Downloaded from http://www.everyspec.com

METRIC

DOD-STD-1838
9 October 1986

MILITARY STANDARD

COMMON ADA® PROGRAMMING SUPPORT ENVIRONMENT (APSE) INTERFACE SET (CAIS)

NO DELIVERABLE DATA
REQUIRED BY THIS DOCUMENT

AREA MCCR/IPSC

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

Downloaded from http://www.everyspec.com

DEPARTMENT of DEFENSE
Washington, DC 20302

Marking for Shipment and Storage

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 sent to the Ada Joint Program Office, Room 3E114, Pentagon, Washington, DC 20301-3081, by using the self-addressed Standardization Document Improvement Proposal (DD Form 1426) appearing at the end of this document or by letter.

Downloaded from http://www.everyspec.com

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 Department of Defense (DoD) Ada[1] 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 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 AIE and ALS 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 KIT: J. Foidl (TRW), J. Kramer (Institute for Defense Analyses), 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 B. Schaar (Veda), T. Harrison (Texas Instruments) and KITIA members: H. Fischer (Mark V Systems), E. Lamb (Bell Labs), T. Lyons (Software Sciences Ltd., U.K.), D. McGonagle (General Electric), H. Morse (Oracle Corporation), E. Ploedereder (Tartan Laboratories), H. Willman (Raytheon), and L. Yelowitz (Ford Aerospace).

During 1984, the following people assisted in preparation of this document: F. Belz (TRW) and the TRW prototype team, J. Kerner (TRW), K. Connolly (TRW), S. Ferdman (Data General), G. Fitch (Intermetrics), R. Gouw (TRW), B. Grant (Intermetrics), N. Lee (Institute for Defense Analyses), J. Long (TRW), and R. Robinson (Institute for Defense Analyses).

During 1985 and 1986, the team was again expanded to include M. Lake and C. Roby (Institute for Defense Analyses), LCDR P. Myers (Ada Joint Program Office), and F. Tadman (TRW). Additional constructive criticism and direction was provided by G. Myers (Naval Ocean Systems Center), O. Roubine (Informatique Internationale), the prototyping teams of Gould, Inc. (Ft. Lauderdale, FL), MITRE (McLean, VA) and TRW (Redondo Beach, CA), and the general memberships of the KIT and KITIA, as well as many independent reviewers. The Ada Joint Program Office is particularly grateful to these individuals and their organizations for providing the time and resources that significantly contributed to this document.

Of all these individuals, those to whom the final form of this document is most attributable are F. Belz, T. Harrison, J. Kramer, P. Oberndorf, E. Ploedereder, C. Roby and F. Tadman. The Ada Joint Program Office regards highly the individual efforts put forth by these and the other people listed here.

This document was prepared with the SCRIBE[2] typesetting tool under VAX/VMS[3] at the Institute for Defense Analyses.

---

[1] ANSI/MIL-STD-1815A 1983.

[2] "SCRIBE" is a registered trademark of UniLogic, Ltd.

[3] VAX and VMS are registered trademarks of Digital Equipment Corporation.

Downloaded from http://www.everyspec.com

FOREWORD

Downloaded from http://www.everyspec.com

Contents

1. SCOPE 1
1.1 Purpose 1
1.2 Application Guidance 2
2. REFERENCED DOCUMENTS 3
2.1 Government documents 3
2.2 Other publications 3
2.3 Order of precedence 5
2.4 Source of documents 5
3. DEFINITIONS 7
4. GENERAL REQUIREMENTS 19
4.1 Introduction 19
4.2 Method of description 19
4.2.1 Allowable differences 20
4.2.2 Semantic descriptions 20
4.2.3 Typographical conventions 21
4.3 CAIS node model 22
4.3.1 Nodes 22
4.3.2 Processes 22
4.3.3 Input and output 23
4.3.4 Relationships and relations 23
4.3.4.1 Kinds of relationships 24
4.3.4.2 Basic predefined relations 25
4.3.4.3 Relation names and relationship keys 29
4.3.5 Paths, pathnames and node identification 29
4.3.6 Attributes 32
4.3.6.1 Predefined attributes 33
4.4 Discretionary and mandatory access control 35
4.4.1 Node access 35
4.4.2 Discretionary access control 36
4.4.2.1 Groups and roles 36
4.4.2.2 Adopting a role 37
4.4.2.3 Granting access rights 40
4.4.2.4 Determining access rights 40
4.4.2.5 Discretionary access checking 48
4.4.3 Mandatory access control 48
4.4.3.1 Labeling of CAIS nodes 51
4.4.3.2 Labeling of process nodes 52
4.4.3.3 Labeling of non-process nodes 52
4.4.3.4 Labeling of nodes for devices 52
4.4.3.5 Mandatory access checking 52
5. DETAILED REQUIREMENTS 53
5.1 General node management 53
5.1.1 Package CAIS_DEFINITIONS 54
5.1.2 Package CAIS_NODE_MANAGEMENT 57
5.1.2.1 Opening a node handle 63
5.1.2.2 Closing a node handle 66
5.1.2.3 Changing the intent regarding node handle usage 67
5.1.2.4 Examining the open status of a node handle 69
5.1.2.5 Querying the intention of a node handle 70
5.1.2.6 Querying the kind of a node 71
5.1.2.7 Querying the number of open file handles on a file node 72

Downloaded from http://www.everyspec.com

5.1.2.8 Obtaining the unique primary pathname 73
5.1.2.9 Obtaining the relationship key of a primary relationship 74
5.1.2.10 Obtaining the relation name of a primary relationship 75
5.1.2.11 Obtaining the relationship key of the last relationship traversed 76
5.1.2.12 Obtaining the relation name of the last relationship traversed 77
5.1.2.13 Obtaining a partial pathname 78
5.1.2.14 Obtaining the name of the last relationship in a pathname 79
5.1.2.15 Obtaining the key of the last relationship in a pathname 80
5.1.2.16 Querying the existence of a node 81
5.1.2.17 Querying sameness 82
5.1.2.18 Obtaining an index for a node handle 84
5.1.2.19 Obtaining an open node handle to the parent 85
5.1.2.20 Copying a node 87
5.1.2.21 Copying trees 89
5.1.2.22 Renaming the primary relationship of a node 92
5.1.2.23 Deleting the primary relationship to a node 94
5.1.2.24 Deleting the primary relationships of a tree 96
5.1.2.25 Creating secondary relationships 98
5.1.2.26 Deleting secondary relationships 100
5.1.2.27 Setting inheritance property of a relationship 102
5.1.2.28 Determining if a secondary relationship is inheritable 104
5.1.2.29 Node iteration types and subtypes 106
5.1.2.30 Creating an iterator over nodes 107
5.1.2.31 Determining iteration status 109
5.1.2.32 Determining the approximate size of the iterator 110
5.1.2.33 Getting the next node in an iteration 111
5.1.2.34 Skipping the next node in an iteration 113
5.1.2.35 Obtaining the path element for the next node in an iteration 114
5.1.2.36 Deleting an iterator 115
5.1.2.37 Setting the current node relationship 116
5.1.2.38 Opening a node handle to the current node 117
5.1.2.39 Determining the creation time of a node 119
5.1.2.40 Determining the last time a relationship was modified 120
5.1.2.41 Determining the last time that node contents were written 121
5.1.2.42 Determining the last time an attribute was modified 122
5.1.3 Package CAIS_ATTRIBUTE_MANAGEMENT 123
5.1.3.1 Creating node attributes 125
5.1.3.2 Creating path attributes 127
5.1.3.3 Deleting node attributes 129
5.1.3.4 Deleting path attributes 131
5.1.3.5 Setting node attributes 133
5.1.3.6 Setting path attributes 135
5.1.3.7 Getting node attributes 137
5.1.3.8 Getting path attributes 139
5.1.3.9 Attribute iteration types and subtypes 141
5.1.3.10 Creating an iterator over node attributes 142
5.1.3.11 Creating an iterator over relationship attributes 144
5.1.3.12 Determining iteration status 146
5.1.3.13 Determining the approximate size of the iterator 147
5.1.3.14 Getting the next attribute name 148
5.1.3.15 Getting the next attribute value 149
5.1.3.16 Skipping the next attribute in an iteration 150
5.1.3.17 Deleting an attribute iterator 151
5.1.4 Package CAIS_ACCESS_CONTROL_MANAGEMENT 152
5.1.4.1 Subtypes 152
5.1.4.2 Value of all access rights 153
5.1.4.3 Setting access control 154

Downloaded from http://www.everyspec.com

5.1.4.4 Deleting access relationships 156
5.1.4.5 Obtaining the value of a GRANT attribute 158
5.1.4.6 Examining access rights 159
5.1.4.7 Adopting a role 160
5.1.4.8 Unlinking an adopted role 162
5.1.5 Package CAIS_STRUCTURAL_NODE_MANAGEMENT 163
5.1.5.1 Creating structural nodes 164
5.2 CAIS process nodes 168
5.2.1 Package CAIS_PROCESS_DEFINITIONS 171
5.2.2 Package CAIS_PROCESS_MANAGEMENT 172
5.2.2.1 Spawning a process 174
5.2.2.2 Awaiting termination or abortion of another process 178
5.2.2.3 Invoking a new process 180
5.2.2.4 Creating a new job 185
5.2.2.5 Deleting a job 188
5.2.2.6 Appending results 190
5.2.2.7 Overwriting results 191
5.2.2.8 Getting results from a process 192
5.2.2.9 Determining the status of a process 194
5.2.2.10 Getting the parameter list 195
5.2.2.11 Aborting a process 196
5.2.2.12 Suspending a process 198
5.2.2.13 Resuming a process 200
5.2.2.14 Determining the number of open node handles 202
5.2.2.15 Determining the number of input and output units used 203
5.2.2.16 Determining the time of activation 204
5.2.2.17 Determining the time of termination or abortion 205
5.2.2.18 Determining the time a process has been active 206
5.2.2.19 Determining the size of a process 207
5.3 CAIS input and output 208
5.3.1 Package CAIS_DEVICES 214
5.3.2 Package CAIS_IO_DEFINITIONS 215
5.3.3 Package CAIS_IO_ATTRIBUTES 217
5.3.3.1 Determining the access method 218
5.3.3.2 Determining the file kind 219
5.3.3.3 Determining the queue kind 220
5.3.3.4 Determining the device kind 221
5.3.3.5 Determining the current file size 222
5.3.3.6 Determining the maximum file size 223
5.3.3.7 Determining the current queue size 224
5.3.3.8 Determining the maximum queue size 225
5.3.4 Package CAIS_DIRECT_IO 226
5.3.4.1 Definition of Types 227
5.3.4.2 Creating a direct file 228
5.3.4.3 Opening a direct file handle 231
5.3.4.4 Closing a direct file handle 232
5.3.4.5 Resetting a direct file handle 233
5.3.4.6 Synchronizing the internal file with file node contents 234
5.3.5 Package CAIS_SEQUENTIAL_IO 235
5.3.5.1 Definition of types 236
5.3.5.2 Creating a sequential file 237
5.3.5.3 Opening a sequential file handle 240
5.3.5.4 Closing a sequential file handle 241
5.3.5.5 Resetting a sequential file handle 242
5.3.5.6 Synchronizing the internal file with file node contents 243
5.3.6 Package CAIS_TEXT_IO 244
5.3.6.1 Definition of types 245

Downloaded from http://www.everyspec.com

5.3.6.2 Creating a text file 246
5.3.6.3 Opening a text file handle 249
5.3.6.4 Closing a text file handle 250
5.3.6.5 Resetting a text file handle 251
5.3.6.6 Synchronizing the internal file with file node contents 252
5.3.7 Package CAIS_QUEUE_MANAGEMENT 253
5.3.7.1 Creating a nonsynchronous copy queue node 257
5.3.7.2 Creating a nonsynchronous mimic queue node 262
5.3.7.3 Creating a nonsynchronous solo text queue node 267
5.3.7.4 Creating a nonsynchronous solo sequential queue node 271
5.3.7.5 Creating a synchronous solo text queue node 274
5.3.7.6 Creating a synchronous solo sequential queue node 278
5.3.7.7 Opening a queue file node handle 281
5.3.7.8 Closing a queue file node handle 281
5.3.7.9 Opening a queue file handle 281
5.3.7.10 Closing a queue file handle 282
5.3.7.11 Reading elements from a queue file 282
5.3.7.12 Writing elements to a queue file 282
5.3.7.13 Resetting a queue file handle 282
5.3.7.14 Determining end of file of a queue file 283
5.3.8 Package CAIS_SCROLL_TERMINAL_IO 284
5.3.8.1 Types and subtypes 286
5.3.8.2 Opening a scroll terminal file handle 287
5.3.8.3 Closing a scroll terminal file handle 288
5.3.8.4 Determining whether a file handle is open 289
5.3.8.5 Determining the number of function keys 290
5.3.8.6 Determining intercepted input characters 291
5.3.8.7 Determining intercepted output characters 292
5.3.8.8 Enabling and disabling function key usage 293
5.3.8.9 Determining function key usage 294
5.3.8.10 Setting the active position 295
5.3.8.11 Determining the active position 296
5.3.8.12 Determining the size of the terminal 297
5.3.8.13 Setting a tab stop 298
5.3.8.14 Clearing a tab stop 299
5.3.8.15 Clearing all tab stops 300
5.3.8.16 Advancing to the next tab position 301
5.3.8.17 Sounding a terminal bell 302
5.3.8.18 Writing to the terminal 303
5.3.8.19 Reading a character from a terminal 304
5.3.8.20 Reading all available characters from a terminal 305
5.3.8.21 Creating a function key descriptor 306
5.3.8.22 Deleting a function key descriptor 307
5.3.8.23 Determining the number of function keys that were read 308
5.3.8.24 Determining function key usage 309
5.3.8.25 Determining the identification of a function key 310
5.3.8.26 Determining the mode of a terminal 311
5.3.8.27 Backspacing the active position 312
5.3.8.28 Advancing the active position to the next line 313
5.3.8.29 Advancing the active position to the next page 314
5.3.8.30 Resetting a scroll terminal file handle 315
5.3.8.31 Synchronizing the internal file with file node contents 316
5.3.8.32 Setting terminal file handle synchronization 317
5.3.8.33 Determining the synchronization of a terminal file handle 318
5.3.9 Package CAIS_PAGE_TERMINAL_IO 319
5.3.9.1 Types, subtypes and constants 321
5.3.9.2 Opening a page terminal file handle 323

Downloaded from http://www.everyspec.com

5.3.9.3 Closing a page terminal file handle 324
5.3.9.4 Determining whether a file handle is open 325
5.3.9.5 Determining the number of function keys 326
5.3.9.6 Determining intercepted input characters 327
5.3.9.7 Determining intercepted output characters 328
5.3.9.8 Enabling and disabling function key usage 329
5.3.9.9 Determining function key usage 330
5.3.9.10 Setting the active position 331
5.3.9.11 Determining the active position 332
5.3.9.12 Determining the size of the terminal 333
5.3.9.13 Setting a tab stop 334
5.3.9.14 Clearing a tab stop 335
5.3.9.15 Clearing all tab stops 336
5.3.9.16 Advancing to the next tab position 337
5.3.9.17 Sounding a terminal bell 338
5.3.9.18 Writing to the terminal 339
5.3.9.19 Reading a character from a terminal 340
5.3.9.20 Reading all available characters from a terminal 341
5.3.9.21 Creating a function key descriptor 342
5.3.9.22 Deleting a function key descriptor 343
5.3.9.23 Determining the number of function keys that were read 344
5.3.9.24 Determining function key usage 345
5.3.9.25 Determining the identification of a function key 346
5.3.9.26 Determining the mode of a terminal 347
5.3.9.27 Deleting characters 348
5.3.9.28 Deleting lines 349
5.3.9.29 Replacing characters in a line with space characters 350
5.3.9.30 Erasing characters in a display 351
5.3.9.31 Erasing characters in a line 352
5.3.9.32 Inserting space characters in a line 353
5.3.9.33 Inserting blank lines in the output terminal file 354
5.3.9.34 Determining graphic rendition support 355
5.3.9.35 Selecting the graphic rendition 356
5.3.9.36 Determining the effect of writing to the end position 357
5.3.9.37 Resetting a page terminal file handle 358
5.3.9.38 Synchronizing the internal file with file node contents 359
5.3.9.39 Setting terminal file handle synchronization 360
5.3.9.40 Determining the synchronization of a terminal file handle 361
5.3.10 Package CAIS_FORM_TERMINAL_IO 362
5.3.10.1 Types and subtypes 363
5.3.10.2 Opening a form terminal file handle 364
5.3.10.3 Closing a form terminal file handle 365
5.3.10.4 Determining whether a file handle is open 366
5.3.10.5 Determining the number of function keys 367
5.3.10.6 Determining intercepted input characters 368
5.3.10.7 Determining intercepted output characters 369
5.3.10.8 Creating a form 370
5.3.10.9 Deleting a form 371
5.3.10.10 Copying a form 372
5.3.10.11 Defining a qualified area 373
5.3.10.12 Removing an area qualifier 374
5.3.10.13 Changing the active position 375
5.3.10.14 Querying the active position 376
5.3.10.15 Advancing forward to qualified area 377
5.3.10.16 Writing to a form 378
5.3.10.17 Erasing a qualified area 379
5.3.10.18 Erasing a form 380

Downloaded from http://www.everyspec.com

5.3.10.19 Activating a form on a terminal 381
5.3.10.20 Reading from a form 382
5.3.10.21 Determining changes to a form 383
5.3.10.22 Determining the identification of a function key 384
5.3.10.23 Determining the termination key 385
5.3.10.24 Determining the size of a form 386
5.3.10.25 Determining the size of the terminal 387
5.3.10.26 Determining if the area qualifier requires space in the form 388
5.3.10.27 Determining if the area qualifier requires space on a terminal 389
5.3.11 Package CAIS_MAGNETIC_TAPE_IO 390
5.3.11.1 Types, subtypes and exceptions 392
5.3.11.2 Opening a tape drive file handle 395
5.3.11.3 Closing a tape drive file handle 396
5.3.11.4 Determining whether a file handle is open 397
5.3.11.5 Determining the mode of a magnetic tape drive 398
5.3.11.6 Requesting the mounting of a tape 399
5.3.11.7 Loading a tape 400
5.3.11.8 Unloading a tape 401
5.3.11.9 Requesting the dismounting of a tape 402
5.3.11.10 Determining the position of the tape 403
5.3.11.11 Rewinding the tape 404
5.3.11.12 Skipping tape marks 405
5.3.11.13 Writing a tape mark 406
5.3.11.14 Determining the status of a magnetic tape drive 407
5.3.11.15 Determining the recording method of a magnetic tape 408
5.3.11.16 Determining whether a write ring is installed 409
5.3.11.17 Skipping blocks in a magnetic tape file 410
5.3.11.18 Reading a block from a magnetic tape file 411
5.3.11.19 Writing a block to a magnetic tape file 412
5.3.11.20 Resetting a magnetic tape file handle 413
5.3.12 Package CAIS_IMPORT_EXPORT 414
5.3.12.1 Importing a file 415
5.3.12.2 Exporting a file 417
5.4 CAIS List Management 419
5.4.1 Package CAIS_LIST_MANAGEMENT 425
5.4.1.1 Types, subtypes, constants and exceptions 425
5.4.1.2 Copying a list 427
5.4.1.3 Making a list empty 428
5.4.1.4 Converting from text to list form 429
5.4.1.5 Converting a list to its text representation 430
5.4.1.6 Determining the equality of two lists 431
5.4.1.7 Deleting an item from a linear list 432
5.4.1.8 Determining the kind of list 433
5.4.1.9 Determining the kind of list item 434
5.4.1.10 Inserting a sequence of items into a linear list 435
5.4.1.11 Concatenating two linear lists 436
5.4.1.12 Extracting a sequence of items from a linear list 437
5.4.1.13 Determining the length of a linear list 438
5.4.1.14 Determining the position of the current linear list 439
5.4.1.15 Determining whether the current linear list is outermost 440
5.4.1.16 Making the next outer linear list current 441
5.4.1.17 Making a nested sublist the current linear list 442
5.4.1.18 Determining the length of the text form of a list or a list item 444
5.4.1.19 Determining the name of a named item 446
5.4.1.20 Determining the position of a named item 447
5.4.1.21 Package CAIS_LIST_ITEM 448
5.4.1.21.1 Extracting a list value from a list item 449

Downloaded from http://www.everyspec.com

5.4.1.21.2 Replacing a list value in a list item 451
5.4.1.21.3 Inserting a list-valued item into a linear list 453
5.4.1.21.4 Locating a list-valued item by value within a linear list 455
5.4.1.22 Package CAIS_IDENTIFIER_ITEM 457
5.4.1.22.1 Copying a token 458
5.4.1.22.2 Converting an identifier from text to token form 459
5.4.1.22.3 Converting an identifier from token to text form 460
5.4.1.22.4 Determining the equality of two identifier tokens 461
5.4.1.22.5 Extracting an identifier value from a list item 462
5.4.1.22.6 Replacing an identifier value in a list item 464
5.4.1.22.7 Inserting an identifier-valued item into a linear list 466
5.4.1.22.8 Locating an identifier-valued item by value within a linear list 468
5.4.1.23 Generic package CAIS_INTEGER_ITEM 469
5.4.1.23.1 Converting an integer value to its canonical text representation 470
5.4.1.23.2 Extracting an integer value from a list item 471
5.4.1.23.3 Replacing an integer value in a list item 473
5.4.1.23.4 Inserting an integer-valued item into a linear list 475
5.4.1.23.5 Locating an integer-valued item by value within a linear list 477
5.4.1.24 Generic package CAIS_FLOAT_ITEM 478
5.4.1.24.1 Converting a floating point value to its canonical text form 479
5.4.1.24.2 Extracting a floating point value from a list item 480
5.4.1.24.3 Replacing a floating point value in a list item 482
5.4.1.24.4 Inserting a floating point-valued item into a linear list 484
5.4.1.24.5 Locating a floating point-valued item by value within a linear list 486
5.4.1.25 Package CAIS_STRING_ITEM 488
5.4.1.25.1 Extracting a string value from a list item 489
5.4.1.25.2 Replacing a string value in a list item 491
5.4.1.25.3 Inserting a string-valued item into a linear list 493
5.4.1.25.4 Locating a string-valued item by value within a linear list 495
5.5 Package CAIS_STANDARD 496
5.6 Package CAIS_CALENDAR 497
5.6.1 Definition of types, subtypes and exceptions 498
5.6.2 Getting the current time 499
5.6.3 Getting the year part of the time 500
5.6.4 Getting the month part of the time 501
5.6.5 Getting the day part of the time 502
5.6.6 Getting the seconds part of the time 503
5.6.7 Splitting time into its components 504
5.6.8 Combining components of time 505
5.6.9 Adding time and duration 506
5.6.10 Subtracting time and duration 507
5.6.11 Comparing two values of time 508
5.7 CAIS Pragmatics 509
6. NOTES 515
6.1 Keywords 515
Appendix A. Predefined Relations, Attributes and Attribute Values 517
Appendix B. CAIS Specification 527
Appendix C. Cross Reference of CAIS Procedures and Functions 571
Appendix D. Syntax Summary 595
Appendix E. CAIS Access Control Management 597
Appendix F. Implementation Dependencies 599

Downloaded from http://www.everyspec.com

Index *601*

Downloaded from http://www.everyspec.com

Figures

FIGURE 1.	CAIS system concept	26
FIGURE 2.	Tree formed by primary relationships	28
FIGURE 3.	Some predefined relations	30
FIGURE 4.	Some predefined attributes	34
FIGURE 5.	Discretionary access control	38
FIGURE 6.	Discretionary access control after process creation	41
FIGURE 7.	Access relationships, Example 1	46
FIGURE 8.	Access relationships, Example 2	49
FIGURE 9.	Access relationships, Example 3	50
FIGURE 10.	COPY_TREE Example	91
FIGURE 11.	Magnetic tape status transitions	393

Downloaded from http://www.everyspec.com

Tables

TABLE I. Pathname BNF 32
TABLE II. GRANT Attribute Value BNF 42
TABLE III. Predefined Access Rights 43
TABLE IV. Classification Attribute Value BNF 51
TABLE V. Intents 58
TABLE VI. Matrix of Access Synchronization Constraints 62
TABLE VII. Process Status Transition 168
TABLE VIII. Relationships Created as a Result of CREATE_JOB 172
TABLE IX. Relationships Created and Inherited for Process Nodes 173
TABLE X. Modes and Intents for Input and Output 209
TABLE XI. Input and Output Packages for File Attributes 210
TABLE XII. File Node Predefined Entities 213
TABLE XIII. Queue File Node Predefined Entities 255
TABLE XIV. Allowed Magnetic Tape Characters 391
TABLE XV. List External Representation BNF 422

Downloaded from http://www.everyspec.com

1. SCOPE

1.1 Purpose

This document provides specifications for a set of Ada[4] packages, with their intended semantics, which together form a set of common interfaces for Ada Programming Support Environments (APSEs). This set of interfaces is known as the Common APSE Interface Set (CAIS) and is designed to promote the source-level portability of Ada programs, particularly Ada software development tools.

The goal of the CAIS is to promote interoperability and transportability of Ada software across Department of Defense (DoD) APSEs.

Interoperability is defined as the ability of APSEs to exchange database 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 Kernel Ada Programming Support Environment (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.[5]

Those Ada programs that are used in support of software and firmware development are defined as *tools*. This includes the spectrum of support software from project management through code development, configuration management and life cycle support. Tools are not restricted to only those software items normally associated with program generation, such as editors, compilers, debuggers, and linker-loaders. Groups of tools that are composed of a number of independent but interrelated programs (such as a debugger which is related to a specific compiler) are referred to as *tool sets*.[6] The CAIS establishes interface requirements for the transportability of Ada tool sets to be used in DoD APSEs. Strict adherence to this interface set will ensure that Ada tools and tool sets will possess a high degree of transportability across conforming APSEs. Where tools function as a set, the CAIS facilitates transportability of the tool set as a whole. Individual tools in this set might not be individually transportable because they depend on inputs from other tools in the set.

The scope of the CAIS includes interfaces to those services, traditionally provided by an operating system, that affect tool transportability. The CAIS is intended to provide the transportability interfaces most often required by common software development tools and includes the following interface areas:

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

---

[4] ANSI/MIL-STD-1815A 1983.

[5] *KAPSE Interface Team: Public Report, Volume I*, 1 April 1982; p. C1.

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

Downloaded from http://www.everyspec.com

b. Processes. This area covers program invocation and control.

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

d. Utilities. This area covers list operations useful for manipulation of parameters and attribute values.

e. Pragmatics. This area presents the pragmatic limitations that a conforming CAIS implementation is allowed to impose.

The CAIS as specified in this document is believed to provide the tool writer with significant advantages in tool portability, since the most crucial host dependencies are transparently encapsulated by the interfaces of the CAIS.

The CAIS is not a replacement for a host operating system. It standardizes those tool-to-host interfaces that are most crucial for tool portability. Other less frequently used or inherently host-dependent interfaces must complement the CAIS in order to provide a basis for the construction of an APSE. The degree of portability of tools will depend on the degree to which they can obtain the required functionality of host interfaces through the CAIS, rather than through host-dependent interfaces.

It is assumed that the reader of this document does not have a detailed knowledge of operating systems concepts and the Ada programming language, but is familiar enough with these topics to understand the concepts defined and used herein.

1.2 Application Guidance

The CAIS applies to all APSEs, in particular those which are to become the basic software life-cycle environments for DoD mission-critical computer systems (MCCS). It is being issued as a military standard in order to allow its application to government contracts. Initially the principal purpose of such an application is to allow contracts to specify the use of the CAIS in experimental implementations whose objective is to learn about the viability, feasibility, implementability and usability of the interface set as a component of a programming support environment. Implementations of this proposed interface set should provide knowledge about the implementation of its features and feedback to the CAIS designers relevant to the development of Revision A of the CAIS.

Proper application of this standard is as follows: (1) prototype implementations of the interface set, either wholly or in part; (2) prototype implementations of tools written to utilize the CAIS interfaces; (3) implementation studies designed for such purposes as determining the probable ease of implementing the CAIS on new operating systems or bare machines; and (4) experimental studies designed to utilize a prototype CAIS and/or tool implementation in order to gather information regarding performance, usability, viability, etc.

It is anticipated that the success of these implementation experiments will establish the viability, feasibility, implementability and usability of the interface set. When these properties have been satisfactorily demonstrated, the standard will be mature enough for use on actual development projects. It is also anticipated that such experimentation will guide the manner in which the standard will be revised in the future.

Downloaded from http://www.everyspec.com

2. REFERENCED DOCUMENTS

2.1 Government documents

Unless otherwise specified, the following specifications, standards, and handbooks of the issue listed in that issue of the Department of Defense Index of Specifications and Standards (DoDISS) specified in the solicitation form a part of this standard to the extent specified herein.

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

[962A]: *Military Standards and Handbooks, Preparation of, MIL-STD-962A*; United States Department of Defense; 26 October 1984.

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

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

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

2.2 Other publications

The following documents form a part of this standard to the extent specified herein. Unless otherwise specified, the issues of the documents which are DoD adopted shall be those listed in the issue of the DoDISS specified in the solicitation. The issues of documents which have not been adopted shall be those in effect on the date of the cited DoDISS.

[ANSI 73a]: American National Standards Institute, *Recorded Magnetic Tape for Information Interchange (800 CPI, NRZI) (ANSI Standard x3.22-1973)*. (Application for copies should be addressed to American National Standards Institute, Inc., 1430 Broadway, New York, NY 10018.)

[ANSI 73b]: American National Standards Institute, *Recorded Magnetic Tape for Information Interchange (1600 CPI, PE) (ANSI Standard x3.39-1973)*. (Application for copies should be addressed to American National Standards Institute, Inc., 1430 Broadway, New York, NY 10018.)

[ANSI 76]: American National Standards Institute, *Recorded Magnetic Tape for Information Interchange (6250 CPI, Group-coded Recording) (ANSI Standard x3.54-1976)*. (Application

Downloaded from http://www.everyspec.com

for copies should be addressed to American National Standards Institute, Inc., 1430 Broadway, New York, NY 10018.)

[ANSI 78]: American National Standards Institute, *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.)

[ANSI 79]: American National Standards Institute, *Additional Controls for Use with American National Standard Code for Information Interchange (ANSI Standard X3.64-1979)*. (Application for copies should be addressed to American National Standards Institute, Inc., 1430 Broadway, New York, NY 10018.)

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

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

[ISO 76a]: *ISO 1863, Information Processing - 9 track, 12,7 mm (0.5 in) wide magnetic tape for information interchange recorded at 32 rpmm (800 cpi)*. (Application for copies should be addressed to International Standards Organization (ISO), 1 rue de Varembe, Case Postale 56, GH12111 Geneva 20, Switzerland)

[ISO 76b]: *ISO 3788, Information Processing - 9 track, 12,7 mm (0.5 in) wide magnetic tape for information interchange recorded at 63 rpmm (1600 cpi) phase encoded*. (Application for copies should be addressed to International Standards Organization (ISO), 1 rue de Varembe, Case Postale 56, GH12111 Geneva 20, Switzerland)

[ISO 84]: *ISO 5652, Information Processing - 9 track, 12,7 mm (0.5 in) wide magnetic tape for information interchange - Format and recording using group coding at 246 cpmm (6250 cpi)*. (Application for copies should be addressed to International Standards Organization (ISO), 1 rue de Varembe, Case Postale 56, GH12111 Geneva 20, Switzerland)

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

[WEBS]: *Webster's Ninth New Collegiate Dictionary*; Merriam-Webster Inc., Springfield, Massachusetts, 1985.

(Nongovernment standards are generally available for reference from libraries. They are also distributed among nongovernment standards bodies and using Federal agencies.)

Downloaded from http://www.everyspec.com

2.3 Order of precedence

In the event of a conflict between the text of this standard and the references cited herein, the text of this standard shall take precedence.

2.4 Source of documents

Copies of listed military standards, specifications, and associated documents listed in the Department of Defense Index of Specifications and Standards, are available from the Department of Defense Single Stock Point, Commanding Officer, Naval Publications and Forms Center, 5801 Tabor Avenue, Philadelphia, PA 19120. Copies of industry association documents should be obtained from the sponsoring industry association. Copies of all other listed documents should be obtained from the contracting activity or as directed by the contracting officer.

