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                      is not PAGE or the value of COUNT is greater than the

                      number of positions including and after the active
                      position.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

163

    STATUS_ERROR   is raised if TERMINAL is not open.

    DEVICE_ERROR   is raised if an input or output operation cannot be
                       completed because of a malfunction of the underlying
                       system.

Additional Interace:

    procedure ERASE_CHARACTER (COUNT ; in POSITIVE :=1)
    is
    begin
        ERASE_CHARACTER(CURRENT_OUTPUT, COUNT);
    end ERASE_CHARACTER;

5.3.7.21 Erasing characters in the display

    procedure ERASE_IN_DISPLAY (TERMINAL : in out FILE_TYPE;
                                SELECTION : in   SELECT_ENUMERATION);

Purpose:
    This procedure erases the characters in the entire 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:
    TERMINAL   is an open handle on a terminal file.

    SELECTION   is the portion of the display to be erased.

Exceptions:
    USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                       is not PAGE.

    MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

    STATUS_ERROR   is raised if TERMINAL is not open.

    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;

164

5.3.7.22 Erasing in a line

```
procedure ERASE_IN_LINE (TERMINAL  : in out 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 handle on a terminal file.

   SELECTION   is the portion of the line to be erased.

Exceptions:
   USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                   is not PAGE.

   MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

   STATUS_ERROR   is raised if TERMINAL is not open.

   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 characters in a line

```
procedure INSERT_SPACE (TERMINAL : in out 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 rightmost
   characters on the line may be lost. The active position is advanced
   to the right COUNT character positions.

Parameters:
   TERMINAL   is an open handle on a terminal file.

   COUNT   is the number of SPACE characters to be inserted.

165

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                  is not PAGE or the value of COUNT is greater than the
                  number of columns including and after the active
                  position.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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_SPACE (COUNT : in POSITIVE := 1)
is
begin
    INSERT_SPACE(CURRENT_OUTPUT, COUNT);
end INSERT_SPACE;
```

5.3.7.24 Inserting lines in the display

```
procedure INSERT_LINE (TERMINAL : in out FILE_TYPE;
                       COUNT    : in     POSITIVE := 1);
```

Purpose:
This procedure inserts COUNT blank lines into the display at the
active line. The lines at and below the top of the display are
lost. The active position remains unchanged.

Parameters:

TERMINAL   is an open handle on a terminal file.

COUNT   is the number of blank lines to be inserted.

Exceptions:

USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                  is not PAGE or the value of COUNT is greater than the
                  number of rows including and after the active
                  position.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be
                  completed because of a malfunction of the underlying
                  system.

Additional Interface:

166

```
procedure INSERT_LINE (COUNT : in POSITIVE)
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 TERMINAL; otherwise it returns FALSE.

Parameters:
TERMINAL   is an open handle on a terminal file.

RENDITION   is a combination of graphic renditions.

Exceptions:
USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                is not PAGE.

MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

STATUS_ERROR   is raised if TERMINAL is not open.

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;
```

167

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 PUT.

Parameters:
  TERMINAL   is an open handle on a terminal file.

  RENDITION   is the graphic rendition to be used in subsequent
                    PUTs.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                    is not PAGE or the selected graphic renditions are
                    not supported by TERMINAL.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

  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 CAIS_FORM_TERMINAL

This package provides functionality for manipulating 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.

168

A form is a two-dimensional matrix of character positions. The rows of a form are indexed by POSITIVE numbers starting with row 1 at the top of the display. The columns of a form are indexed by POSITIVE numbers starting with column 1 at the left side of the form. The position identified by row 1, column 1, is called the starting position of the form. The position with the highest row and column indices 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, subtypes, constants, and exceptions

```
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
    (SIZE                           : POSITION_TYPE;
     AREA_QUALIFIER_REQUIRES_SPACE : BOOLEAN)
```

169

        is private;

    subtype FILE_TYPEis CAIS_IO_CONTROL.FILE_TYPE;

    subtype PRINTABLE_CHARACTER 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. A 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.

    USE_ERROR   : exception renames CAIS_IO_EXCEPTIONS.USE_ERROR;
    MODE_ERROR   : exception renames CAIS_IO_EXCEPTIONS.MODE_ERROR;
    STATUS_ERROR : exception renames CAIS_IO_EXCEPTIONS.STATUS_ERROR;
    LAYOUT_ERROR : exception renames CAIS_IO_EXCEPTIONS.LAYOUT_ERROR;
    DEVICE_ERROR : exception renames CAIS_IO_EXCEPTIONS.DEVICE_ERROR;

USE_ERROR is raised if an operation is attempted that is not possible for reasons that depend on the characteristics of the external file. MODE_ERROR is raised by an attempt to read from a file of mode OUT_FILE or write to a file of mode IN_FILE. STATUS_ERROR is raised if the handle on the terminal file is not open. DEVICE_ERROR is raised if an input or output operation cannot be completed because of a malfunction of the underlying system. LAYOUT_ERROR is raised by an attempt to set column or row numbers in excess of specified maximum values.

5.3.8.2 Determining the number of function keys

    function FUNCTION_KEYS(TERMINAL : in   FILE_TYPE)
        return NATURAL;

Purpose:
  This function returns the maximum value that can be returned by the function TERMINATION_KEY.

Parameters:
  TERMINAL   is an open handle on a terminal file.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE is not FORM.

  MODE_ERROR   is raised if TERMINAL is of mode OUT_FILE or APPEND_FILE.

  STATUS_ERROR   is raised if TERMINAL is not open.

170

DEVICE_ERROR   is raised if an input or output operation cannot be
                completed because of a malfunction of the underlying
                system.

Additional Interface:

```
function FUNCTION_KEYS return NATURAL
is
begin
    return FUNCTION_KEYS(CURRENT_INPUT);
end FUNCTION_KEYS;
```

5.3.8.3 Opening a form

```
procedure OPEN
    (FORM                           :    out FORM_TYPE;
     SIZE                           : in     POSITION_TYPE;
     AREA_QUALIFIER_REQUIRES_SPACE : in     BOOLEAN);
```

Purpose:
This procedure opens a FORM of the specified size to be used with a
form terminal.

Parameters:
FORM   is the form to be opened for manipulation.

SIZE   indicates the size of form to be opened (which should
                correspond with the size of the form terminal on
                which it will be activated).

AREA_QUALIFIER_REQUIRES_SPACE indicates whether the area qualifier
                requires space on this form (i.e., the position in
                which the area qualifier is defined may not be used
                for the display of data).

Exceptions: none

5.3.8.4 Determining whether a form is open

```
function IS_OPEN(FORM : in FORM_TYPE) return BOOLEAN;
```

Purpose:
This function returns TRUE if the FORM has been opened; otherwise
it returns FALSE.

Parameters:
FORM   is the form being queried.

Exceptions: none

171

5.3.8.5 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. 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 an open form.

 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 FORM is not open.

5.3.8.6 Removing an area qualifier

```
procedure REMOVE_AREA_QUALIFIER
        (FORM  : in out FORM_TYPE);
```

Purpose:
 This procedure removes an area qualifier from the active position.

Parameters:
 FORM   is the open 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 FORM is not open.

172

5.3.8.7 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:
  STATUS_ERROR   is raised if FORM is not open.

  LAYOUT_ERROR   is raised if POSITION does not identify a position on

                  FORM.

5.3.8.8 Moving to the next qualified area

```
procedure NEXT_QUALIFIED_AREA(FORM  : in out FORM_TYPE;
                              COUNT : in     POSITIVE := 1);
```

Purpose:
  This procedure moves the active position COUNT qualified areas
  toward the end of the form.

Parameters:
  FORM   is an open form.

  COUNT   is the number of qualified areas the active position
                  is to be advanced.
Exceptions:
  STATUS_ERROR   is raised if either FORM is not open or FORM has
                  fewer than COUNT qualified areas after the active
                  position.

173

5.3.8.9 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 an open form.

 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.

 STATUS_ERROR   is raised if FORM is not open.

Additional interfaces:

```
procedure PUT (FORM : in out FORM_TYPE;
               ITEM : in     STRING)
is
begin
    for INDEX in ITEM'FIRST .. ITEM'LAST loop
        PUT(FORM, ITEM(INDEX)); -- Write a single character
    end loop;
end PUT;
```

5.3.8.10 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 is located.

Parameters:
 FORM   is an open form.

Exceptions:
 STATUS_ERROR   is raised if FORM is not open or no area
                     qualifiers have been defined for FORM.

174

5.3.8.11 Erasing the form

```
procedure ERASE_FORM
    (FORM : in out FORM_TYPE);
```

Purpose:
 This procedure removes all area qualifiers and places SPACE
 characters in all positions.

Parameters:
 FORM   is an open form.

Exceptions:
 STATUS_ERROR   is raised if FORM is not open.

5.3.8.12 Activating a form on a terminal

```
procedure ACTIVATE(TERMINAL : in     FILE_TYPE;
                   FORM      : in out FORM_TYPE);
```

Purpose:
 This procedure activates the form on the terminal. The terminal
 display is modified to reflect the contents of the form.   When the
 user of the terminal enters a termination key the modified form is
 returned. Only the unprotected areas of the form may be modified by
 the user.

Parameters:
 TERMINAL   is an open handle on a terminal file.

 FORM   is an open form.

Exceptions:
 USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                   is not FORM. It is also raised if either the size of
                   the form is not compatible with the terminal (i.e.
                   the sizes differ or the area qualifier requires space
                   on the terminal, but not on the form.

 STATUS_ERROR   is raised if either TERMINAL is not open or FORM is
                   not open.

175

5.3.8.13 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.
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 an open form.

ITEM   is the character that was read.

Exceptions:
STATUS_ERROR   is raised if FORM is not open.

Additional interfaces:

```
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;
```

5.3.8.14 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 an open form.

Exceptions:
STATUS_ERROR   is raised if FORM is not open.

176

5.3.8.15 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_FORM procedure. A value of
  zero indicates the normal termination key (e.g., the ENTER key).

Parameters:
  FORM   is an open form.

Exceptions:
  STATUS_ERROR   is raised if FORM is not open.

5.3.8.16 Determining the size of a form/terminal

    function SIZE(FORM   : in FORM_TYPE) return POSITION_TYPE;
    function SIZE(TERMINAL : in FILE_KIND) return POSITION_TYPE;

Purpose:
  These functions return the position of the last column of the last
  row of the form/terminal.

Parameters:
  FORM   is an open form.

  TERMINAL   is an open handle on a terminal file.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                  is not FORM.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if FORM/TERMINAL is not open.

  DEVICE_ERROR   is raised if an input or output operation cannot be
                  completed because of a malfunction of the underlying
                  system.

Additional Interface:

    function SIZE return POSITION_TYPE
    is
    begin
        return SIZE(CURRENT_OUTPUT);
    end SIZE;

177

5.3.8.17 Determining if the area qualifier requires space