Downloaded from http://www.everyspec.com

3. DEFINITIONS

The following is an alphabetical listing of terms which are used in the description of the CAIS. Where a document named in Section 2 was used to obtain the definition, the definition is preceded by a bracketed reference to that document. Definitions that have been interpreted and tailored to fit the CAIS include the phrase "in the CAIS" as part of that definition.

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

3.2 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. See also **access to a node**.

3.3 **access checking.** The act of determining the access rights, checking them against those rights required for the intended operation, and either permitting or denying the intended operation.

3.4 **access control.** [TCSEC] (1) discretionary access control: a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control). (2) mandatory access control: a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (i.e., clearance) of subjects to access information of such sensitivity. In the CAIS, access control refers to all the aspects of controlling access to information. It consists of access rights and access checking.

3.5 **access relationship.** A relationship of the predefined relation ACCESS.

3.6 **access right constraints.** The restrictions placed on certain kinds of operations by access control.

3.7 **access rights.** Descriptions of the kinds of operations that processes are allowed to perform on nodes.

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

3.9 **accessible.** A node is accessible if the current process as a subject has sufficient discretionary access rights to have knowledge of the existence of the node as an object and if mandatory access controls permit the current process as a subject to have knowledge of the existence of the node as an object. In the CAIS, a node is accessible if the current process has (adopted a role which has) been granted at least the access right EXISTENCE to that node and mandatory access control rules permit the process to have knowledge of the existence of the node.

Downloaded from http://www.everyspec.com

3.10 active position. The position at which an operation on a terminal device is to be performed.

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

3.12 adopt a role. The action of a process to acquire the access rights which have been, or will be, granted to adopters of that role; in the CAIS this is accomplished by establishing a secondary relationship of the predefined relation ADOPTED_ROLE from the process node to the group node representing the role.

3.13 advance the active position. Scroll, page or form terminal: Occurs whenever (i) the row number of a new position is greater than the row number of the old or (ii) the row number of the new position is the same and the column number of the new position is greater than that of the old.

3.14 approved access rights. Approved access rights are access rights whose names appear in the resulting_rights_list of any grant_item of those GRANT attribute values for which either (1) no necessary_right is given or (2) the necessary_right names an approved access right (under finite recursive application of this definition). See Appendix D for the definitions of resulting_rights_list, grant_item, and necessary_right.

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

3.16 attribute. A value named by an attribute name and associated with a node or relationship which provides information about that node or relationship such as the kind of a node or the invocation parameters of a process. This value is a list.

3.17 attribute iterator. See iterator.

3.18 base node. A node that serves as the starting point, usually for a path element or a pathname.

3.19 canonical list text representation. The result of transforming a list representation to list text.

3.20 closed node handle. A node handle that is not associated with a particular node. A closed node handle cannot be used to access any node.

3.21 contents. A file or process associated with a CAIS node. In the CAIS, a node is said to contain its contents.

Downloaded from http://www.everyspec.com

3.22 copy queue. A queue with initial contents that are the same as the contents of another file in which all write operations append information to the end of the queue and all read operations are destructive.

3.23 coupled file. A secondary storage file (containing either text or sequential elements) used to initialize a mimic queue file and which is mutually dependent upon the mimic queue file.

3.24 current job. The process node tree containing the current process node; the root process node of this tree is the target node of a secondary relationship of the predefined relation CURRENT_JOB.

3.25 current linear list. A linear list, within a list structure, to which the linear list manipulations implicitly refer. Exactly one current linear list is associated with each LIST_TYPE value.

3.26 current node. The node that is currently the focus or context for the activities of the current process; this node is the target node of a secondary relationship of the predefined relation CURRENT_NODE.

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

3.28 current user. The user's top-level node; it is the target node of the secondary relationship of the predefined relation CURRENT_USER.

3.29 default group node. A group node that is the target of a secondary relationship of the predefined relation DEFAULT_ROLE from either (i) a top-level user node or (ii) a node that represents the executable image of a program. No node may have multiple default group nodes.

3.30 dependent process. A process other than a root process.

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

3.32 device file. An external file that represents a device; in the CAIS, the predefined device files are magnetic tape drive files and terminal files.

3.33 device name. The key of a primary relationship of the predefined relation DEVICE.

3.34 discretionary access control. See access control.

3.35 element. (of a file) A value of the generic data type with which the input and output package was instantiated (see [1815A] 14.1, 14.2, 14.2.2, and 14.2.4).

Downloaded from http://www.everyspec.com

3.36 **empty list.** A linear list that contains no items. It is not considered to be either a named list or an unnamed list.

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

3.38 **external file.** (see [1815A] 14.1(1) based on the definition of 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.

3.39 **file.** See external file.

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

3.41 **file node.** A node that contains an Ada external file, e.g., a host system file, a device, or a queue.

3.42 **form.** A two-dimensional matrix of character positions.

3.43 **group.** A set of users; in the CAIS, a group is represented by a group node.

3.44 **group name.** The key of a primary relationship of the predefined relation GROUP emanating from the system-level node or of a secondary relationship of the predefined relation GROUP emanating from a process node.

3.45 **group node.** A structural node representing a group. This node may have emanating relationships of the predefined relations POTENTIAL_MEMBER and DOT to other group nodes.

3.46 **identification of a node.** A means to identify a node; in the CAIS, identification of a node is provided either by a pathname or by specifying a base node and the identification of a relationship emanating from the base node by means of its relation name and a relationship key designator.

3.47 **identification of a relationship.** A means to identify a relationship; in the CAIS, identification of a relationship is provided by specifying the base node from which it emanates, its relation name and its relationship key in terms of a relationship key designator.

3.48 **identifier text.** An external representation of an identifier value of a list item.

3.49 **illegal identification.** A node identification in which the pathname or the relationship key or relation name is syntactically illegal with respect to the syntax defined in Table I (Pathname BNF), page 32.

Downloaded from http://www.everyspec.com

3.50 **inaccessible.** A node is inaccessible if the current process node 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.

3.51 **inheritable.** A property of a secondary relationship that describes whether or not it is copied when the node from which it emanates is copied. Also, a property of a secondary relationship emanating from a creating process node that describes whether or not it is copied, upon process creation, to emanate from a newly created process node. In the CAIS, a boolean predefined attribute INHERITABLE on relationships establishes whether or not the relationships are inheritable.

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

3.53 **initiated process.** The process whose program has been placed into execution.

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

3.55 **interface.** [DACS] A shared boundary.

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

3.57 **interoperability.** The ability of APSEs to exchange database objects and their relationships in forms usable by tools and user programs without conversion.

3.58 **item name.** A name that may be associated with a list item.

3.59 **item value.** A value associated with a list item. There are five kinds of values that an item can have: string, integer, floating point number, identifier and linear list.

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

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

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

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

Downloaded from http://www.everyspec.com

3.64 **linear list.** A linearly ordered set of data elements called list items.

3.65 **list.** [IEEE] An ordered set of items of data. In the CAIS, an entity of type LIST_TYPE whose value is a linearly ordered set of data elements or is empty.

3.66 **list item.** A data element in a list.

3.67 **magnetic tape drive file.** An external file that represents a magnetic tape drive.

3.68 **mandatory access control.** See access control.

3.69 **mimic queue.** A queue with initial contents that are the same as the contents of another secondary storage file and that is mutually dependent with that file, in which all write operations append information to the end of the file and queue and all read operations are destructive on the queue.

3.70 **name.** An identifier by which a thing is known.

3.71 **named item.** A list item that has a name associated with it, i.e., a list item that has an item name.

3.72 **named list.** A non-empty linear list that contains only named items.

3.73 **nested list structure.** A linear list together with all of its nested sublists (and all of their nested sublists, recursively including all the nested sublists).

3.74 **nested sublist.** A linear list item of a linear list is called a nested sublist of the linear list containing the list item.

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

3.76 **node handle.** An Ada object of type NODE_TYPE which is used to identify a CAIS node for access, deletion, or creation; it is internal to a process.

3.77 **node iterator.** See iterator.

3.78 **node kind.** A predefined attribute on every relationship indicating the kind of the target node; the value of the attribute is STRUCTURAL, PROCESS or FILE.

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

3.80 **nonsynchronous queue.** A queue which permits an implementation-dependent number of write operations to occur independently of any read operations on the queue.

3.81 **null key.** A single distinguished key represented by the empty string.

Downloaded from http://www.everyspec.com

3.82 object. [TCSEC] A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. In the CAIS, an object is any node to be accessed.

3.83 obtainable. A node is obtainable if it has been created and its primary relationship has not been deleted.

3.84 open file handle. A file handle that is associated with a particular file. A file handle that is not open cannot be used to access any file.

3.85 open node handle. A node handle that is associated with a particular node. A node handle that is not open cannot be used to access any node.

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

3.87 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.

3.88 path element. A portion of a pathname representing the traversal of a single relationship; it consists of a relation name and relationship key.

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

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

3.91 potential member. A group that may dynamically acquire membership in another group; in the CAIS, a group node is termed a potential member of a group if it is reachable from the node representing the group by traversing only relationships of the predefined relations DOT and POTENTIAL_MEMBER.

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

3.93 precede the active position. Scroll, page or form terminal: Occurs whenever (i) the row number of a new position is less than the row number of the old or (ii) the row number of the new position is the same and the column number of the new position is less than that of the old.

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

Downloaded from http://www.everyspec.com

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

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

3.97 process tree. For a given process, the set of processes consisting of the given process plus each process whose node's unique primary path traverses the node of the given process.

3.98 program. [1815A] A program is composed of a number of compilation units, one of which is a subprogram called the main program.

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

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

3.101 queue file. An external file that represents a sequence of information that is accessed in a first-in, first-out manner. There are three kinds of queue files in the CAIS: solo, copy, and mimic queue files.

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

3.103 relation name. The string that identifies a relation.

3.104 relationship. In the CAIS 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. A relationship is either a primary relationship or a secondary relationship.

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

3.106 relationship key designator. The way that relationship keys are designated to the interfaces. There are two forms of relationship key designators: an identifier (or the empty string), or the string "#", optionally preceded by an identifier prefix.

3.107 role. A role is associated with a group node. It is the set of all relationships of the predefined relation ACCESS (called access relationships) emanating from nodes and targeted at that group node or any of the group nodes reachable recursively from that group node by secondary relationships of the predefined relation PARENT.

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

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

Downloaded from http://www.everyspec.com

3.110 secondary storage file. An external file that represents a disk or other random access storage file.

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

3.112 solo queue. A queue, initially empty, in which all write operations append information to the end and all read operations are destructive.

3.113 source node. The node from which a relationship emanates.

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

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

3.116 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 any process (acting on behalf of a given user) performing an operation requiring access to an object.

3.117 suspend. To stop the execution of a process such that it can be resumed.

3.118 synchronization. The act of forcing all data written to the internal file identified by a scroll or page terminal file handle to be transmitted to the contents of the file node with which the terminal file handle is associated.

3.119 synchronous queue. A queue which contains no elements; a write operation on the queue is not completed until a corresponding read operation on the same queue has been completed.

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

3.121 target node. The node at which a relationship terminates.

3.122 task. [1815A] A task operates in parallel with other parts of the program.

3.123 terminal file. An external file that represents an interactive terminal device. There are three kinds of terminal files in the CAIS: scroll, page, and form terminals.

3.124 termination of a process. Termination (see [1815A] 9.4) of the execution of the subprogram which is the main program (see [1815A] 10.1) of the process.

Downloaded from http://www.everyspec.com

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

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

3.127 tool sets. [STONEMAN] Groups of tools that are composed of a number of independent but interrelated programs (such as a debugger which is related to a specific compiler).

3.128 top-level node. A node whose parent is the system-level node; may be a structural node representing a user or group or a file node representing a device.

3.129 track. An open node handle or secondary relationship is guaranteed always to refer to the same node, regardless of any changes to relationships that could cause pathnames to become illegal or to refer to different nodes. An open node handle is said to **track** the node to which it refers. Similarly, secondary relationships **track** their target nodes.

3.130 transportability. The ability of a tool to be installed on a different Kernel Ada Programming Support Environment (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.

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

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

3.133 unadopt a role. The action of a process to disown any of its adopted roles, excepting the role of the current user; in the CAIS this is accomplished by deleting a secondary relationship of the predefined relation ADOPTED_ROLE from a process node to a group node representing the role.

3.134 undefined token. A distinguished value of a variable of type TOKEN_TYPE that represents an undefined value for that variable.

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

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

3.137 unnamed item. A list item that has no name associated with it, i.e., a list item that has no item name.

Downloaded from http://www.everyspec.com

3.138 unnamed list. A non-empty linear list that contains only unnamed items.

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

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

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

Downloaded from http://www.everyspec.com

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

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

Appendix A provides descriptions of the entities in the CAIS which are predefined. Appendix B provides a set of the Ada package specifications which have been organized for compilation of the CAIS interfaces. The material contained in this Appendix is a mandatory part of the standard. Appendix C provides a list of all CAIS procedures and functions in order to allow the reader ready access to a description of a particular capability in the CAIS. Appendix D summarizes the syntax descriptions given throughout the document.

Appendix E describes what an implementation-defined replacement for Package CAIS_ ACCESS_CONTROL_MANAGEMENT as described in Section 5.1.4 should contain. The material contained in this Appendix is a mandatory part of the standard.

Appendix F describes how and where those aspects of a CAIS implementation which are implementation-dependent must be documented. The material contained in this Appendix is a mandatory part of the standard.

Examples contained in the document are not part of the standard.

4.2 Method of description

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

a. the syntax as defined by canonical Ada package specifications, and

b. the semantics as defined by the descriptions both of the general node model and of the particular packages and procedures.

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

Downloaded from http://www.everyspec.com

4.2.1 Allowable differences

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

a. The packages of a particular CAIS implementation are allowed to import additional library units; this may cause name conflicts with the names of library units required by otherwise legal and non-erroneous Ada programs. Furthermore, a particular CAIS implementation may extend the enumeration types in the package CAIS_DEVICES with additional enumeration literals; this may cause name conflicts by virtue of section 8.3 and 8.4 of [1815A]. Barring such name conflicts, however, any Ada program that is legal and not erroneous in the presence of the canonical package specifications as library units must be legal and not erroneous if the canonical packages are replaced by the packages of a particular CAIS implementation. [Note: It is recommended, although not required, that any Ada program that is illegal in the presence of the canonical package specifications as library units is also illegal if the canonical packages are replaced by the packages of a particular CAIS implementation.]

b. The CAIS interfaces provided by the subprograms declared in the packages of a particular CAIS implementation must have semantics whose effects are indistinguishable from those in the standard.

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

4.2.2 Semantic descriptions

The interface semantics are described in most cases through narrative. These narratives are divided into as many as five paragraphs and appear in the following format:

Purpose:

This paragraph describes the semantics of the interface.

Parameters:

PARAMETER briefly describes each of the parameters.

Exceptions:

EXCEPTION briefly describes the conditions under which each exception is raised.

Downloaded from http://www.everyspec.com

Additional Interfaces:

```
-- In cases where an interface is overloaded and the additional
-- versions can be described in terms of the basic form of the
-- interface and other CAIS interfaces, these versions are
-- described in this paragraph using Ada. This method of
-- presenting the semantics of the Additional Interfaces is a
-- conceptual model. It does not imply that the Additional
-- Interfaces must be implemented in terms of the existing ones
-- exactly as specified, merely that their behavior is equivalent
-- to such an implementation. The semantics described in the
-- Purpose, Parameters and Exceptions paragraphs apply only to
-- the principal interface; the Additional Interfaces may have
-- additional semantics as implied by the given bodies.
```

Notes:

Any relevant information that does not fall under one of the previous four headings is included in this paragraph.

4.2.3 Typographical conventions

This document follows the typographical conventions of [1815A] where these are not in conflict with those of [962A]. In particular:

a. **boldface** type is used for Ada language reserved words,

b. UPPER CASE is used for Ada language identifiers which are not reserved words,

c. in the text, syntactic category names are written in normal typeface with any embedded underscores removed,

d. in the text, where reference is made to the actual value of an Ada variable (for example, a procedure parameter), the Ada name is used in normal typeface. However, where reference is made to the Ada object itself (see [1815A] 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 [1815A] 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.

Downloaded from http://www.everyspec.com

4.3 CAIS node model

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

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

Both nodes and relationships possess attributes. Attributes of nodes describe properties of the entities represented by the nodes; attributes of relationships describe properties of the interrelations represented by the relationships as well as the kind of the target node.

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; a node is said "to contain" its contents. If a node is a *file node*, it contains an Ada external file. There are four kinds of CAIS supported Ada external files: secondary storage (represented by a host file), queue (used for interprocess communication), terminal, and tape drive. If a node is a *process node*, it contains a representation of the execution of an Ada program. If a node is a *structural node*, it has no contents and the node is used strictly as a holder of relationships and attributes. The *kind* of a node is a predefined and implicitly established attribute on every relationship which points to the node. Nodes can be created, renamed, accessed (as part of other operations), and deleted.

4.3.2 Processes

A *process* is the CAIS mechanism used to represent the execution of an Ada program. A process is represented as the contents of a process node. 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 initial execution of the program are allocated to the process, and execution is started. It is possible for a process to request additional resources after execution has begun. 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 [1815A] 9(5):

Downloaded from http://www.everyspec.com

> Parallel tasks (parallel logical processors) may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can detect that the same effect can be guaranteed if parts of the actions of a given [Ada] task are executed by different physical processors acting in parallel, it may choose to execute them in this way; in such a case several physical processors implement a single logical processor.

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

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

4.3.3 Input and output

Ada input and output in Chapter 14 of [1815A] involves the transfer of data to and from Ada external files. The underlying model for the contents of a file node is that of a file of data items, accessible either in a sequential order or in a random order (i.e., directly by some index) through the packages specified in Section 5.3. These file nodes may represent disk or other secondary storage files, queues, or other devices. Devices supported by the CAIS include magnetic tape drives and terminals. CAIS file nodes contain Ada external files and also represent information about them.

4.3.4 Relationships and relations

The relationships of CAIS nodes form the edges of a directed graph. 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 or the kind of its target node.

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

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

Nodes in the environment are attainable by following relationships. Operations are provided

Downloaded from http://www.everyspec.com

to *traverse a relationship*, that is, to follow a relationship from its source node to its target node.

4.3.4.1 Kinds of relationships

There are two kinds of relationships: primary and secondary. When a node is created, an initial relationship is established from some other node to the newly created node. This initial relationship is called the *primary relationship* to this new node; it is the only primary relationship of which the new node is a target node. The source node of this initial relationship is called the *parent*. In addition, the new node will be connected back to this parent via a relationship of the predefined relation PARENT. There is no requirement that all primary relationships emanating from a node have the same relation name. Primary relationships form a strictly hierarchical tree; that is, for every node (except the root) there is one and only one sequence of primary relationships leading to it from the node that is the root of the tree. No cycles can be created in this tree using only primary relationships. A node is *obtainable* if it has been created and its primary relationship has not been deleted.

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

A *secondary relationship* is an arbitrary connection which may be established between two existing nodes; secondary relationships may form an arbitrary directed graph. User-defined secondary relationships are created with the CREATE_SECONDARY_RELATIONSHIP procedure and deleted with the DELETE_SECONDARY_RELATIONSHIP procedure. Secondary relationships may exist to unobtainable nodes since a secondary relationship to the node could have existed before the deletion of its primary relationship and that deletion does not affect the secondary relationship. A RENAME operation has no effect on any secondary relationships which have the renamed node as a target node, i.e., the secondary relationships still *track* the renamed node (see Section 5.1.2.22, page 92). It is a general underlying principle of the CAIS that CAIS-internal node identifications used to implement relationships are unique and persist over time, even if the respective nodes are deleted.

Certain secondary relationships are *inheritable*. Upon copying a node (see Section 5.1.2.20, page 87 and Section 5.1.2.21, page 89), inheritable relationships emanating from the node are copied to emanate from the copied node. Similarly, upon creation of a dependent process node (see Section 5.2, page 168), inheritable relationships emanating from the node of the creating process are copied to emanate from the dependent process node. Thus, an inherited relationship is unaffected by any changes to the original relationship and its attributes. Primary relationships are never inheritable. Secondary relationships of user-defined relations can, upon creation or by subsequent calls on the interface SET_INHERITANCE, be specified to be inheritable or not inheritable. Secondary relationships of several relations predefined in the CAIS are never inheritable; these relations are EXECUTABLE_IMAGE, MIMIC_FILE, DEFAULT_ROLE, STANDARD_INPUT, STANDARD_OUTPUT, STANDARD_ERROR and CURRENT_NODE. For the last four of these relations a mechanism equivalent to

Downloaded from http://www.everyspec.com

inheritance is provided by means of default parameters in the interfaces which create process nodes (see Section 5.2.2, page 172). Secondary relationships of several relations predefined in the CAIS are always inheritable; these relations are USER, DEVICE, GROUP, CURRENT_JOB and CURRENT_USER. All other secondary relationships of relations predefined in the CAIS are created as inheritable relationships; however, the inheritance property can be subsequently changed by calls on SET_INHERITANCE.

4.3.4.2 Basic predefined relations

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

The CAIS node model incorporates the concept of a system, of which an example is shown in Figure 1. This concept provides the means of administering all the entities represented within one CAIS implementation. This concept implies the existence of a *system-level node* which acts as the root of the CAIS primary relationship tree spanning the entire node structure. The system-level node cannot be accessed explicitly by the user via the CAIS interfaces. It may only be manipulated by interfaces outside the CAIS.

A *top-level node* is one whose parent is the system-level node. There are three kinds of top-level nodes: user and device nodes (explained below) and group nodes (explained in Section 4.4).

The CAIS node model incorporates the concept of a user. A *user* may be an individual, project, or other organizational entity; this concept is not equated with only an individual person. Each user has one top-level node. This top-level node is a structural node which represents the user and from it the user can access other structural, file and process nodes. Each user node is reachable from the system-level node along a primary relationship of the predefined relation USER emanating from the system-level node. The key of this relationship is the *user name.* Each user name has a top-level user node associated with it. The CAIS does not define interfaces for creating nodes which represent users; such interfaces are to be provided outside the CAIS.

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

Figure 2 shows an example of a hierarchical tree formed by primary relationships. The top-level user node that is the target node of the primary relationship 'USER(JONES) has two relationships emanating from it. The relationship 'DOT(LOGIN_SCRIPT) has a file node as its target node and the relationship 'DOT(TRACKER) has a structural node as its target node. Both of these relationships are primary relationships of the (default) relation DOT (see Section 4.3.5, page 29). The primary relationship 'DOT(LANDING_SYSTEM) emanates

Downloaded from http://www.everyspec.com

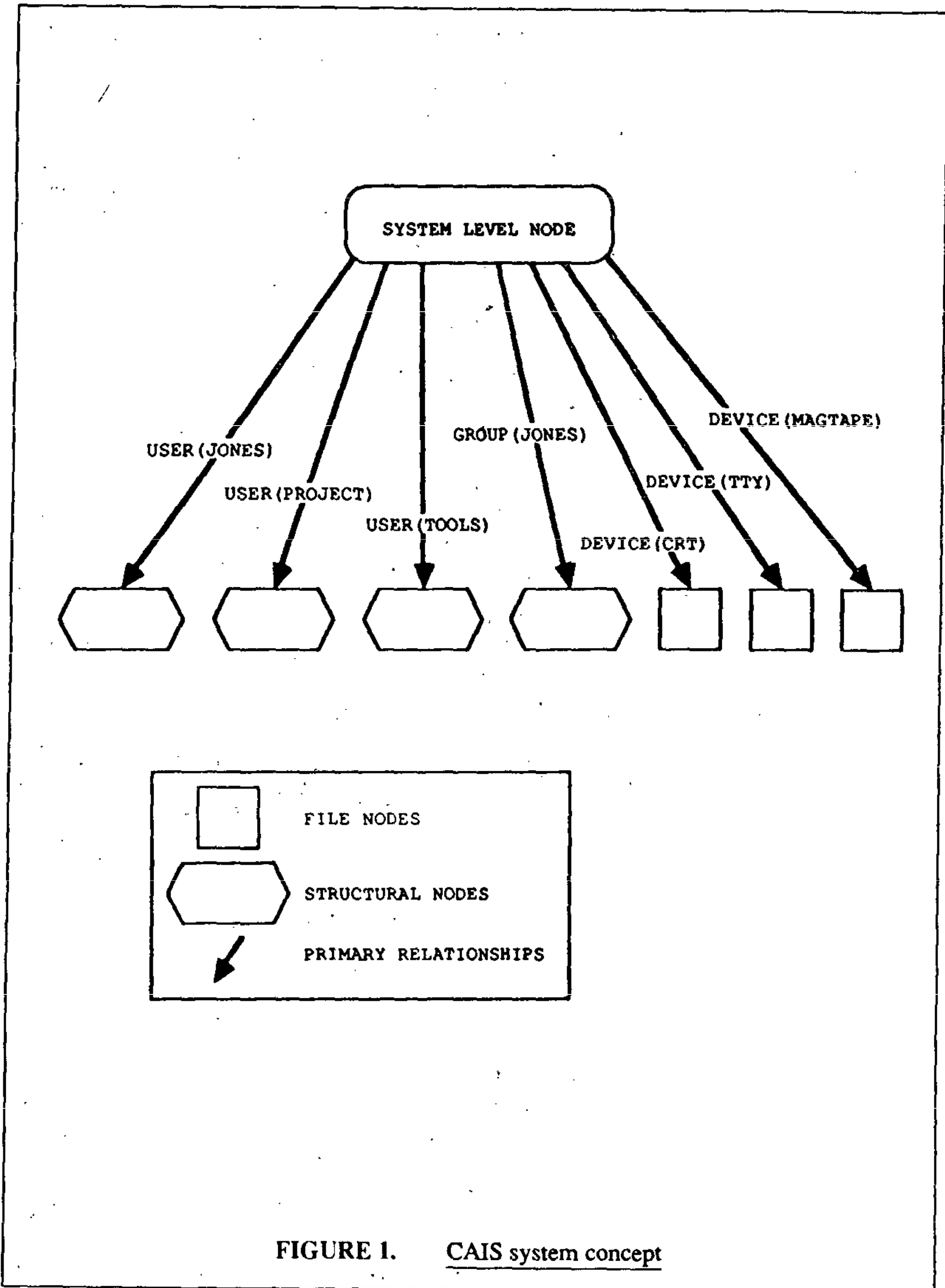

FIGURE 1. CAIS system concept

Downloaded from http://www.everyspec.com

from the structural node identified by 'USER(JONES)'DOT(TRACKER) and has a structural node as its target node. The relationship of the user-defined relation WITH_UNIT with relation key RADAR emanates from this structural node and has a file node as its target node. The top-level user node identified by the relationship 'USER(TOOLS) has several primary relationships of the (default) relation DOT emanating from it. The target nodes identified by two of these relationships, 'DOT(EDIT) and 'DOT(CLI), are file nodes. These file nodes might contain executable images of programs, EDIT and CLI. In addition, another structural node identified by 'USER(TOOLS)'DOT(TRACKER) is the target node of a primary relationship of the (default) relation DOT which emanates from the structural node identified by 'USER(TOOLS). The file node identified by 'USER(TOOLS)'DOT (TRACKER)'DOT(SIMULATOR) may have an executable image of a project-specific tool, i.e., specific to the TRACKER project. The top-level structural node 'GROUP(PROJECT) is a group node (see Section 4.4.2.1) and the top-level device node 'DEVICE(CRT) is a file node.

The CAIS node model incorporates the concept of a job. When a user logs onto the APSE or calls the CREATE_JOB procedure (see Section 5.2.2.4, page 185), a *root process node* is created which often represents a command interpreter or other user-communication process. It is left to each CAIS implementation to set up a methodology for users to log onto the APSE and for enforcing any constraints that limit the top-level user nodes at which users may log on. After logging onto the APSE, the user will be regarded by the CAIS as the user associated with the top-level user node at which he logged on. A process node tree, spanned by primary relationships, develops from the root process node as other processes (called *dependent processes*) are initiated for the user. A particular user may have several root process nodes concurrently. Each corresponding process node tree is referred to as a *job*. The predefined JOB relation is provided for locating each of the root process nodes from the user's top-level node. A primary relationship of the predefined relation JOB emanates from each user's top-level node to the root process node of each of the user's jobs. The key of this relationship is assigned by the mechanism of interpreting the LATEST_KEY constant (see Section 4.3.5) unless otherwise specified in the CREATE_JOB procedure call.

While the CAIS does not specify an interface for creating the initial root process node when a user logs onto the APSE, the effect is to be the same as a call to the CREATE_JOB procedure. The secondary relationships which the implementation must establish are found in Section 5.2.2 (page 172). In particular, secondary relationships of the predefined relations USER and DEVICE must be established, with the appropriate user and device names as keys. These relationships emanate from the root process node being created to an implementation-defined subset of top-level user nodes and file nodes representing users and devices, respectively. Dependent process nodes in the job inherit these relationships. File nodes representing devices and top-level nodes of other users can be reached from a process node via these secondary relationships of the relation DEVICE or USER and a relationship key which is interpreted as the respective device or user name.

CURRENT_JOB, CURRENT_USER, and CURRENT_NODE are predefined relations which provide a convenient means for identifying other CAIS nodes. These relationships emanate from each process node. The relationship of the predefined relation CURRENT_JOB always points to the root process node of a process node's job, i.e., that process node's *current job*. The relationship of the predefined relation CURRENT_USER always points to the user's top-level node, i.e., the *current user*. The relationship of the predefined relation CURRENT_NODE can be used to point to a node called the *current node* which represents

Downloaded from http://www.everyspec.com

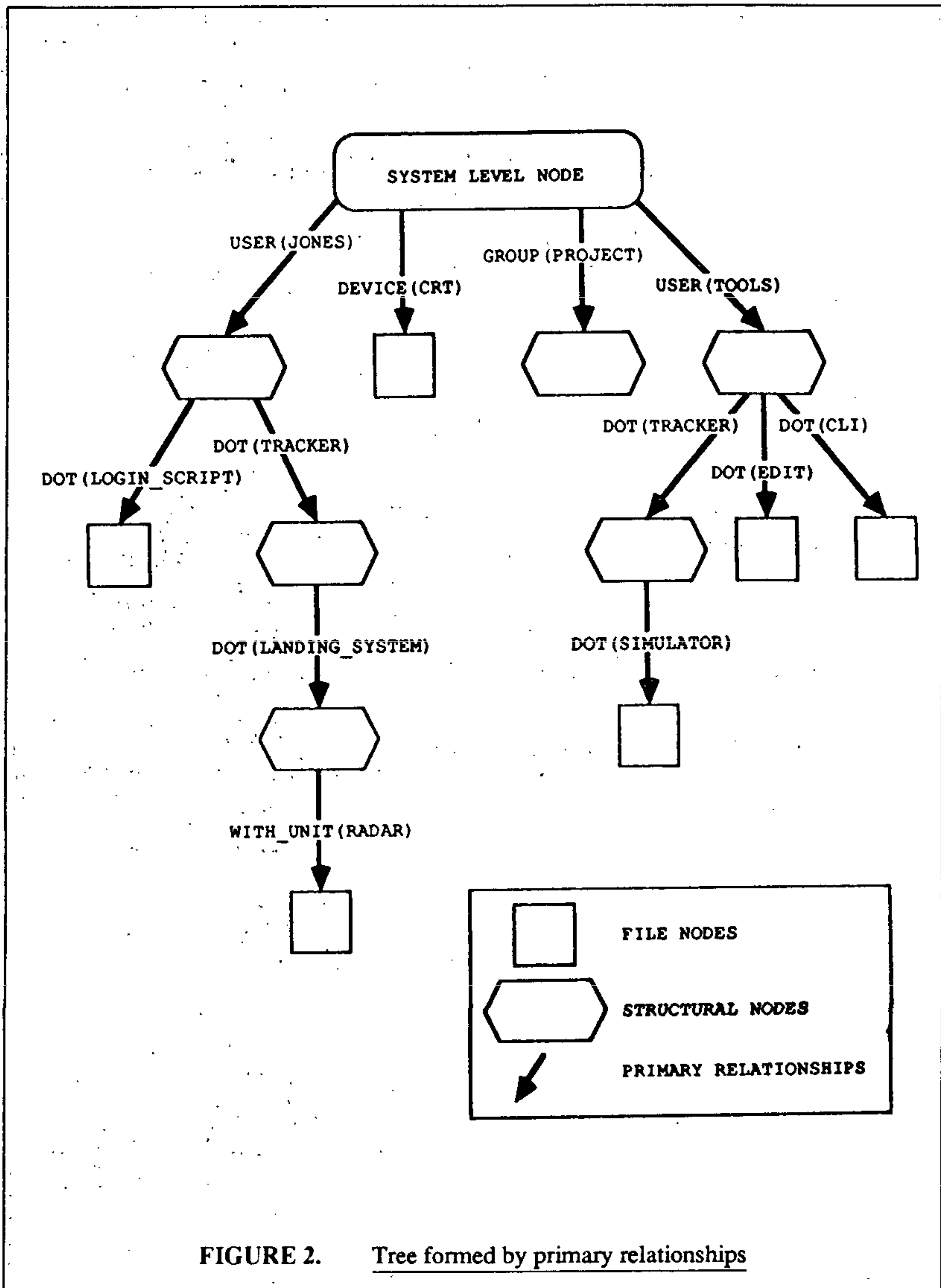

FIGURE 2. Tree formed by primary relationships

Downloaded from http://www.everyspec.com

the current focus of the process or the context for its activities. The process node can thus use the current node for a *base node*, i.e., a starting point, when specifying pathnames (see Section 4.3.5). The CAIS requires that, when a root process node is created when the user logs onto the APSE, it has a secondary relationship of the predefined relation CURRENT_NODE pointing to the top-level node for the user. Figure 3 shows an example of some of these predefined relations.

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

4.3.4.3 Relation names and relationship keys

Relation names have the syntax of Ada identifiers. Relationship keys also have the syntax of Ada identifiers. In addition, there is a single, distinguished key called the *null key* which is represented by the empty string. Within such identifiers upper and lower case are treated as equivalent. Interfaces which return such identifiers must return all alphabetic characters in upper case.

Relations are designated to the interfaces by their names. These names are passed to CAIS interfaces via parameters of the subtype RELATION_NAME (section 5.1.1, page 54) or included in pathnames. Relationships are designated by relation names and relationship keys. Relationship keys are designated to the interfaces by a *relationship key designator*. Relationship key designators are passed to CAIS interfaces via parameters of subtype RELATIONSHIP_KEY (section 5.1.1, page 54) or included in pathnames. Relationship key designators have two forms: an identifier (or the empty string), or the string "#", optionally preceded by an identifier prefix. The relationship key designator "#" is referred to as the *latest key*. A relationship key designator of the first form (an identifier or the empty string) specifies the relationship key. When selecting a relationship, a relationship key designator of the second form specifies the relationship key of an existing relationship, lexicographically last in the sequence of all keys of relationships of the same relation emanating from that node which begin with the given prefix. When creating a node or relationship, a relationship key designator of the second form causes a relationship key to be automatically assigned. This relationship key will lexicographically follow all existing relationship keys of relationships of the same relation emanating from that node which begin with the given prefix.

4.3.5 Paths, pathnames and node identification

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

Paths are specified by *pathnames*. When a pathname is supplied to or returned from a CAIS interface, it always refers to a path which starts at the current process node. Starting from this node, the path specified by the pathname is followed by traversing a sequence of relationships until the desired node is reached. The pathname for this path is made up of the

Downloaded from http://www.everyspec.com

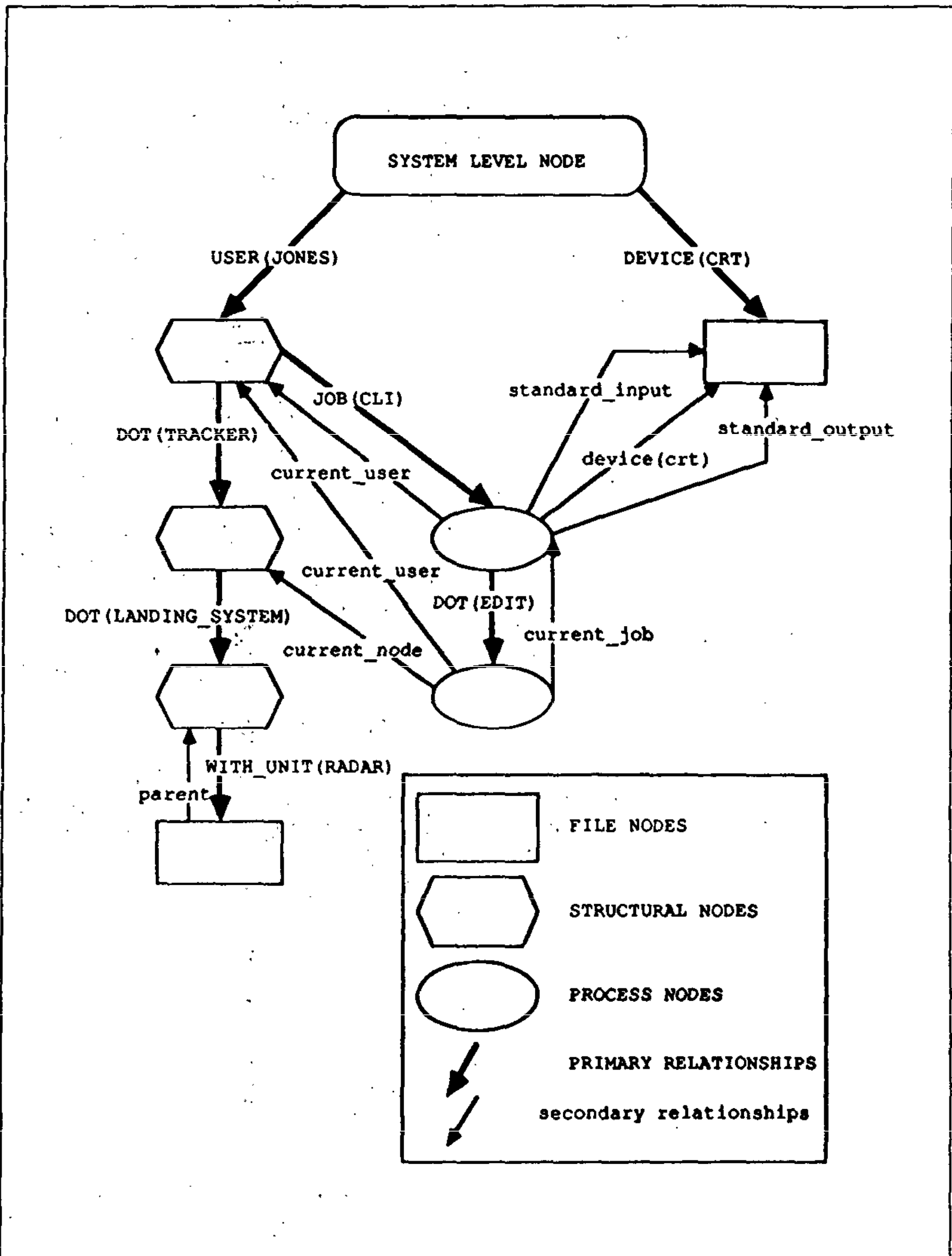

FIGURE 3. Some predefined relations

Note: The relation WITH_UNIT is not predefined.

Downloaded from http://www.everyspec.com

concatenation of *path elements* each of which identifies the traversed relationships in the same order in which they are traversed.

The syntax of a path element is an apostrophe (pronounced ''tick'') followed by a relation name and a parenthesized relationship key designator. If the relationship key designator of a path element is the null key, the parentheses may be omitted. Thus, 'PARENT and 'PARENT() identify the same relationship.

The CAIS predefines the relation DOT; it allows for an abbreviated form of a path element. If the relation name in a path element is DOT, then the path element may be represented simply by a dot (''.'') followed by the relationship key designator. Thus, 'DOT(TRACKER) is the same as .TRACKER. DOT has no CAIS-specific semantics other than this abbreviation quality, except when it is used in a group tree where it has properties similar to any other predefined relationship (section 4.4.2.1, page 36). Relationship keys of relationships of the DOT relation must not be the empty string. Instances of the DOT relation may be manipulated by the user within access right constraints. Relationships of the DOT relation are not restricted to be primary relationships.

A pathname may begin simply with a relationship key designator, not prefixed by either an apostrophe or period. This is an abbreviation for a pathname obtained by prefixing the given pathname with 'CURRENT_NODE., i.e., the interpretation follows the relationship of the predefined relation CURRENT_NODE and then the relationship of the predefined relation DOT with the given key. Thus LANDING_SYSTEM is the same as 'CURRENT_NODE.LANDING_SYSTEM.

A pathname may also be a '':''. This refers to the current process node.

For example, in Figure 3, all of the following are legal pathnames within the context of the process node identified by 'USER(JONES)'JOB(CLI).EDIT. They all refer to the same node, since the relationship of the predefined relation CURRENT_NODE points to the same node as 'USER(JONES).TRACKER and the relationship of the predefined relation CURRENT_USER points to the same node as 'USER(JONES):

LANDING_SYSTEM'WITH_UNIT(RADAR)

'USER(JONES).TRACKER.LANDING_SYSTEM'WITH_UNIT(RADAR)

'CURRENT_USER.TRACKER.LANDING_SYSTEM'WITH_UNIT(RADAR)

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

Identification of a node is provided either by a pathname or by specifying a base node and the identification of a relationship emanating from the base node by means of its relation name and a relationship key designator. The phrase ''to identify a node'' means to provide an identification for a node. A node identification is considered an *illegal identification* if either the pathname or the relationship key designator or the relation name is syntactically illegal with respect to the syntax defined in Table I.

Downloaded from http://www.everyspec.com

Identification by pathname implies *traversal of a node* if a relationship emanating from the node is traversed; consequently all nodes on the path to a node are traversed, while the node at the end of the path is not traversed. An identification that would require traversal of an unobtainable or inaccessible (see Section 4.4.1, page 36) node is treated as the identification for a non-existing node.

Identification of a relationship is provided by specifying the base node from which it emanates, its relation name and its relationship key in terms of a relationship key designator.

TABLE I. Pathname BNF

pathname	::=	relationship_key_designator{path_element} | path_element{path_element} | :
path_element	::=	'relation_name [([relationship_key_designator])] | .relationship_key_designator
relation_name	::=	identifier
relationship_key_designator	::=	relationship_key | [identifier_prefix] #
relationship_key	::=	identifier
identifier_prefix	::=	letter { [underline] letter_or_digit } [underline]

See Appendix D for a description of the notation used.

4.3.6 Attributes

Both nodes and relationships may have attributes which provide information about the node or relationship. Attributes are identified by an attribute name. Each attribute has a name and a value. The value is a list represented using the CAIS_LIST_MANAGEMENT type called LIST_TYPE (see Section 5.4.1).

Attribute names follow the syntax of an Ada identifier. There is no requirement that relation names and attribute names be different from each other. Interfaces which return attribute names must return any alphabetic characters in upper case.

Downloaded from http://www.everyspec.com

The user can create and delete user-defined attributes as well as manipulate their values by means of the interfaces specified in Section 5.1.3, page 123.

4.3.6.1 Predefined attributes

The CAIS predefines several attributes on nodes and relationships, as well as the nature of the values of these attributes in terms of appropriate Ada types, e.g., enumeration types. Predefined attributes cannot be created, modified or deleted by the user, except where explicitly noted in the CAIS specification.

Special interfaces are provided to retrieve the values of predefined attributes and update them as appropriate in terms of their respective Ada types. The predefined attributes can also be accessed by the interfaces provided for attributes in general; in this case, the attribute values are uniformly expressed in terms of values of type LIST_TYPE (see Section 5.4, page 419). The correspondence of values of type LIST_TYPE used in, or obtained from, the general attribute manipulation interfaces with the values used in, or obtained from, the special interfaces for predefined attributes is explained in Section 5.1.3, page 123.

In particular, the CAIS predefines the following attributes on all nodes: TIME_CREATED, TIME_RELATIONSHIP_WRITTEN, and TIME_ATTRIBUTE_WRITTEN. These attributes describe the date and time of the last activity on the node corresponding to the mnemonic name of the respective attribute. For details, refer to the interface descriptions for TIME_CREATED (see Section 5.1.2.39, page 119), TIME_RELATIONSHIP_WRITTEN (see Section 5.1.2.40, page 120) and TIME_ATTRIBUTE_WRITTEN (see Section 5.1.2.42, page 122) and to the package CAIS_CALENDAR (see Section 5.6, page 497).

An additional attribute on all nodes is predefined in the suggested model of mandatory access control described in Section 4.4.3: the attribute OBJECT_CLASSIFICATION describes the mandatory access control classification of the node as an object.

The CAIS predefines the following attributes on all relationships: NODE_KIND and INHERITABLE. The NODE_KIND attribute discriminates the kind of node to which the relationship is targeted, i.e., to a process, file or structural node. The boolean INHERITABLE attribute establishes whether or not the relationship is inheritable according to the rules given in Section 4.3.4.1, page 24.

All other predefined attributes are restricted to certain kinds of nodes or to relationships of certain relations. They are explained in the appropriate subsections of Section 5.

A summary of all predefined attributes and their possible values is provided in Appendix A. Figure 4 shows an example of some of these predefined attributes.

Downloaded from http://www.everyspec.com

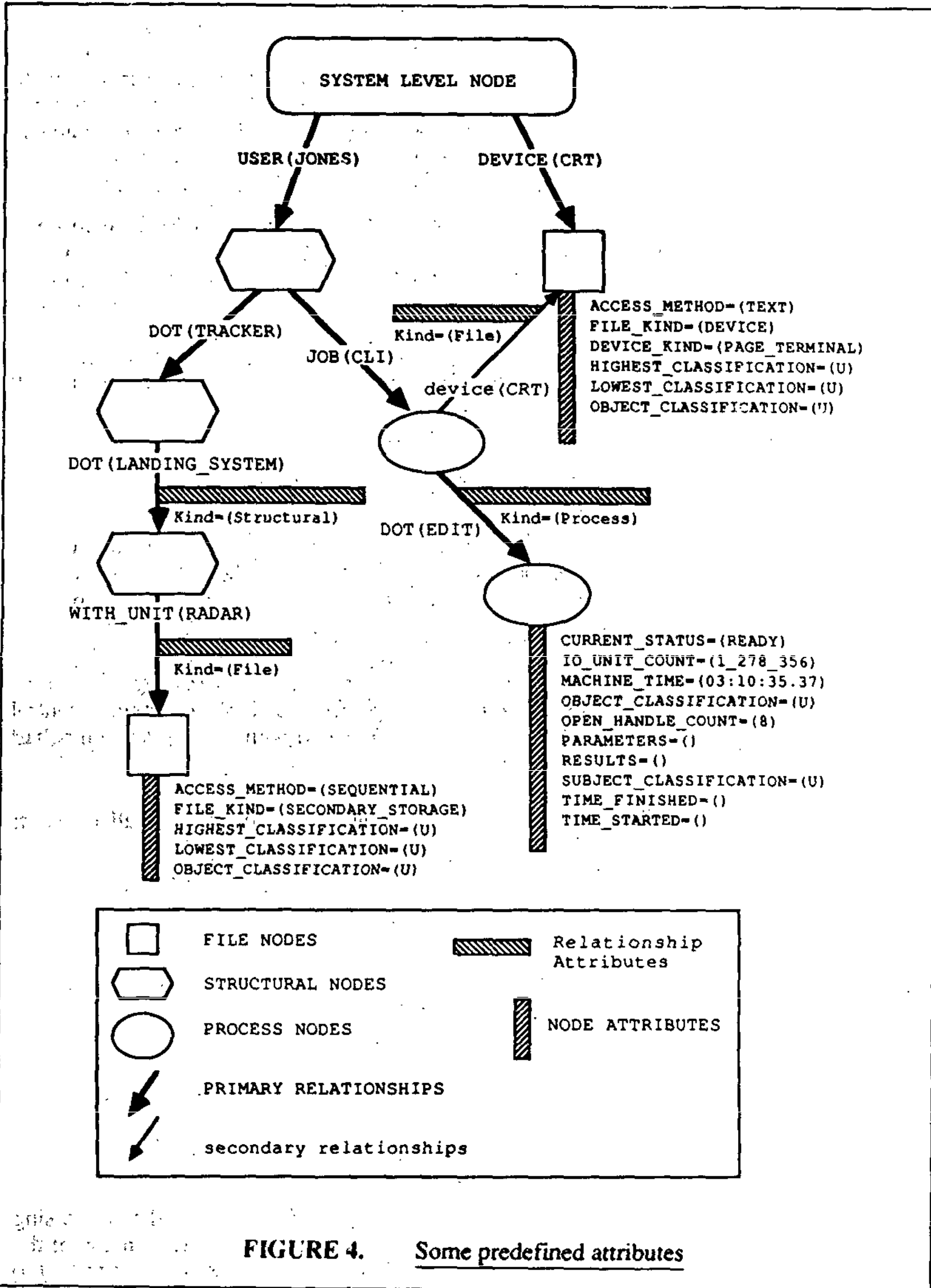

FIGURE 4. Some predefined attributes

Downloaded from http://www.everyspec.com

4.4 Discretionary and mandatory access control

The CAIS specifies mechanisms for discretionary and mandatory access control (see [TCSEC]). Alternate discretionary or mandatory access control mechanisms can be substituted by an implementation provided that the semantics of all interfaces in Section 5 (with the exception of Section 5.1.4) are implemented as specified. All alternate mechanisms as well as the implementation behavior of such a replacement package must be included in an implementer's CAIS reference manual as described in Appendix E of this document.

> *Access* is a specific type of interaction between a subject and an object that results in the flow of information from one to the other. A subject is 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. An object is a passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. [TCSEC]

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

a. *access rights* - Descriptions of the kinds of operations that processes are allowed to perform on nodes. The granting of these rights is explained in Section 4.4.2.3.

b. *access checking* - The act of determining the access rights, checking them against those rights required for the intended operation, and either permitting or denying the intended operation. These operations are described in Section 4.4.2.4 and Section 4.4.2.5.

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

The restrictions placed on certain kinds of operations by access control are called *access right constraints*.

4.4.1 Node access

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

a. reading or writing of the contents of the node,

b. reading or writing of attributes of the node,

c. reading or writing of relationships emanating from a node or of their attributes, and

d. traversing a node (see Section 4.3.5).

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

Downloaded from http://www.everyspec.com

In the CAIS, the following operations do not constitute access to a node: closing node handles to a node, opening a node handle with intent NO_ACCESS (see Section 5.1.2), reading or writing of relationships of which a node is the target node or of the attributes of such relationships, querying the kind of a node and querying the status of node handles to a node.

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

4.4.2 Discretionary access control

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

4.4.2.1 Groups and roles

The CAIS node model incorporates the concepts of groups and roles. A *group* is a set of users that is represented by a group node. Group membership of a subject determines the set of access rights it has or can acquire with respect to objects. Each group is represented by a structural node, termed a *group node*. Group nodes are organized in group node trees. Each top-level group node is reachable from the system-level node along a primary relationship of the predefined relation GROUP emanating from the system-level node. The key of this relationship is the *group name*. The primary relationship of a group node other than a top-level group node must be of the predefined relation DOT emanating from another group node. Upon creation of a root process node, secondary relationships of the predefined relation GROUP are created emanating from the process node and targeted at an implementation-defined subset of top-level group nodes. The relationship keys of these relationships are the respective group names. Dependent process nodes inherit these relationships (see Table IX, page 173).

The CAIS associates a role with each group node. The *role* associated with a group node is the set of all relationships of the predefined relation ACCESS (called *access relationships*) emanating from nodes and targeted at this group node or any of the group nodes reachable recursively from this group node by secondary relationships of the predefined relation PARENT. These access relationships provide information from which access rights of a subject to an object are determined as further described in Section 4.4.2.4.

Secondary relationships of the predefined relation DEFAULT_ROLE are used to associate with group nodes all top-level user nodes and some nodes representing the executable images of programs. Each top-level user node has a secondary relationship of the predefined relation DEFAULT_ROLE emanating from it and targeted at a group node. A node which represents the executable image of a program may also be associated with a group node by means of a secondary relationship of the predefined relation DEFAULT_ROLE emanating from that node to the group node. This group node is termed the *default group node* of the user or program, respectively. No node may have multiple default group nodes.

Downloaded from http://www.everyspec.com

Group nodes may have secondary relationships of the predefined relation POTENTIAL_MEMBER emanating from them to other group nodes. A group node is termed a *potential member* of a group if it is reachable from the node representing the group by traversing only relationships of the predefined relations DOT and POTENTIAL_MEMBER. Figure 5 shows an example of a hierarchical group structure and its embedding in the node model.

Group nodes, their attributes and their emanating relationships cannot be modified by means of CAIS interfaces. It is left to each CAIS implementation to set up a methodology and to provide interfaces for the creation, modification or deletion of group nodes and for the creation, modification and deletion of the relationships of the predefined relation DEFAULT_ROLE, in particular for those emanating from nodes representing the executable image of a program. It is suggested that such interfaces not be generally available to the CAIS user community. The effects of the deletion of group nodes and of alterations of group memberships or of relationships of the predefined relation DEFAULT_ROLE on concurrently executing processes are implementation-defined.

4.4.2.2 Adopting a role

When a process *adopts* a particular role, that process acquires the access rights which have been, or will be, granted to adopters of that role. In the CAIS, a secondary relationship of the predefined relation ADOPTED_ROLE is created from the process node to the group node representing this role. The group node and the role associated with this group node are said to be adopted by that process. A process is said to execute ''under the authority of its adopted roles''.

Roles are adopted either implicitly at creation of the process node or explicitly. When a root process node is created, it implicitly adopts the role associated with the default group node of the current user. It also implicitly adopts the role associated with the default group node of the node containing the executable image of the program it is executing, if such a default group node exists. When any other process node is created, it implicitly adopts all adopted roles of its creating process and the role associated with the default group node of the node containing the executable image of the program it is executing, if such a default group node exists.

There may be multiple relationships of the predefined relation ADOPTED_ROLE emanating from a process node. Keys of relationships of the predefined relation ADOPTED_ROLE are implementation-defined when the relationship is created implicitly or are specified by the user as a parameter of the ADOPT_ROLE procedure when the relationship is created explicitly.

An executing process may explicitly adopt roles associated with certain group nodes in addition to its existing roles, using the ADOPT_ROLE procedure (Section 5.1.4.7, page 160). For a process to adopt a role associated with a given group node, some group node already adopted by the process must be a potential member of the group represented by this group node. Once a process has adopted a role, it executes under the authority of this role as well as any other roles it has adopted.

Similarly, a process may disown any of its adopted roles, excepting the role of the current user; the process is said to *unadopt a role*. When the process unadopts a role, using the UNADOPT_ROLE procedure (Section 5.1.4.8, page 162), the respective relationship of the

Downloaded from http://www.everyspec.com

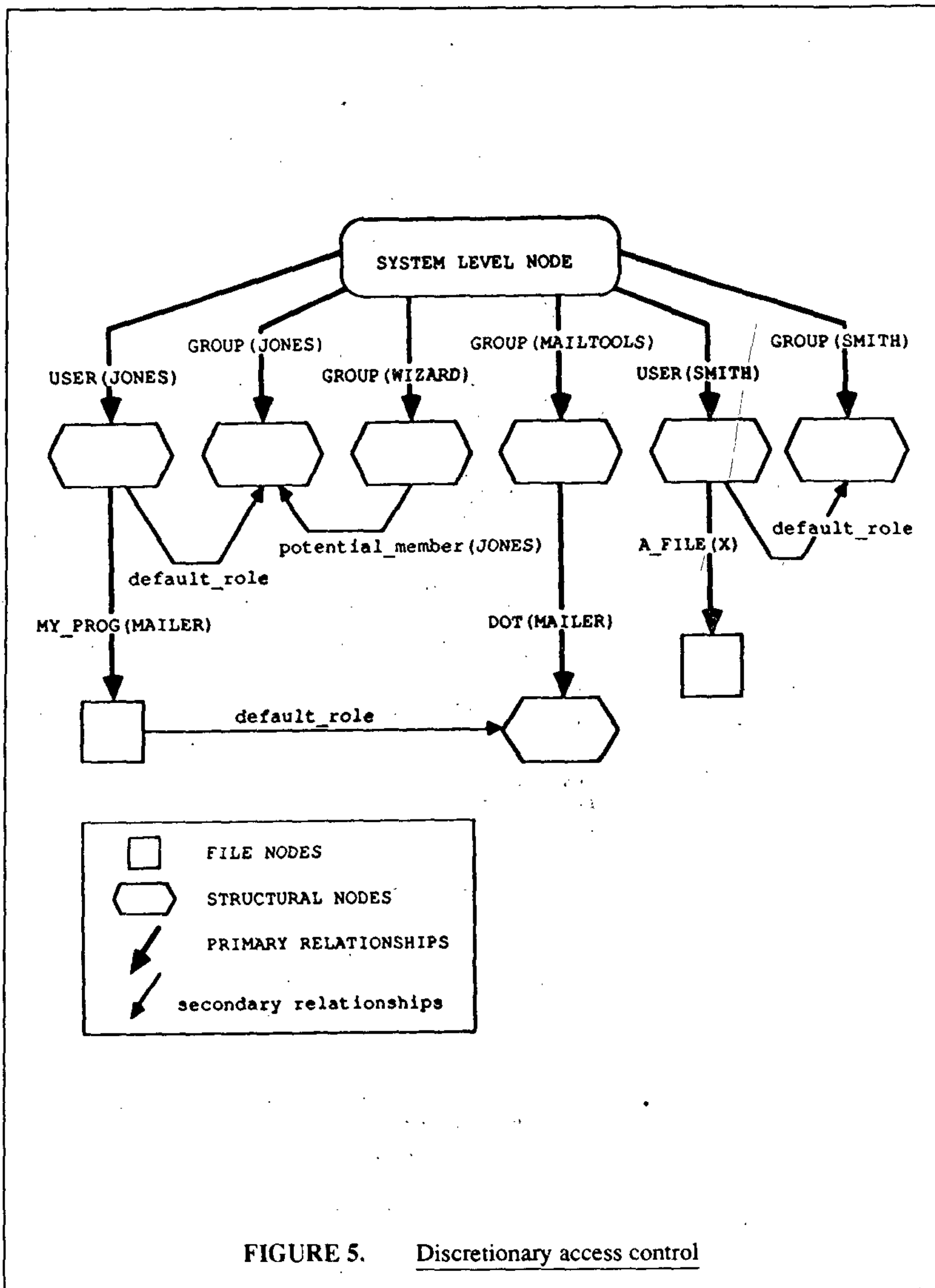

FIGURE 5. Discretionary access control

Downloaded from http://www.everyspec.com

predefined relation ADOPTED_ROLE is deleted. The process then executes under the authority of its remaining adopted roles.

Figure 6 depicts an example for the node model structure shown in Figure 5 after the creation of the process node by user JONES and the explicit adoption of 'GROUP(WIZARD) by this process.

Downloaded from http://www.everyspec.com

The relationship of the predefined relation POTENTIAL_MEMBER shown in Figure 6 is a prerequisite for user JONES to adopt the role 'GROUP(WIZARD). The relationship of the predefined relation ADOPTED_ROLE from 'JOB(MAILER) to 'GROUP(WIZARD) exists only if an explicit call was issued on the procedure ADOPT_ROLE (see Section 5.1.4.7, page 160) for the creation of this relationship.

4.4.2.3 Granting access rights

An object may be the source node of zero or more access relationships targeted at group nodes. Each access relationship has a predefined attribute, called GRANT, that specifies which access rights to the object are granted to processes (subjects) executing under the authority of roles that include this relationship.

Access relationships and their GRANT attributes are established for objects either at node creation or by using the interfaces provided in the package CAIS_ACCESS_CONTROL_MANAGEMENT.

At node creation, the DISCRETIONARY_ACCESS parameter of the node creation interface provides a set of access rights to be granted to the current user. A relationship of the predefined relation ACCESS with a GRANT attribute value given by the DISCRETIONARY_ACCESS parameter is established to the group node associated with the current user, i.e., the node identified by the pathname 'CURRENT_USER'DEFAULT_ROLE.

The SET_GRANTED_RIGHTS procedure (see Section 5.1.4.3, page 154) can be used by a process to establish an access relationship between an object node and a group node and to set the value of the GRANT attribute. This procedure can also be used to change the value of the GRANT attribute of an existing access relationship.

In order to limit the set of group nodes to which access relationships can be established, a CAIS implementation is allowed to restrict the set of group nodes which may be target nodes of access relationships to an implementation-defined subset which may depend on the process (subject) and its acquired roles. A violation of such restrictions causes the exception ACCESS_VIOLATION to be raised by the CAIS interfaces attempting to create the access relationship.

4.4.2.4 Determining access rights

The value of the GRANT attribute on access relationships must conform to the syntax of Table II, which corresponds to the syntax of named aggregates in Ada. The syntax is consistent with that given in Section 5.4. The interfaces in Section 5.4 can be used to construct and manipulate the value of the GRANT attribute.

Downloaded from http://www.everyspec.com

FIGURE 6. Discretionary access control after process creation

Note: The relationship keys on the relationships of the predefined relation ADOPTED_ROLE are omitted, although required by the CAIS node model, since these keys are not relevant to the discussion in this section.

Downloaded from http://www.everyspec.com

TABLE II. GRANT Attribute Value BNF

```
grant_attribute_value  ::= ( [ grant_item {, grant_item} ] )

grant_item             ::= ( [ necessary_right => ] resulting_rights_list )

necessary_right        ::= identifier

resulting_rights_list  ::= identifier
                           | ( identifier {, identifier} )
```

See Appendix D for a description of the notation used.

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

A node is *accessible* if the current process as a subject has sufficient discretionary access rights to have knowledge of the existence of the node as an object and if mandatory access controls permit the current process as a subject to have knowledge of the existence of the node as an object. In the CAIS, a node is accessible if the current process has (adopted a role which has) been granted at least the access right EXISTENCE to that node and mandatory access control rules permit the process to have knowledge of the existence of the node.

Determining the discretionary access rights that a given process (subject) has to a given object involves: (1) all adopted roles under which the current process is executing and (2) the GRANT attributes of the subset of the access relationships comprising these roles and emanating from the node representing the object.

The values of the GRANT attributes of these access relationships are used to determine the set of approved access rights. *Approved access rights* are access rights whose names appear in the resulting_rights_list of any grant_item of these GRANT attribute values for which either (1) no necessary_right is given or (2) the necessary_right names an approved access right (under finite recursive application of this definition).

Figure 7, Figure 8 and Figure 9 show an example for the relationships relevant to the determination of approved access rights. These figures are derived from Figure 6. In Figure 7, Figure 8 and Figure 9, keys on access relationships and on relationships of the predefined relation ADOPTED_ROLE are omitted where they are not needed for discussion. In Figure 7, the process 'USER(JONES)'JOB(MAILER) has implicitly adopted the roles 'GROUP (JONES) and 'GROUP(MAILTOOLS).MAILER. Since the latter is reachable via a primary relationship of the DOT relation from the group MAILTOOLS, any access relationships to

Downloaded from http://www.everyspec.com

'GROUP(MAILTOOLS) are automatically part of the role of 'GROUP (MAILTOOLS).MAILER. In addition, this process has adopted the role 'GROUP(WIZARD) explicitly.

TABLE III. Predefined Access Rights

Access Right	Explanation
EXISTENCE	The minimum access rights without which the object is inaccessible to the subject. Without additional access rights the subject may neither read nor write attributes, relationships or contents of the object. This access right is necessary to open the object with intent NO_ACCESS.
READ_RELATIONSHIPS	The subject may read attributes of relationships emanating from the object or use it for traversal to another node; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent READ_ RELATIONSHIPS.
WRITE_RELATIONSHIPS	The subject may create or delete relationships emanating from the object or may create, delete, or modify attributes of these relationships; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent WRITE_RELATIONSHIPS. This access right is sufficient to open the object with (exclusive or non-exclusive) intent APPEND_RELATIONSHIPS.
APPEND_RELATIONSHIPS	The subject may create relationships emanating from the object and attributes of these relationships; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent APPEND_ RELATIONSHIPS.
READ_ATTRIBUTES	The subject may read attributes of the object; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent READ_ ATTRIBUTES.
WRITE_ATTRIBUTES	The subject may create, write, or delete attributes of the object; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent WRITE_ATTRIBUTES. This access right is sufficient to open the object with (exclusive or non-exclusive) intent APPEND_ ATTRIBUTES.
APPEND_ATTRIBUTES	The subject may create attributes of the object; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent APPEND_ATTRIBUTES.
READ_CONTENTS	The subject may read contents of the object; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent READ_ CONTENTS.