```
function AREA_QUALIFIER_REQUIRES_SPACE
   (FORM : in FORM_TYPE)
   return BOOLEAN;
function AREA_QUALIFIER_REQUIRES_SPACE
   (TERMINAL : in FILE_TYPE)
   return BOOLEAN;
```

Purpose:
  These functions return TRUE if the area qualifier requires space on
  the form/terminal; otherwise returns FALSE.

Parameters:
  FORM   is an open form.

  TERMINAL   is an open handle on a terminal file.

Exceptions:
  USE_ERROR   is raised if a value of the attribute TERMINAL_TYPE
                    is not FORM.

  MODE_ERROR   is raised if TERMINAL is of mode IN_FILE.

  STATUS_ERROR   is raised if FORM/TERMINAL is not open.

  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 CAIS_GENERAL_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
[ANSI 78].

The CAIS supports the transfer of information to and from a single tape
volume. The CAIS supports the transfer of source programs.   Data
transferred to and from tapes may consist of the following characters:

  Characters   Representation of Characters

  all printable characters   corresponding characters

178

horizontal tab	ASCII.HT
vertical tab	ASCII.VT
carriage return	ASCII.CR
line feed	ASCII.LF
form feed	ASCII.FF
end-of-file	zero or more NULLs followed immediately by a tape mark.

Use of other characters is not defined with the exception of NULL which is used in the end-of-file definition. The end-of-line terminator is the line feed. The end-of-page terminator is the form feed. An end-of-page terminator must be preceded by an end-of-line terminator. An end-of-file must be preceded by an end-of-page terminator.

To use a tape drive, a handle on the 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 mounted as unlabeled tapes; however, only initialized labeled tapes may be mounted as labeled tapes. The semantics of both initializing and mounting imply the loading of the tape. Once a tape has been mounted, CAIS_TEXT_IO routines are used to get information to and from the tape.

When information transfer is completed, the tape is dismounted or rewound using either the UNLOAD or NO_UNLOAD option. If UNLOAD is chosen, the tape is completely rewound. If NO_UNLOAD is chosen, the tape is rewound to the beginning of the tape and may be remounted.

Once a tape is unloaded, another tape may be mounted. When the user is finished utilizing the drive, he closes the file handle on the drive (see Section 5.3.4).

The following is the format of files on unlabeled tape where an '*' represents a tape mark and BOT is the beginning of the tape.

BOT file * file *   ...   * file **

The following is the format of files on labeled tape where an '*' represents a tape mark, BOT is the beginning of the tape, VOL is the Volume-Header Label, HDR is the File-Header label, and EOF is the End-of-File label.

BOT VOL HDR * file * EOF * HDR * file * EOF * ... * HDR * file * EOF **

(If a labeled tape is mounted as an unlabeled tape, then each label group is considered to be a file.)

179

5.3.9.1 Types, subtypes, constants, and exceptions

```
type STATUS is
     (START_OF_TAPE,      -- at beginning-of-tape reflector mark
      END_OF_TAPE,        -- at the end-of-tape reflector mark
      TAPE_MARK_READ,     -- tape mark has just been read
      DOUBLE_TAPE_MARK,   -- delimits the logical end of tape
      DRIVE_READY,        -- device is on line
      MOUNTED,            -- tape has been mounted
      LABELED,            -- tape is a labeled tape
      WRITE_ENABLED);     -- write ring is in place
type STATUS_ARRAY is array(STATUS) of BOOLEAN;
type LOAD_TYPE is (UNLOAD, NO_UNLOAD);
type DENSITY_TYPE is (DENSITY_800, DENSITY_1600, DENSITY_6250);
```

STATUS_ARRAY contains current information about a tape drive. LOAD_TYPE
determines whether a rewind will stop at the beginning-of-tape reflector
mark or rewind all of the way.

```
subtype VOLUME_STRING is STRING(1..6);
subtype FILE_STRING is STRING(1..17);
subtype NAME_STRING is STRING;
subtype TAPE_TYPE is CAIS.FILE_TYPE;


MODE_ERROR    :  exception renames CAIS_IO_EXCEPTIONS.MODE_ERROR;
STATUS_ERROR  :  exception renames CAIS_IO_EXCEPTIONS.STATUS_ERROR;
DEVICE_ERROR  :  exception renames CAIS_IO_EXCEPTIONS.DEVICE_ERROR;
USE_ERROR     :  exception renames CAIS_IO_EXCEPTIONS.USE_ERROR;
```

VOLUME_STRING and FILE_STRING both have the syntax of an Ada identifier.
NAME_STRING is used for the external name of a tape, i.e., the name
written on the tape container. The file type TAPE_TYPE is used for
controlling all operations on tape drives.

MODE_ERROR is raised by an attempt to read from a file of mode OUT_FILE
or write to a file of mode IN_FILE.   STATUS_ERROR is raised if the
handle on the terminal file is not open. DEVICE_ERROR is raised if an
input or output operation cannot be completed because of a malfunction
of the underlying system.   USE_ERROR is raised if an operation is
attempted that is not possible for reasons that depend on
characteristics of the external file.

5.3.9.2 Mounting an unlabeled tape

        procedure UNLABELED_MOUNT (TAPE_DRIVE: in   TAPE_TYPE;
                                   TAPE_NAME: in   NAME_STRING);

Purpose:
  This procedure mounts a tape whose external name is TAPE_NAME on the
  drive identified by TAPE_DRIVE and sets the status of MOUNTED to
  TRUE.

  The tape is stopped at the beginning of the tape. TAPE_MARK_READ is
  set TRUE. This procedure checks for a write ring and sets
  WRITE_ENABLED accordingly. This procedure sets LABELED to FALSE.

Parameters:
  TAPE_DRIVE   is an open handle on a tape drive file.

  TAPE_NAME   is an external label which identifies the volume to
                     be mounted.

Exceptions:
  MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is
                     IN_FILE.

  STATUS_ERROR   is raised if TAPE_DRIVE is not open.

  DEVICE_ERROR   is raised if an input or output operation cannot be
                     completed because of a malfunction of the underlying
                     system.

  USE_ERROR   is raised if an attempt is made to mount a mounted
                     tape or an attempt is made to mount a tape that is
                     not yet initialized.

5.3.9.3 Mounting a labeled tape

        procedure LABELED_MOUNT (TAPE_DRIVE: in   TAPE_TYPE;
                                 VOLUME_ID: in   VOLUME_STRING;
                                 TAPE_NAME: in   NAME_STRING);

Purpose:
  This procedure mounts a labeled tape whose external name is
  TAPE_NAME on the drive identified by TAPE_DRIVE. It checks to see
  that the first block on the volume is a Volume-Header label (VOL1 -
  See Table XIV). The VOLUME_ID in the parameter list must match the
  volume identifier in the Volume-Header label on the tape.

  The tape is stopped at the beginning-of-tape reflector mark.

Parameters:
  TAPE_DRIVE   is an open handle on a tape drive file.

181

VOLUME_ID   is the name which identifies volume;it must match
                     the name in the volume header.

Exceptions:
MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is
                     IN_FILE.

STATUS_ERROR   is raised if TAPE_DRIVE is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be
                     completed because of a malfunction of the underlying
                     system.

USE_ERROR   is raised if an attempt is made to mount a mounted
                     tape, if the VOLUME_ID does not match the volume
                     identifier, if an attempt is made to mount a tape
                     that is not yet initialized, or if an attempt is made

                     to mount an unlabeled tape.

5.3.9.4 Dismounting a tape

```
procedure DISMOUNT(TAPE_DRIVE: in    TAPE_TYPE;
                   LOAD:       in    LOAD_TYPE := UNLOAD);
```

Purpose:
This procedure dismounts the tape on the drive identified by
TAPE_DRIVE and sets the status of DRIVE_READY to FALSE. If the
parameter LOAD has the value NO_UNLOAD, then the tape is left at the
beginning-of-tape reflector mark and the status of START_OF_TAPE is
set TRUE; otherwise, the tape is unloaded. Dismounting a
dismounted tape has no effect.

Parameters:
TAPE_DRIVE   is an open handle on a tape drive file.

LOAD   determines whether the tape will be unloaded or left
                     at the beginning-of-tape mark.

Exceptions:
MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is
                     IN_FILE.

STATUS_ERROR   is raised if TAPE_DRIVE is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be
                     completed because of a malfunction of the underlying
                     system.

182

5.3.9.5 Determining tape status

    function TAPE_STATUS (TAPE_DRIVE: in   TAPE_TYPE)
        return STATUS_ARRAY;

Purpose:
  This procedure obtains current tape status information. This
  procedure may be invoked while the calling process has an open
  handle on the tape drive.

Parameters:
  TAPE_DRIVE   is an open handle on a tape drive file.

Exceptions:
  MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is
                  IN_FILE.

  STATUS_ERROR   is raised if TAPE_DRIVE is not open.

  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 Skipping tape marks

    procedure SKIP_TAPE_MARKS (TAPE_DRIVE: in   TAPE_TYPE;
                               NUMBER: in   INTEGER:=1);

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, no operation is
  performed.

  Unless NUMBER is zero, the status of TAPE_MARK_READ is TRUE after
  this procedure. If two adjacent tape marks are encountered, the
  status of DOUBLE_TAPE_MARK is set TRUE and the tape is stopped
  following the second tape mark. If the end-of-tape (EOT) reflector
  mark is encountered, END_OF_TAPE is set TRUE. If the
  beginning-of-tape (BOT) reflector mark is encountered, the tape is
  stopped at the BOT reflector mark and the status of START_OF_TAPE is
  set TRUE.

Parameters:
  TAPE_DRIVE   is an open handle on a tape drive file.

  NUMBER   is the number of tape marks to skip and the direction
                  of movement.

Exceptions:
  MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is

183

3-210

IN_FILE.

STATUS_ERROR   is raised if TAPE_DRIVE is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be
                    completed because of a malfunction of the underlying
                    system.

Notes:

Nothing beyond a double tape mark is accessible.

If the status of END_OF_TAPE is set TRUE, then it remains TRUE until
the end-of-tape reflector mark is passed in the opposite (reverse)
direction.

5.3.9.7 Writing a tape mark

    procedure WRITE_TAPE_MARK (TAPE_DRIVE: in   TAPE_TYPE;
                               NUMBER: in   POSITIVE := 1);

Purpose:
    This procedure writes NUMBER consecutive tape marks on the tape
    which is mounted on the drive identified by TAPE_DRIVE. The tape is
    stopped following the last tape mark written.

    A single tape mark is written following each file except the last
    file on the tape which is followed by a double tape mark.   For the
    CAIS, a file on a magnetic tape is either a text file or a label
    group where 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.

    If a single tape mark is written the status of TAPE_MARK_READ is set
    to TRUE. If a double tape mark is written, then the status of
    TAPE_MARK_READ is set to TRUE, and the status of DOUBLE_TAPE_MARK is
    set to TRUE. If an end-of-tape (EOT) reflector mark is encountered,
    the status of END_OF_TAPE is set to TRUE.

Parameters:
    TAPE_DRIVE   is an open handle on a tape drive file.

    NUMBER   is the number of consecutive tape marks to be written

Exceptions:
    MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is
                      IN_FILE.

    STATUS_ERROR   is raised if TAPE_DRIVE is not open.

    DEVICE_ERROR   is raised if an input or output operation cannot be

184

completed because of a malfunction of the underlying system.

Notes:

The status of END_OF_TAPE remains TRUE until the BOT mark is passed again in the opposite (reverse) direction.

5.3.9.8 Initializing a tape

procedure INITIALIZE_UNLABELED(TAPE_DRIVE: in TAPE_TYPE);

Purpose:
This procedure initializes an unlabeled tape which is loaded on the drive identified by TAPE_DRIVE and sets the status of LABELED to FALSE.

If the tape is not located at the beginning-of-tape (BOT) reflector mark, then the tape is rewound to the BOT reflector mark. Two adjacent tape marks are written following the BOT reflector mark. The tape is stopped following the beginning-of-tape reflector mark. The status of TAPE_MARK_READ is set to TRUE. The status of DOUBLE_TAPE_MARK is set to TRUE. The status of START_OF_TAPE is set to FALSE.

Parameters:
TAPE_DRIVE   is an open handle on a tape drive file.

Exceptions:
MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is IN_FILE.

STATUS_ERROR   is raised if TAPE_DRIVE is not open.

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 reflector mark in front of the two tape marks written at initialization.

To recycle a tape, it must be reinitialized; initialization places the logical end of tape at the beginning of the tape.

185

5.3.9.9 Initializing a labeled tape