Downloaded from http://www.everyspec.com

TABLE III. Predefined Access Rights -- Continued.

Access Right	Explanation
WRITE_CONTENTS	The subject may create, write, or delete contents of the object; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent WRITE_CONTENTS. This access right is sufficient to open the object with (exclusive or non-exclusive) intent APPEND_CONTENTS.
APPEND_CONTENTS	The subject may append contents of the object; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent APPEND_CONTENTS.
READ	This is the union of READ_RELATIONSHIPS, READ_ATTRIBUTES, READ_CONTENTS and EXISTENCE access rights. This access right is necessary to open the object with (exclusive or non-exclusive) intent READ. It is sufficient to open the object with (exclusive or non-exclusive) intent READ_RELATIONSHIPS, READ_ATTRIBUTES or READ_CONTENTS.
WRITE	This is the union of WRITE_RELATIONSHIPS, WRITE_ATTRIBUTES, WRITE_CONTENTS and EXISTENCE access rights. This access right is necessary to open the object with (exclusive or non-exclusive) intent WRITE. It is sufficient to open the object with (exclusive or non-exclusive) intent WRITE_RELATIONSHIPS, WRITE_ATTRIBUTES, WRITE_CONTENTS, APPEND_RELATIONSHIPS, APPEND_ATTRIBUTES and APPEND_CONTENTS.
APPEND	This is the union of APPEND_RELATIONSHIPS, APPEND_ATTRIBUTES, APPEND_CONTENTS and EXISTENCE access rights. This access right is necessary to open the object with (exclusive or non-exclusive) intent APPEND. It is sufficient to open the object with (exclusive or non-exclusive) intent APPEND_RELATIONSHIPS, APPEND_ATTRIBUTES or APPEND_CONTENTS.
EXECUTE	The subject may create a process that takes the contents of the object as its executable image; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with intent EXECUTE.
CONTROL	The subject may modify access control information of the object; the access right EXISTENCE is implicitly granted. This access right is necessary to open the object with (exclusive or non-exclusive) intent CONTROL.
ALL_RIGHTS	This access right is the union of all other predefined access rights. All access rights in this table are granted to the subject.

Downloaded from http://www.everyspec.com

Given a process node 'USER(JONES)'JOB(MAILER), an object 'USER(SMITH) 'A_FILE(X), five group nodes 'GROUP(JONES), 'GROUP(WIZARD), 'GROUP (MAILTOOLS), 'GROUP(MAILTOOLS).MAILER and 'GROUP(SMITH) representing roles (where 'GROUP(WIZARD) was adopted explicitly), the following relationships might exist as depicted in Figure 7.

a. a relationship of the relation ACCESS from the object 'A_FILE(X) to the group node 'GROUP(JONES) with a GRANT attribute value of ((NEWMAIL)),

b. a relationship of the relation ACCESS from the object 'A_FILE(X) to the group node 'GROUP(MAILTOOLS) with a GRANT attribute value of ((NEWMAIL=>APPEND),(READMAIL=>READ)),

c. a relationship of the relation ACCESS from the object 'A_FILE(X) to the group node 'GROUP(SMITH) with a GRANT attribute value of (((NEWMAIL,READMAIL,CONTROL))),

d. a relationship of the relation ACCESS from the object 'A_FILE(X) to the group node 'GROUP(WIZARD) with a GRANT attribute value of (((READ, WRITE))),

e. a relationship of the relation ADOPTED_ROLE from the subject (process) 'JOB(MAILER) to the group node 'GROUP(JONES),

f. a relationship of the relation ADOPTED_ROLE from the subject (process) 'JOB(MAILER) to the group node 'GROUP(WIZARD), by explicit adoption.

g. a relationship of the relation ADOPTED_ROLE from the subject (process) 'JOB(MAILER) to the group node 'GROUP(MAILTOOLS).MAILER,

The values of the GRANT attributes of the access relationships are used to determine the set of approved access rights; in the example shown in the figure, the values of the grant_items which are used to determine the approved access rights of the job are ((READ,WRITE)), (NEWMAIL), (NEWMAIL=>APPEND) and (READMAIL=>READ).

Approved access rights are access rights whose names appear in the resulting_rights_list of any grant_item of these GRANT attribute values for which (1) no necessary_right is given or (2) the necessary_right names an approved access right (under finite recursive application of the definition of approved access rights). In this example, the access rights NEWMAIL, READ and WRITE have no necessary_right given, thus the set of approved access rights under rule (1) contains NEWMAIL, READ and WRITE.

The access right APPEND can be added to the set of approved access rights under rule (2) because APPEND appears in the resulting_rights_list of the grant_item whose necessary_ right name, NEWMAIL, matches an access right name already in the set. The set of approved access rights now consists of NEWMAIL, READ, WRITE and APPEND.

If the process had not explicitly adopted 'GROUP(WIZARD), the access rights would be determined by the GRANT attribute values ((NEWMAIL)) and ((NEWMAIL=>APPEND), (READMAIL=>READ)) which results in the set of approved access rights NEWMAIL and APPEND.

Downloaded from http://www.everyspec.com

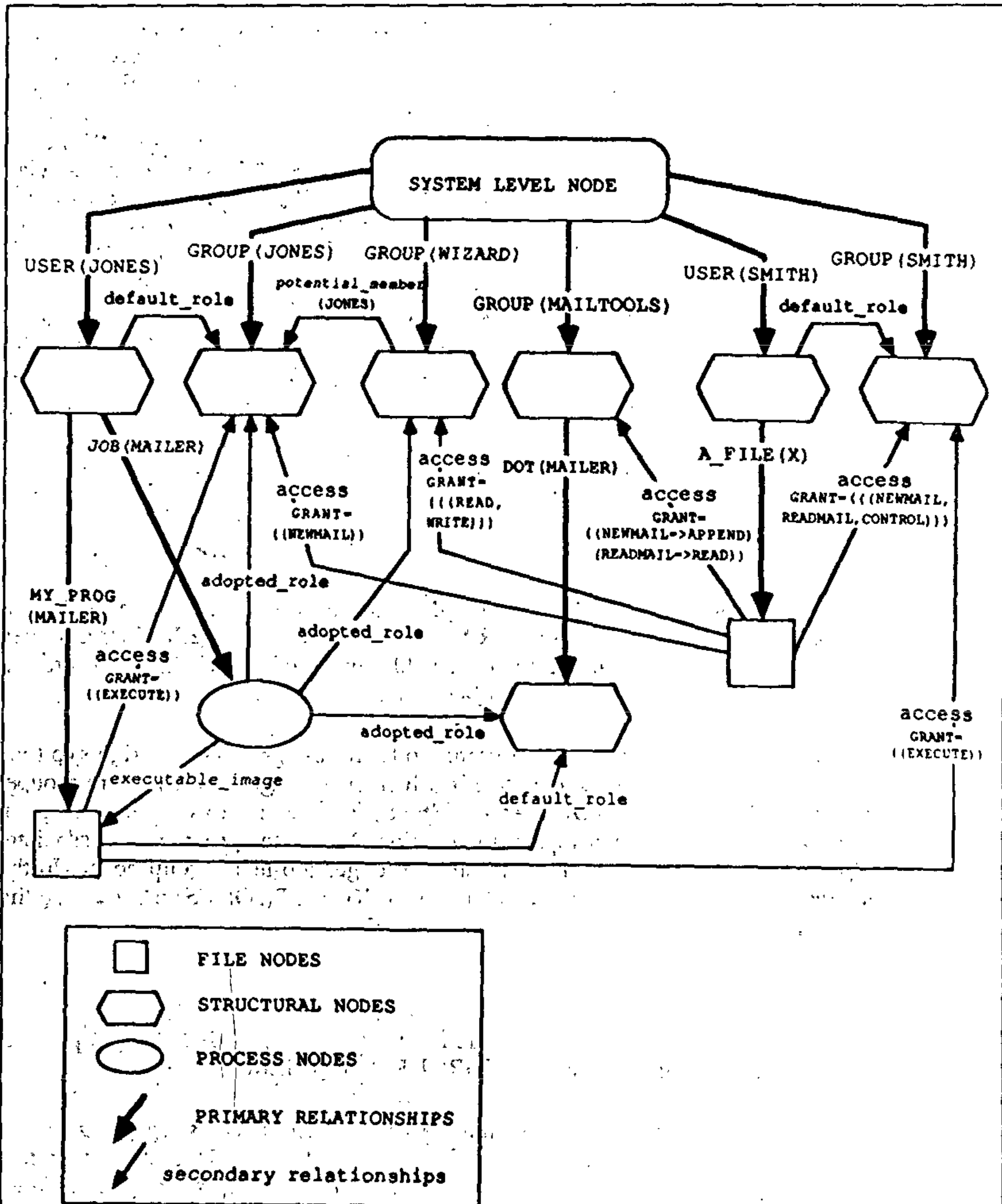

FIGURE 7. Access relationships, Example 1

Note: The relationship keys on the relationships of the predefined relation ADOPTED_ROLE are omitted, although required by the CAIS node model, since these keys are not relevant to the discussion in this section.

Downloaded from http://www.everyspec.com

If 'USER(SMITH) were to run 'USER(JONES)'MY_PROG(MAILER), as shown in Figure 8, his access rights would be determined by the GRANT attribute values (((NEWMAIL, READMAIL,CONTROL))) and ((NEWMAIL=>APPEND), (READMAIL=>READ)) which results in the approved access rights NEWMAIL, READMAIL, APPEND, READ and CONTROL. Note that 'USER(SMITH) cannot read or append the contents of the file 'A_FILE(X) except through the services of a tool whose default role is reachable via a primary relationship of the DOT relation from 'GROUP(MAILTOOLS) or has the capability as a potential member to adopt the role 'GROUP(MAILTOOLS). By having access right CONTROL, he may change the access rights at will. Of course, if 'USER(SMITH) could adopt the role WIZARD or could use a tool that adopts this role, he would not need to use a tool in the family of MAILTOOLS to read or write the contents of the file 'A_FILE(X). It is a matter of policy in setting up the access control structures whether highly privileged roles, such as the role WIZARD, are adoptable by a large number of users and tools.

While the access rights READ, WRITE, APPEND and CONTROL used in this example are predefined access rights and therefore carry meaning to the CAIS interfaces that open node handles, NEWMAIL and READMAIL are user-defined access rights without CAIS-specific semantics. An approved user-defined access right does not influence the checking of access rights (see Section 4.4.2.5); it only influences the determination of approved access rights.

In each of these examples, users SMITH and JONES were allowed to run the MAILER program because access relationships with a GRANT attribute value of (((EXECUTE))) were established from the file node containing the executable image of the MAILER program to the group nodes representing their respective default roles.

While Figure 7 and Figure 8 show the use of group nodes for assigning access rights to tools or groups of tools, Figure 9 exhibits an example in which access rights for users are grouped in a similar manner. The group 'GROUP(ALL_USERS) is introduced as a top-level group node with emanating primary relationships of the relation DOT to the group nodes associated with the users SMITH and JONES. The latter are no longer top-level group nodes. In this new example, the access relationship that was targeted at 'GROUP(JONES) is moved so that it is targeted at 'GROUP(ALL_USERS). The NEWMAIL access right is deleted from the access relationship to the group node associated with user SMITH. Now mail can be sent to SMITH by all users, but he can read it only via the MAILER program. However, WIZARDs can read or edit the file 'A_FILE(X) without using a program whose DEFAULT_ROLE is a member of the MAILTOOLS group. All users are now allowed to run the MAILER program since the EXECUTE access right is granted to 'GROUP(ALL_USERS).

A change to either of the above scenarios so that all access right changes regarding the file 'A_FILE(X) have to be accomplished through a tool in 'GROUP(MAILTOOLS) can be done by simply moving the GRANT attribute value CONTROL to the access relationship targeted at 'GROUP(MAILTOOLS).

Downloaded from http://www.everyspec.com

4.4.2.5 Discretionary access checking

CAIS discretionary access control rules state that any access right required for a subject to access an object must be contained in the set of approved access rights of that object with respect to that subject. The CAIS model requires discretionary access checking to be performed at the time a node handle is opened (see Section 5.1.2.1, page 63). At this point access rights implied by the INTENT parameter of the open operation must be a subset of the approved access rights. If this is not the case, the operation is terminated and an exception is raised. If the access rights implied by the INTENT parameter are a subset of the approved access rights, then for subsequent access using a successfully opened node handle, the access rights required may be compared to the rights implied by the intent, rather than the approved access rights.

4.4.3 Mandatory access control

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

A mandatory access control classification is a combination of a hierarchical classification level and zero or more non-hierarchical categories. 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. 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 is included in the set of categories assigned to the object.

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

In the CAIS, subjects are CAIS processes, while an object may be any CAIS node. Operations are CAIS operations and are classified as read, write, or read and write operations. Mandatory access checking is performed by comparing the classification of the subject with that of the object with respect to the type of operations intended to be performed. The CAIS model requires mandatory access checking to be performed at the time a node handle is opened (see Section 5.1.2.1, page 63) with respect to the intents expressed in the INTENT parameter. For subsequent access using a successfully opened node handle, mandatory access checking may, but need not, be repeated. The effects of alterations of classifications of objects on concurrently executing processes are implementation-defined.

Downloaded from http://www.everyspec.com

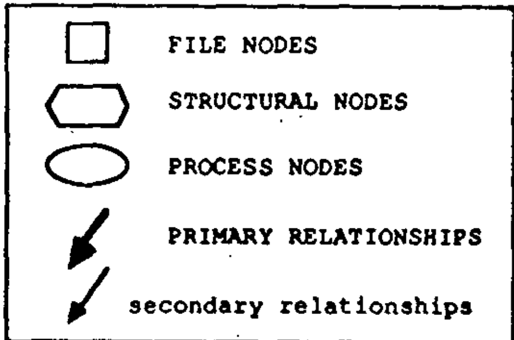

FIGURE 8. Access relationships, Example 2

Note: The relationship keys on the relationships of the predefined relation ADOPTED_ROLE are omitted, although required by the CAIS node model, since these keys are not relevant to the discussion in this section.

Downloaded from http://www.everyspec.com

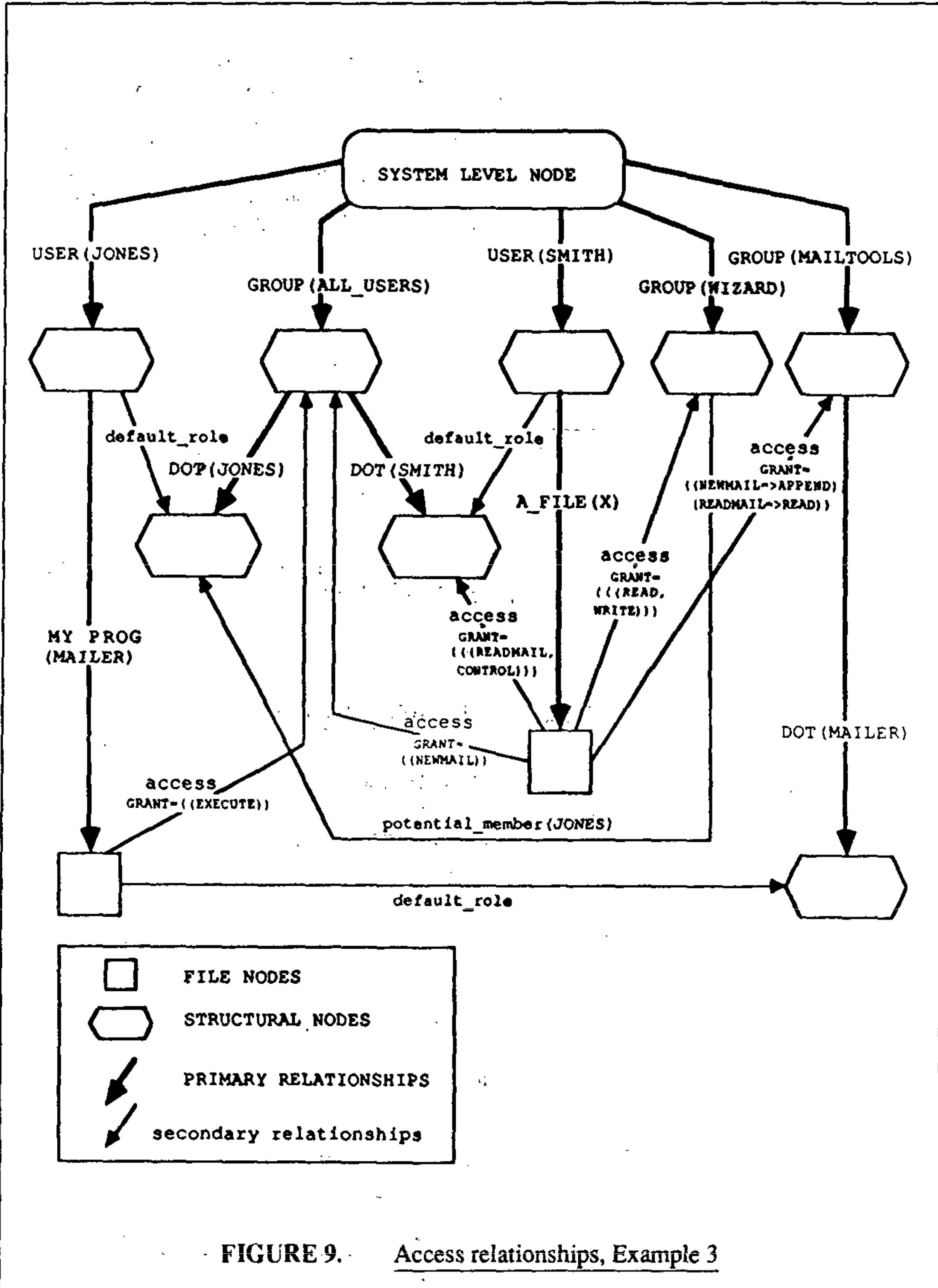

FIGURE 9. Access relationships, Example 3

Downloaded from http://www.everyspec.com

4.4.3.1 Labeling of CAIS nodes

The labeling of nodes is provided by predefined node attributes. A predefined attribute, called SUBJECT_CLASSIFICATION, is assigned to each process node and represents the classification of that process 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 cannot be read or written directly through the CAIS interfaces, except for their initial setting at node creation. The value of the attribute is a parenthesized list containing one or two items, the hierarchical classification level and, optionally, 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 one or more keyword members of the set of non-hierarchical categories. The hierarchical classification level set and the non-hierarchical category set are implementation-defined. For example, the following are possible classification attribute values:

(TOP_SECRET, (MAIL_USER, OPERATOR, STAFF))

(UNCLASSIFIED)

(SECRET, (STAFF))

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

TABLE IV. Classification Attribute Value BNF

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

See Appendix D for a description of the notation used.

Downloaded from http://www.everyspec.com

4.4.3.2 Labeling of process nodes

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

When a root process node 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 creating process may specify the classification attributes associated with the node. If no classification is specified, the classification is inherited from the creating process. The assigned classification must adhere to the requirements for mandatory access control over write operations.

4.4.3.3 Labeling of non-process nodes

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

4.4.3.4 Labeling of nodes for devices

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

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

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

4.4.3.5 Mandatory access checking

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

Downloaded from http://www.everyspec.com

5. DETAILED REQUIREMENTS

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

5.1 General node management

This section describes the CAIS interfaces for the general manipulation of nodes, relationships and attributes. These interfaces are defined in five CAIS packages: CAIS_DEFINITIONS defines types, subtypes, exceptions, and constants used throughout the CAIS; CAIS_NODE_MANAGEMENT defines interfaces for general operations on nodes and relationships; CAIS_ATTRIBUTE_MANAGEMENT defines interfaces for general operations on attributes; CAIS_ACCESS_CONTROL_MANAGEMENT defines interfaces for setting access rights and adopting roles; and CAIS_STRUCTURAL_NODE_MANAGEMENT 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 Section 5.2 and Section 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*. Ada objects of this type can be associated with a node by means of CAIS procedures, causing an *open node handle* to be assigned to the object. While such an association is in effect, the node handle is said to be open; otherwise, the node handle is said to be *closed*. Most procedures expect either a parameter of type NODE_TYPE, a pathname, or a combination of a base node (specified by a parameter of type NODE_TYPE) and a path element relative to it, to identify a node.

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

The package CAIS_STANDARD (see Section 5.5, page 496) contains certain scalar types predefined in the CAIS. The intent of providing this package is to make these types reasonably independent of any predefined types in the Ada language, whose characteristics may vary among compilers. These types are CAIS_INTEGER, CAIS_NATURAL, CAIS_POSITIVE and CAIS_DURATION and are analogous to the Ada types INTEGER, NATURAL, POSITIVE and DURATION, respectively.

Downloaded from http://www.everyspec.com

5.1.1 Package CAIS_DEFINITIONS

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

```
type NODE_TYPE is limited private;

type NODE_KIND is (FILE, STRUCTURAL, PROCESS);

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

type INTENT_ARRAY is array (CAIS_POSITIVE range <>) of INTENT_SPECIFICATION;

subtype PATHNAME          is  STRING;
subtype RELATIONSHIP_KEY  is  STRING;
subtype RELATION_NAME     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. INTENT_ARRAY is the type of the parameter INTENT of CAIS procedures which open or change the intent of a node handle, as further explained in Section 5.1.2.

PATHNAME, RELATIONSHIP_KEY, and RELATION_NAME are string subtypes for pathnames, relationship key designators, and relation names. Values of these subtypes are subject to certain syntactic restrictions whose violation causes exceptions to be raised.

```
subtype ATTRIBUTE_NAME is STRING;
```

ATTRIBUTE_NAME is a subtype for the names of attributes.

```
subtype ATTRIBUTE_LIST              is CAIS_LIST_MANAGEMENT.LIST_TYPE;
subtype DISCRETIONARY_ACCESS_LIST   is CAIS_LIST_MANAGEMENT.LIST_TYPE;
subtype MANDATORY_ACCESS_LIST       is CAIS_LIST_MANAGEMENT.LIST_TYPE;
```

ATTRIBUTE_LIST is the subtype for lists of attributes. DISCRETIONARY_ACCESS_LIST and MANDATORY_ACCESS_LIST are the subtypes, respectively, of the discretionary and mandatory access control information. Values of these LIST_TYPE subtypes are subject to certain syntactic restrictions whose violation causes exceptions to be raised.

Downloaded from http://www.everyspec.com


```
CURRENT_USER:      constant PATHNAME          := "'CURRENT_USER";
CURRENT_NODE:      constant PATHNAME          := "'CURRENT_NODE";
CURRENT_PROCESS:   constant PATHNAME          := ":";
LATEST_KEY:        constant RELATIONSHIP_KEY  := "#";
DEFAULT_RELATION:  constant RELATION_NAME     := "DOT";
LONG_DELAY:        constant CAIS_DURATION     := CAIS_DURATION'LAST;
```

CURRENT_USER, CURRENT_NODE, and CURRENT_PROCESS are standard pathnames for the current user's top-level node, current node (base node for pathnames beginning with a relationship key), and current process node, respectively. LATEST_KEY and DEFAULT_RELATION are standard names for the latest key and the default relation name, respectively. LONG_DELAY is a constant of type CAIS_DURATION (see [1815A] 9.6) used for time limits.

```
ACCESS_VIOLATION:              exception;
ATTRIBUTE_ERROR:               exception;
DEVICE_ERROR:                  exception;
EXISTING_NODE_ERROR:           exception;
INTENT_VIOLATION:              exception;
ITERATOR_ERROR:                exception;
LOCK_ERROR:                    exception;
NAME_ERROR:                    exception;
NODE_KIND_ERROR:               exception;
PATHNAME_SYNTAX_ERROR:         exception;
PREDEFINED_ATTRIBUTE_ERROR:    exception;
PREDEFINED_RELATION_ERROR:     exception;
RELATIONSHIP_ERROR:            exception;
SECURITY_VIOLATION:            exception;
STATUS_ERROR:                  exception;
SYNTAX_ERROR:                  exception;
USE_ERROR:                     exception;
```

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

ATTRIBUTE_ERROR is raised if the interface expects an attribute of the given name and none exists or if the interface expects there to be no attribute of the given name but one already exists.

DEVICE_ERROR is raised if a CAIS operation cannot be completed because of a malfunction of the underlying system. All interfaces may raise this exception, so it is not explicitly mentioned in the descriptions of the interfaces.

EXISTING_NODE_ERROR is raised if a node already exists with the identification given and an attempt is made to create a node with this identification.

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

ITERATOR_ERROR is raised if an iterator is used that has not been set or is exhausted.

Downloaded from http://www.everyspec.com

LOCK_ERROR is raised if the opening of a node handle to a node is delayed beyond a specified time limit due to the existence of a lock on this node, its attributes, relationships, or contents. Such opening of a node handle may also be an implicit action of a CAIS interface call (see Section 5.1.2, page 57). LOCK_ERROR may be raised prior to expiration of the timeout if the CAIS implementation can determine that a deadlock situation has occurred.

NAME_ERROR is raised if 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.

NODE_KIND_ERROR is raised if the kind of the node is incorrect for the attempted operation.

PATHNAME_SYNTAX_ERROR is raised if the pathname information is syntactically illegal.

PREDEFINED_ATTRIBUTE_ERROR is raised if an attempt is made to create or modify a predefined attribute that cannot be created or modified by the user.

PREDEFINED_RELATION_ERROR is raised if an attempt is made to create or modify a relationship of a predefined relation that cannot be created or modified by the user.

RELATIONSHIP_ERROR is raised if the relationship identified by the parameters BASE, KEY and RELATION of a procedure or function does not exist.

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

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

SYNTAX_ERROR is raised if the information given in certain parameters other than pathname parameters is syntactically illegal.

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

The CAIS does not prescribe a particular preference ordering among these exceptions in the presence of multiple error situations, except where the lack of such preference would create inference paths violating security models. If access to information that is inaccessible according to the access control models is required to detect an additional error situation, the exception relating to the attempt to access inaccessible information takes precedence over the exception for the additional error.

Downloaded from http://www.everyspec.com

5.1.2 Package CAIS_NODE_MANAGEMENT

This package defines the general primitives for manipulating, copying, renaming and deleting nodes and their relationships. The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_PRAGMATICS.

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_NODE_MANAGEMENT (see Section 5.1.5), the creation of nodes for processes is performed by INVOKE_PROCESS, SPAWN_PROCESS and CREATE_JOB of package CAIS_PROCESS_MANAGEMENT (see Section 5.2.2), and the creation of nodes for files is performed by the CREATE procedures of the input and output packages (see Section 5.3).

Three CAIS interfaces for manipulating node handles are OPEN (opens a node handle), CLOSE (closes a node handle), and CHANGE_INTENT (alters the specification of the intention of node handle usage). In addition, OPEN_PARENT, GET_CURRENT_NODE, GET_NEXT and the node creation procedures also open node handles. These interfaces perform access synchronization in accordance with an intent specified by the parameter INTENT. One or more of the intents defined in Table V can be expressed by the INTENT parameters.

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

Open node handles to a node can delay attempts to open other node handles to this node or to change the intent of other node handles to this node, as explained in Table V and summarized graphically in Table VI. Most CAIS interfaces that may be delayed upon opening a node allow the specification of a time limit after which a call to such interfaces is allowed to terminate with an exception.

Downloaded from http://www.everyspec.com

TABLE V. Intents

Intent	Explanation
NO_ACCESS	The established access right for subsequent operations is to query properties of the node handle and obtainability of the node only. Locks on the node have no delaying effect.
READ EXCLUSIVE_READ	Open and CHANGE_INTENT operations are delayed if the node, its contents, attributes or relationships are locked against read operations. The established access right for subsequent operations is to read node contents, attributes and relationships. For EXCLUSIVE_READ, the node is locked against opens with any write or control intent as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with write or control intent.
WRITE EXCLUSIVE_WRITE	Open and CHANGE_INTENT operations are 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 opens with any read, write, append, execute or control intent as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with read, write, append, execute or control intent.
APPEND EXCLUSIVE_APPEND	Open and CHANGE_INTENT operations are delayed if the node, its contents, attributes or relationships are locked against append operations. The established access right for subsequent operations is to append to the contents and to create attributes or relationships. For EXCLUSIVE_APPEND, the node is locked against opens with write, append or execute intent as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with write, append or execute intent.
READ_CONTENTS EXCLUSIVE_READ_CONTENTS	Open and CHANGE_INTENT operations are delayed if the node or its contents are locked against read operations. The established access right for subsequent operations is to read the node contents. For EXCLUSIVE_READ_CONTENTS, the node contents are locked against all opens with write intent as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with intent to write its contents.

Downloaded from http://www.everyspec.com

TABLE V. Intents -- Continued.

Intent	Explanation
WRITE_CONTENTS EXCLUSIVE_WRITE_CONTENTS	Open and CHANGE_INTENT operations are delayed if the node or its contents are locked against write operations. The established access right for subsequent operations is to write or append to the node contents. For EXCLUSIVE_WRITE_CONTENTS, the node contents are locked against opens with read, write, append or execute intent as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with intent to read, write, execute or append to its contents.
APPEND_CONTENTS EXCLUSIVE_APPEND_CONTENTS	Open and CHANGE_INTENT operations are delayed if the node or its contents are locked against append operations. The established access right for subsequent operations is to append to the node contents. For EXCLUSIVE_APPEND_CONTENTS, the node contents are locked against opens with write, append, or execute intent as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with intent to write, execute or append to its contents.
READ_ATTRIBUTES EXCLUSIVE_READ_ATTRIBUTES	Open and CHANGE_INTENT operations are delayed if the node or its attributes are locked against read operations. The established access right for subsequent operations is to read node attributes. For EXCLUSIVE_READ_ATTRIBUTES, the node is locked against opens with intent to write attributes as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with intent to write attributes.
WRITE_ATTRIBUTES EXCLUSIVE_WRITE_ATTRIBUTES	Open and CHANGE_INTENT operations are delayed if the node or its attributes are locked against write operations. The established access right for subsequent operations is to modify and create node attributes. For EXCLUSIVE_WRITE_ATTRIBUTES, the node is locked against opens with intent to read, write or append attributes as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with intent to read, write or append attributes.

Downloaded from http://www.everyspec.com

TABLE V. Intents -- Continued.

Intent	Explanation
APPEND_ATTRIBUTES EXCLUSIVE_APPEND_ATTRIBUTES	Open and CHANGE_INTENT operations are delayed if the node or its attributes are locked against append operations. The established access right for subsequent operations is to create node attributes. For EXCLUSIVE_APPEND_ATTRIBUTES, the node is locked against opens with intent to write or append attributes as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with intent to write or append attributes.
READ_RELATIONSHIPS EXCLUSIVE_READ_RELATIONSHIPS	Open and CHANGE_INTENT operations are delayed if the node or its relationships are locked against read operations. The established access right for subsequent operations is to read node relationships, including their attributes. For EXCLUSIVE_READ_RELATIONSHIPS, the node is locked against opens with control intent or intent to write relationships as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with control intent or intent to write relationships.
WRITE_RELATIONSHIPS EXCLUSIVE_WRITE_RELATIONSHIPS	Open and CHANGE_INTENT operations are delayed if the node or its relationships are locked against write operations. The established access right for subsequent operations is to write or create node relationships, including their attributes. For EXCLUSIVE_WRITE_RELATIONSHIPS, the node is locked against opens with control intent or intent to read, write or append relationships as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with control intent or intent to read, write or append relationships.
APPEND_RELATIONSHIPS EXCLUSIVE_APPEND_RELATIONSHIPS	Open and CHANGE_INTENT operations are delayed if the node or its relationships are locked against append operations. The established access right for subsequent operations is to create node relationships, including their attributes. For EXCLUSIVE_APPEND_RELATIONSHIPS, the node is locked against opens with control intent or intent to write or append relationships as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with control intent or intent to write or append relationships.

Downloaded from http://www.everyspec.com

TABLE V. Intents -- Continued.

Intent	Explanation
CONTROL EXCLUSIVE_CONTROL	Open and CHANGE_INTENT operations are delayed if the node or its relationships are locked against write, control or (exclusive or non-exclusive) READ and READ_RELATIONSHIPS operations. The established access right for subsequent operations is to read, write or append access control information. For EXCLUSIVE_CONTROL, the node is locked against opens to read, write or append relationships or to read, write, or append access control information as specified in Table VI. Open and CHANGE_INTENT operations are additionally delayed if there are open node handles to the node with intent to read, write or append relationships or to read, write or append access control information.
EXECUTE	Open and CHANGE_INTENT operations are delayed if the node contents are locked against read or append operations. The established access right for subsequent operations is the permission to initiate a process taking the node contents as executable image.

Downloaded from http://www.everyspec.com

TABLE VI. Matrix of Access Synchronization Constraints

I1 \ I2		NON-EXCLUSIVE															EXCLUSIVE													
		NO	R	W	A	RC	WC	AC	RA	WA	AA	RR	WR	AR	C	EX	R	W	A	RC	WC	AC	RA	WA	AA	RR	WR	AR	C	
NON-EXCLUSIVE	NO	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	NO
	R	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	X	.	.	X	.	.	X	.	X	R
	W	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	X	X	X	X	X	X	X	X	X	X	X	W
	A	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	.	X	X	.	X	X	.	X	X	X	A
	RC	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	X	.	.	.	.	.	.	.	.	RC
	WC	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	X	X	X	X	.	.	.	.	.	.	.	WC
	AC	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	.	X	X	.	.	.	.	.	.	.	AC
	RA	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	X	.	.	.	.	.	RA
	WA	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	X	.	.	.	X	X	X	.	.	.	.	WA
	AA	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	.	.	.	.	X	X	.	.	.	.	AA
	RR	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	.	.	.	.	.	.	.	.	X	.	X	RR
	WR	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	X	.	.	.	.	.	.	X	X	X	X	WR
	AR	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	.	.	.	.	.	.	.	X	X	X	AR
	C	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	X	.	.	.	.	.	.	X	X	X	X	C
	EX	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	X	X	.	X	X	.	.	.	.	.	.	.	EX
EXCLUSIVE	R	.	.	X	.	.	X	.	.	X	.	.	X	.	X	.	.	X	.	.	X	.	.	X	.	.	X	.	X	R
	W	.	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	W
	A	.	.	X	X	.	X	X	.	X	X	.	X	X	X	X	.	X	X	.	X	X	.	X	X	.	X	X	X	A
	RC	.	.	X	.	.	X	.	.	.	.	.	.	.	.	.	.	X	.	.	X	.	.	.	.	.	.	.	.	RC
	WC	.	X	X	X	X	X	X	.	.	.	.	.	.	.	X	X	X	X	X	X	X	.	.	.	.	.	.	.	WC
	AC	.	.	X	X	.	X	X	.	.	.	.	.	.	.	X	.	X	X	.	X	X	.	.	.	.	.	.	.	AC
	RA	.	.	X	.	.	.	.	.	X	.	.	.	.	.	.	.	X	.	.	.	.	.	X	.	.	.	.	.	RA
	WA	.	X	X	X	.	.	.	X	X	X	.	.	.	.	.	X	X	X	.	.	.	X	X	X	.	.	.	.	WA
	AA	.	.	X	X	.	.	.	.	X	X	.	.	.	.	.	.	X	X	.	.	.	.	X	X	.	.	.	.	AA
	RR	.	.	X	.	.	.	.	.	.	.	.	X	.	X	.	.	X	.	.	.	.	.	.	.	.	X	.	X	RR
	WR	.	X	X	X	.	.	.	.	.	.	X	X	X	X	.	X	X	X	.	.	.	.	.	.	X	X	X	X	WR
	AR	.	.	X	X	.	.	.	.	.	.	.	X	X	X	.	.	X	X	.	.	.	.	.	.	.	X	X	X	AR
	C	.	X	X	X	.	.	.	.	.	.	X	X	X	X	.	X	X	X	.	.	.	.	.	.	X	X	X	X	C
	I1 / I2	NO	R	W	A	RC	WC	AC	RA	WA	AA	RR	WR	AR	C	EX	R	W	A	RC	WC	AC	RA	WA	AA	RR	WR	AR	C	

X = Open with intent I2 is delayed if there are open handles opened with intent I1.

NO = NO_ACCESS
R = READ
RC = READ_CONTENTS
RA = READ_ATTRIBUTES
RR = READ_RELATIONSHIPS
C = CONTROL
W = WRITE
WC = WRITE_CONTENTS
WA = WRITE_ATTRIBUTES
WR = WRITE_RELATIONSHIPS
EX = EXECUTE
A = APPEND
AC = APPEND_CONTENTS
AA = APPEND_ATTRIBUTES
AR = APPEND_RELATIONSHIPS

Downloaded from http://www.everyspec.com

5.1.2.1 Opening a node handle