```
procedure INITIALIZE_LABELED ( TAPE_DRIVE: in TAPE_TYPE;
                               VOLUME_ID:        in VOLUME_STRING;
                               ACCESSIBILITY :   in VOLUME_ACCESS:=" ");
```

Purpose:
 This procedure initializes a labeled tape which is loaded on the
 drive identified by TAPE_DRIVE. A file header, two tape marks, an
 end-of-file label, and a double tape mark are written. The tape is
 stopped at the beginning-of-tape reflector mark.

 The expiration date is set to a space followed by five zeroes. The
 file name is arbitrary. The section number is 0001. The block
 count is 000000.

Parameters:
 TAPE_DRIVE   is an open handle on a tape drive file.

 VOLUME_ID   is a string identifying the volume name.

 ACCESSIBILITY   are the access rights to the volume; a space
                  indicates NO access control.

Exceptions:
 MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is
                  IN_FILE.

 STATUS_ERROR   is raised if TAPE_DRIVE is not open.

 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 created
 by this procedure will be overwritten.

 To recycle a tape, it must be reinitialized; initialization places
 the logical end of tape at the beginning of the tape.

186

5.3.9.10 Creating a volume header label

```
procedure VOLUME_HEADER (TAPE_DRIVE:    in  TAPE_TYPE;
                         ACCESSIBILITY: in  VOLUME_ACCESS := "";
                         VOLUME_ID:     in  VOLUME_STRING );
```

Purpose:
  This procedure creates a volume header, as described in Table X, for
  the volume mounted on the drive identified by TAPE_DRIVE.

TABLE X. 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

The   value   of   VOLUME_ID   may   not   be   the   empty   string.   The
accessibility character is obtained from the parameters. The   owner
identification is the key of the 'CURRENT_USER relationship of the
current process.   The Label-Standard-Version indicates the ANSI
standard version to which these labels conform.

187

Parameters:

TAPE_DRIVE   is an open handle on a tape drive file.

VOLUME_ID   identifies the volume.

ACCESSIBILITY   is a character representing the access rights to the volume.

Exceptions:

MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is IN_FILE.

STATUS_ERROR   is raised if TAPE_DRIVE is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be completed because of a malfunction of the underlying system.

USE_ERROR   is raised if tape on drive was mounted as an unlabeled tape.

5.3.9.11 Creating a file header label

```
procedure FILE_HEADER (TAPE_DRIVE:      in  TAPE_TYPE;
                       TEXT_FILE:        in  FILE_TYPE;
                       EXPIRATION_DATE:  in  STRING :=" 99366");
```

Purpose:

This procedure creates a file header, as described in Table XI, for the tape mounted on the drive identified by TAPE_DRIVE.

188

TABLE XI. 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	First VOLUME_ID in the file set
28 to 31	File Section Number	First volume of file is '0001'. For each volume after, increment by one base 10.
32 to 35	File Sequence Number	Distinguishes files in a file set. First file in set gets '0001'. For each file after, increment 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

189

A file identifier is a unique identifier generated by the CAIS. The
creation date is the time the file header label is written on the
tape. The expiration date is the date the file may be overwritten.

Parameters:

TAPE_DRIVE   is an open handle on a tape drive file.

EXPIRATION_DATE   is a string identifying the date the file may be
                  overwritten (six characters ' YYDDD' where YY is
                  the year and DDD is the day (001-366)). When
                  the expiration date is a space followed by five
                  zeroes, the file has expired.

TEXT_FILE   is the file to be written to tape.

Exceptions:

MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is
                  IN_FILE.

STATUS_ERROR   is raised if TAPE_DRIVE is not open.

DEVICE_ERROR   is raised if an input or output operation cannot be
                  completed because of a malfunction of the underlying
                  system.

USE_ERROR   is raised if tape on drive was mounted as an
                  unlabeled tape.

Notes:

When overwriting a file, first check the expiration date in the file
header label. If the existing file has expired, then back up and
overwrite the old File-Header label with the new File-Header label.
If the existing file has not expired, it may not be overwritten.

5.3.9.12 Creating an end file label

```
procedure END_FILE_LABEL(TAPE_DRIVE:      in  TAPE_TYPE;
                         TEXT_FILE:       in  FILE_TYPE;
                         EXPIRATION_DATE: in  STRING := " 99366");
```

Purpose:

This procedure creates an end file label, as shown in Table XII, for
the tape mounted on the drive identified by TAPE_DRIVE. This label
is written at the end of a complete file.

190

TABLE XII. 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 HDR1	Same as corresponding fields in HDR1
55 to 60	Block Count	Number of blocks in file
61 to 80	Same as corresponding fields in HDR1	Same as corresponding fields in HDR1

The creation date, the file identifier, and the expiration date match

the corresponding fields in the file header label.

Parameters:
TAPE_DRIVE   is an open handle on a tape drive file.

EXPIRATION_DATE is a string identifying the date the file may be
                 overwritten (six characters ' YYDDD', where YY is
the
                 year and DDD is the day (001-366). When expiration
                 date is a space followed by five zeroes, the file
has
                 expired.

TEXT_FILE   is the file to be written to tape.

Exceptions:
MODE_ERROR   is raised if the file mode of the TAPE_DRIVE is
                 IN_FILE.

STATUS_ERROR   is raised if TAPE_DRIVE is not open.

DEVICE_ERROR   is raised if an input-output operation cannot be
                 completed because of a malfunction of the underlying
                 system.

USE_ERROR   is raised if tape on drive was mounted as an
                 unlabeled tape.

191

5.4 CAIS Utilities

This section defines the abstract data type LIST_TYPE.

5.4.1 Package CAIS_LIST_UTILITIES

This package 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 items 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. A null list is a list which contains null items. Such a list is neither of a named or unnamed kind. A null list can be obtained by using the NULL_LIST function or 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, integer and real numbers, enumeration values, and lists. The kind of an item is a value of the enumeration type ITEM_KIND.

The specifications within this package allow for the manipulation of lists which are either of unnamed or named 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.

The value of an entity of LIST_TYPE can be externally represented 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.

The BNF for a list's external representation is given in Table XIII.

192

TABLE XIII   List External Representation BNF

```
list_type ::= named_list | list
named_list ::= ( [ named_item { , named_item } ] )
positional_list ::= ( [ item { , item } ] )
named_item ::= name_string => item
item ::= list_type
        | quoted_string
        | integer_nr
        | real_nr
        | identifier
integer_nr ::= integer_literal
real_nr ::= real_literal
quoted_string ::= " { letter_or_digit | "" } "
identifier ::= ada_identifier
name_string ::= ada_identfier
```

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

5.4.1.1 Types, subtypes, constants, and exceptions -

```
type     LIST_TYPE       is limited private;
type     LIST_KIND       is (UNNAMED, NAMED, NULL);
type     ITEM_KIND       is (LIST_ITEM, STRING_ITEM, INTEGER_ITEM,
                             REAL_ITEM, IDENTIFIER_ITEM);
subtype LIST_TEXT        is STRING;
subtype ELEMENT_TEXT     is STRING;
subtype NAME_STRING      is STRING;
type    COUNT            is range 0 .. implementation_defined;
subtype POSITION_COUNT is COUNT range COUNT'FIRST + 1 .. COUNT'LAST;
```

LIST_KIND enumerates the unnamed and named kinds of lists. ITEM_KIND enumerates
the classifications of list items. LIST_TEXT is the type of a list's external
representation. ITEM_TEXT is the type of a list-item's external representation.
NAME_STRING is the type of an item's name in a named item.

193

3-220

SEARCH_ERROR : exception;

The exception SEARCH_ERROR is raised if a search for a named item fails or if an item's position falls outside the range of the list's length.

5.4.1.2 Establishing a null-list

  function NULL_LIST return LIST_TYPE;

Purpose:
  This function returns a null list.

Parameters:
  None.

Exceptions:
  None.

5.4.1.3 Converting from an external list representation

  function TO_LIST (LIST_LITERAL: in LIST_TEXT) return LIST_TYPE;

Purpose:

  This function converts the external representation of a list to LIST_TYPE and returns that converted value.   This function establishes the list to be of a named, unnamed or null kind.
Parameters:

  LIST_LITERAL   is the external representation of the list.

Exceptions:
  USE_ERROR   is raised if LIST_LITERAL does not conform to
                        the syntax as specified in Table XVII.

5.4.1.4 Converting to external list representation

  function TO_TEXT (LIST: in LIST_TYPE) return LIST_TEXT;

  procedure TO_TEXT (LIST: in LIST_TYPE;
                     LIST_LITERAL: out LIST_TEXT;
                     LIST_RANGE: out natural);

Purpose:
  This function/procedure converts a list to its external representation.

194

Parameters:
    LIST   is the list to be converted.
    LIST_LITERAL   is the external representation of LIST.
    LIST_RANGE   is the length of the string returned in
                       LIST_LITERAL.

Exceptions:
    CONSTRAINT_ERROR is raised if the length of the string resulting
                       from the conversion exceeds the size of
                       LIST_LITERAL.

5.4.1.5 Inserting an item into a list

```
                    LIST_ELEMENT: in ELEMENT_TEXT;
                    POSITION: in COUNT;
                    IS_STRING: in BOOLEAN:=FALSE);


  procedure INSERT (LIST  : in out LIST_TYPE;
                    LIST_ELEMENT: in ELEMENT_TEXT;
                    NAMED: in NAME_STRING;
                    POSITION: in COUNT;
                    IS_STRING: in BOOLEAN:=FALSE);
```

Purpose:
This procedure inserts an item into a list after the list item
specified by POSITION. A value of zero (=0) in POSITION specifies a
position at the head of the list. The position order of the items
in the list following the inserted item will assume new ordinal
values starting with the value POSITION +1. An insertion of a named
list item into an empty list will determine that list to be of named
kind from then on. Conversely, an insertion of a unnamed list item
will determine that list to be of unnamed kind.

Parameters:
    LIST   is the list into which the item will be inserted.
    LIST_ELEMENT   is the list item to be inserted.
    NAMED   is the name of the new item. It may only be used
                       with named lists.
    POSITION   is the position specification.
    IS_STRING   allows a list element of STRING_ITEM kind to be
                       expressed as a non-quoted string if set TRUE.

Exceptions:
    SEARCH_ERROR   is raised if POSITION specifies a value larger
                       than the (existing) length of the list.
    USE_ERROR   is raised if an attempt is made to insert a named
                       list item into an unnamed list or conversely, an
                       attempt is made to insert an unnamed list item
                       into a named list.

195

5.4.1.6 Resetting the value of a named item

```
procedure RESET      (LIST:          in out LIST_TYPE;
                      POSITION:       in POSITION_COUNT;
                      LIST_ELEMENT: in ELEMENT_TEXT);

procedure RESET      (LIST : in out LIST_TYPE;
                      NAMED : in    NAME_STRING;
                      LIST_ELEMENT: in ELEMENT_TEXT);
```

Purpose:

  This procedure replaces an item in the specified list. The new item
  must be of the same item kind as that it replaces.

Parameters:

LIST	is the list containing the item to be replaced.
POSITION	is the position within the list to identify the item to be replaced.
NAMED	is the name of the item to be replaced.
LIST_ELEMENT	is the new item.

Exceptions:

USE_ERROR	is raised if LIST_ELEMENT does not replace a list item of the same item kind.
SEARCH_ERROR	is raised if there is no item with the NAMED component or if POSITION has a value larger than the (existing) length of the list.

5.4.1.7 Extracting an item