```
procedure OPEN (NODE:       in out NODE_TYPE;
                NAME:       in     PATHNAME;
                INTENT:     in     INTENT_ARRAY;
                TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY);

procedure OPEN (NODE:       in out NODE_TYPE;
                BASE:       in     NODE_TYPE;
                KEY:        in     RELATIONSHIP_KEY;
                RELATION:   in     RELATION_NAME := DEFAULT_RELATION;
                INTENT:     in     INTENT_ARRAY;
                TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY);
```

Purpose:

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

Parameters:

NODE	is a node handle, initially closed, to be opened to the identified node.
NAME	is the pathname identifying the node to be opened.
BASE	is an open node handle to a base node for node identification.
KEY	is the relationship key designator 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 CAIS_DURATION, specifying a time limit for the delay on waiting for the unlocking of a node in accordance with the desired INTENT.

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the NAME, KEY or RELATION given is syntactically illegal (see Table I, page 32).

Downloaded from http://www.everyspec.com

NAME_ERROR is raised if any traversed node in the path specified is unobtainable or inaccessible, if the node to which the handle is to be opened is inaccessible or unobtainable and the given INTENT includes any intent other than NO_ACCESS, or if the relationship specified by BASE, KEY and RELATION or by any path element of NAME does not exist.

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

STATUS_ERROR
is raised if the node handle NODE is already open at the time of the call on OPEN or if BASE is not an open node handle.

LOCK_ERROR is raised if the OPEN operation is delayed beyond the specified time limit due to the existence of locks in conflict with the specified INTENT. This includes any delays caused by locks on nodes traversed on the path specified by the pathname NAME or locks on the node identified by BASE, preventing the reading of relationships emanating from these nodes. LOCK_ERROR may be raised prior to expiration of the timeout if the CAIS implementation can determine that a deadlock situation has occurred.

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

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

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

Downloaded from http://www.everyspec.com

Additional Interfaces:

```
procedure OPEN (NODE:        in out NODE_TYPE;
                NAME:        in     PATHNAME;
                INTENT:      in     INTENT_SPECIFICATION := READ;
                TIME_LIMIT:  in     CAIS_DURATION := LONG_DELAY)
is
begin
    OPEN (NODE, NAME, (1 => INTENT), TIME_LIMIT);
end OPEN;

procedure OPEN (NODE:        in out NODE_TYPE;
                BASE:        in     NODE_TYPE;
                KEY:         in     RELATIONSHIP_KEY;
                RELATION:    in     RELATION_NAME := DEFAULT_RELATION;
                INTENT:      in     INTENT_SPECIFICATION := READ;
                TIME_LIMIT:  in     CAIS_DURATION := LONG_DELAY)
is
begin
    OPEN (NODE, BASE, KEY, RELATION, (1 => INTENT), TIME_LIMIT);
end OPEN;
```

Notes:

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

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

Downloaded from http://www.everyspec.com

5.1.2.2 Closing a node handle

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

Purpose:

This procedure severs any association between the node handle NODE and the node and releases any associated lock on the node imposed by the intent of the node handle NODE. Closing an already closed node handle has no effect. If there are any open file handles associated with the contents of the node identified by this node handle, the file handles are also closed.

Parameter:

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

Exceptions:

None.

Notes:

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

Downloaded from http://www.everyspec.com

5.1.2.3 Changing the intent regarding node handle usage

```
procedure CHANGE_INTENT (NODE:       in out NODE_TYPE;
                         INTENT:     in     INTENT_ARRAY;
                         TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY);
```

Purpose:

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

Parameters:

NODE
: is an open node handle.

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

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

Exceptions:

NAME_ERROR
: is raised if the node handle NODE refers to an unobtainable or inaccessible node and INTENT contains any intent specification other than NO_ACCESS.

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

LOCK_ERROR
: is raised if the operation is delayed beyond the specified time limit due to the existence of locks on the node in conflict with the specified INTENT. LOCK_ERROR may be raised prior to expiration of the timeout if the CAIS implementation can determine that a deadlock situation has occurred.

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

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

Downloaded from http://www.everyspec.com

USE_ERROR is raised if there are open file handles associated with the node and the new INTENT differs from the existing intent regarding the contents of the node identified by NODE.

Additional Interface:

```
procedure CHANGE_INTENT (NODE:       in out NODE_TYPE;
                         INTENT:     in     INTENT_SPECIFICATION;
                         TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY)
is
begin
    CHANGE_INTENT (NODE, (1 => INTENT), TIME_LIMIT);
end CHANGE_INTENT;
```

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.

Downloaded from http://www.everyspec.com

5.1.2.4 Examining the open status of a node handle

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

Purpose:

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

Parameter:

NODE   is a node handle.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.1.2.5 Querying the intention of a node handle

```
function INTENT (NODE: in NODE_TYPE)
    return INTENT_ARRAY;
```

Purpose:

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

Parameter:

NODE is an open node handle.

Exception:

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

Downloaded from http://www.everyspec.com

5.1.2.6 Querying the kind of a node

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

Purpose:

This function returns the kind of a node, either FILE, PROCESS or STRUCTURAL. It is possible to query the kind of inaccessible and unobtainable nodes. The query does not constitute access to the node. The "kind" of the target node is regarded as an implicit attribute of all relationships to the node and of any open node handle.

Parameter:

NODE is an open node handle.

Exception:

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

Downloaded from http://www.everyspec.com

5.1.2.7 Querying the number of open file handles on a file node

```
function OPEN_FILE_HANDLE_COUNT (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns the number of open file handles associated with the node handle NODE.

Parameter:

NODE is an open node handle.

Exceptions:

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

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

Downloaded from http://www.everyspec.com

5.1.2.8 Obtaining the unique primary pathname

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

Purpose:

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

Parameter:

NODE is an open node handle identifying the node.

Exceptions:

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

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

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

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

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

Downloaded from http://www.everyspec.com

5.1.2.9 Obtaining the relationship key of a primary relationship

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

Purpose:

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

Parameter:

NODE is an open node handle identifying the node.

Exceptions:

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

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

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

Downloaded from http://www.everyspec.com

5.1.2.10 Obtaining the relation name of a primary relationship

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

Purpose:

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

Parameter:

NODE is an open node handle identifying the node.

Exceptions:

NAME_ERROR is raised if the parent 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 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 discretionary access control rights of the current process 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.

Downloaded from http://www.everyspec.com

5.1.2.11 Obtaining the relationship key of the last relationship traversed

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

Purpose:

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

Parameter:

NODE is an open node handle.

Exceptions:

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

USE_ERROR is raised if the node handle NODE was opened using the pathname ":".

Downloaded from http://www.everyspec.com

5.1.2.12 Obtaining the relation name of the last relationship traversed

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

Purpose:

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

Parameter:

NODE is an open node handle.

Exceptions:

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

USE_ERROR is raised if the node handle NODE was opened using the pathname ":".

Downloaded from http://www.everyspec.com

5.1.2.13 Obtaining a partial pathname

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

Purpose:

This function returns the pathname obtained by deleting the last path element from NAME. It does not establish whether the pathname identifies an existing node; only the syntactic properties of the pathname are examined. This function also checks the syntactic legality of the pathname NAME. If the submitted name is an abbreviated name according to the rules of Section 4.3.5, the function returns the corresponding portion of the fully expanded pathname. If the fully expanded pathname has only a single path element, ":" is returned.

Parameter:

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

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if NAME is a syntactically illegal pathname (see Table I, page 32).

USE_ERROR is raised if NAME has the value ":".

Downloaded from http://www.everyspec.com

5.1.2.14 Obtaining the name of the last relationship in a pathname

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

Purpose:

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

Parameter:

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

Exceptions:

PATHNAME_SYNTAX_ERROR
: is raised if NAME is a syntactically illegal pathname (see Table I, page 32).

USE_ERROR is raised if NAME has the value ":".

Downloaded from http://www.everyspec.com

5.1.2.15 Obtaining the key of the last relationship in a pathname

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

Purpose:

This function returns the relationship key designator of the last path element of the pathname NAME. The empty string is returned if the last path element of the pathname NAME has no relationship key designator. This function does not establish whether the pathname identifies an existing node; only the syntactic properties of the pathname are examined. This function checks the syntactic legality of the pathname NAME.

Parameter:

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

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if NAME is a syntactically illegal pathname (see Table I, page 32).

USE_ERROR is raised if NAME has the value ":".

Downloaded from http://www.everyspec.com

5.1.2.16 Querying the existence of a node

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

Purpose:

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

Parameter:

NODE is an open node handle identifying the node.

Exception:

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

Additional Interfaces:

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

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

Notes:

IS_OBTAINABLE can be used to determine whether a node identified via a secondary relationship has been made unobtainable or is inaccessible to the current process.

Downloaded from http://www.everyspec.com

5.1.2.17 Querying sameness

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

Purpose:

This function returns TRUE if the nodes identified by its arguments are the same node; otherwise it returns FALSE. If both nodes are unobtainable or inaccessible, IS_SAME returns its result as if the nodes were not unobtainable or inaccessible. If only one of the nodes is unobtainable or inaccessible, IS_SAME always returns FALSE.

Parameters:

NODE1 is an open node handle to a node.

NODE2 is an open node handle to a node.

Exception:

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

Additional Interface:

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

Downloaded from http://www.everyspec.com

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.

Open node handles to unobtainable or inaccessible nodes can exist, if the intent under which these node handles were opened is NO_ACCESS only. No security violation arises in the IS_SAME interface, since the past existence and ''sameness'' of the nodes cannot be denied due to the visible existence of the relationships used in opening the node handles.

Downloaded from http://www.everyspec.com

5.1.2.18 Obtaining an index for a node handle

```
function INDEX (NODE:    in NODE_TYPE;
                MODULO:  in CAIS_POSITIVE)
    return CAIS_NATURAL;
```

Purpose:

This function returns an implementation-defined number in the range of 0 to MODULO-1. This number is guaranteed to be the same if the interface is called with the same MODULO parameter value for two open node handles N1 and N2, for which IS_SAME(N1,N2) is TRUE. Otherwise, the function result may but need not differ.

Parameters:

NODE	is an open node handle.
MODULO	is a number used to restrict the result to be in the range 0..MODULO-1.

Exception:

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

Notes:

This interface is intended to allow users to create hash tables for open node handles and fast checking for sameness. The fact that the type NODE_TYPE is limited private implies that the components of such hash tables need to be implemented in terms of access types, since the hash value can only be obtained after the node handle is opened.

An implementation of the interface ought to return numbers that are randomly distributed over the range 0..MODULO-1 for open node handles associated with different nodes.

Downloaded from http://www.everyspec.com

5.1.2.19 Obtaining an open node handle to the parent

```
procedure OPEN_PARENT (PARENT:     in out NODE_TYPE;
                       NODE:       in     NODE_TYPE;
                       INTENT:     in     INTENT_ARRAY;
                       TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY);
```

Purpose:

This procedure returns an open node handle in PARENT to the parent 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 OPEN_PARENT is equivalent to

```
OPEN (PARENT, NODE, "", "PARENT", INTENT, TIME_LIMIT).
```

Parameters:

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

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 CAIS_DURATION, specifying a time limit for the delay on waiting for the unlocking of the parent in accordance with the desired INTENT.

Exceptions:

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

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

STATUS_ERROR is raised if the node handle PARENT is open at the time of the call or if the node handle NODE is not open.

LOCK_ERROR is raised if the opening of the parent is delayed beyond the specified TIME_LIMIT due to the existence of locks in conflict with the specified INTENT. LOCK_ERROR may be raised prior to expiration of the timeout if the CAIS implementation can determine that a deadlock situation has occurred.

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

Downloaded from http://www.everyspec.com

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

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

Additional Interface:

```
procedure OPEN_PARENT (PARENT:     in out NODE_TYPE;
                       NODE:       in     NODE_TYPE;
                       INTENT:     in     INTENT_SPECIFICATION := READ;
                       TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY)
is
begin
    OPEN_PARENT (PARENT, NODE, (1=>INTENT), TIME_LIMIT);
end OPEN_PARENT;
```

Downloaded from http://www.everyspec.com

5.1.2.20 Copying a node

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

Purpose:

This procedure copies a file or structural node that does not have emanating primary relationships. The node copied is identified by the open node handle FROM and is copied to a newly created node. The new node is identified by the combination of the TO_BASE, TO_KEY and TO_RELATION parameters. The newly created node is of the same kind as the node identified by FROM. If the node is a file node, its contents are also copied, i.e., a new copied file is created. Any inheritable secondary relationship emanating from the original node is recreated in the copy. A secondary relationship of the predefined relationship PARENT is created from the newly created node to the node identified by TO_BASE. If the target node of the original node's relationship is the node itself, then the copy has an analogous relationship to itself. Any other secondary relationship whose target node 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 designator 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:

PATHNAME_SYNTAX_ERROR
is raised if the new node identification given by TO_KEY and TO_RELATION is syntactically illegal (see Table I, page 32).

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

NODE_KIND_ERROR
is raised if the original node is a process node.

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

PREDEFINED_RELATION_ERROR
is raised if TO_RELATION is the name of a predefined relation that cannot be created by the user.

Downloaded from http://www.everyspec.com

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

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

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

Additional Interface:

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

Downloaded from http://www.everyspec.com

5.1.2.21 Copying trees

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

Purpose:

This procedure copies a tree of file or structural nodes formed by primary relationships emanating from the node identified by the open node handle FROM. Primary relationships are recreated between corresponding copied nodes. The root node of the newly created tree corresponding to the FROM node is the node identified by the combination of the TO_BASE, TO_KEY and TO_RELATION parameters. If an exception is raised by the procedure, none of the nodes is copied. Secondary relationships, attributes, and node contents are copied as described for COPY_NODE with the following additional rules:

a. Secondary relationships between two nodes which both are copied are recreated between the two copies.

b. Any inheritable secondary relationships emanating from a node which is copied, but which refer to nodes outside the tree being copied, are copied so that they emanate from the copy, but still refer to the original target node.

c. Secondary relationships emanating from a node which is not copied, but which refer to nodes inside the tree being copied, are unaffected.

If the node identified by TO_BASE is part of the tree to be copied, then the copy of the node identified by FROM will not be copied recursively.

Figure 10, page 91, shows an example of copying a tree.

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

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the new node identification given by TO_KEY and TO_RELATION is syntactically illegal (see Table I, page 32).

Downloaded from http://www.everyspec.com

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given for the root node of the copied tree (given by TO_BASE, TO_KEY and TO_RELATION).

NODE_KIND_ERROR
is raised if any node to be copied is a process node.

PREDEFINED_RELATION_ERROR
is raised if TO_RELATION is the name of a predefined relation that cannot be created by the user.

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

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

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

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

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

Additional Interface:

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

Downloaded from http://www.everyspec.com

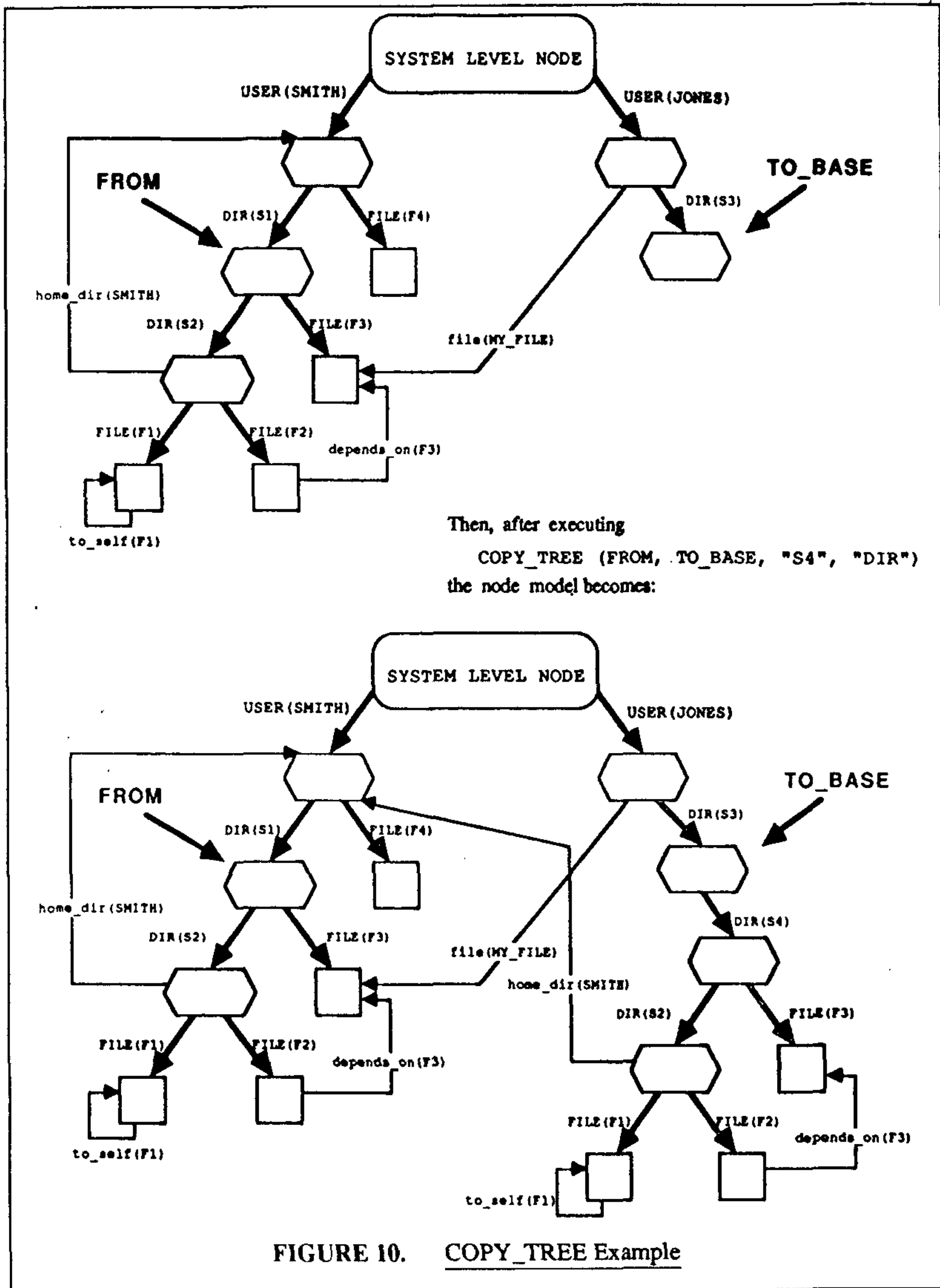

FIGURE 10. COPY_TREE Example

Downloaded from http://www.everyspec.com

5.1.2.22 Renaming the primary relationship of a node

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

Purpose:

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

Parameters:

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

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

NEW_KEY is a relationship key designator for the new primary relationship.

NEW_RELATION
is a relation name for the new primary relationship.

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the new node identification given by NEW_KEY and NEW_RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given.

NODE_KIND_ERROR
is raised if the node identified by NODE is a process node.

USE_ERROR is raised if the renaming cannot be accomplished while still maintaining non-circularity of primary relationships.

PREDEFINED_RELATION_ERROR
is raised if NEW_RELATION is the name of a predefined relation that cannot be created by the user or if the primary relationship to be deleted belongs to a predefined relation which cannot be modified by the user.

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

Downloaded from http://www.everyspec.com

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

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.

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

Notes:

Open node handles from existing processes track the renamed node.

Downloaded from http://www.everyspec.com

5.1.2.23 Deleting the primary relationship to a node

```
procedure DELETE_NODE (NODE:        in out NODE_TYPE;
                       TIME_LIMIT:  in     CAIS_DURATION := LONG_DELAY);
```

Purpose:

This procedure deletes the primary relationship to a node identified by NODE. The node becomes unobtainable. The node handle NODE is closed. If the node is a process node and the process is not yet TERMINATED (see Section 5.2), DELETE_NODE aborts the process. The TIME_LIMIT parameter allows the specification of a time limit for the delay imposed by the existence of locks on the parent of the node. A delayed call completes after the node is unlocked or the specified time limit has elapsed. In the latter case, the exception LOCK_ERROR is raised.

Parameters:

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

TIME_LIMIT — is a value of type CAIS_DURATION, specifying a time limit for the delay on waiting for the unlocking of a node in accordance with the intent WRITE_RELATIONSHIPS.

Exceptions:

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

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

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

STATUS_ERROR
is raised if the node handle NODE is not open at the time of the call.

LOCK_ERROR is raised if access, with intent WRITE_RELATIONSHIPS, to the parent of the node to be deleted cannot be obtained within the specified TIME_LIMIT due to an existing lock on the node. LOCK_ERROR may be raised prior to expiration of the timeout if the CAIS implementation can determine that a deadlock situation has occurred.

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

Downloaded from http://www.everyspec.com

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

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

Additional Interface:

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

Notes:

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

Downloaded from http://www.everyspec.com

5.1.2.24 Deleting the primary relationships of a tree

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

Purpose:

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

Parameter:

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

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

STATUS_ERROR
is raised if the node handle NODE is not open at the time of the call.

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

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

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

Downloaded from http://www.everyspec.com

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

Notes:

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

Downloaded from http://www.everyspec.com

5.1.2.25 Creating secondary relationships

```
procedure CREATE_SECONDARY_RELATIONSHIP
                    (TARGET_NODE:  in NODE_TYPE;
                     SOURCE_BASE:  in NODE_TYPE;
                     NEW_KEY:      in RELATIONSHIP_KEY;
                     NEW_RELATION: in RELATION_NAME  := DEFAULT_RELATION;
                     INHERITABLE:  in BOOLEAN  := FALSE);
```

Purpose:

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

Parameters:

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

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

NEW_KEY
: is the relationship key designator for the new secondary relationship.

NEW_RELATION
: is the relation name for the new secondary relationship.

INHERITABLE
: specifies the value of the predefined attribute INHERITABLE of the newly created relationship.

Exceptions:

NAME_ERROR
: is raised if either a primary or secondary relationship already exists with the identification given by SOURCE_BASE, NEW_KEY and NEW_RELATION.

PATHNAME_SYNTAX_ERROR
: is raised if the node identification given by NEW_KEY and NEW_RELATION is syntactically illegal (see Table I, page 32).

PREDEFINED_RELATION_ERROR
: is raised if NEW_RELATION is the name of a predefined relation that cannot be created by the user.

STATUS_ERROR
: is raised if the node handles TARGET_NODE and SOURCE_BASE are not both open.

Downloaded from http://www.everyspec.com

INTENT_VIOLATION
: is raised if SOURCE_BASE was not opened with an intent establishing the right to create relationships.

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

Additional Interface:

```
procedure CREATE_SECONDARY_RELATIONSHIP
                    (TARGET_NODE: in NODE_TYPE;
                     NEW_NAME:    in PATHNAME;
                     INHERITABLE: in BOOLEAN := FALSE)
is
    SOURCE_BASE: NODE_TYPE;
begin
    OPEN (SOURCE_BASE, BASE_PATH(NEW_NAME),
          (1=>APPEND_RELATIONSHIPS));
    CREATE_SECONDARY_RELATIONSHIP (TARGET_NODE, SOURCE_BASE,
                                   LAST_KEY(NEW_NAME),
                                   LAST_RELATION(NEW_NAME),
                                   INHERITABLE);
    CLOSE (SOURCE_BASE);
exception
    when others =>
        CLOSE (SOURCE_BASE);
        raise;
end CREATE_SECONDARY_RELATIONSHIP;
```

Notes:

CREATE_SECONDARY_RELATIONSHIP can be used to create secondary relationships to nodes that are unobtainable or inaccessible.

Downloaded from http://www.everyspec.com

5.1.2.26 Deleting secondary relationships

```
procedure DELETE_SECONDARY_RELATIONSHIP
                        (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 designator of the relationship to be deleted.

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

Exceptions:

PATHNAME_SYNTAX_ERROR
: is raised if the node identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

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

USE_ERROR
: is raised if the relationship given by BASE, KEY and RELATION is a primary relationship.

PREDEFINED_RELATION_ERROR
: is raised if RELATION is the name of a predefined relation that cannot be created by the user.

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

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

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

Downloaded from http://www.everyspec.com

Additional Interface:

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

Notes:

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

Downloaded from http://www.everyspec.com

5.1.2.27 Setting inheritance property of a relationship

```
procedure SET_INHERITANCE
                    (BASE:         in NODE_TYPE;
                     KEY:          in RELATIONSHIP_KEY;
                     RELATION:     in RELATION_NAME := DEFAULT_RELATION;
                     INHERITABLE:  in BOOLEAN);
```

Purpose:

This procedure sets the value of the predefined attribute INHERITABLE of the relationship identified by the BASE, KEY and RELATION parameters to the value of the parameter INHERITABLE.

Parameters:

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

KEY is the relationship key designator of the affected relationship.

RELATION is the relation name of the affected relationship.

INHERITABLE specifies the new value of the predefined attribute INHERITABLE.

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the relationship identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

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

USE_ERROR is raised if the identified relationship is a primary relationship or if it is a relationship of a predefined relation whose inheritance property cannot be altered by the user as specified in section 4.3.4.1.

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.

Downloaded from http://www.everyspec.com

Additional Interface:

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

Downloaded from http://www.everyspec.com

5.1.2.28 Determining if a secondary relationship is inheritable

```
function IS_INHERITABLE (BASE:     in NODE_TYPE;
                         KEY:      in RELATIONSHIP_KEY;
                         RELATION: in RELATION_NAME := DEFAULT_RELATION)
    return BOOLEAN;
```

Purpose:

This function returns the value of the predefined attribute INHERITABLE of the relationship identified by the BASE, KEY and RELATION parameters. For primary relationships, this function always returns FALSE.

Parameters:

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

KEY — is the relationship key designator of the affected relationship.

RELATION — is the relation name of the affected relationship.

Exceptions:

PATHNAME_SYNTAX_ERROR
: is raised if the relationship identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

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

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.

Downloaded from http://www.everyspec.com

Additional Interface:

```
function IS_INHERITABLE (NAME: in PATHNAME)
    return BOOLEAN
is
    BASE: NODE_TYPE;
    VALUE: BOOLEAN;
begin
    OPEN (BASE, BASE_PATH(NAME), (1=>READ_RELATIONSHIPS));
    VALUE := IS_INHERITABLE (BASE, LAST_KEY(NAME), LAST_RELATION(NAME));
    CLOSE (BASE);
    return VALUE;
exception
    when others =>
        CLOSE (BASE);
        raise;
end IS_INHERITABLE;
```

Downloaded from http://www.everyspec.com

5.1.2.29 Node iteration types and subtypes

The following types and subtypes are used in the interfaces for iterating over a set of nodes.

```
type NODE_ITERATOR is limited private;
```

A *node iterator* is an Ada object of the type NODE_ITERATOR, which is a limited private type assumed to contain the bookkeeping information necessary for the implementation of the MORE, GET_NEXT, SKIP_NEXT and NEXT_NAME interfaces. The nodes are returned by GET_NEXT in ASCII lexicographical order by relation name and then by relationship key. The effect on existing iterators of creation or deletion of relationships is implementation-defined.

```
subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
subtype RELATION_NAME_PATTERN    is RELATION_NAME;
```

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

```
type RELATIONSHIP_KIND is (PRIMARY, SECONDARY, BOTH);
type NODE_KIND_ARRAY   is array (CAIS_NATURAL range <>)
                            of NODE_KIND;
```

RELATIONSHIP_KIND is an enumerated type which determines whether the iterator will be based on primary, secondary, or both primary and secondary relationships. NODE_KIND_ARRAY is a type indicating the kind(s) of nodes on which the iterator will be based.

Downloaded from http://www.everyspec.com

5.1.2.30 Creating an iterator over nodes

```
procedure CREATE_ITERATOR
     (ITERATOR:          in out NODE_ITERATOR;
      NODE:              in     NODE_TYPE;
      KIND:              in     NODE_KIND_ARRAY := (FILE, STRUCTURAL);
      KEY:               in     RELATIONSHIP_KEY_PATTERN := "*";
      RELATION:          in     RELATION_NAME_PATTERN := DEFAULT_RELATION;
      KIND_OF_RELATION:  in     RELATIONSHIP_KIND := PRIMARY);
```

Purpose:

This procedure establishes a node iterator ITERATOR over the set of nodes that are the target nodes of relationships emanating from a given node identified by NODE and matching the specified KEY and RELATION patterns. Nodes that are of a kind not contained in the component values of KIND are omitted by subsequent calls to GET_NEXT (see Section 5.1.2.33, page 111), SKIP_NEXT (see Section 5.1.2.34, page 113) or NEXT_NAME (see Section 5.1.2.35, page 114) using the resulting ITERATOR. Depending on the value of KIND_OF_RELATION, nodes reachable by primary or secondary or both primary and secondary relationships will be included on the iterator.

Parameters:

ITERATOR — is the node iterator returned.

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

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

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

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

KIND_OF_RELATION
is an enumeration value; it determines whether the iterator will be based on primary, secondary, or both primary and secondary relationships.

Exceptions:

SYNTAX_ERROR
is raised if the pattern given in KEY or RELATION is syntactically illegal (see Table I, page 32 and Section 5.1.2.29, page 106).

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.

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure CREATE_ITERATOR
     (ITERATOR:          in out NODE_ITERATOR;
      NAME:              in     PATHNAME;
      KIND:              in     NODE_KIND_ARRAY := (FILE, STRUCTURAL);
      KEY:               in     RELATIONSHIP_KEY_PATTERN := "*";
      RELATION:          in     RELATION_NAME_PATTERN := DEFAULT_RELATION;
      KIND_OF_RELATION:  in     RELATIONSHIP_KIND := PRIMARY)
is
     NODE: NODE_TYPE;
begin
     OPEN (NODE, NAME, (1=>READ_RELATIONSHIPS));
     CREATE_ITERATOR (ITERATOR, NODE, KIND, KEY, RELATION,
                      KIND_OF_RELATION);
     CLOSE (NODE);
exception
     when others =>
          CLOSE (NODE);
          raise;
end CREATE_ITERATOR;
```

Notes:

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

Downloaded from http://www.everyspec.com

5.1.2.31 Determining iteration status

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

Purpose:

This function returns FALSE if all nodes contained on the node iterator have been retrieved with the GET_NEXT (see Section 5.1.2.33, page 111) procedure or skipped over with the SKIP_NEXT (see Section 5.1.2.34, page 113) procedure; otherwise it returns TRUE.

Parameter:

ITERATOR — is a node iterator previously set by the procedure CREATE_ITERATOR, page 107.

Exception:

ITERATOR_ERROR
is raised if the ITERATOR has not been previously set by the procedure CREATE_ITERATOR (see Section 5.1.2.30, page 107) or has been set but subsequently deleted by the procedure DELETE_ITERATOR (see Section 5.1.2.36, page 115) at the time of the call on MORE.

Downloaded from http://www.everyspec.com

5.1.2.32 Determining the approximate size of the iterator

```
function APPROXIMATE_SIZE (ITERATOR: in NODE_ITERATOR)
    return CAIS_NATURAL;
```

Purpose:

This function returns the approximate number of elements on the iterator at the moment of the call. Calls on GET_NEXT or SKIP_NEXT have no influence on the value returned by this function.

Parameter:

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

Exception:

ITERATOR_ERROR
is raised if the ITERATOR has not been previously set by the procedure CREATE_ITERATOR (see Section 5.1.2.30, page 107) or has been set but subsequently deleted by the procedure DELETE_ITERATOR (see Section 5.1.2.36, page 115) at the time of the call on APPROXIMATE_SIZE.

Notes:

This interface should not be used in loops of the form:

```
for I in 1 .. APPROXIMATE_SIZE (ITERATOR) loop
    GET_NEXT (ITERATOR, NEXT_NODE);
end loop;
```

since the deletion of relationships may reduce the number of node handles returned by the repeated calls on GET_NEXT.

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

Downloaded from http://www.everyspec.com

5.1.2.33 Getting the next node in an iteration

```
procedure GET_NEXT (ITERATOR:   in out  NODE_ITERATOR;
                    NEXT_NODE:  in out  NODE_TYPE;
                    INTENT:     in      INTENT_ARRAY;
                    TIME_LIMIT: in      CAIS_DURATION := LONG_DELAY);
```

Purpose:

This procedure returns an open node handle to the next node on the iterator in the parameter NEXT_NODE; the intent under which the node handle is opened is specified by the INTENT parameter. If NEXT_NODE is open prior to the call to GET_NEXT, it is closed prior to being opened to the next node. A time limit can be specified for the maximum delay permitted if the node to be opened is locked against access with the specified INTENT. If an exception is raised by the call on GET_NEXT, the next call on GET_NEXT for the same iterator will attempt to return an open node handle to the same node, i.e., the iterator is not advanced by a call resulting in an exception.

Parameters:

ITERATOR — is a node iterator previously set by the procedure CREATE_ITERATOR (see Section 5.1.2.30, page 107).

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 CAIS_DURATION, specifying a time limit for the delay on waiting for the unlocking of the node in accordance with the desired INTENT.

Exceptions:

NAME_ERROR — is raised if the node whose node handle is to be returned in NEXT_NODE is unobtainable or inaccessible and the INTENT includes any intent other than NO_ACCESS.

ITERATOR_ERROR — is raised if the ITERATOR has not been previously set by the procedure CREATE_ITERATOR (see Section 5.1.2.30, page 107), if the iterator has been set but subsequently deleted by the procedure DELETE_ITERATOR (see Section 5.1.2.36, page 115) prior to the call on GET_NEXT, or if the iterator is exhausted.

USE_ERROR — is raised 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. LOCK_ERROR may be raised prior to expiration of the timeout if the CAIS implementation can determine that a deadlock situation has occurred.

Downloaded from http://www.everyspec.com

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

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

Additional interface:

```
procedure GET_NEXT (ITERATOR:   in out NODE_ITERATOR;
                    NEXT_NODE:  in out NODE_TYPE;
                    INTENT:     in     INTENT_SPECIFICATION := NO_ACCESS;
                    TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY)
is
begin
    GET_NEXT (ITERATOR, NEXT_NODE, (1=>INTENT), TIME_LIMIT);
end GET_NEXT;
```

Downloaded from http://www.everyspec.com

5.1.2.34 Skipping the next node in an iteration

```
procedure SKIP_NEXT (ITERATOR: in out NODE_ITERATOR);
```

Purpose:

This procedure advances the iterator to the next node on the iterator without returning an open handle to this node.

Parameter:

ITERATOR — is a node iterator previously set by the procedure CREATE_ITERATOR (see Section 5.1.2.30, page 107).

Exception:

ITERATOR_ERROR
: is raised if the ITERATOR has not been previously set by the procedure CREATE_ITERATOR (see Section 5.1.2.30, page 107), if the iterator has been set but subsequently deleted by the procedure DELETE_ITERATOR (see Section 5.1.2.36, page 115) prior to the call on SKIP_NEXT, or if the iterator is exhausted.

Notes:

This procedure can be used to advance the iterator across a node for which GET_NEXT resulted in an exception.

Downloaded from http://www.everyspec.com

5.1.2.35 Obtaining the path element for the next node in an iteration

```
function NEXT_NAME (ITERATOR: in NODE_ITERATOR)
    return PATHNAME;
```

Purpose:

This function returns a path element, composed of the relation name and relationship key of the relationship which caused the next node to be included in the iteration. The returned value has the syntax of a path element (see Table I, page 32); it can be submitted to LAST_KEY and LAST_RELATION for obtaining the constituent values. The iterator is not advanced by this call, i.e., it is possible to call GET_NEXT to obtain a node handle on this node or to call SKIP_NEXT to advance the iterator to the next node.

Parameter:

ITERATOR is a node iterator previously set by the procedure CREATE_ITERATOR, page 107.

Exception:

ITERATOR_ERROR
is raised if the ITERATOR has not been previously set by the procedure CREATE_ITERATOR (see Section 5.1.2.30, page 107), or if the iterator has been set but subsequently deleted by the procedure DELETE_ITERATOR (see Section 5.1.2.36, page 115) prior to the call on NEXT_NAME or if the iterator is exhausted (i.e., the value of MORE (ITERATOR) is FALSE).

Downloaded from http://www.everyspec.com

5.1.2.36 Deleting an iterator

```
procedure DELETE_ITERATOR (ITERATOR: in out NODE_ITERATOR);
```

Purpose:

This procedure deletes an iterator. The value of its parameter after the call is as if it were never set by CREATE_ITERATOR (see Section 5.1.2.30, page 107). Deleting an iterator that is not set has no effect.

Parameter:

ITERATOR is a node iterator.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.1.2.37 Setting the current node relationship

```
procedure SET_CURRENT_NODE (NODE:       in NODE_TYPE;
                            TIME_LIMIT: in CAIS_DURATION := LONG_DELAY);
```

Purpose:

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

Parameters:

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

TIME_LIMIT
: is a value of type CAIS_DURATION specifying a time limit for the delay on waiting for access to the current process node with intent WRITE_RELATIONSHIPS.

Exceptions:

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

LOCK_ERROR
: is raised if access, with intent WRITE_RELATIONSHIPS, to the current process node cannot be obtained within the specified TIME_LIMIT due to an existing lock on the node. LOCK_ERROR may be raised prior to expiration of the timeout if the CAIS implementation can determine that a deadlock situation has occurred.

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 PATHNAME;
                            TIME_LIMIT: in CAIS_DURATION := LONG_DELAY)
is
    NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (1=>NO_ACCESS));
    SET_CURRENT_NODE (NODE, TIME_LIMIT);
exception
    when others =>
        CLOSE (NODE);
        raise;
end SET_CURRENT_NODE;
```

Downloaded from http://www.everyspec.com

5.1.2.38 Opening a node handle to the current node

```
procedure GET_CURRENT_NODE
              (NODE:        in out NODE_TYPE;
               INTENT:      in     INTENT_ARRAY;
               TIME_LIMIT:  in     CAIS_DURATION := LONG_DELAY);
```

Purpose:

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

Parameters:

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

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

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

Exceptions:

NAME_ERROR is raised if the current node is inaccessible or unobtainable and the INTENT contains any intent specification other than NO_ACCESS.

USE_ERROR is raised if INTENT is an empty array.

STATUS_ERROR
is raised if NODE is an open node handle at the time of the call.

LOCK_ERROR is raised if access, with intent READ_RELATIONSHIPS, to the current process node cannot be obtained within the specified TIME_LIMIT due to an existing lock on the node. LOCK_ERROR may be raised prior to expiration of the timeout if the CAIS implementation can determine that a deadlock situation has occurred.

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

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

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure GET_CURRENT_NODE
                   (NODE:        in out NODE_TYPE;
                    INTENT:      in     INTENT_SPECIFICATION := NO_ACCESS;
                    TIME_LIMIT:  in     CAIS_DURATION := LONG_DELAY)
is
begin
     GET_CURRENT_NODE (NODE, (1=>INTENT), TIME_LIMIT);
end GET_CURRENT_NODE;
```

Notes:

The call on GET_CURRENT_NODE is equivalent to

```
OPEN (NODE, "'CURRENT_NODE", INTENT, TIME_LIMIT);
```

Downloaded from http://www.everyspec.com

5.1.2.39 Determining the creation time of a node

```
function TIME_CREATED (NODE: in NODE_TYPE)
    return CAIS_CALENDAR.TIME;
```

Purpose:

This function returns a value of type CAIS_CALENDAR.TIME representing the value of the predefined attribute TIME_CREATED of the node identified by NODE. The value returned is the time at which the node was created.

Parameter:

NODE — is an open node handle identifying the node whose attribute is being queried.

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 to read attributes.

Additional Interface:

```
function TIME_CREATED (NAME: in PATHNAME)
    return CAIS_CALENDAR.TIME
is
    NODE:    NODE_TYPE;
    RESULT: CAIS_CALENDAR.TIME;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := TIME_CREATED (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end TIME_CREATED;
```

Downloaded from http://www.everyspec.com

5.1.2.40 Determining the last time a relationship was modified

```
function TIME_RELATIONSHIP_WRITTEN (NODE: in NODE_TYPE)
    return CAIS_CALENDAR.TIME;
```

Purpose:

This function returns a value of type CAIS_CALENDAR.TIME representing the value of the predefined attribute TIME_RELATIONSHIP_WRITTEN of the node identified by NODE. The value returned is the time at which any relationship on the node identified by NODE was modified (i.e., a new relationship added or an existing relationship deleted) or at which any attributes of any relationship emanating from the node were modified (i.e., value of an attribute of the relationship changed by the user, a new attribute of the relationship added or an existing attribute of the relationship deleted). Changes to relationships that are maintained by the implementation and cannot be set using CAIS interfaces are not reflected in TIME_RELATIONSHIP_WRITTEN.

Parameter:

NODE — is an open node handle identifying the node whose attribute is being queried.

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 to read attributes.

Additional Interface:

```
function TIME_RELATIONSHIP_WRITTEN (NAME: in PATHNAME)
    return CAIS_CALENDAR.TIME
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_CALENDAR.TIME;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := TIME_RELATIONSHIP_WRITTEN (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end TIME_RELATIONSHIP_WRITTEN;
```

Downloaded from http://www.everyspec.com

5.1.2.41 Determining the last time that node contents were written

```
function TIME_CONTENTS_WRITTEN (NODE: in NODE_TYPE)
    return CAIS_CALENDAR.TIME;
```

Purpose:

This function returns a value of type CAIS_CALENDAR.TIME representing the value of the predefined attribute TIME_CONTENTS_WRITTEN of the file node identified by NODE. The value returned is the time at which the file contents of the node were last modified (i.e., written).

Parameter:

NODE — is an open node handle identifying the node whose attribute is being queried.

Exceptions:

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

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a file node.

INTENT_VIOLATION
: is raised if NODE was not opened with an intent to read attributes.

Additional Interface:

```
function TIME_CONTENTS_WRITTEN (NAME: in PATHNAME)
    return CAIS_CALENDAR.TIME
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_CALENDAR.TIME;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := TIME_CONTENTS_WRITTEN (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end TIME_CONTENTS_WRITTEN;
```

Downloaded from http://www.everyspec.com

5.1.2.42 Determining the last time an attribute was modified

```
function TIME_ATTRIBUTE_WRITTEN (NODE: in NODE_TYPE)
    return CAIS_CALENDAR.TIME;
```

Purpose:

This function returns a value of type CAIS_CALENDAR.TIME representing the value of the predefined attribute TIME_ATTRIBUTE_WRITTEN of the node identified by NODE. The value returned is the time at which any attribute on the node identified by NODE was modified (i.e., an attribute value changed by the user, a new attribute added or an existing attribute deleted) by a call on a CAIS interface. Changes to attributes that are made implicitly by the CAIS implementation are not reflected in the result of TIME_ATTRIBUTE_WRITTEN.

Parameter:

NODE — is an open node handle identifying the node whose attribute is being queried.

Exception:

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

INTENT_VIOLATION
: is raised if NODE was not opened with an intent to read attributes.

Additional Interface:

```
function TIME_ATTRIBUTE_WRITTEN (NAME: in PATHNAME)
    return CAIS_CALENDAR.TIME
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_CALENDAR.TIME;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := TIME_ATTRIBUTE_WRITTEN (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end TIME_ATTRIBUTE_WRITTEN;
```

Notes:

Updating the attributes TIME_CONTENTS_WRITTEN and TIME_RELATIONSHIP_WRITTEN does not affect the value of the TIME_ATTRIBUTE_WRITTEN attribute.

Downloaded from http://www.everyspec.com

5.1.3 Package CAIS_ATTRIBUTE_MANAGEMENT

This package supports the definition and manipulation of attributes for nodes and relationships. The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_PRAGMATICS.

The name of an attribute follows the syntax of an Ada identifier. The value of each attribute is a list; the format of the list is defined by the package CAIS_LIST_MANAGEMENT (see Section 5.4). Upper and lower case distinctions are not significant within the attribute names.

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

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

While there are special interfaces to retrieve the value of predefined attributes in the various CAIS packages, such values can also be retrieved by the general attribute manipulation interfaces of the package CAIS_ATTRIBUTE_MANAGEMENT. It can be reasonably assumed that it will not be possible to implement the latter retrievals as efficiently as the former ones.

For predefined attributes, the following rules apply regarding the nature of the LIST_TYPE value returned by calls on the general attribute manipulation interfaces:

a. Retrieval of the value of predefined attributes described as being of integer type by the interfaces of package CAIS_ATTRIBUTE_MANAGEMENT will yield an unnamed list of a single integer-valued list item.

b. Retrieval of the value of predefined attributes described as being of type LIST_TYPE by the interfaces of package CAIS_ATTRIBUTE_MANAGEMENT will yield the respective list value.

c. Retrieval of the value of predefined attributes described as being of enumeration type by the interfaces of package CAIS_ATTRIBUTE_MANAGEMENT will yield an unnamed list of a single token-valued list item. The string representation of this token is equal to the result of applying the IMAGE attribute of the enumeration type to the enumeration value corresponding to the token.

d. Retrieval of the value of predefined attributes, whose value is described as being a combination of one or more values of enumeration type, by the interfaces of package CAIS_ATTRIBUTE_MANAGEMENT will yield an unnamed list of token-valued list items. The string representation of each token is equal to the result of applying the IMAGE attribute of the enumeration type to the enumeration value corresponding to the token.

Downloaded from http://www.everyspec.com

e. Retrieval of the value of predefined attributes described as being of type CAIS_DURATION will yield an unnamed list of a single integer-valued list item. This integer value represents the attribute value in multiples of CAIS_PRAGMATICS.SMALL_FOR_CAIS_DURATION.

f. Retrieval of the value of predefined attributes of type CAIS_CALENDAR.TIME by the interfaces of package CAIS_ATTRIBUTE_MANAGEMENT will yield a named list with four integer-valued components. The component names are, in order, YEAR, MONTH, DAY and SECONDS. The component values are the values as obtained when applying the procedure CAIS_CALENDAR.SPLIT to a value of type CAIS_CALENDAR.TIME, except that the SECONDS component is an integer value which represents the SECONDS component of CAIS_CALENDAR.TIME in multiples of CAIS_PRAGMATICS.SMALL_FOR_CAIS_DURATION.

Downloaded from http://www.everyspec.com

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:

SYNTAX_ERROR
is raised if the attribute name given is not a valid Ada identifier.

PREDEFINED_ATTRIBUTE_ERROR
is raised if ATTRIBUTE is the name of a predefined node attribute that cannot be created by the user.

ATTRIBUTE_ERROR
is raised if the node already has 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 create 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.

Downloaded from http://www.everyspec.com

Additional Interface:

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

Downloaded from http://www.everyspec.com

5.1.3.2 Creating path attributes

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

Purpose:

This procedure creates an attribute, named by ATTRIBUTE, of a relationship and sets its initial value to VALUE. The relationship is identified by the BASE, KEY and RELATION parameters.

Parameters:

BASE	is an open node handle to the node from which the relationship emanates.
KEY	is the relationship key designator 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:

PATHNAME_SYNTAX_ERROR
: is raised if the relationship identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

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

SYNTAX_ERROR
: is raised if the attribute name given is not a valid Ada identifier.

PREDEFINED_RELATION_ERROR
: is raised if RELATION is the name of a predefined relation that cannot be modified by the user.

PREDEFINED_ATTRIBUTE_ERROR
: is raised if ATTRIBUTE is the name of a predefined relationship attribute that cannot be created by the user.

ATTRIBUTE_ERROR
: is raised if the relationship already has an attribute of the given name.

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

Downloaded from http://www.everyspec.com

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

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

Additional Interface:

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

Downloaded from http://www.everyspec.com

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:

SYNTAX_ERROR
is raised if the attribute name given is not a valid Ada identifier.

PREDEFINED_ATTRIBUTE_ERROR
is raised if ATTRIBUTE is the name of a predefined node attribute that cannot be modified by the user.

ATTRIBUTE_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.

Downloaded from http://www.everyspec.com

Additional Interface:

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

Downloaded from http://www.everyspec.com

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 BASE, KEY and RELATION.

Parameters:

BASE	is an open node handle to the node from which the relationship emanates.
KEY	is the relationship key designator of the affected relationship.
RELATION	is the relation name of the affected relationship.
ATTRIBUTE	is the name of the attribute to be deleted.

Exceptions:

PATHNAME_SYNTAX_ERROR
: is raised if the relationship identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

SYNTAX_ERROR
: is raised if the attribute name given is not a valid Ada identifier.

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

PREDEFINED_RELATION_ERROR
: is raised if RELATION is the name of a predefined relation that cannot be modified by the user.

PREDEFINED_ATTRIBUTE_ERROR
: is raised if ATTRIBUTE is the name of a predefined relationship attribute that cannot be modified by the user.

ATTRIBUTE_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.

Downloaded from http://www.everyspec.com

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

Downloaded from http://www.everyspec.com

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 the open node handle NODE.

Parameters:

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

ATTRIBUTE — is the name of the attribute.

VALUE — is the new value of the attribute.

Exceptions:

SYNTAX_ERROR
is raised if the attribute name given is not a valid Ada identifier.

PREDEFINED_ATTRIBUTE_ERROR
is raised if ATTRIBUTE is the name of a predefined node attribute that cannot be modified by the user.

ATTRIBUTE_ERROR
is raised if the node does not have an 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.

Downloaded from http://www.everyspec.com

Additional Interface:

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

Downloaded from http://www.everyspec.com

5.1.3.6 *Setting path attributes*

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

Purpose:

This procedure sets the value of the relationship attribute named by ATTRIBUTE to the value specified by VALUE. The relationship is identified by BASE, KEY and RELATION.

Parameters:

BASE	is an open node handle to the node from which the relationship emanates.
KEY	is the relationship key designator 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:

PATHNAME_SYNTAX_ERROR
: is raised if the relationship identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

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

SYNTAX_ERROR
: is raised if the attribute name given is not a valid Ada identifier.

PREDEFINED_RELATION_ERROR
: is raised if RELATION is the name of a predefined relation that cannot be modified by the user.

PREDEFINED_ATTRIBUTE_ERROR
: is raised if ATTRIBUTE is the name of a predefined relationship attribute that cannot be modified by the user.

ATTRIBUTE_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.

Downloaded from http://www.everyspec.com

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

Downloaded from http://www.everyspec.com

5.1.3.7 Getting node attributes

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

Purpose:

This procedure returns the value of the node attribute named by ATTRIBUTE in the parameter VALUE, in accordance with the rules given in Section 5.1.3, page 123. The node is identified by the open node handle NODE.

Parameters:

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

ATTRIBUTE — is the name of the attribute.

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

Exceptions:

SYNTAX_ERROR
is raised if the attribute name given is not a valid Ada identifier.

ATTRIBUTE_ERROR
is raised if the node does not have an attribute of the given name or if the name designates a predefined attribute for mandatory access control purposes.

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.

Downloaded from http://www.everyspec.com

Additional Interface:

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

Downloaded from http://www.everyspec.com

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 returns the value of the relationship attribute named by ATTRIBUTE in the parameter VALUE, in accordance with the rules given in Section 5.1.3, page 123. The relationship is identified by BASE, KEY and RELATION.

Parameters:

BASE	is an open node handle to the node from which the relationship emanates.
KEY	is the relationship key designator of the affected relationship.
RELATION	is the relation name of the affected relationship.
ATTRIBUTE	is the name of the attribute.
VALUE	is the result parameter containing the value of the attribute.

Exceptions:

PATHNAME_SYNTAX_ERROR
: is raised if the relationship identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

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

SYNTAX_ERROR
: is raised if the attribute name given is not a valid Ada identifier.

ATTRIBUTE_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.

Downloaded from http://www.everyspec.com

Additional Interface:

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

Downloaded from http://www.everyspec.com

5.1.3.9 Attribute iteration types and subtypes

```
type    ATTRIBUTE_ITERATOR     is limited private;
subtype ATTRIBUTE_NAME_PATTERN is STRING;
```

An *attribute iterator* is an Ada object of the type ATTRIBUTE_ITERATOR, which is a limited private type assumed to contain the bookkeeping information necessary for the implementation of the MORE, NEXT_NAME, GET_NEXT_VALUE and SKIP_NEXT interfaces. The attributes are returned by NEXT_NAME and GET_NEXT_VALUE in ASCII lexicographical order by attribute name. Predefined attributes for mandatory access control purposes are omitted by the iterator. The effect on existing iterators of creation or deletion of attributes or relationships is implementation-defined.

These types and subtypes are used in the following interfaces for iteration over a set of attributes of nodes or relationships. An ATTRIBUTE_NAME_PATTERN has the same syntax as an ATTRIBUTE_NAME, except that "?" will match any single character and "*" will match any string of zero or more characters.

Downloaded from http://www.everyspec.com

5.1.3.10 Creating an iterator over node attributes

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

Purpose:

This procedure returns in the parameter ITERATOR an attribute iterator according to the semantic rules for attribute selection given in Section 5.1.3.9. The iterator can then be processed by means of the MORE, NEXT_NAME, GET_NEXT_VALUE and SKIP_NEXT interfaces.

Parameters:

ITERATOR is the attribute iterator returned.

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

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

Exceptions:

SYNTAX_ERROR
is raised if the PATTERN is syntactically illegal (see Section 4.3.6 and Section 5.1.3.9).

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 CREATE_NODE_ATTRIBUTE_ITERATOR
                    (ITERATOR: in out ATTRIBUTE_ITERATOR;
                     NAME:     in     PATHNAME;
                     PATTERN:  in     ATTRIBUTE_NAME_PATTERN := "*")
is
    NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    CREATE_NODE_ATTRIBUTE_ITERATOR (ITERATOR, NODE, PATTERN);
    CLOSE (NODE);
exception
    when others =>
        CLOSE (NODE);
        raise;
end CREATE_NODE_ATTRIBUTE_ITERATOR;
```

Downloaded from http://www.everyspec.com

Notes:

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

Downloaded from http://www.everyspec.com

5.1.3.11 Creating an iterator over relationship attributes

```
procedure CREATE_PATH_ATTRIBUTE_ITERATOR
                    (ITERATOR:  in out ATTRIBUTE_ITERATOR;
                     BASE:      in     NODE_TYPE;
                     KEY:       in     RELATIONSHIP_KEY;
                     RELATION:  in     RELATION_NAME := DEFAULT_RELATION;
                     PATTERN:   in     ATTRIBUTE_NAME_PATTERN := "*");
```

Purpose:

This procedure is provided to obtain an attribute iterator for relationship attributes. The relationship is identified by BASE, KEY and RELATION. 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, NEXT_NAME, GET_NEXT_VALUE and SKIP_NEXT interfaces.

Parameters:

ITERATOR	is the attribute iterator returned.
BASE	is an open node handle to the node from which the relationship emanates.
KEY	is the relationship key designator of the affected relationship.
RELATION	is the relation name of the affected relationship.
PATTERN	is a pattern for attribute names (see Section 5.1.3.9).

Exceptions:

PATHNAME_SYNTAX_ERROR
: is raised if the relationship identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

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

SYNTAX_ERROR
: is raised if PATTERN is syntactically illegal (see Section 4.3.6 and Section 5.1.3.9).

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.

Downloaded from http://www.everyspec.com

Additional Interface:

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

Notes:

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

Downloaded from http://www.everyspec.com

5.1.3.12 Determining iteration status

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

Purpose:

This function returns FALSE if all attributes contained on the attribute iterator have been retrieved with the subprograms SKIP_NEXT and GET_NEXT_VALUE; otherwise, it returns TRUE.

Parameter:

ITERATOR is an attribute iterator previously constructed.

Exception:

ITERATOR_ERROR
is raised if the ITERATOR has not been previously set by the procedures CREATE_NODE_ATTRIBUTE_ITERATOR (Section 5.1.3.10, page 142) or CREATE_PATH_ATTRIBUTE_ITERATOR (Section 5.1.3.11, page 144) or if the iterator has been subsequently deleted by the procedure DELETE_ITERATOR (Section 5.1.3.17, page 151) prior to the call on MORE.

Downloaded from http://www.everyspec.com

5.1.3.13 Determining the approximate size of the iterator

```
function APPROXIMATE_SIZE (ITERATOR: in ATTRIBUTE_ITERATOR)
    return CAIS_NATURAL;
```

Purpose:

This function returns the approximate number of elements on the attribute iterator at the moment of the call. Calls on NEXT_NAME, GET_NEXT_VALUE or SKIP_NEXT have no influence on the value returned by this function.

Parameter:

ITERATOR is an attribute iterator previously constructed.

Exception:

ITERATOR_ERROR
is raised if the ITERATOR has not been previously set by the procedures CREATE_NODE_ATTRIBUTE_ITERATOR (Section 5.1.3.10, page 142) or CREATE_PATH_ATTRIBUTE_ITERATOR (Section 5.1.3.11, page 144) or if the iterator has been subsequently deleted by the procedure DELETE_ITERATOR (Section 5.1.3.17, page 151) prior to the call on APPROXIMATE_SIZE.

Notes:

This interface should not be used in loops of the form

```
for I in 1 .. APPROXIMATE_SIZE (ITERATOR) loop
    NEXT_ATTRIBUTE_NAME := NEXT_NAME (ITERATOR);
    GET_NEXT_VALUE (ITERATOR, NEXT_ATTRIBUTE_VALUE);
end loop;
```

since the deletion of attributes may reduce the number of attributes returned by the repeated calls on NEXT_NAME and GET_NEXT_VALUE.

Downloaded from http://www.everyspec.com

5.1.3.14 Getting the next attribute name

```
function NEXT_NAME (ITERATOR: in ATTRIBUTE_ITERATOR)
    return ATTRIBUTE_NAME;
```

Purpose:

This function returns the name of the next attribute on the iterator without advancing the iterator.

Parameter:

ITERATOR is an attribute iterator previously constructed.

Exception:

ITERATOR_ERROR
is raised if the ITERATOR has not been previously set by the procedures CREATE_NODE_ATTRIBUTE_ITERATOR (Section 5.1.3.10, page 142) or CREATE_PATH_ATTRIBUTE_ITERATOR (Section 5.1.3.11, page 144), if the iterator has been subsequently deleted by the procedure DELETE_ITERATOR (Section 5.1.3.17, page 151) prior to the call on NEXT_NAME, or if the iterator is exhausted.

Downloaded from http://www.everyspec.com

5.1.3.15 Getting the next attribute value

```
procedure GET_NEXT_VALUE (ITERATOR: in out ATTRIBUTE_ITERATOR;
                          VALUE:    in out LIST_TYPE);
```

Purpose:

This procedure returns the value of the next attribute on the iterator in VALUE, in accordance with the rules given in Section 5.1.3, page 123, and then advances the iterator to the next attribute on the iterator (i.e., the one corresponding to the returned value).

Parameters:

ITERATOR is an attribute iterator previously constructed.

VALUE is the value of the next attribute on the iterator.

Exception:

ITERATOR_ERROR

is raised if the ITERATOR has not been previously set by the procedures CREATE_NODE_ATTRIBUTE_ITERATOR (Section 5.1.3.10, page 142) or CREATE_PATH_ATTRIBUTE_ITERATOR (Section 5.1.3.11, page 144), if the iterator has been subsequently deleted by the procedure DELETE_ITERATOR (Section 5.1.3.17, page 151) prior to the call on GET_NEXT_VALUE, or if the iterator is exhausted.

Downloaded from http://www.everyspec.com

5.1.3.16 Skipping the next attribute in an iteration