```
procedure EXTRACT    (LIST :          in    LIST_TYPE;
                      POSITION:       in    POSITION_COUNT;
                      LIST_ELEMENT:      out ELEMENT_TEXT;
                      ITEM_RANGE:        out POSITIVE);

procedure EXTRACT    (LIST:           in    LIST_TYPE;
                      NAMED:          in    NAME_STRING;
                      LIST_ELEMENT:      out ELEMENT_TEXT;
                      ITEM_RANGE:        out POSITIVE);

function EXTRACT (LIST : in LIST_TYPE;
                  POSITION : in POSITION_COUNT)
                                        return ELEMENT_TEXT;

function EXTRACT (LIST  : in LIST_TYPE;
                  NAMED : in NAME_STRING) return ELEMENT_TEXT;
```

Purpose:

  This procedure returns the list item in LIST_ELEMENT as specified by
  POSITION or NAMED without removing the item from the list.

196

The function counterparts simply return the list item.

Parameters:

LIST	is the list containing the item.
POSITION	is the position within the list that identifies the item to be extracted.
NAMED	is the name of the item to be extracted.
LIST_ELEMENT	is the item read.
ITEM_RANGE	is the length of the string returned in LIST_ELEMENT.

Exceptions:

USE_ERROR	is raised if LIST is empty.
CONSTRAINT_ERROR	is raised if the length of the string in the returned item exceeds the size of LIST_ELEMENT.
SEARCH_ERROR	is raised if there is no item with the NAMED component or if POSITION has a value larger than the (existing) length of the LIST.

5.4.1.8 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);
```

Purpose:

This procedure deletes the list item specified by POSITION or NAMED from LIST.

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:

SEARCH_ERROR	is raised if there is no item with the NAMED component or if POSITION has a value larger than the (existing) length of LIST.
USE_ERROR	is raised if the parameter NAMED is used with an unnamed list.

197

3-224

5.4.1.9 Determining the kind of list or the kind of list item

    function KIND (LIST : in LIST_TYPE) return LIST_KIND;

    function KIND (LIST: in LIST_TYPE;
                   POSITION: in POSITION_COUNT) return ITEM_KIND;

    function KIND (LIST: in LIST_TYPE;
                   NAMED: in NAME_STRING) return ITEM_KIND;

Purpose:

  This function returns either the kind of list or the kind of item in
  the referenced 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:
    SEARCH_ERROR   is raised if there is no item with the NAMED
                      component or if POSITION has a value larger than
                      the (existing) length of 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 from the lists
  FRONT and BACK. The lists FRONT and BACK must be of the same kind.

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.

Exceptions:
    USE_ERROR   is raised if FRONT and BACK are not of the same
                      kind.

198

5.4.1.11 Determining the length of the 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.

Parameters:
  LIST   is the list of interest.

Exceptions:
  None.

5.4.1.12 Determining the name of a named item

    procedure ITEM_NAME (LIST :   in   LIST_TYPE
                         POSITION: in   POSITION_COUNT;
                         NAMED: out NAME_STRING;
                         NAME_RANGE: out POSITIVE);

    function ITEM_NAME (LIST :   in LIST_TYPE;
                        POSITION : in POSITION_COUNT)
                                             return NAME_STRING;
Purpose:

  This procedure/function returns the name of the list item in a named
  list, specified by POSITION.

Parameters:

    LIST   is the list of interest.
    POSITION   is the position within the list that identifies
                      the item.
    NAMED   is the string representation of the list item name.
    NAME_RANGE   is the length of the string returned in NAMED.

Exceptions:
    SEARCH_ERROR   is raised if POSITION has a value larger than the
                      (existing) length of LIST.
    USE_ERROR   is raised if LIST is not a named list.

199

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;

Purpose:
  This function returns the length of a string representing either a
  list or the list item identified by POSITION in a list.

Parameters:
  LIST   is the list of interest.
  POSITION   is the position within the list that identifies the
              item.

Exceptions:
  SEARCH_ERROR is raised if POSITION has a value larger than the
               (existing) length of LIST.

200

APPENDIX A

Predefined Relations
Relation Names and Attributes

Predefined Relations:

        ACCESS
        ADOPTED_ROLE
        ASSOCIATE
        CURRENT_ERROR
        CURRENT_INPUT
        CURRENT_JOB
        CURRENT_NODE
        CURRENT_OUTPUT
        CURRENT_USER
        DEVICE
        JOB
        ROLE
        PARENT
        PERMANENT_MEMBER
        POTENTIAL_MEMBER
        STANDARD_INPUT
        STANDARD_OUTPUT
        STANDARD_ERROR
        USER

Predefined Attributes:

        ACCESS_METHOD
        CURRENT_STATUS
        FILE_KIND
        FINISH_TIME
        GRANT
        HANDLES_OPEN
        HIGHEST_CLASSIFICATION
        IO_UNITS
        LOWEST_CLASSIFICATION
        MACHINE_TIME

201

OBJECT_CLASSIFICATION
PARAMETER_LIST
QUEUE_TYPE
RESULTS_LIST
SIZE
START_TIME
SUBJECT_CLASSIFICATION
TERMINAL_TYPE

Predefined Attribute values:

CONTROL
COPY
DIRECT
EXECUTE
EXISTENCE
FORM
MAGNETIC_TAPE
MIMIC
PAGE
QUEUE
READ
READ_ATTRIBUTES
READ_CONTENTS
READ_RELATIONSHIPS
SCROLL
SECONDARY_STORAGE
SEQUENTIAL
SOLO
TERMINAL
TEXT
WRITE
WRITE_ATTRIBUTES
WRITE_CONTENTS
WRITE_RELATIONSHIPS

202

APPENDIX B

CAIS SPECIFICATION

This appendix contains a set of Ada package spcifications of the CAIS interfaces which compiles correctly. It brings togethr most of 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.

(To be supplied in January 1985 Military Standard.)

INDEX

284

205

208

209

210

3-238

212

213

214

215

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

216

217

218

219

Postscript : Submission of Comments

For submission of comments on this CAIS Version 1.4, we would appreciate them being sent by Arpanet to the address

    CAIS-COMMENT at ECLB

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

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

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

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

        !section ...
        !version CAIS 1.4
        !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). The paragraph number is the one given in the margin of the paper form of this document (it is not contained in the ECLB files). As an example, here is the section line of a comment from a previous version:

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

The version line, for comments on the current document, should only contain "!version CAIS 1.4 ". 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)D.Taffs 82-04-26
!version CAIS 1.4
!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	OMB Approval No. 22-R255

INSTRUCTIONS: The purpose of this form is to solicit beneficial comments which will help achieve procurement of suitable products at reasonable cost and minimum delay, or will otherwise enhance use of the document. DoD contractors, government activities, or manufacturers/vendors who are prospective suppliers of the product are invited to submit comments to the government. Fold on lines on reverse side, staple in corner, and send to preparing activity. Comments submitted on this form do not constitute or imply authorization to waive any portion of the referenced document(s) or to amend contractual requirements. Attach any pertinent data which may be of use in improving this document. If there are additional papers, attach to form and place both in an envelope addressed to preparing activity.

DOCUMENT IDENTIFIER AND TITLE

NAME OF ORGANIZATION AND ADDRESS	CONTRACT NUMBER
	MATERIAL PROCURED UNDER A
	☐ DIRECT GOVERNMENT CONTRACT ☐ SUBCONTRACT

1. HAS ANY PART OF THE DOCUMENT CREATED PROBLEMS OR REQUIRED INTERPRETATION IN PROCUREMENT USE?
   A. GIVE PARAGRAPH NUMBER AND WORDING.

   B. RECOMMENDATIONS FOR CORRECTING THE DEFICIENCIES

2. COMMENTS ON ANY DOCUMENT REQUIREMENT CONSIDERED TOO RIGID

3. IS THE DOCUMENT RESTRICTIVE?

   ☐ YES   ☐ NO (If "Yes", in what way?)

4. REMARKS

SUBMITTED BY (Printed or typed name and address - Optional)	TELEPHONE NO.
	DATE

DD FORM 1 JAN 72 **1426**   REPLACES EDITION OF 1 JAN 68 WHICH MAY BE USED

3-249

Defense Electronics Supply Center
Directorate of Engineering Standardization, DESC-E
Dayton, Ohio 45444

POSTAGE AND FEES PAID
DEFENSE SUPPLY AGENCY
DoD-304

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE $300

Defense Electronics Supply Center
Directorate of Engineering Standardization, DESC-E
Dayton, Ohio 45444

DoD

Requirements

and

Design Criteria

for the Common APSE Interface Set (CAIS)

October 1984

PREFACE

The KAPSE Interface Team (KIT), and its companion Industry-Academia team (KITIA), were formed by a Memorandum of Agreement (MOA) signed by the three services and the Undersecretary of Defense in January, 1982. Their purpose is to contribute to the achievement of Interoperability of applications databases and Transportability of software development tools ("I&T"). These are important economic objectives, identified at the outset of the DoD common language initiative in the mid-1970's and now acknowledged to require an integrated Ada Programming Support Environment (APSE), in addition to the standard language Ada, for fulfillment. The core of the KIT/KITIA strategy to fulfill I&T objectives is to define a standard set of Ada Programming Support Environment (APSE) interfaces ("CAIS" for "Common APSE Interface Set") to which all Ada-related tools can be written, thus assuring the ability to share tools and databases between conforming Ada Programming Support Environments (APSEs). Note that a large number of these interfaces are at the Kernel APSE (KAPSE) level. This document establishes requirements and design objectives (called "criteria") on the definition of a CAIS.

This document is related to the DoD "Stoneman" Requirements for Ada Programming Support Environments in identifying and refining the derived requirements which are imposed upon a CAIS and which effect the I&T-related objectives. Additional influences on this document were the DoD "Steelman" Requirements for High Order Computer Programming Languages and the several sets of ANSI "OSCRL" requirements and design objectives for Operating System Command and Response Languages.

Table of Contents

i

1. INTRODUCTION

1.1 Scope. This document provides the Department of Defense's requirements and design criteria for the definition and specification of a Common APSE Interface Set (CAIS) for Ada Programming Support Environments (APSEs).

1.2 Terminology. The precise and consistent use of terms has been attempted throughout the document.

Many potentially ambiguous terms have been used in the document. Most are defined in the Glossary of KIT/KITIA terminology. Some are defined in the sections where they are used with definitions tailored to the context of this document.

Additionally, the following verbs and verb phrases have been used consistently throughout the document to indicate where and to what degree individual constraints apply. Any sentence not containing one of the following verbs or verb phrases is a definition, explanation or comment.

> "shall"   indicates a requirement on the definition
>   of the CAIS; sometimes "shall" is followed
>   by "provide" or "support," in which cases
>   the following two definitions supercede
>   this one.
>
> "shall provide" indicates a requirement for the CAIS to
>   provide interface(s) with prescribed
>   capabilities.
>
> "shall support" indicates a requirement for the CAIS to
>   provide interface(s) with prescribed
>   capabilities or for CAIS definers to
>   demonstrate that the capability may be
>   constructed from CAIS interfaces.
>
> "should"   indicates a desired goal but one for which
>   there is no objective test.