```
procedure SKIP_NEXT (ITERATOR: in ATTRIBUTE_ITERATOR);
```

Purpose:

This procedure advances the iterator to the next attribute on the iterator.

Parameter:

ITERATOR is an attribute iterator previously constructed.

Exception:

ITERATOR_ERROR
is raised if the ITERATOR has not been previously set by the procedures CREATE_NODE_ATTRIBUTE_ITERATOR (Section 5.1.3.10, page 142) or CREATE_PATH_ATTRIBUTE_ITERATOR (Section 5.1.3.11, page 144), if the iterator has been subsequently deleted by the procedure DELETE_ITERATOR (Section 5.1.3.17, page 151) prior to the call on SKIP_NEXT, or if the iterator is exhausted.

Downloaded from http://www.everyspec.com

5.1.3.17 Deleting an attribute iterator

```
procedure DELETE_ITERATOR (ITERATOR: in out ATTRIBUTE_ITERATOR);
```

Purpose:

This procedure deletes an attribute iterator. The value of its parameter after the call is as if it were never set by the procedures CREATE_NODE_ATTRIBUTE_ITERATOR (Section 5.1.3.10, page 142) or CREATE_PATH_ATTRIBUTE_ITERATOR (Section 5.1.3.11, page 144). Deleting an iterator that is not set has no effect.

Parameter:

ITERATOR is an attribute iterator.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.1.4 Package CAIS_ACCESS_CONTROL_MANAGEMENT

This package provides primitives for manipulating access control information for CAIS nodes. In addition, certain CAIS subprograms declared elsewhere (e.g., the node creation interfaces, the node open interfaces, SPAWN_PROCESS, INVOKE_PROCESS and CREATE_JOB) allow the specification of initial access control information. The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_PRAGMATICS.

The CAIS specifies mechanisms for discretionary and mandatory access control (see [TCSEC]). Alternate discretionary or mandatory access control mechanisms can be substituted by an implementation provided that the semantics of all interfaces in Section 5 (with the exception of Section 5.1.4) are implemented as specified. These alternate mechanisms as well as the implementation behavior of such a replacement package must be included in an implementer's CAIS reference manual as described in Appendix E of this document.

5.1.4.1 Subtypes

```
subtype GRANT_VALUE   is CAIS_LIST_MANAGEMENT.LIST_TYPE;
subtype ACCESS_RIGHTS is STRING;
```

GRANT_VALUE is a subtype for values of GRANT attributes; it is a list in the syntax described in Table II, page 42.

ACCESS_RIGHTS is a subtype for values of access rights.

Downloaded from http://www.everyspec.com

5.1.4.2 Value of all access rights

```
function ALL_RIGHTS
    return DISCRETIONARY_ACCESS_LIST;
```

Purpose:

This function returns a value of type DISCRETIONARY_ACCESS_LIST, which, when installed as the value of the GRANT attribute on an access relationship, grants all predefined discretionary access rights. The list value consists of a single grant item (see Table II, page 42) without necessary right and a resulting rights list consisting of the identifier ALL_RIGHTS (see Table III, page 44).

Parameters:

None.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.1.4.3 Setting access control

```
procedure SET_GRANTED_RIGHTS (NODE:       in NODE_TYPE;
                              GROUP_NODE: in NODE_TYPE;
                              GRANT:      in GRANT_VALUE);
```

Purpose:

This procedure sets access control information for a given node. If a relationship of the predefined relation ACCESS does not exist from the node identified by NODE to the node identified by GROUP_NODE, such a relationship with an implementation-defined relationship key is created from the node specified by NODE to the node specified by GROUP_NODE. If necessary, the predefined attribute GRANT is created on this relationship. The value of the GRANT attribute is set to the value of the GRANT parameter (see Table II, page 42, for the syntax).

Parameters:

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

GROUP_NODE is an open node handle to a group node.

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

Exceptions:

SYNTAX_ERROR
is raised if the value specified for the parameter GRANT is syntactically illegal (see Table II, page 42).

NODE_KIND_ERROR
is raised if GROUP_NODE is not an open node handle to a group node.

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

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

ACCESS_VIOLATION
is raised if the executing process (subject) is not allowed to establish or alter an access relationship to the given group node according to implementation-defined criteria.

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.

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure SET_GRANTED_RIGHTS (NAME:        in PATHNAME;
                              GROUP_NAME:  in PATHNAME;
                              GRANT:       in GRANT_VALUE)
is
    NODE, GROUP_NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (1=>CONTROL));
    OPEN (GROUP_NODE, GROUP_NAME, (1=>NO_ACCESS));
    SET_GRANTED_RIGHTS (NODE, GROUP_NODE, GRANT);
    CLOSE (NODE);
    CLOSE (GROUP_NODE);
exception
    when others =>
        CLOSE (NODE);
        CLOSE (GROUP_NODE);
        raise;
end SET_GRANTED_RIGHTS;
```

Downloaded from http://www.everyspec.com

5.1.4.4 Deleting access relationships

```
procedure DELETE_GRANTED_RIGHTS (NODE:       in NODE_TYPE;
                                 GROUP_NODE: in NODE_TYPE);
```

Purpose:

This procedure deletes access control information for a given node. If a relationship of the predefined relation ACCESS exists from the node identified by NODE to the node identified by GROUP_NODE, it is deleted. If no access relationship exists from the node identified by NODE to the node identified by GROUP_NODE, this interface has no effect, and no error indication is given.

Parameters:

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

GROUP_NODE is an open node handle to a group node.

Exceptions:

NODE_KIND_ERROR
is raised if GROUP_NODE is not an open node handle to a group node.

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

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

ACCESS_VIOLATION
is raised if the executing process (subject) is not allowed to establish or alter an access relationship to the given group node according to implementation-defined criteria.

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.

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure DELETE_GRANTED_RIGHTS (NAME:        in PATHNAME;
                                 GROUP_NAME:  in PATHNAME)
is
    NODE, GROUP_NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (1=>CONTROL));
    OPEN (GROUP_NODE, GROUP_NAME, (1=>NO_ACCESS));
    DELETE_GRANTED_RIGHTS (NODE, GROUP_NODE);
    CLOSE (NODE);
    CLOSE (GROUP_NODE);
exception
    when others =>
        CLOSE (NODE);
        CLOSE (GROUP_NODE);
        raise;
end DELETE_GRANTED_RIGHTS;
```

Downloaded from http://www.everyspec.com

5.1.4.5 Obtaining the value of a GRANT attribute

```
procedure GET_GRANTED_RIGHTS (NODE:       in     NODE_TYPE;
                              GROUP_NODE: in     NODE_TYPE;
                              GRANT:      in out GRANT_VALUE);
```

Purpose:

This procedure retrieves the value of the GRANT attribute of the access relationship between the object NODE and the group node GROUP_NODE. It returns the empty list if no such relationship exists.

Parameters:

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

GROUP_NODE is an open node handle to a group node.

GRANT is, upon return, a list describing what access rights are granted to the group represented by the group node for the object NODE.

Exceptions:

NODE_KIND_ERROR
is raised if GROUP_NODE is not an open node handle to a group node.

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

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

Additional Interface:

```
procedure GET_GRANTED_RIGHTS (NAME:       in     PATHNAME;
                              GROUP_NAME: in     PATHNAME;
                              GRANT:      in out GRANT_VALUE)
is
    NODE, GROUP_NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (1=>READ_RELATIONSHIPS));
    OPEN (GROUP_NODE, GROUP_NAME, (1=>NO_ACCESS));
    GET_GRANTED_RIGHTS (NODE, GROUP_NODE, GRANT);
    CLOSE (NODE);
    CLOSE (GROUP_NODE);
exception
    when others =>
        CLOSE (NODE);
        CLOSE (GROUP_NODE);
        raise;
end GET_GRANTED_RIGHTS;
```

Downloaded from http://www.everyspec.com

5.1.4.6 Examining access rights

```
function IS_APPROVED (OBJECT_NODE:  in NODE_TYPE;
                      ACCESS_RIGHT: in ACCESS_RIGHTS)
    return BOOLEAN;
```

Purpose:

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

Parameters:

OBJECT_NODE
: is an open node handle to the object node.

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

Exceptions:

SYNTAX_ERROR
: is raised if the parameter ACCESS_RIGHT is not a valid Ada identifier.

STATUS_ERROR
: is raised if OBJECT_NODE is not an open node handle.

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

Additional Interface:

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

Downloaded from http://www.everyspec.com

5.1.4.7 Adopting a role

```
procedure ADOPT_ROLE (GROUP_NODE:   in NODE_TYPE;
                      KEY:          in RELATIONSHIP_KEY := LATEST_KEY;
                      INHERITABLE:  in BOOLEAN := TRUE);
```

Purpose:

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

Parameters:

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

KEY is a relationship key designator to be used in creating the relationship.

INHERITABLE specifies the value of the predefined attribute INHERITABLE of the newly created relationship.

Exceptions:

SYNTAX_ERROR
is raised if KEY is syntactically illegal (see Table I, page 32).

NODE_KIND_ERROR
is raised if GROUP_NODE is not an open node handle to a group node.

USE_ERROR is raised if there is no adopted role of the current process that is a potential member of the group represented by GROUP_NODE or if there already exists a relationship of the predefined relation ADOPTED_ROLE with relationship key designator KEY emanating from the current process node.

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

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

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

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure ADOPT_ROLE (GROUP_NAME:  in PATHNAME;
                      KEY:         in RELATIONSHIP_KEY := LATEST_KEY;
                      INHERITABLE: in BOOLEAN := TRUE)
is
    GROUP_NODE: NODE_TYPE;
begin
    OPEN (GROUP_NODE, GROUP_NAME);
    ADOPT_ROLE (GROUP_NODE, KEY, INHERITABLE);
exception
    when others =>
        CLOSE (GROUP_NODE);
end ADOPT_ROLE;
```

Downloaded from http://www.everyspec.com

5.1.4.8 Unlinking an adopted role

```
procedure UNADOPT_ROLE (KEY: in RELATIONSHIP_KEY);
```

Purpose:

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

Parameter:

KEY is the relationship key designator of the relation ADOPTED_ROLE.

Exceptions:

SYNTAX_ERROR
is raised if KEY is syntactically illegal (see Table I, page 32).

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.

Downloaded from http://www.everyspec.com

5.1.5 Package CAIS_STRUCTURAL_NODE_MANAGEMENT

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.

The package CAIS_STRUCTURAL_NODE_MANAGEMENT defines the primitive operations for creating structural nodes. The exceptions raised by all subprograms in this package are defined in the package CAIS_DEFINITIONS.

Downloaded from http://www.everyspec.com

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;
    INTENT:                in     INTENT_ARRAY := (1=>WRITE);
    ATTRIBUTES:            in     ATTRIBUTE_LIST := EMPTY_LIST;
    DISCRETIONARY_ACCESS:  in     DISCRETIONARY_ACCESS_LIST :=
                               CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:      in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
```

Purpose:

This procedure creates a structural node and installs the primary relationship to it as well as the corresponding relationship of the predefined relation PARENT to the node identified by BASE. The relation name and relationship key designator of the primary relationship to the node are given by the parameters RELATION and KEY, respectively; the base node from which the primary relationship emanates is given by the parameter BASE. An open node handle to the newly created node with intent as specified by the INTENT parameter is returned in NODE.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

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 designator of the primary relationship to be created.
RELATION	is the relation name of the primary relationship to be created.
INTENT	is the intent of subsequent operations on the node; the actual parameter takes the form of an array aggregate.

Downloaded from http://www.everyspec.com

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

DISCRETIONARY_ACCESS
is the initial access control information associated with the created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given.

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

STATUS_ERROR
is raised if BASE is not an open node handle or if NODE is an open node handle at the time of the call.

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

Downloaded from http://www.everyspec.com

Additional Interfaces:

```
procedure CREATE_NODE
   (NODE:                    in out NODE_TYPE;
    NAME:                    in     PATHNAME;
    INTENT:                  in     INTENT_ARRAY := (1=>WRITE);
    ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
    DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST)
is
     BASE: NODE_TYPE;
begin
     OPEN (BASE, BASE_PATH(NAME), (1=>APPEND_RELATIONSHIPS));
     CREATE_NODE (NODE, BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
                  INTENT, ATTRIBUTES, DISCRETIONARY_ACCESS,
                  MANDATORY_ACCESS);
     CLOSE (BASE);
exception
     when others =>
          CLOSE (BASE);
          raise;
end CREATE_NODE;

procedure CREATE_NODE
   (BASE:                    in NODE_TYPE;
    KEY:                     in RELATIONSHIP_KEY := LATEST_KEY;
    RELATION:                in RELATION_NAME := DEFAULT_RELATION;
    INTENT:                  in INTENT_ARRAY := (1=>WRITE);
    ATTRIBUTES:              in ATTRIBUTE_LIST := EMPTY_LIST;
    DISCRETIONARY_ACCESS:    in DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:        in MANDATORY_ACCESS_LIST := EMPTY_LIST)
is
     NODE: NODE_TYPE;
begin
     CREATE_NODE (NODE, BASE, KEY, RELATION, INTENT, ATTRIBUTES,
                  DISCRETIONARY_ACCESS, MANDATORY_ACCESS);
     CLOSE (NODE);
end CREATE_NODE;

procedure CREATE_NODE
   (NAME:                    in PATHNAME;
    INTENT:                  in INTENT_ARRAY := (1=>WRITE);
    ATTRIBUTES:              in ATTRIBUTE_LIST := EMPTY_LIST;
    DISCRETIONARY_ACCESS:    in DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:        in MANDATORY_ACCESS_LIST := EMPTY_LIST)
is
     NODE: NODE_TYPE;
begin
     CREATE_NODE (NODE, NAME, INTENT, ATTRIBUTES,
                  DISCRETIONARY_ACCESS, MANDATORY_ACCESS);
     CLOSE (NODE);
end CREATE_NODE;
```

Downloaded from http://www.everyspec.com

Notes:

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

Downloaded from http://www.everyspec.com

5.2 CAIS process nodes

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

TABLE VII. Process Status Transition

EVENT	STATUS: non-existent	READY	SUSPENDED	ABORTED	TERMINATED
Process Creation	READY	Not Applicable	Not Applicable	Not Applicable	Not Applicable
Termination of Main Program	Not Applicable	TERMINATED	Not Applicable	Not Applicable	Not Applicable
Invocation of Interface ABORT_ PROCESS	Not Applicable	ABORTED	ABORTED	No Effect	No Effect
Invocation of Interface SUSPEND_ PROCESS	Not Applicable	SUSPENDED	No Effect	No Effect	No Effect
Invocation of Interface RESUME_ PROCESS	Not Applicable	No Effect	READY	No Effect	No Effect

A process is said to be *terminated* when its main program (in the sense of [1815A] 10.1) has terminated (in the sense of [1815A] 9.4). See also the notes in [1815A] 9.4. Thus, termination of a process takes place when the main program has completed and all tasks dependent on the main program have terminated.

A process may be *suspended* either by itself or by another process. When a process is suspended, its execution is stopped such that it can later be resumed. If an Ada program is suspended, all tasks are suspended and no tasks may be activated until the process is resumed. A suspended process may be resumed by another process. A process may be resumed even if its parent process is not resumed.

A process may be *aborted* either by itself or by another process.

Downloaded from http://www.everyspec.com

For a given process, a *process tree* is the set of processes consisting of the given process plus each process whose node's unique primary path traverses the node of the given process. When a process is aborted, suspended or resumed, all of the processes in its process tree are likewise aborted, suspended or resumed, subject to discretionary access control; in any case their process nodes remain until deleted. Any open node handles of a process are closed when the process terminates or is aborted.

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

a. Spawn - the procedure SPAWN_PROCESS returns after initiating the specified program. The initiating process and the initiated process run in parallel, and, within each of them, their tasks may execute in parallel.

b. Invoke - the procedure INVOKE_PROCESS returns control to the calling task after the initiated process has terminated or aborted. Execution of the calling task is blocked until termination or abortion of the initiated process, but other tasks in the initiating process may execute in parallel with the initiated process and its tasks.

c. Create - the procedure CREATE_JOB returns after initiating the specified program. The initiating process and the initiated process run in parallel, and, within each of them, their tasks may execute in parallel.

Every process node has several predefined attributes. Three of these are RESULTS, which can be used to store multiple user-defined strings giving (intermediate) results of the process; PARAMETERS, which contains the parameters with which the process was initiated; and CURRENT_STATUS, which gives the current status of the process (see Table VII, page 168). In addition, every process node has several predefined attributes which provide information for standardized debugging and performance measurement of processes within the CAIS implementation. One of these predefined attributes, OPEN_NODE_HANDLE_COUNT, gives the number of node handles the process currently has open. The remaining predefined attributes have implementation-dependent values and should not be used for comparison with values from other CAIS implementations. TIME_STARTED and TIME_FINISHED give the time of initiation and the time of termination or abortion of the process. MACHINE_TIME gives the length of time the process was active on the logical processor, if the process has terminated or aborted, or zero, if the process has not terminated or aborted. IO_UNIT_COUNT gives the number of GET and PUT operations that have been performed by the process. PROCESS_SIZE gives the amount of memory currently in use by the process. The CURRENT_STATUS, OPEN_NODE_HANDLE_COUNT, TIME_STARTED, TIME_FINISHED, MACHINE_TIME, IO_UNIT_COUNT and PROCESS_SIZE predefined attributes are maintained by the implementation and cannot be set using CAIS interfaces.

When a process has terminated or aborted, the final status, recorded in the predefined process node attribute CURRENT_STATUS, will persist as long as the process node exists. CURRENT_STATUS may also be examined by the CAIS interfaces CURRENT_STATUS and GET_RESULTS.

For purposes of input and output, every process node has one relationship of each of the following predefined relations: STANDARD_INPUT, STANDARD_OUTPUT and STANDARD_ERROR. STANDARD_INPUT, STANDARD_OUTPUT and STANDARD_

Downloaded from http://www.everyspec.com

ERROR are relation names of relationships established at process creation. The STANDARD_INPUT and STANDARD_OUTPUT files conform to the semantics given for these in [1815A] 14.3.2, except that these files are not automatically open upon initiation of process execution. Interfaces are provided in the CAIS input and output packages (see Section 5.3) to read relationships of these predefined relations.

Downloaded from http://www.everyspec.com

5.2.1 Package CAIS_PROCESS_DEFINITIONS

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

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

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

```
subtype RESULTS_LIST   is CAIS_LIST_MANAGEMENT.LIST_TYPE;
subtype RESULTS_STRING is STRING;
subtype PARAMETER_LIST is CAIS_LIST_MANAGEMENT.LIST_TYPE;
```

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

```
ROOT_PROCESS:    constant PATHNAME := "'CURRENT_JOB";
STANDARD_INPUT:  constant PATHNAME := "'STANDARD_INPUT";
STANDARD_OUTPUT: constant PATHNAME := "'STANDARD_OUTPUT";
STANDARD_ERROR:  constant PATHNAME := "'STANDARD_ERROR";
```

ROOT_PROCESS is a standard pathname for the root process node of the current job. STANDARD_INPUT, STANDARD_OUTPUT and STANDARD_ERROR are predefined pathnames for the standard input, output and error files, respectively, of the current process.

```
EXECUTABLE_IMAGE_ERROR:  exception;
```

EXECUTABLE_IMAGE_ERROR is raised if it can be determined that the file node does not contain an executable image.

Downloaded from http://www.everyspec.com

5.2.2 Package CAIS_PROCESS_MANAGEMENT

This package specifies interfaces for the creation and termination of processes and examination and modification of process node attributes. The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_PROCESS_DEFINITIONS.

As part of the creation of root process nodes, new secondary relationships emanating from the newly created process node are created as described in Table VIII.

As part of the creation of process nodes other than root process nodes, secondary relationships of several predefined relations are created, emanating from the newly created process node. In addition, the newly created process node inherits all secondary relationships from the node of the creating process for which IS_INHERITABLE (see Section 5.1.2.28, page 104) is TRUE.

Table IX summarizes the inheritance or creation of all predefined relationships which emanate from the created process node.

TABLE VIII. Relationships Created as a Result of CREATE_JOB

A Secondary Relationship of the Predefined Relation:	Is Created to the Node Identified by the:
ACCESS	Pathname 'CURRENT_USER'DEFAULT_ROLE (the GRANT attribute is set by the interface parameter DISCRETIONARY_ACCESS).
ADOPTED_ROLE	Pathname 'CURRENT_USER'DEFAULT_ROLE. This secondary relationship is also created to the default group node of the node containing the executable image of the program, if such a group node exists.
CURRENT_JOB	Newly-created root process node.
CURRENT_NODE	Interface parameter ENVIRONMENT_NODE.
CURRENT_USER	The user's top-level user node.
DEVICE	Implementation-defined subset of top-level device nodes.
EXECUTABLE_IMAGE	Interface parameter FILE_NODE.
GROUP	Implementation-defined subset of top-level group nodes.
PARENT	Predefined constant CURRENT_USER.
STANDARD_ERROR	Interface parameter ERROR_FILE.
STANDARD_INPUT	Interface parameter INPUT_FILE.
STANDARD_OUTPUT	Interface parameter OUTPUT_FILE.
USER	Implementation-defined subset of top-level user nodes.

Downloaded from http://www.everyspec.com

TABLE IX. Relationships Created and Inherited for Process Nodes

A Secondary Relationship of the Predefined Relation:	Is
ACCESS	Inherited from the creating process node; in addition, the access relationship to the node identified by the pathname 'CURRENT_USER'DEFAULT_ROLE is created or altered to have a GRANT attribute set to the value of the interface parameter DISCRETIONARY_ACCESS.
ADOPTED_ROLE	Inherited from the creating process node; in addition, created to the default group node of the node containing the executable image of the program, if such a group node exists.
CURRENT_JOB	Inherited from the creating process node.
CURRENT_NODE	Created to the node identified by the interface parameter ENVIRONMENT_NODE.
CURRENT_USER	Inherited from the creating process.
DEVICE	Inherited from the creating process.
EXECUTABLE_IMAGE	Created to the node identified by the interface parameter FILE_NODE.
GROUP	Inherited from the creating process.
PARENT	Created to the node for the creating process.
STANDARD_ERROR	Created to the node identified by the interface parameter ERROR_FILE.
STANDARD_INPUT	Created to the node identified by the interface parameter INPUT_FILE.
STANDARD_OUTPUT	Created to the node identified by the interface parameter OUTPUT_FILE.
USER	Inherited from the creating process.

Downloaded from http://www.everyspec.com

5.2.2.1 Spawning a process

```
procedure SPAWN_PROCESS
  (NODE:                      in out NODE_TYPE;
   FILE_NODE:                 in     NODE_TYPE;
   INTENT:                    in     INTENT_ARRAY;
   INPUT_PARAMETERS:          in     PARAMETER_LIST := EMPTY_LIST;
   KEY:                       in     RELATIONSHIP_KEY := LATEST_KEY;
   RELATION:                  in     RELATION_NAME := DEFAULT_RELATION;
   DISCRETIONARY_ACCESS:      in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
   MANDATORY_ACCESS:          in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
   ATTRIBUTES:                in     ATTRIBUTE_LIST := EMPTY_LIST;
   INPUT_FILE:                in     PATHNAME := STANDARD_INPUT;
   OUTPUT_FILE:               in     PATHNAME := STANDARD_OUTPUT;
   ERROR_FILE:                in     PATHNAME := STANDARD_ERROR;
   ENVIRONMENT_NODE:          in     PATHNAME := CURRENT_NODE);
```

Purpose:

This procedure creates a new process node whose contents represent the execution of the program contained in the specified file node. The primary relationship to the newly created process node emanates from the current process node and has relation name and relationship key identified by the RELATION and KEY parameters. The process is then activated, i.e., its status is set to READY. Control returns to the calling task after the new node is created. The process node containing the calling task must have execution rights for the file node. An open node handle NODE on the new node is returned, with an intent as specified by the INTENT parameter. The new process, as a subject, has all discretionary access rights to its own process node (as the object).

Secondary relationships emanating from the new process node are created and inherited as described in Table IX, page 173.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51). Object and subject classification labels of a process are the same.

Parameters:

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

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

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

Downloaded from http://www.everyspec.com

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

KEY
: is the relationship key designator of the primary relationship from the current process node to the new process node.

RELATION
: is the relation name of the primary relationship from the current process node to the new process node.

DISCRETIONARY_ACCESS
: is the initial access control information associated with the created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
: is a list defining the classification label for the created node (see Table IV, page 51).

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

INPUT_FILE
: is a pathname to a file node that will be the target node of the secondary relationship of the predefined relation STANDARD_INPUT.

OUTPUT_FILE
: is a pathname to a file node that will be the target node of the secondary relationship of the predefined relation STANDARD_OUTPUT.

ERROR_FILE
: is a pathname to a file node that will be the target node of the secondary relationship of the predefined relation STANDARD_ERROR.

ENVIRONMENT_NODE
: is a pathname to a node the new process will have as its initial current node.

Exceptions:

PATHNAME_SYNTAX_ERROR
: is raised if the node identification given by KEY and RELATION is syntactically illegal.

RELATIONSHIP_ERROR
: is raised if any of the nodes identified by INPUT_FILE, OUTPUT_FILE, ERROR_FILE, or ENVIRONMENT_NODE does not exist.

Downloaded from http://www.everyspec.com

EXISTING_NODE_ERROR
is raised if a relationship of the relation RELATION with the relationship key designator KEY already exists.

SYNTAX_ERROR
is raised if any of the parameters INPUT_PARAMETERS, MANDATORY_ACCESS (see Table IV, page 51), DISCRETIONARY_ACCESS (see Section 4.4.2.3) or ATTRIBUTES (see description above) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

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

USE_ERROR is raised if any of the parameters INPUT_PARAMETERS, MANDATORY_ACCESS, DISCRETIONARY_ACCESS, or ATTRIBUTES is semantically illegal.

STATUS_ERROR
is raised if NODE is an open node handle at the time of the call or if FILE_NODE is not an open node handle.

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

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

SECURITY_VIOLATION
may be raised if the attempt to obtain access to the node identified by NODE for the specified intent represents a violation of mandatory access controls. SECURITY_VIOLATION may also be raised if a process is created either of a higher or of a lower classification than the current process. SECURITY_VIOLATION may be raised only if the conditions for raising the other exceptions are not satisfied.

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure SPAWN_PROCESS
   (NODE:                       in out NODE_TYPE;
    FILE_NODE:                  in     NODE_TYPE;
    INTENT:                     in     INTENT_SPECIFICATION := READ_ATTRIBUTES;
    INPUT_PARAMETERS:           in     PARAMETER_LIST := EMPTY_LIST;
    KEY:                        in     RELATIONSHIP_KEY := LATEST_KEY;
    RELATION:                   in     RELATION_NAME := DEFAULT_RELATION;
    DISCRETIONARY_ACCESS:       in     DISCRETIONARY_ACCESS_LIST :=
                                   CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:           in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
    ATTRIBUTES:                 in     ATTRIBUTE_LIST := EMPTY_LIST;
    INPUT_FILE:                 in     PATHNAME := STANDARD_INPUT;
    OUTPUT_FILE:                in     PATHNAME := STANDARD_OUTPUT;
    ERROR_FILE:                 in     PATHNAME := STANDARD_ERROR;
    ENVIRONMENT_NODE:           in     PATHNAME := CURRENT_NODE)
is
begin
    SPAWN_PROCESS (NODE, FILE_NODE, (1=>INTENT), INPUT_PARAMETERS,
                   KEY, RELATION, DISCRETIONARY_ACCESS,
                   MANDATORY_ACCESS, ATTRIBUTES, INPUT_FILE,
                   OUTPUT_FILE, ERROR_FILE, ENVIRONMENT_NODE);
end SPAWN_PROCESS;
```

Notes:

If the default value of the INTENT parameter is used, the exception SECURITY_ VIOLATION may be raised when a process is created at a higher classification than that of the current process.

Downloaded from http://www.everyspec.com

5.2.2.2 Awaiting termination or abortion of another process

```
procedure AWAIT_PROCESS_COMPLETION
                (NODE:       in   NODE_TYPE;
                 TIME_LIMIT: in   CAIS_DURATION := LONG_DELAY);
```

Purpose:

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

Parameters:

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

TIME_LIMIT — is the limit on the time that the calling task will be suspended awaiting the process. When the limit is exceeded the calling task resumes execution.

Exceptions:

NODE_KIND_ERROR
is raised if NODE does not identify a process node.

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

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

SECURITY_VIOLATION
may be raised if the attempt to wait for completion of the process represents a violation of the mandatory access control of the CAIS implementation.

Additional interface:

```
procedure AWAIT_PROCESS_COMPLETION
                (NODE:              in      NODE_TYPE;
                 RESULTS_RETURNED:  in out  RESULTS_LIST;
                 STATUS:               out  PROCESS_STATUS_KIND;
                 TIME_LIMIT:        in      CAIS_DURATION := LONG_DELAY)
is
begin
    AWAIT_PROCESS_COMPLETION (NODE, TIME_LIMIT);
    GET_RESULTS (NODE, RESULTS_RETURNED);
    STATUS := CURRENT_STATUS (NODE);
end AWAIT_PROCESS_COMPLETION;
```

Downloaded from http://www.everyspec.com

Notes:

The description of the interface GET_RESULTS can be found on page 192.

Downloaded from http://www.everyspec.com

5.2.2.3 Invoking a new process

```
procedure INVOKE_PROCESS
     (NODE:                   in out NODE_TYPE;
      FILE_NODE:              in     NODE_TYPE;
      INTENT:                 in     INTENT_ARRAY;
      RESULTS_RETURNED:       in out RESULTS_LIST;
      STATUS:                    out PROCESS_STATUS_KIND;
      INPUT_PARAMETERS:       in     PARAMETER_LIST;
      KEY:                    in     RELATIONSHIP_KEY := LATEST_KEY;
      RELATION:               in     RELATION_NAME := DEFAULT_RELATION;
      DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
      ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
      INPUT_FILE:             in     PATHNAME := STANDARD_INPUT;
      OUTPUT_FILE:            in     PATHNAME := STANDARD_OUTPUT;
      ERROR_FILE:             in     PATHNAME := STANDARD_ERROR;
      ENVIRONMENT_NODE:       in     PATHNAME := CURRENT_NODE;
      TIME_LIMIT:             in     CAIS_DURATION := LONG_DELAY);
```

Purpose:

This procedure creates a new process node whose contents represent the execution of the program contained in the specified file node. The primary relationship to the newly created process node emanates from the current process node and has relation name and relationship key identified by the RELATION and KEY parameters. The process is then activated, i.e., its status is set to READY. Control returns to the calling task after the newly created process terminates or is aborted or the amount of time specified by the parameter TIME_LIMIT expires. The process node containing the calling task must have execution rights for the file node. An open node handle NODE on the new node is returned, with an intent as specified by the INTENT parameter. The new process, as a subject, has all discretionary access rights to its own process node (as the object).

Secondary relationships emanating from the new process node are created and inherited as described in Table IX, page 173.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51). Object and subject classification labels of a process are the same.

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

Downloaded from http://www.everyspec.com


```
SPAWN_PROCESS (NODE, FILE_NODE, INTENT, INPUT_PARAMETERS,
               KEY, RELATION, DISCRETIONARY_ACCESS, MANDATORY_ACCESS,
               ATTRIBUTES, INPUT_FILE, OUTPUT_FILE, ERROR_FILE,
               ENVIRONMENT_NODE);
AWAIT_PROCESS_COMPLETION (NODE, TIME_LIMIT);
GET_RESULTS (NODE, RESULTS_RETURNED);
STATUS := CURRENT_STATUS (NODE);
```

Parameters:

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

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

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

RESULTS_RETURNED
: is a list of results from the new process, which are represented by strings. The individual results may be extracted from the list using the subprograms of CAIS_LIST_MANAGEMENT.

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

INPUT_PARAMETERS
: is a list containing process parameter information. The list is constructed and parsed using the subprograms of CAIS_LIST_MANAGEMENT. The value of INPUT_PARAMETERS is stored in the predefined attribute PARAMETERS of the new node.