1.3 Relationship to CAIS Specifications and Implementations. This document specifies functional capabilities which are to be provided in the semantics of a CAIS specification and are therefore to be provided by conforming CAIS implementations. In general, the specifications of software fulfilling those capabilities (and decisions about including or not including CAIS interfaces for certain capabilities as suggested by the "shall support" definition in the previous section) are delegated to the CAIS definers. If

a particular facility specified in the CAIS is independent of other CAIS facilities, then a CAIS implementor may elect to reuse CAIS facilities to provide the particular specified facility, thereby achieving a "layered implementation" of the CAIS. Therefore, the realization of a specific CAIS implementation is the result of intentionally divided decision-making authority among 1) this requirements document, 2) CAIS definers, and 3) CAIS implementors.

2. GENERAL DESIGN OBJECTIVES

2.1 Scope of the CAIS. The CAIS shall consist of the interfaces necessary and sufficient to support the use of APSEs throughout the lifecycle, and to promote I&T among APSEs. The CAIS should be broad enough to support wide sets of tools and classes of projects. The CAIS is not required to provide all general operating system capabilities.

2.2 Basic Services. The CAIS should provide simple-to-use mechanisms for achieving common, simple actions. Features which support less frequently used tool needs should be given secondary consideration.

2.3 Implementability. The CAIS specification shall be machine independent and implementation independent. The CAIS shall be implementable on bare machines and on machines with any of a variety of operating systems. The CAIS shall contain only interfaces which provide facilities which have been demonstrated in existing operating systems, kernels, or command processors. CAIS features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for unneeded generality, and to ensure that unused portions of a CAIS implementation will not add to execution costs of a non-using tool. The measures of the efficiency criterion are, primarily, minimum interactive response time for APSE tools and, secondarily, consumption of tool-chargeable resources.

2.4 Modularity. Interfaces should be designed in a modular fashion such that they may be understood in isolation and such that there are no hidden interactions between interfaces. This permits a tool writer to employ a subset of the CAIS.

2.5 Extensibility. The design of the CAIS should facilitate development and use of portable extensions of the CAIS; i.e., CAIS interfaces should be reusable so that they can be combined to create new interfaces and facilities which are also portable.

2.6 Technology Compatibility. The CAIS shall adopt existing standards where applicable. For example, recognized standards for device characteristics are provided by ANSI, ISO, IEEE, and DoD.

2.7 Consistency.   The design of the CAIS should minimize the number of underlying concepts. It should have few special cases and should consist of features that are individually simple. These objectives are not to be pursued to the extreme of providing inconvenient mechanisms for the expression of some common, reasonable actions.

2.8 Security.   The CAIS shall be implementable as a secure system that fulfills the requirements for a Class (B2) system in the DoD document titled *Trusted Computer System Evaluation Criteria.* The CAIS shall be designed to mediate all tool access to underlying system services (i.e., no *by-passing* the conforming CAIS implementation is necessary to implement any APSE function). The CAIS should accommodate implementations that coexist with (without compromising) and operate within a variety of security mechanisms.

3. GENERAL SYNTAX AND SEMANTICS

3.1 Syntax

3.1A General Syntax.   The syntax of the CAIS shall be expressed as Ada package specifications. The syntax of the CAIS shall conform to the character set as defined by the Ada standard (section 2.1 of ANSI/MIL-STD-1815A).

3.1B Uniformity.   The CAIS should employ uniform syntactic conventions and should not provide several notations for the same concept. CAIS syntax issues (including, at least, limits on name lengths, abbreviation styles, other naming conventions, relative ordering of input and output parameters, etc.) should be resolved in a uniform and integrated manner for the whole CAIS.

3.1C Name Selection.   The CAIS should avoid coining new words (literals or identifiers) and should avoid using words in an unconventional sense. Ada identifiers (names) defined by the CAIS should be natural language words or industry accepted terms whenever possible. The CAIS should define Ada identifiers which are visually distinct and not easily confused (including, at least, that the CAIS should avoid defining two Ada identifiers that are only a 2-character transposition away from being identical). The CAIS should use the same name everywhere in the interface set, and not its possible synonyms, when the same meaning is intended.

3.1D Pragmatics.   The CAIS should impose only those restrictive rules, constraints, or anomalies required to achieve I&T. The CAIS specification shall enumerate all instances of syntactic constraint setting which are deferred to the implementor. CAIS implementors will be required to provide the complete specifications of all syntactic restrictions imposed by their CAIS implementations.

3.2 Semantics

3.2A General Semantics.   The CAIS shall be completely and unambiguously defined. The specification of semantics should be both precise and understandable. The semantic specification of each CAIS interface shall include precise statement of assumption (including execution-time preconditions for calls)s, effects on global data and packages, and interactions with other interfaces.

3-1

3.2B   Responses.   The CAIS shall provide standard responses for all interfaces, including a unique, non-null response (return value or exception) for each type of unsuccessful completion. All responses returned across CAIS interfaces shall be defined in an implementation-independent manner. Everytime a CAIS interfaces is called under the same circumstances, it should return the same response.

3.2C   Exceptions.   All named exceptions raised and propagated by the CAIS shall be documented. The CAIS specification shall require CAIS implementations to provide handlers for all unnamed exceptions raised in the implementations' bodies.

3.2D   Consistency.   The description of CAIS semantics should use the same word or phrase everywhere, and not its possible synonyms, when the same meaning is intended.

3.2E   Cohesiveness.   Each CAIS interface should provide only one function.

3.2F   Pragmatics.   The CAIS specification shall enumerate all aspects of the meanings of CAIS interfaces and facilities which must be defined by CAIS implementors. CAIS implementors will be required to provide the complete specifications for these implementation-defined semantics.

4. ENTITY MANAGEMENT SUPPORT

This characterization of Entity Management Support is based on the STONEMAN requirements for a database, using a model based on the entity-relationship concept. Although a CAIS design meeting these requirements is expected to demonstrate the characteristics and capabilities reflected here, it is not necessary that such a design directly employ this entity-relationship model.

The general capabilities required in the model specified by the CAIS are the following. The entity-relationship model, for which definitions and requirements follow in 4.1 - 4.7 provides these capabilities, and any alternative model of CAIS requirements must also.

a. There must be a means for retaining data.

b. There must be a way for retaining relationships and properties of data.

c. There must be a way of operating upon data and creating new data (and deleting data).

d. There must be a means to determine whether the properties of an item of data are valid and whether operations upon it are valid.

e. There must be a way to restrict operations on an item of data to valid ones.

f. There must be a description of each item of data and that description may be operated upon.

g. The relationships and properties of data must be separate from their existence and separate from the tools that operate upon them.

h. There must be a way to develop new data by inheriting (some of) the properties of existing data.

4.1 Entities, Relationships, and Attributes   The following definitions pertain specifically to this section:

ENTITY   A representation of a person, place, event or thing.

RELATIONSHIP An ordered connection or association among entities. A relationship among N entities (not necessarily distinct) is known as an "N-ary" relationship.

ATTRIBUTE   An association of an entity or relationship with an elementary value

4-1

ELEMENTARY VALUE
> One of two kinds of representations of data: interpreted and uninterpreted.

INTERPRETED DATA
> A data representation whose structure is controlled by CAIS facilities and may be used in the CAIS operations. Examples are representations of integer, string, real, data and enumeration data, and aggregates of such data.

UNINTERPRETED DATA
> A data representation whose structure is not controlled by CAIS facilities and is not used in the CAIS operations. Examples might be representations of files, such as requirements documents, program source code, and program object code.

4.1A Data. The CAIS shall provide facilities for representing data using entities, attributes or binary relationships. The CAIS may provide facilities for more general N-ary relationships, but it is not required to do so.

4.1B Elementary Values. The CAIS shall provide facilities for representing data as elementary values.

4.1C System Integrity. The CAIS facilities shall ensure the integrity of the CAIS-managed data.

4.2 Typing The following definition pertains specifically to this section:

TYPING
> An organization of entities, relationships and attributes in which they are partitioned into sets, called entity types, relationship types and attribute types, according to designated type definitions.

4.2A Types. The facilities provided by the CAIS shall enforce typing by providing that all operations conform to the type definitions. Every entity, relationship and attribute shall have one and only one type.

4.2B Rules about Type Definitions. The CAIS type definitions shall

- specify the entity types and relationship types to which each attribute type may apply

- specify the type or types of entities that each relationship type may connect and the attribute types allowed for each relationship type

- specify the set of allowable elementary values for each attribute type

- specify the relationship types and attribute types for each entity type

- permit relationship types that represent either functional mappings (one-to-one or many-to-one) or relational mappings (one-to-many or many-to-many)

● permit multiple distinct relationships among the same entities

● impose a lattice structure on the types which includes inheritance of attributes, attribute value ranges (possibly restricted), relationships and allowed operations.

4.2C Type Definition.   The CAIS shall provide facilities for defining new entity, relationship and attribute types.

4.2D Changing Type Definitions.   The CAIS shall provide facilities for changing type definitions. These facilities shall be controlled such that data integrity is maintained.

4.2E Triggering.   The CAIS shall provide a conditional triggering mechanism so that prespecified procedures or operations (such as special validation techniques employing multiple attribute value checking) may be invoked whenever values of indicated attributes change.   The CAIS shall provide facilities for defining such triggers and the operations or procedures which are to be invoked.

4.3 Identification   The following definitions pertain specifically to this section:

EXACT IDENTITY
>   A designation of an entity (or relationship) that is always associated with the entity (or relationship) that it designates. This exact identity will always designate exactly the same entity (or relationship), and it cannot be changed.

IDENTIFICATION
>   A means of specifying the entities, relationships and attributes to be operated on by a designated operation.

4.3A Exact Identities.   The CAIS shall provide exact identities for all entities. The CAIS shall support exact identities for all relationships. The exact identify shall be unique within an instance of a CAIS implementation, and the CAIS shall support a mechanism for the utilization of exact identities across all CAIS implementations.

4.3B Identification.   The CAIS shall provide identification of all entities, attributes and relationships The CAIS shall provide identification of all entities by their exact identify. The CAIS shall support identification of all relationships by their exact identity.

4-3

4.3C Identification Methods.   The CAIS shall provide identification of entities and relationships by at least the following methods:

- identification of some "start" entity(s), the specification of some relationship type and the specification of some predicate involving attributes or attribute types associated with that relationship type or with some entity type. This method shall identify those entities which are related to the identified start entity(s) by relationships of the given relationships type and for which the predicate is true. Subject to the security constraints of section 2.8, all relationships and entities shall be capable of identification via this method, and all attributes and attribute types (except uninterpreted data) shall be permitted in the predicates.

- identification of an entity type or relationship type and specification of some predicate on the value of any attribute of the entity type or relationship type. This method shall identify those entities or relationships of the given type for which the predicate is true. Subject to the security constraints of section 2.8, all attributes (except uninterpreted data) shall be permitted in the predicates.

4.4 Operations

4.4A Entity Operations.   The CAIS shall provide facilities to:

- create entities

- delete entities

- examine entities (by examining their attributes and relationships)

- modify entities (by modifying their attributes)

- identify entities (as specified in Section 4.3)

4.4B Relationship Operations.   The CAIS shall provide facilities to:

- create relationships

- delete relationships

- examine relationships (by examining their attributes)

- modify relationships (by modifying their attributes)

- identify relationships (as specified in Section 4.3)

4.4C Attribute Operations.   The CAIS shall provide facilities to:

- examine attributes

- modify attributes

4.4D Exact Identify Operations. The CAIS shall provide facilities to:

- pass exact identities between processes

- compare exact identities

4.4E Uninterpreted Data Operations. The CAIS shall provide that use of the input-output facilities of the Ada language (as defined in Chapter 14 of ANSI/MIL-STD-1815A) results in reading/writing an uninterpreted data attribute of an entity. The facilities of Section 6 shall then apply.

4.4F Synchronization. The CAIS shall provide dynamic access synchronization mechanisms to individual entities, relationships and attributes.

4.5 Transaction. The following definition pertains specifically to this section:

TRANSACTION A grouping of operations, including a designated sequence of operations, which requires that either all of the designated operations are applied or none are; e.g., a transaction is uninterruptible from the user's point of view.

4.5A Transaction Mechanism. The CAIS shall support a transaction mechanism. The effect of running transactions concurrently shall be as if the concurrent transactions were run serially.

4.5B Transaction Control. The CAIS shall support facilities to start, end and abort transactions. When a transaction is aborted, all effects of the designated sequence of operations shall be as if the sequence was never started.

4.5C System Failure. System failure while a transaction is in progress shall cause the effects of the designated sequence of operations to be as if the sequence was never started.

4.6 History. The following definition pertains specifically to this section:

HISTORY   A recording of the manner in which entities, relationships and attribute values were produced and of all information which was relevant in the production of those entities, relationships or attribute values.

4.6A History Mechanism. The CAIS shall support a mechanism for collecting and utilizing history. The history mechanism shall provide sufficient information to support comprehensive configuration control.

4.6B History Integrity.   The CAIS shall support mechanisms for ensuring the fidelity of the history.

4.7 Robustness and Restoration.   The following definitions pertain specifically to this section:

BACKUP   A redundant copy of some subset of the CAIS-managed data. The subset is capable of restoration to active use by a CAIS implementation, particularly in the event of a loss of completeness or integrity in the data in use by implementation.

ARCHIVE   A subset of the CAIS-managed data that has been relegated to backing storage media while retaining the integrity, consistency and availability of all information in the entity management system.

4.7A Robustness and Restoration.   The CAIS shall support facilities which ensure the robustness of and ability to restore CAIS-managed data. The facilities shall include at least those required to support the backup and archiving capabilities provided by modern operating systems.

5. PROGRAM EXECUTION FACILITIES

Access controls and security rights will apply to all CAIS facilities required by this section.

The following definitions pertain specifically to this section:

PROCESS   The CAIS facility used to represent the execution of any program.

PROGRAM   A set of compilation units, one of which is a subprogram called the *main program.* Execution of the program consists of execution of the main program, which may invoke subprograms declared in the compilation units of the program.

RESOURCE   Any capacity which must be scheduled, assigned, or controlled by the operating system to assure consistent and non-conflicting usage by programs under execution. Examples of resources include: CPU time, memory space (actuals and virtual), and shared facilities (variables, devices, spoolers, etc.).

ACTIVATE   To create a CAIS process. The activation of a program binds that program to its execution environment, which are the resources required to support the process's execution, and includes the program to be executed. The activation of a process marks the earliest point in time which that process can be referenced as an entity within the CAIS environment.

TERMINATE   To stop the execution of a process such that it cannot be resumed.

DEACTIVATE   To remove a terminated process so that it may no longer be referenced within the CAIS environment.

SUSPEND   To stop the execution of a process such that it can resumed. In the context of an Ada program being executed, this implies the suspension of all tasks, and the prevention of the activation of any task until the process is resumed. It specifically does not imply the release of any resources which a process has assigned to it, or which it has acquired, to support its execution.

RESUME   To resume the execution of a suspended process.

TASK WAIT   The execution of a task within a process is delayed until a CAIS service requested by this task has been performed. Other tasks in the same process are not delayed.

5.1 Activation of Program

5.1A Activation.   The CAIS shall provide a facility for a process to create a process for a program that has been made ready for execution. This event is called activation.

5.1B   Unambiguous Identification.   The CAIS shall provide facilities for the unambiguous identification of a process at any time between its activation and deactivation; one such capability shall be as an indivisible part of activation. This act of identification establishes a reference to that process. Once such a reference is established, that reference will refer to the same process until the reference is dissolved. A reference is always dissolved upon termination of the process that established the reference. A terminated process may not be deactivated while there are references to that process.

5.1C Activation Data.   The CAIS shall provide a facility to make data available to a program upon its activation.

5.1D Dependent Activation.   The CAIS shall provide a facility for the activation of programs that depend upon the activating process for their existence.

5.1E Independent Activation.   The CAIS shall provide a facility for the activation of programs that do not depend upon the activating process for their existence.

5.2 Termination

5.2A Termination.   The CAIS shall provide a facility for a process to terminate a process. There shall be two forms of termination; the voluntary termination of a process (termed completion) and the abnormal termination of a process. Completion of a process is always self-determined, whereas abnormal termination may be initiated by other processes.

5.2B Termination of Dependent Processes.   The CAIS shall support clear, consistent rules defining the termination behavior of processes dependent on a terminating process.

5.2C Termination Data.   The CAIS shall provide a facility for termination data to be made available. This data shall provide at least an indication of success or failure for processes that complete. For processes that terminate abnormally the termination data shall indicate abnormal termination.

5.3 Communication

5.3A Data Exchange. The CAIS shall provide a facility for the exchange of data among processes.

5.4 Synchronization

5.4A Task Waiting. The CAIS shall support task waiting.

5.4B Parallel Execution. The CAIS shall provide for the parallel execution of processes.

5.4C Synchronization. The CAIS shall provide a facility for the synchronization of cooperating processes.

5.4D Suspension. The CAIS shall provide a facility for suspending a process.

5.4E Resumption. The CAIS shall provide a facility to resume a process that has been suspended.

5.5 Monitoring

5.5A Identify Reference. The CAIS shall provide a facility for a process to determine an unambiguous identity of a process and to reference that process using that identity.

5.5B RTS Independence. CAIS program execution facilities shall be designed to require no additional functionality in the Ada Run-Time System (RTS) from that provided by Ada semantics. Consequently, the implementation of the Ada RTS shall be independent of the CAIS.

5.5C Instrumentation. The CAIS shall provide a facility for a process to inspect and modify the execution environment of another process. This facility is intended to promote support for portable debuggers and other instrumentation tools.

6. INPUT/OUTPUT

The requirements specified in this section pertain to input/output between/among objects (e.g. processes. data entities, communication devices, and storage devices) unless otherwise stated. All facilities specified in the following requirements are to be available to non-privileged processes, unless otherwise specified.

The following definitions pertain specifically to this section:

DATA UNIT   a representation of a value of an Ada discrete type.

DATAPATH   the mechanism by which data units are transmitted from a producer to a consumer.

DATASTREAM   the data units flowing from a producer to a consumer (without regard to the implementing mechanism).

CONSUMER   an entity that is receiving data units via a datapath.

PRODUCER   an entity that is transmitting data units via a datapath.

TYPE-AHEAD   the ability of a producer to transmit data units before the consumer requests the data units

6.1 General Input/Output

a. Waiting. The CAIS shall cause only the task requesting a synchronous input/output operation to await completion.

6.2 Virtual I/O Drivers

6.2A Data Unit Transmission

a. Hardcopy terminals. The CAIS shall provide interfaces for the control of hardcopy terminals.

b. Page terminals. The CAIS shall provide interfaces for the control of page terminals. A page terminal transmits/receives one data unit at a time.

c. Printers. The CAIS shall provide interfaces for the control of character-imaging printers and bit-map printers.

d. Paper tape drives. The CAIS shall provide interfaces for the control of paper tape drives.

e. Graphics support. The CAIS shall support the control of interactive graphical input/output devices.

6.2B Data Block Transmission

a. Block terminals. The CAIS shall provide interfaces for the control of character-imaging block terminals. A block terminal transmits/receives a block of data units at a time.

b. Tape drives. The CAIS shall provide interfaces for the control of magnetic tape drives.

6.3 Datapath Control

a. Interface level. The datapath control facilities of the CAIS shall be provided at a level comparable to that of Ada LRM File I/O. That is, control of datapaths shall be provided via subprogram calls rather than via the data units transmitted to the device.

b. Timeout. The CAIS shall provide facilities to permit timeout on input and output operations.

c. Exclusive access. The CAIS shall provide facilities to obtain exclusive access to a producer/consumer; such exclusive access does not prevent a privileged process from transmitting to the consumer.

6.3A Data Unit Transmission

a. Data unit size. The CAIS shall provide input/output facilities for communication with devices requiring 5-bit, 7-bit, and 8-bit data units, minimally.

b. Raw input/output. The CAIS shall provide the ability to transmit/receive data units and data unit sequences without modification (e.g. transformation of units, addition of units, removal of units).

c. Single data unit transmission. The CAIS shall provide facilities for the input/output of single data units. The completion of this operation makes the data unit available to its consumer(s) without requiring another input/output event, including the receipt of a termination or escape sequence, the filling of a buffer, or the invocation of an operation to force input/output.

d. Datapath buffer size. The CAIS shall provide facilities for the specification of the sizes of input/output data path buffers during process execution.

e. Datapath flushing. The CAIS shall provide facilities for the removal of all buffered data from an input/output datapath.

f. Output datapath processing. The CAIS shall provide facilities to force the output of all data in an output datapath.

g. Input/output sequencing. The CAIS shall provide facilities to ensure the servicing of input/output requests in the order of their invocation.

h. Padding. The CAIS shall specify the set of data units and data unit sequences (including the null set) which can be added to an input/output datastream. The CAIS shall provide facilities permitting a process to select/query at execution time the subset of data units and data unit sequences which may be added (including the null set).

i. Filtering. The CAIS shall specify the set of data units and data unit sequences (including the null set) which may be filtered from an input or output datastream. The CAIS shall provide facilities permitting a process to select/query at execution time the subset of data units and data unit sequences which may be filtered (including the null set).

j. Modification. The CAIS shall specify the set of modifications that can occur to data units in an input/output datastream (e.g., mapping from lower case to upper case). The CAIS shall provide facilities permitting a process to select/query at execution time the subset of modifications that may occur (including the null set).

k. Datastream redirection. The CAIS shall provide facilities to associate at execution time the producer/consumer of each input/output datastream with a specific device, data entity, or process.

l. Input Sampling. The CAIS shall provide facilities to sample an input datapath for available data without having to wait if data are not available.

m. Transmission characteristics. The CAIS shall support control at execution time of host transmission characteristics (e.g., rates, parity, number of bits, half/full duplex).

n. Type-ahead. The CAIS shall provide facilities to disable/enable type-ahead. The CAIS shall provide facilities to indicate whether type-ahead is supported in the given implementation. The CAIS shall define the results of invoking the facilities to disable/enable type-ahead in those implementations that do not support type-ahead (e.g., null-effect or exception raised)

o. Echoing. The CAIS shall provide facilities to disable/enable echoing of data units to their source. The CAIS shall provide facilities to indicate whether echo-suppression is supported in the given implementation. The CAIS shall define the results of invoking the facilities to disable/enable echoing in those implementations that do not support echo-suppression (e.g., null effect or exception raised).

p. Control input datastream. The CAIS shall provide facilities to designate an input datastream as a control input datastream.

q. Control input trap. The CAIS shall provide the ability to abort a process by means of trapping a specific data unit or data unit sequence in a control input datastream of that process.

r. Trap sequence. The CAIS shall provide facilities to specify/query the data unit or data unit sequence that may be trapped. The CAIS shall provide facilities to disable/enable this facility at execution time.

s. Data link control. The CAIS shall support facilities for the dynamic control of data links, including, at least, self-test, automatic dialing, hang-up, and broken-link handling.

6.3B Data Block Transmission

a. Data block size. The CAIS shall provide facilities for the specification of the size of data blocks during program execution.

b. Datapath buffer size. The CAIS shall provide facilities for the specification of the sizes of input/output datapath buffers during process execution.

c. Datapath flushing. The CAIS shall provide facilities for the removal of all buffered data from an input/output datapath.

d. Output datapath processing. The CAIS shall provide facilities to force the output of all data in an output datapath.

e. Input/output sequencing. The CAIS shall provide facilities to ensure the servicing of input/output requests in the order of their invocation.

f. Datastream redirection. The CAIS shall provide facilities to associate at execution time each input/output datastream with a specific device, data entity, or process.

6.4 Data Entity Transfer

    a. Common external form. The CAIS shall specify a representation on physical media of a set of related data entities (referred to as the Common External Form).

    b. Transfer. The CAIS shall provide facilities using the Common External Form to support the transfer among CAIS implementations of sets of related data entities such that contents, attributes, and relationships are preserved.

CAIS SPECIFICATION COORDINATION REPORT

Bernard Abrams

Grumman Aerospace Corp

INTRODUCTION

This report contains a summary and analysis of standards and specifications that could possibly conflict with CAIS. A list of applicable standards was obtained from various indexes and by asking knowledgeable people. The primary index used was DODISS (Department of Defense Index of Standards and Specifications). Both government and industry standards were examined. Standards that were suspected of conflicting or of being redundant with CAIS were read and are reported herein.

This report would not have been possible without the assistance of the Grumman Aerospace Corp. Engineering Standards Department.

DOCUMENT ID: ANSI/ANS 10.2 1982

TITLE   Recommend Programming Practices to Facilitate
the Portability of Scientific Computer Programs

DOCUMENT DATE   12 March 1982

AGENCY   American Nuclear Society

STATUS   Approved

SUMMARY   Programming practices are recommended for
making application programs portable. The emphasis is on
scientific and engineering applications in FORTRAN. Typical
recommendations are to avoid extensions to ANSI Fortran.

CONNECTION TO CAIS   There is no connection. ANSI 10.2 is
concerned with achieving portability by using a common subset
of a variety of FORTRAN versions. CAIS achieves portability by
standardizing on an operating system interface in an
environment where the programming language is standard.

. . . .

DOCUMENT ID: ANSI/ANS 10.5 1979

TITLE   Guidelines for Considering User Needs in
Computer Program Development

DOCUMENT DATE   29 August, 1979

AGENCY   American Nuclear Society

STATUS   Approved

SUMMARY   User concerns are listed including proper
application, ease of use, reliability, and time required to
obtain results. Design practices to achieve programs that meet
the users concerns are modularity, automated adjustment to
hardware differences, and minimized input by using default
values.

CONNECTION TO CAIS   This standard is a good summary of design
practices for building user friendly programs, but it is not
directly applicable to CAIS because CAIS does not define a
human user interface.

....

DOCUMENT ID: ANSI X3H1

TITLE   OSCRL (Operating System Command & Response
  Language) Specification 09SD

DOCUMENT DATE   2 February 1984   Revision 20

AGENCY   ANSI

STATUS   Draft

SUMMARY   OSCRL specifies the command language used by a
  human user to request operating system services. The purpose is
  to promote portability of people and programs among general
  purpose computer systems. OSCRL has commands for managing files
  (COPY, CREATE, DELETE), commands for managing processes
  (SUBMIT), and a procedural language for controlling commands
  (IF, LOOP, GO TO, EXIT).

CONNECTION TO CAIS   There is a strong connection between CAIS
  and OSCRL. CAIS specifies the language used by a computer
  program to call for operating system services. These are the
  same services that a human user requests with OSCRL. The two
  languages should be compatible. If both specifications are
  adopted, then a user will use OSCRL to enter requests which
  will be translated by a command interpreter to CAIS calls.

      There have been discussions of having the user enter
  commands in Ada. The OSCRL language is not Ada.

      There is a definite need to coordinate OSCRL and CAIS
  since they overlap in many areas. One example is file naming
  conventions. A detailed comparison of the OSCRL draft with CAIS
  should be prepared.

....

DOCUMENT ID: ANSI/MIL-STD 1815A

TITLE   Ada Programming Language

DOCUMENT DATE   22 January 1983

AGENCY   Ada Joint Programming Office, DoD

STATUS   Approved

SUMMARY   Ada Language Reference Manual

CONNECTION TO CAIS   In addition to the requirement that CAIS
  conform to the Ada language, MIL-STD 1815A is a prototype of
  the format for CAIS. For example the precedent of allowing an
  exception to the outline of MIL-STD 962 was set by MIL-STD
  1815A and followed by CAIS.

....

DOCUMENT ID: DoD 4120.3-M

TITLE   Defense Standardization and Specification
  Program, Policies, Procedures and Instructions

DOCUMENT DATE   July, 1980

AGENCY   DoD

STATUS   Mandatory for use by all DoD activities

SUMMARY   The organizational procedure for making
  standards is described. It includes organization and
  assignments, planning, programming, policies and procedures for
  standardizing documents.

CONNECTION TO CAIS   The procedure for making CAIS a standard
  is described here.

CAIS Specification Coordination Report

. . . .

DOCUMENT ID: DOD-STD 7935

TITLE   Automatic Data System (ADS) Documentation

DOCUMENT DATE   13 September 1977

AGENCY   Department of Defense

STATUS   Approved

SUMMARY   All the documents that must be produced when
developing a computer system are describe. Standard outlines
are given.

CONNECTION TO CAIS An implementation of CAIS would be an ADS, but
CAIS itself is not. It might be considered an FD (Functional
Description) which is one of the documents of the development
life cycle required by DOD-STD 7935.

. . . .

DOCUMENT ID: FIPS PUB 30

TITLE   Software Summary for Describing Computer
Programs and Automated Data Systems

DOCUMENT DATE   30 June 1974

AGENCY   Institute for Computer Science, NBS

STATUS   Approved

SUMMARY   This publication provides a standard software
summary form (SF-185) for describing computer programs and
automated data systems for identification, reference, and
dissemination purposes.

CONNECTION TO CAIS none

....

DOCUMENT ID: FIPS PUB 41

TITLE   Computer Security Guidelines for Implementing
the Privacy Act of 1974

DOCUMENT DATE   30 May, 1975

AGENCY   National Bureau of Standards

STATUS   Approved

SUMMARY   This is general guidelines for computer
security including physical security, entry controls, data
encryption, and programming practices.

CONNECTION TO CAIS Nothing in CAIS prevents the implementation of
the security provisions of FIPS PUB 41.

. . . .

DOCUMENT ID: FIPS PUB 46

TITLE   Data Encryption Standard

DOCUMENT DATE   15 January 1977

AGENCY   Institute for Computer Science, NBS

STATUS   Approved

SUMMARY   An algorithm is described for enciphering and deciphering a block of data. The algorithm is applied to data when it leaves a system, and again when it reenters the system. The implementation method is not described. Whether or not the encryption is done in the CPU or in separate modules is not specified.

CONNECTION TO CAIS Encryption translates a given bit pattern into a random pattern. Therefore the transmission system after the point of encryption must pass all bit combinations. Any CAIS feature that prevents the transmission of specific characters could interfere with encryption.

. . . .

DOCUMENT ID: IEEE 162-63

TITLE   Standard Definition of Terms for Electronic Digital Computers

DOCUMENT DATE   Dec 63

AGENCY   IEEE

STATUS   Approved

SUMMARY   This is a hardware oriented glossary.

CONNECTION TO CAIS Possible use in definitions.

CAIS Specification Coordination Report

....

DOCUMENT ID: IEEE STD 730-81

TITLE   Standard for Software Quality Assurance Plans

DOCUMENT DATE   1981

AGENCY   IEEE

STATUS   Approved

SUMMARY   This standard describes what a SQAP (Software
Requirements Assurance Plan) should be. A SQAP applies to a
software development project. The activities and documents
needed for QA are listed. The minimum activities are SRR
(Software Requirements Review), PDR (Preliminary Design
Review), CDR (Critical Design Review), SVR (Software
Verification Review), Functional Audit, Physical Audit, and In
Process Audit. The minimum documents are SRS (Software
Requirements Specification), SDD (Software Data Design) SVP
(Software Verification Plan) and SVR (Software Verification
Report).

CONNECTION TO CAIS The CAIS itself is an interface standard, not
executable software, and therefore is not subject to the same
QA requirements as a software product. However, CAIS is a
product and as such should have some QA plan. A CAIS
implementation is software and needs a full QA plan.

....

DOCUMENT ID: MIL S 52779A

TITLE   Software   Quality   Assurance   Program
Requirements

DOCUMENT DATE   1 August, 1979

AGENCY   US Army Computer Systems Command

STATUS   Approved by Department of Defense

SUMMARY   This standard is applicable to computer
programs and related data and documentation. A Software Quality
Assurance Program is described. Included is the requirement for
practices and procedures to assure compliance. In addition the
tools, techniques, and methodologies

CONNECTION TO CAIS   An implementation of CAIS will require a
QA program.

....

DOCUMENT ID: MIL-STD 12D

TITLE   Abbreviations for Use on Drawings, and in
    Specifications, Standards, and Technical Drawings.

DOCUMENT DATE   29 May 1981

AGENCY   Department of Defense

STATUS   Approved

SUMMARY   This standard contains a list of approved
    abbreviations for use in specifications and standards. The list
    is by the full spelling and is cross referenced by
    abbreviation. An interesting quote is "when in doubt, spell it
    out".

CONNECTION TO CAIS Abbreviations in CAIS should not
    contradict MIL-STD 12D. An abbreviation such as "CAIS" that is
    too specific to appear in MIL-STD 12D is acceptable.

    The approved abbreviation for Identification is IDENT. The
    use of ID in the CAIS text violates MIL-STD 12D. A non-
    standard abbreviation may be used in a computer name. Agreement
    between computer names and MIL-STD 12D is optional. The
    standard is concerned with abbreviations in text.

....

DOCUMENT ID: MIL-STD 483

TITLE   Configuration Management Practices For
    Systems, Equipment, Munitions, And Computer Programs

DOCUMENT DATE   1 June 1971

AGENCY   USAF

STATUS   Approved

SUMMARY   This is one of several standards for
    configuration management.

CONNECTION TO CAISThis will be important during
    implementation.

. . . .

DOCUMENT ID: MIL STD 961a

TITLE   Military Specification, Preparation Of

DOCUMENT DATE   22 September 1981

AGENCY   DoD

STATUS   Approved

SUMMARY   The format of a MIL Specification is described. The use if "will" and "shall", the standard section numbering, style and word usage are specified.

CONNECTION TO CAIS   CAIS is a standard, not a specification. A companion document, MIL STD 962 applies to CAIS.

. . . .

DOCUMENT ID: MIL STD 962

TITLE   Outline of Forms and Instructions for the Preparation of Military Standards

DOCUMENT DATE   22 September, 1975

AGENCY   DoD, Defense Electronics Supply Center

STATUS   Approved

SUMMARY   This standard gives the format of a MIL Standard including word usage, paragraph identification, symbols, format for tables, use of footnotes, and figure sizes. A standardized outline is described.

CONNECTION TO CAIS   MIL STD 962 is applicable to CAIS. CAIS does conform in terms of format and style. However CAIS does not follow the standard outline. This exception is necessary because the standard outline does not fit an interface definition like CAIS.

. . . .

CAIS Specification Coordination Report

DOCUMENT ID: MIL-STD 1644

TITLE   Trainer System Software Development

DOCUMENT DATE   7 March 1979

AGENCY   Navel Trainer Equipment Center

STATUS   Approved

SUMMARY   This is one of several standards on the
procedure for developing software. The others are MIL STD 1679
and MIL STD SDS. The documents required and the procedures to
be followed are specified.

CONNECTION TO CAIS   None since CAIS is not training equipment

....

DOCUMENT ID: MIL-STD 1679

TITLE   Weapon System Software Development

DOCUMENT DATE   1 December 1978

AGENCY   NAVMAT 09Y

STATUS   Approved

SUMMARY   This standard controls the way software is
developed with emphasis on documents and procedures.

CONNECTION TO CAIS   Weapon System software is defined broadly
enough to include software development facilities of which CAIS
is a part. An implementation of CAIS should follow one of the
software development standards.

....

DOCUMENT ID: MIL-STD SDS

TITLE   Defense System Software Development

DOCUMENT DATE   20 December 1983

AGENCY   USAF RADC

STATUS   Draft Not Approved

SUMMARY   The methods and documents for software development are specified. Structured programming constructs are required. This standard is a replacement for MIL-STD 1679 and MIL-STD 1644.

Important quotes are "the contractor is encouraged to incorporate commercially available software" and "the contractor shall produce code that can be regenerated and maintained using only government-owned or contractually deliverable software."

CONNECTION TO CAIS   There is no conflict. This Standard controls the way CAIS is implemented.

....

CONCLUSION

The only standard found to have potential overlap is OSCRL. There are other standards that specify the way a standards document is formatted or the quality control of a program. These must be considered in making CAIS a standard, but have no direct conflict.

CAIS Specification Coordination Report

KITIA DRAFT PROPOSAL

The KITIA believes that positive action should be taken to address the perceived risks incurred by the current approach to achieving a common APSE interface set. In January of 1985, and RFP asking for a design of CAIS 2.0 is expected to be issued. This project will provide a CAIS design derived from and compatible with the current version.

The current version of the CAIS (1.3) has been criticized as too ambitious. This is, in aprt, because it prescribes a model of an operating system which is based on an entity-relationship database (the "node model"). Because current operating systems do not have this structure, concerns have been expressed about the risks in its schedule, its cost, and its technical viability. In addition, the current CAIS has made no explicit provision for implementations 'on' and 'beside' existing data management and operating systems. To insure uninterrupted progress in the Ada program and to provide transition paths for current compilers and environment efforts, the KITIa recommends that AJPO initiate a second, parallel effort.

We recommend that a second RPF be issued simultaneously with that for CAIS 2.0. This second RFP should contract a project to explore alternative CAIS designs to reduce risk, and enhance the Ada community's ability to rapidly deploy Ada tools into existing software development environments. This RFP should request an alternative CAIS design and working prototype which addresses the following objectives:

1. The interfaces shall be capable of being implemented in at least two existing environments without modification to their host operating systems (kernel).

2. There shall be minimal data management coding needed in th implementation of the interfaces to supplement the underlying facilities. This shall maximize use of existing structures and resources.

3. Ada tools using the interfaces shall be capable of operating on existing objects and devices, and cooperate with existing tools.

4. Pre-existing tools shall be capable of operating on data objects created by the Ada tools using the new interfaces.

5. Ada tools which use the interfaces shall be capable of being invoked and controlled by pre-existing tools and command languages.

6. The interfaces shall be specified by a set of Ada package specifications.

ADA PAPER
by
Dr. Chris Napjus

The following paper is presented for your information and consideration by Dr. Chris Napjus. Dr. Napjus is a respected member of the Software Engineering Community and is currently employed by the DoD. Dr. Napjus has followed Ada from its inception and is one of its strongest proponents.

PREAMBLE/PARABLE

Charles Goren revolutionized the world of bridge forty years ago with his new point-count system of bidding. Today, all American bridge players know and understand Goren, or "Standard American" as it is now called. Yet very, very few use the system in unmodified form, and the variations used, encompassing various combinations of features, are enormous. Yet most people use "straight Goren" features for a large plurality of their bids, as the variations apply to specific, often fairly rare, situations. Moreover, all can "fall back on Standard American" whenever they are playing with a new partner, gradually deviating from it over a period of time as they agree to variations which are to their mutual liking.

The fact that "Goren" still forms the basis for most of the bidding systems used today, despite forty years of evolution and the attempted introduction of a great many different "complete" systems, is a testament not only to the quality of the system, but also a reminder that a total revision of the way things are done constitutes an enormous undertaking. There are probably few - perhaps no - bridge players who believe that Standard American (or their particular variant of it) is the best possible system. What they agree to is that it is a good one; that it is understood by anyone with whom they might wish to play; that it is adaptable to whatever particular style they may like; that developing an entire, consistent system of their own is essentially impossible; and that even learning a new system, developed by someone else, would cost more in time and effort than any benefit which might ultimately accrue.

The analogy between bidding systems and the Ada environment should be obvious, and should be heeded. It is always possible to develop a better system than one you have, but the effort involved may well not be worth it. There is no single "best" answer, and one will never be able to design a system that any sizeable percentage of computer scientists would agree was the best attainable at that time. What is important is to have a good, useful system; better than what one has currently; which can be learned fairly readily so that it will be understood and capable of being used by any Ada developer. Like bridge players, software developers will find it far more useful to have a system which is solid and consistent, but easily modifiable in particulars to suit a given environment.

SOME RELEVANT CONSIDERATIONS

1. Ada was conceptualized, designed, funded, and developed to solve a pressing DoD need. The fact that it is more widely useful than initially envisioned is not a reason for deviating from its intended focus. The degree to which it will benefit non-DoD users is largely synergistic. Its benefit to DoD is essential.

2. As a percentage of the entire field of users, few (particularly within DoD) have any meaningful tools at all, let alone integrated environments. What is essential, as a first step, is to change this

situation, even if the result is not "state-of-the-art" for any particular state-of-the-art. There is vastly more to be gained by moving the bulk of the users SOON from 15 years behind the S-O-T-A to only 5 years behind than there is by providing the relatively few who are working at the S-O-T-A with a much more advanced environment SOMETIME in the future. In short, we should not be applying our effort to defining an environment for the 90's when the bulk of the users are still working in the 70's.

3. There is also a significant danger to the entire Ada effort by waiting too long to provide something practical. Aside from the obvious material loss in having to maintain too many non-standard systems that much longer, Ada will not be used extensively until it is able to promise a SUBSTANTIAL benefit over other systems. This requires a usable environment. The longer we wait, the longer it will be until a large number of people are trained on Ada and, as the widely proclaimed benefits recede ever-farther into the twilight, the emphasis on training people will become less and less. The community is ready for Ada NOW. The number of non-believers will grow exponentially the longer it continues to be "something for the future."

4. Even a relatively small set of manually-invoked tools would be a significant improvement over the common situation today. Moreover, incrementality does not preclude improvement. We can have it both ways if we do it right. Perhaps most important, the average user is not ready to utilize some beautiful state-of-the-art system. He needs experience with tools in general before he will, or can, make use of an automated system which he doesn't understand. There is much to be said for crawling before walking.

5. It would certainly be nice to have an Ada environment which is as methodology- independent as posssible. But:

   a. this is an argument for keeping the number of MAPSE tools small

   b. the need for methodology independence is far greater when looking at the total class of potential users vice the DoD class

   c. ANY consistent methodology, suboptimal as it might (and surely will) be, is better than NONE (the usual situation now), and may indeed be better than a combination of (presumably) optimized methodologies within too small an organization.

6. It is useful to think of software environment technology as being in four tiers:

   a. research technology (most appropriate to academia)

   b. prototype usage (industrial research/technology groups)

   c. well understood, common technology (some usage, especially larger firms)

   d. outmoded/outdated methods (the bulk of the community, especially DoD)

7. With respect to these tiers, DoD (and much of industry) has its efforts in 3 and 4, predominately 4. The crying need is to move all 4-level projects to 3 as soon as possible, and the KIT should be devot-

3-290

ing most of its attention to this goal. Level-3 technology, under-
stood, easily-teachable, not-too-drastic a departure is what project
managers need NOW. Such a move from the technology of the 60's and
70's to that of the 80's is consistent with:

  a. need

  b. maximum cost savings

  c. the impetus behind Ada

  d. Congressional guidance

  e. provision of a vehicle for subsequent improvements.

8. I think that the KIT is likely a victim of its own constitution of
   very intelligent, highly competent computer scientists. It is all too
   normal to have little interest in that which you consider mundane; to
   have too little concern with that part of the world you don't see; to
   try to design the best system of which you are capable (which may not
   be the most cost-effective); and to concentrate on aspects that are
   of personal interest to you (and, not incidentally, of sufficient
   research interest to permit publication). I think that this may ex-
   plain much of the KIT's lack of tangible achievement to date. There
   is too much commonality of interests and background, which almost
   quarantees that many problems will be viewed within a limited con-
   text. In fact, the KIT would benefit from infusion of others with
   very diverse backgrounds. What is needed is a broad range of back-
   grounds, interests, and goals ranging from the very blue sky to the
   in-the-trenches users, and everything in between.

9. The KIT, and the Ada community, cannot afford to wait for fruition of
   a "best possible solution," which may require "only a few more months
   (or years)" of study in order to become feasible on a wide scale.

MY OWN (DISJOINTED) VIEW OF WHAT THE MAPSE/STANDARD TOOL SET SHOULD BE

1. It must be limited, modifiable, substitutable, and expandable. But
   modifications must be carefully controlled, determined to be for very
   solid reasons, and centrally disseminated. I believe that we need
   tool validation for MAPSE tools to ensure that they do AT LEAST
   everything which it has been decided MUST be available in the stan-
   dard MAPSE.

2. Like Ada itself, the tools must be oriented toward ease of use, re-
   gardless of any added burden this may impose on the developers of the
   tools.

3. I agree that the existing STONEMAN is inadequate, but believe we need
   an expanded, updated, consolidated revision (a la STRAWMAN, WOODEN-
   MAN, TINMAN, STEELMAN) rather than a new, replacement document.

4. The basic MAPSE must provide a standard set of tools, with a standard
   MINIMAL set of functions which they perform, and a standard, common
   command language which can invoke them. Supersets are fine (of tools,
   of command language, or of functions performed by a specific tool) SO
   LONG AS the basic, reuired tools, functions, and commands are there,
   and the user can be assured that a given command will result in a
   given, known set of transformations. A given environment may have ad-

ditional tools, tools which do more than their minimum required functions, and alternative ways of invoking them (to include, perhaps, self-invoking mechanisms from one tool to the next.) But the basic, STANDARD mechanisms must be there.

5. Like a basic bridge bidding system, with additional conventions, my view of the MAPSE is one of a relatively modest set of relatively simple tools, ALWAYS available in ANY Ada environment, which will be learned by and under stood by ALL Ada developers. Organizations or individuals can add their own favorite modifications to this base, but can count on the basic set at all times, and can always revert to it if desired. Such a scheme should not only maximize transportatbility of programmers, but also accommodate itself well to various skill levels within a project or organization. Newer, less experienced personnel can become productive rapidly by using the more basic (and known) mechanisms, only gradually extending their utilization to more exotic features. Older hands can, simultaneously, make use of whatever features their particular (tailored) environment may offer.

END

FILMED

11-85

DTIC