KEY
: is the relationship key designator of the primary relationship from the current process node to the new process node.

RELATION
: is the relation name of the primary relationship from the current process node to the new node.

DISCRETIONARY_ACCESS
: is the initial access control information associated with the created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
: is a list defining the classification label for the created node (see Table IV, page 51).

Downloaded from http://www.everyspec.com

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

INPUT_FILE is a pathname to a file node that will be the target node of the secondary relationship of the predefined relation STANDARD_INPUT.

OUTPUT_FILE is a pathname to a file node that will be the target node of the secondary relationship of the predefined relation STANDARD_OUTPUT.

ERROR_FILE is a pathname to a file node that will be the target node of the secondary relationship of the predefined relation STANDARD_ERROR.

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

TIME_LIMIT is the limit on the time that the calling task will be suspended awaiting the new process. When the limit is exceeded, the calling task resumes execution.

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by KEY and RELATION is syntactically illegal.

RELATIONSHIP_ERROR
is raised if any of the nodes identified by INPUT_FILE, OUTPUT_FILE, ERROR_FILE, or ENVIRONMENT_NODE does not exist.

EXISTING_NODE_ERROR
is raised if a relationship of the relation RELATION with relationship key designator KEY already exists.

SYNTAX_ERROR
is raised if any of the parameters INPUT_PARAMETERS, MANDATORY_ACCESS (see Table IV, page 51), DISCRETIONARY_ACCESS (see Section 4.4.2.3) or ATTRIBUTES (see description above) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

Downloaded from http://www.everyspec.com

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

USE_ERROR is raised if any of the parameters INPUT_PARAMETERS, MANDATORY_ACCESS, DISCRETIONARY_ACCESS, or ATTRIBUTES is semantically illegal.

STATUS_ERROR
is raised if NODE is an open node handle at the time of the call or if FILE_NODE is not an open node handle.

LOCK_ERROR is raised if access with intent APPEND_RELATIONSHIPS cannot be obtained to the current process node due to an existing lock on the node. LOCK_ERROR may be raised prior to expiration of the timeout if the CAIS implementation can determine that a deadlock situation has occurred.

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

SECURITY_VIOLATION
may be raised if the attempt to wait for completion of the process and to obtain results from it represents a violation of the mandatory access controls for the CAIS. SECURITY_VIOLATION may be raised when a process is created either of a higher or of a lower classification than the current process. SECURITY_VIOLATION may be raised only if the conditions for raising the other exceptions are not satisfied.

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure INVOKE_PROCESS
   (NODE:                      in out NODE_TYPE;
    FILE_NODE:                 in     NODE_TYPE;
    INTENT:                    in     INTENT_SPECIFICATION := READ_ATTRIBUTES;
    RESULTS_RETURNED:          in out RESULTS_LIST;
    STATUS:                       out PROCESS_STATUS_KIND;
    INPUT_PARAMETERS:          in     PARAMETER_LIST;
    KEY:                       in     RELATIONSHIP_KEY := LATEST_KEY;
    RELATION:                  in     RELATION_NAME := DEFAULT_RELATION;
    DISCRETIONARY_ACCESS:      in     DISCRETIONARY_ACCESS_LIST :=
                                  CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:          in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
    ATTRIBUTES:                in     ATTRIBUTE_LIST := EMPTY_LIST;
    INPUT_FILE:                in     PATHNAME := STANDARD_INPUT;
    OUTPUT_FILE:               in     PATHNAME := STANDARD_OUTPUT;
    ERROR_FILE:                in     PATHNAME := STANDARD_ERROR;
    ENVIRONMENT_NODE:          in     PATHNAME := CURRENT_NODE;
    TIME_LIMIT:                in     CAIS_DURATION := LONG_DELAY)
is
begin
    INVOKE_PROCESS (NODE, FILE_NODE, (1=>INTENT), RESULTS_RETURNED,
                    STATUS, INPUT_PARAMETERS, KEY, RELATION,
                    DISCRETIONARY_ACCESS, MANDATORY_ACCESS,
                    ATTRIBUTES, INPUT_FILE, OUTPUT_FILE, ERROR_FILE,
                    ENVIRONMENT_NODE, TIME_LIMIT);
end INVOKE_PROCESS;
```

Notes:

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

Downloaded from http://www.everyspec.com

5.2.2.4 Creating a new job

```
procedure CREATE_JOB
   (FILE_NODE:                in NODE_TYPE;
    INPUT_PARAMETERS:         in PARAMETER_LIST := EMPTY_LIST;
    KEY:                      in RELATIONSHIP_KEY := LATEST_KEY;
    DISCRETIONARY_ACCESS:     in DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:         in MANDATORY_ACCESS_LIST := EMPTY_LIST;
    ATTRIBUTES:               in ATTRIBUTE_LIST := EMPTY_LIST;
    INPUT_FILE:               in PATHNAME := STANDARD_INPUT;
    OUTPUT_FILE:              in PATHNAME := STANDARD_OUTPUT;
    ERROR_FILE:               in PATHNAME := STANDARD_ERROR;
    ENVIRONMENT_NODE:         in PATHNAME := CURRENT_USER;
    DELETE_WHEN_TERMINATED:   in BOOLEAN := TRUE);
```

Purpose:

This procedure creates a new root process node whose contents represent the execution of the program contained in the specified file node. The process is then activated, i.e., its status is set to READY. Control returns to the calling task after the new job is created. The process node containing the calling task must have execution rights for the file node and sufficient rights to append relationships to the node identified by 'CURRENT_USER. A new primary relationship of the predefined relation JOB is established from the current user node to the root process node of the new job. The new root process as a subject has all discretionary access rights to its own process node (the object).

Secondary relationships emanating from the new root process node are created as described in Table VIII, page 172.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51). Object and subject classification labels of a process are the same.

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 subprograms provided in CAIS_LIST_MANAGEMENT. INPUT_PARAMETERS is stored in the predefined attribute PARAMETERS of the new node.

Downloaded from http://www.everyspec.com

KEY is the relationship key designator of the primary relationship of the predefined relation JOB from the current user node to the new process node.

DISCRETIONARY_ACCESS
is the initial access control information associated with the created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

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

INPUT_FILE is a pathname to a file node that will be the target node of the secondary relationship of the predefined relation STANDARD_INPUT.

OUTPUT_FILE is a pathname to a file node that will be the target node of the secondary relationship of the predefined relation STANDARD_OUTPUT.

ERROR_FILE is a pathname to a file node that will be the target node of the secondary relationship of the predefined relation STANDARD_ERROR.

ENVIRONMENT_NODE
is a pathname to a node the new process will have as its initial current node.

DELETE_WHEN_TERMINATED
is a boolean; if TRUE, the process node will be deleted when the job terminates; if FALSE, the process node will be kept when the job terminates. Note that in the latter case, an explicit call to the DELETE_JOB interface (see Section 5.2.2.5, page 188) must be made in order to delete the process node.

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by KEY is syntactically illegal.

RELATIONSHIP_ERROR
is raised if any of the nodes identified by INPUT_FILE, OUTPUT_FILE, ERROR_FILE, or ENVIRONMENT_NODE does not exist.

EXISTING_NODE_ERROR
is raised if a relationship of the relation JOB with relationship key designator KEY already exists.

Downloaded from http://www.everyspec.com

SYNTAX_ERROR
is raised if any of the parameters INPUT_PARAMETERS MANDATORY_ACCESS (see Table IV, page 51), DISCRETIONARY_ACCESS (see Section 4.4.2.3) or ATTRIBUTES (see description above) is syntactically illegal.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

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

USE_ERROR
is raised if any of the parameters INPUT_PARAMETERS, MANDATORY_ACCESS, DISCRETIONARY_ACCESS, or ATTRIBUTES is semantically illegal.

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

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

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

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

Downloaded from http://www.everyspec.com

5.2.2.5 Deleting a job

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

Purpose:

This procedure effectively performs the DELETE_NODE operation for a specified root process node and recursively applies DELETE_NODE (see Section 5.1.2.23, page 94) to all nodes reachable by a unique primary pathname from the designated node. The nodes whose primary relationships are to be deleted are opened with intent EXCLUSIVE_WRITE, thus locking them for other operations. The order in which the deletions of primary relationships is performed is not specified. If the DELETE_JOB operation raises an exception, none of the primary relationships is deleted.

Parameter:

NODE is an open node handle to the root process node of the job to be deleted.

Exceptions:

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

PREDEFINED_RELATION_ERROR
is raised if the primary relationship to the node identified by NODE is not a relationship of the predefined relation JOB.

STATUS_ERROR
is raised if the node handle NODE is not open at the time of the call.

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

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

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

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

Downloaded from http://www.everyspec.com

Additional Interface:

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

Notes:

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

The DELETE_TREE operation (see Section 5.1.2.24, page 96) cannot be applied to jobs, due to the predefined nature of the JOB relation.

Downloaded from http://www.everyspec.com

5.2.2.6 Appending results

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

Purpose:

This procedure inserts the value of its RESULTS parameter as the last item in the list which is the value of the RESULTS attribute of the current process node.

Parameter:

RESULTS is a string to be appended to the RESULTS attribute value of the current process node.

Exception:

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

Downloaded from http://www.everyspec.com

5.2.2.7 Overwriting results

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

Purpose:

This procedure replaces the value of the RESULTS attribute of the current process node with a list containing a single item which is the value of the parameter RESULTS.

Parameter:

RESULTS is a string to be stored in the RESULTS attribute value list of the current process node.

Exception:

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

Downloaded from http://www.everyspec.com

5.2.2.8 Getting results from a process

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

Purpose:

This procedure returns the value of the attribute RESULTS of the process node identified by NODE. The process need not have terminated or aborted. The empty list is returned in RESULTS if WRITE_RESULTS or APPEND_RESULTS has not been called by the process contained in the node identified by NODE.

Parameters:

NODE is an open node handle on a process node.

RESULTS is an unnamed list of strings giving the value of the RESULTS attribute of the process node identified by NODE. The individual strings may be extracted from the list using the subprograms of CAIS_LIST_MANAGEMENT (see Section 5.4).

Exceptions:

NODE_KIND_ERROR
is raised if the node identified by NODE is not a process node.

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

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

Downloaded from http://www.everyspec.com

Additional Interfaces:

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

procedure GET_RESULTS (NAME:     in      PATHNAME;
                       RESULTS:  in out  RESULTS_LIST;
                       STATUS:      out  PROCESS_STATUS_KIND)
is
    NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    GET_RESULTS (NODE, RESULTS);
    STATUS := CURRENT_STATUS (NODE);
    CLOSE (NODE);
exception
    when others =>
        CLOSE (NODE);
        raise;
end GET_RESULTS;

procedure GET_RESULTS (NAME:     in      PATHNAME;
                       RESULTS:  in out  RESULTS_LIST)
is
    NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    GET_RESULTS (NODE, RESULTS);
    CLOSE (NODE);
exception
    when others =>
        CLOSE (NODE);
        raise;
end GET_RESULTS;
```

Notes:

The STATUS parameter in the Additional Interfaces performs in a manner similar to the function CURRENT_STATUS (Section 5.2.2.9, page 194).

Downloaded from http://www.everyspec.com

5.2.2.9 Determining the status of a process

```
function CURRENT_STATUS (NODE: in NODE_TYPE)
    return PROCESS_STATUS_KIND;
```

Purpose:

This function returns the value of the attribute CURRENT_STATUS associated with the process node identified by NODE.

Parameter:

NODE — is an open node handle identifying the node of the process whose status is to be queried.

Exceptions:

NODE_KIND_ERROR
is raised if the node identified by NODE is not a process node.

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

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

Additional Interface:

```
function CURRENT_STATUS (NAME: in PATHNAME)
    return PROCESS_STATUS_KIND
is
    NODE: NODE_TYPE;
    RESULT: PROCESS_STATUS_KIND;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := CURRENT_STATUS (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end CURRENT_STATUS;
```

Downloaded from http://www.everyspec.com

5.2.2.10 Getting the parameter list

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

Purpose:

This procedure returns the value of the predefined attribute PARAMETERS of the current process node.

Parameter:

PARAMETERS is a list containing parameter information. The list is constructed and can be manipulated using the subprograms provided in CAIS_LIST_MANAGEMENT (see Section 5.4).

Exception:

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

Notes:

The value of the predefined attribute PARAMETERS is set during process node creation; see the interfaces SPAWN_PROCESS (page 174), INVOKE_PROCESS (page 180) and CREATE_JOB (page 185).

Downloaded from http://www.everyspec.com

5.2.2.11 Aborting a process

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

Purpose:

This procedure aborts the process represented by NODE (see Table VII, page 168). It also aborts any process in its process tree for which the current process has sufficient discretionary access rights to (1) traverse the primary relationships to the node representing the process and (2) obtain access to the node representing the process with intent WRITE_ATTRIBUTES and WRITE_CONTENTS. If the current process does not have such access rights to the node of any process in the process tree, the respective node's process is not aborted, and after all processes that can be aborted are aborted, ABORT_PROCESS returns by raising the exception ACCESS_VIOLATION. The order in which the processes are aborted is not specified. After return of ABORT_PROCESS the CURRENT_STATUS of the process represented by NODE will be ABORTED or TERMINATED; it will be TERMINATED only if the process terminated before ABORT_PROCESS took effect. The nodes associated with the aborted processes remain until explicitly deleted. If NODE_KIND_ERROR, STATUS_ERROR or INTENT_VIOLATION is raised none of the processes in the process tree is aborted.

Parameters:

NODE	is an open node handle for the node of the process to be aborted.
RESULTS	is a string to be appended to the RESULTS attribute of the node represented by NODE.

Exceptions:

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a process node.

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

INTENT_VIOLATION
: is raised if the node handle NODE was not opened with an intent establishing rights to read relationships and to write attributes and contents.

ACCESS_VIOLATION
: is raised if the current process does not have sufficient discretionary access rights to obtain access to any node of the process tree identified by NODE with intent including READ_RELATIONSHIPS, WRITE_ATTRIBUTES and WRITE_CONTENTS.

Downloaded from http://www.everyspec.com

Additional Interfaces:

```
procedure ABORT_PROCESS (NAME:    in PATHNAME;
                         RESULTS: in RESULTS_STRING)
is
    NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (READ_RELATIONSHIPS, WRITE_CONTENTS,
                       WRITE_ATTRIBUTES));
    ABORT_PROCESS (NODE, RESULTS);
    CLOSE (NODE);
exception
    when others =>
         CLOSE (NODE);
         raise;
end ABORT_PROCESS;

procedure ABORT_PROCESS (NODE: in NODE_TYPE)
is
begin
    ABORT_PROCESS (NODE, "ABORTED");
end ABORT_PROCESS;

procedure ABORT_PROCESS (NAME: in PATHNAME)
is
    NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (READ_RELATIONSHIPS, WRITE_CONTENTS,
                       WRITE_ATTRIBUTES));
    ABORT_PROCESS (NODE, "ABORTED");
    CLOSE (NODE);
exception
    when others =>
        CLOSE (NODE);
        raise;
end ABORT_PROCESS;
```

Notes:

ABORT_PROCESS can be used by a task to abort the process that contains it.

It is intentional that LOCK_ERROR will not be raised by this procedure.

Downloaded from http://www.everyspec.com

5.2.2.12 Suspending a process

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

Purpose:

This procedure suspends the process represented by NODE (see Table VII, page 168). It also suspends any process in its process tree for which the current process has sufficient discretionary access rights to (1) traverse the primary relationships to the node representing the process and (2) obtain access to the node representing the process with intent WRITE_ATTRIBUTES and WRITE_CONTENTS. If the current process does not have such access rights to the node of any process in the process tree, the respective node's process is not suspended, and after all processes that can be suspended are suspended, SUSPEND_PROCESS returns by raising the exception ACCESS_VIOLATION. The order in which the processes are suspended is not specified. After return of SUSPEND_PROCESS, the CURRENT_STATUS of the process represented by NODE will be ABORTED, TERMINATED or SUSPENDED; it will be SUSPENDED unless the process terminated or was aborted before SUSPEND_PROCESS took effect. The nodes associated with the suspended processes remain until explicitly deleted. If NODE_KIND_ERROR, STATUS_ERROR or INTENT_VIOLATION is raised none of the processes in the process tree is suspended.

Parameter:

NODE
: is an open node handle identifying the node of the process to be suspended.

Exceptions:

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a process node.

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

INTENT_VIOLATION
: is raised if the node handle NODE was not opened with an intent establishing rights to read relationships and to write attributes and contents.

ACCESS_VIOLATION
: is raised if the current process does not have sufficient discretionary access rights to obtain access to any node of the process tree identified by NODE with intent including READ_RELATIONSHIPS, WRITE_ATTRIBUTES and WRITE_CONTENTS.

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure SUSPEND_PROCESS (NAME: in PATHNAME)
is
    NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (READ_RELATIONSHIPS, WRITE_ATTRIBUTES,
                       WRITE_CONTENTS));
    SUSPEND_PROCESS (NODE);
    CLOSE (NODE);
exception
    when others =>
        CLOSE (NODE);
        raise;
end SUSPEND_PROCESS;
```

Notes:

SUSPEND_PROCESS can be used by a task to suspend the process that contains it.

Processes in the process tree of the process represented by NODE may be resumed by other CAIS calls prior to completion of the call on SUSPEND_PROCESS.

Downloaded from http://www.everyspec.com

5.2.2.13 Resuming a process

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

Purpose:

This procedure resumes the process represented by NODE (see Table VII, page 168). It also resumes any process in its process tree for which the current process has sufficient discretionary access rights to (1) traverse the primary relationships to the node representing the process and (2) obtain access to the node representing the process with intent WRITE_ATTRIBUTES and WRITE_CONTENTS. If the current process does not have such access rights to the node of any process in the process tree, the respective node's process is not resumed, and after all processes that can be resumed are resumed, RESUME_PROCESS returns by raising the exception ACCESS_VIOLATION. The order in which the processes are resumed is not specified. After return of RESUME_PROCESS, the CURRENT_STATUS of the process represented by NODE will be ABORTED, TERMINATED or READY; it will be READY unless the process terminated or was aborted before RESUME_PROCESS took effect. The nodes associated with the resumed processes remain until explicitly deleted. If NODE_KIND_ERROR, STATUS_ERROR or INTENT_VIOLATION is raised none of the processes in the process tree is resumed.

Parameter:

NODE — is an open node handle identifying the node of the process to be resumed.

Exceptions:

NODE_KIND_ERROR
is raised if the node identified by NODE is not a process node.

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

INTENT_VIOLATION
is raised if the node handle NODE was not opened with an intent establishing rights to read relationships and to write attributes and contents.

ACCESS_VIOLATION
is raised if the current process does not have sufficient discretionary access rights to obtain access to any node of the process tree identified by NODE with intent including READ_RELATIONSHIPS, WRITE_ATTRIBUTES and WRITE_CONTENTS.

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure RESUME_PROCESS (NAME: in PATHNAME)
is
    NODE: NODE_TYPE;
begin
    OPEN (NODE, NAME, (READ_RELATIONSHIPS, WRITE_ATTRIBUTES,
                       WRITE_CONTENTS));
    RESUME_PROCESS (NODE);
    CLOSE (NODE);
exception
    when others =>
        CLOSE (NODE);
        raise;
end RESUME_PROCESS;
```

Notes:

Processes in the process tree of the process represented by NODE may be suspended by other CAIS calls prior to completion of the call on RESUME_PROCESS.

Downloaded from http://www.everyspec.com

5.2.2.14 Determining the number of open node handles

```
function OPEN_NODE_HANDLE_COUNT (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns a natural number representing the value of the predefined attribute OPEN_NODE_HANDLE_COUNT of the process node identified by NODE.

Parameter:

NODE — is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a process node.

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

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

Additional Interface:

```
function OPEN_NODE_HANDLE_COUNT (NAME: in PATHNAME)
    return CAIS_NATURAL
is
    NODE: NODE_TYPE;
    RESULT: CAIS_NATURAL;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := OPEN_NODE_HANDLE_COUNT (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end OPEN_NODE_HANDLE_COUNT;
```

Downloaded from http://www.everyspec.com

5.2.2.15 Determining the number of input and output units used

```
function IO_UNIT_COUNT (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns a natural number representing the value of the predefined attribute IO_UNIT_COUNT of the process node identified by NODE.

Parameter:

NODE — is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a process node.

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

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

Additional Interface:

```
function IO_UNIT_COUNT (NAME: in PATHNAME)
    return CAIS_NATURAL
is
    NODE: NODE_TYPE;
    RESULT: CAIS_NATURAL;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := IO_UNIT_COUNT (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end IO_UNIT_COUNT;
```

Downloaded from http://www.everyspec.com

5.2.2.16 Determining the time of activation

```
function TIME_STARTED (NODE: in NODE_TYPE)
    return CAIS_CALENDAR.TIME;
```

Purpose:

This function returns a value of type CAIS_CALENDAR.TIME representing the value of the predefined attribute TIME_STARTED of the process node identified by NODE.

Parameter:

NODE
: is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a process node.

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

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

Additional Interface:

```
function TIME_STARTED (NAME: in PATHNAME)
    return CAIS_CALENDAR.TIME
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_CALENDAR.TIME;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := TIME_STARTED (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end TIME_STARTED;
```

Downloaded from http://www.everyspec.com

5.2.2.17 Determining the time of termination or abortion

```
function TIME_FINISHED (NODE: in NODE_TYPE)
    return CAIS_CALENDAR.TIME;
```

Purpose:

This function returns a value of type CAIS_CALENDAR.TIME representing the value of the predefined attribute TIME_FINISHED of the process node identified by NODE.

Parameter:

NODE — is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a process node.

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

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

USE_ERROR — is raised if the process is not yet terminated or aborted.

Additional Interface:

```
function TIME_FINISHED (NAME: in PATHNAME)
    return CAIS_CALENDAR.TIME
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_CALENDAR.TIME;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := TIME_FINISHED (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end TIME_FINISHED;
```

Downloaded from http://www.everyspec.com

5.2.2.18 Determining the time a process has been active

```
function MACHINE_TIME (NODE: in NODE_TYPE)
    return CAIS_DURATION;
```

Purpose:

This function returns a value of type CAIS_DURATION representing the value of the predefined attribute MACHINE_TIME of the process node identified by NODE. Zero is returned if the process is not yet terminated or aborted.

Parameter:

NODE — is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a process node.

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

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

Additional Interface:

```
function MACHINE_TIME (NAME: in PATHNAME)
    return CAIS_DURATION
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_DURATION;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := MACHINE_TIME (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end MACHINE_TIME;
```

Downloaded from http://www.everyspec.com

5.2.2.19 Determining the size of a process

```
function PROCESS_SIZE (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns a value of type CAIS_NATURAL representing the value of the predefined attribute PROCESS_SIZE of the process node identified by NODE. The predefined attribute PROCESS_SIZE designates the amount of memory currently in use by the process. If the process is terminated or aborted, the value returned is the amount of memory in use by the process at the time that the process was terminated or aborted. The result of this function is expressed in multiples of CAIS_PRAGMATICS.MEMORY_STORAGE_UNIT_SIZE.

Parameter:

NODE — is an open node handle identifying the process node whose attribute is being queried.

Exceptions:

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a process node.

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

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

Additional Interface:

```
function PROCESS_SIZE (NAME: in PATHNAME)
    return CAIS_NATURAL
is
    NODE: NODE_TYPE;
    RESULT: CAIS_NATURAL;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := PROCESS_SIZE (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end PROCESS_SIZE;
```

Downloaded from http://www.everyspec.com

5.3 CAIS input and output

This section describes the CAIS interfaces for the transfer of data to and from CAIS files, queues or devices. These interfaces are defined in the following CAIS packages:

a. CAIS_DEVICES defines types used throughout the input and output interfaces of the CAIS. These types allow a CAIS implementation to provide additional packages in support of additional devices in a way that is compatible with the CAIS standard;

b. CAIS_IO_DEFINITIONS defines types, subtypes, constants and exceptions used throughout the input and output interfaces of the CAIS;

c. CAIS_IO_ATTRIBUTES defines interfaces for obtaining the values of the predefined input and output attributes on file nodes;

d. CAIS_DIRECT_IO, CAIS_SEQUENTIAL_IO and CAIS_TEXT_IO define interfaces largely corresponding to the interfaces defined in [1815A], Chapter 14. The interfaces are described in terms of their correspondences with and differences from those defined in [1815A].

e. CAIS_QUEUE_MANAGEMENT defines mechanisms for the creation of queue file nodes;

f. CAIS_SCROLL_TERMINAL_IO, CAIS_PAGE_TERMINAL_IO and CAIS_FORM_TERMINAL_IO define interfaces for operating on (abstract) scroll terminals, page terminals and form terminals, respectively;

g. CAIS_MAGNETIC_TAPE_IO defines interfaces for operating on magnetic tape drive files; and

h. CAIS_IMPORT_EXPORT defines interfaces for transferring files between the host operating system and the CAIS implementation node model.

Throughout this document, the word ''file'' is used to mean an Ada external file, which in the CAIS is the contents of a file node, while in [1815A] the word ''file'' is used to mean an internal file. The input and output operations in the packages in this section are expressed as operations on objects of some file type, rather than directly in terms of the contents of the file nodes. These objects are internal to a CAIS process (*internal files*). Internal files are identified by *file handles*. An Ada type FILE_TYPE is defined for values that represent file handles. Ada objects of this type can be associated with a file by means of CAIS interfaces causing an *open file handle* to be assigned to the object. While such an association is in effect, the file handle is said to be open. In this section, a parameter or variable of type FILE_TYPE identifies a file.

An open file handle is always associated with an open file node handle that was used in opening the file handle or returned along with the open file handle by a node creating interface. An open file node handle may be associated with multiple open file handles. File handles may be closed without affecting the status of the associated file node handle. However, closing the file node handle also closes all file handles associated with that file node handle (see Section 5.1.2.2, page 66).

The locking semantics of intents provided upon opening node handles applies only to nodes (see Table V, page 61). Thus, it is possible to open multiple file handles associated with a

Downloaded from http://www.everyspec.com

given file node handle opened under an exclusive intent on the contents of the node. Consequently, a single Ada program can obtain exclusive access to an external file and yet open and operate upon multiple file handles to this file without relinquishing exclusive access for obtaining additional file handles to the file. The opening of a second node handle to this file node by the same or any other process will however be delayed as specified in Table VI, page 62. It is not possible to change the intent relating to the contents of an open file node handle if there are open file handles associated with the node (see Section 5.1.2.3, page 67). The interaction of operations on two or more file handles associated with the same open file node handle is implementation-dependent.

File handles are also opened under a mode which determines the allowed data transfer operations. This mode must be consistent with the intent under which the associated file node was opened, as shown in Table X.

TABLE X. Modes and Intents for Input and Output

Mode	INTENT required
IN_FILE	READ_CONTENTS
OUT_FILE	WRITE_CONTENTS
INOUT_FILE	READ_CONTENTS and WRITE_CONTENTS
APPEND_FILE	APPEND_CONTENTS

Several predefined attributes are applicable to file nodes. The attributes FILE_KIND and ACCESS_METHOD are predefined on all file nodes. The attribute DEVICE_KIND is predefined on all file nodes whose FILE_KIND attribute has the value DEVICE. The attribute QUEUE_KIND is predefined on all file nodes whose FILE_KIND attribute has the value QUEUE. These attributes provide information about the contents of a file node and how it may be accessed.

The predefined values for the predefined file node attribute FILE_KIND are SECONDARY_STORAGE, QUEUE and DEVICE.

The predefined values for the predefined file node attribute ACCESS_METHOD are SEQUENTIAL, DIRECT and TEXT. These values indicate which of the predefined CAIS input and output packages may be used to perform input and output operations on the contents of the file node. A value of SEQUENTIAL indicates that the CAIS_SEQUENTIAL_IO package may be used. A value of DIRECT indicates that either of the packages CAIS_DIRECT_IO or CAIS_SEQUENTIAL_IO may be used. A value of TEXT indicates that the package CAIS_TEXT_IO may be used. A CAIS implementation is permitted to add additional values for this attribute.

The predefined values for the predefined file node attribute DEVICE_KIND are SCROLL_TERMINAL, PAGE_TERMINAL, FORM_TERMINAL and MAGNETIC_TAPE_DRIVE.

Downloaded from http://www.everyspec.com

One or more of these values may be specified for a device file node to indicate which device packages may be used to operate upon the contents of the device file node. A value of SCROLL_TERMINAL indicates that the package CAIS_SCROLL_TERMINAL_IO may be used. A value of PAGE_TERMINAL indicates that the package CAIS_PAGE_TERMINAL_IO may be used. A value of FORM_TERMINAL indicates that the package CAIS_FORM_TERMINAL_IO may be used. A value of MAGNETIC_TAPE_DRIVE indicates that the package CAIS_MAGNETIC_TAPE_IO may be used. A CAIS implementation is permitted to add additional values for this attribute.

The predefined values for the predefined file node attribute QUEUE_KIND are explained in Section 5.3.7, page 253 and following.

Table XI summarizes the applicability of predefined CAIS packages to file nodes with certain values for the predefined file node attributes FILE_KIND, ACCESS_METHOD and DEVICE_KIND. If a CAIS implementation adds additional packages, or adds additional values for the predefined attributes DEVICE_KIND and ACCESS_METHOD, then it must document these extensions by an extended Table XI in Appendix F. A CAIS implementation is not permitted to reduce the applicability of the predefined packages.

TABLE XI. Input and Output Packages for File Attributes

The file attributes listed to the right may use the packages which are listed below according to the constraints in this Table.	FILE_KIND	ACCESS_METHOD	DEVICE_KIND
CAIS_IO_ATTRIBUTES	Any	Any	Any
CAIS_DIRECT_IO	Any	DIRECT	Any
CAIS_SEQUENTIAL_IO	Any	SEQUENTIAL	Any
CAIS_TEXT_IO	Any	TEXT	Any
CAIS_QUEUE_MANAGEMENT	QUEUE	Any	Any
CAIS_SCROLL_TERMINAL_IO	DEVICE	TEXT	SCROLL_TERMINAL
CAIS_PAGE_TERMINAL_IO	DEVICE	TEXT	PAGE_TERMINAL
CAIS_FORM_TERMINAL_IO	DEVICE	TEXT	FORM_TERMINAL
CAIS_MAGNETIC_TAPE_IO	DEVICE	TEXT	MAGNETIC_TAPE_DRIVE
CAIS_IMPORT_EXPORT	SECONDARY_STORAGE	Any	Any

Downloaded from http://www.everyspec.com

The CAIS defines three kinds of files: secondary storage files, queue files and device files, which correspond to the FILE_KIND values SECONDARY_STORAGE, QUEUE and DEVICE, respectively.

a. A *secondary storage file* in the CAIS represents a disk or other random access storage file.

   Each secondary storage file node has two predefined attributes related to the number of file storage units (see FILE_STORAGE_UNIT_SIZE, Section 5.7, page 513) allocated to the contents of the file node. The predefined attribute MAXIMUM_FILE_SIZE indicates the maximum number of file storage units that may be allocated to the contents of the file node. A value of zero indicates that the number of file storage units is unrestricted. The predefined attribute CURRENT_FILE_SIZE indicates the number of file storage units that are allocated to the contents of a file node. The exception CAPACITY_ERROR is raised when an operation on the contents of a secondary storage file node (the result of a write or close operation) causes the maximum permitted number of file storage units to be exceeded.

b. A *queue file* in the CAIS represents a sequence of information that is accessed in a first-in, first-out manner. There are three kinds of CAIS queue files: solo, copy and mimic. Queue files are further described in Section 5.3.7, page 253. Queues may be created by using the procedures specified in the CAIS_QUEUE_MANAGEMENT package. Queues may be read and written using the interfaces of the CAIS_TEXT_IO package or the CAIS_SEQUENTIAL_IO package. The allowable predefined values for the attribute ACCESS_METHOD are SEQUENTIAL and TEXT. A value of SEQUENTIAL indicates that the CAIS_SEQUENTIAL_IO package may be used; a value of TEXT indicates that the CAIS_TEXT_IO package may be used.

c. A *device file* in the CAIS represents a device. Packages to operate on device files are either predefined in the CAIS or can be added by individual CAIS implementations. The FILE_KIND attribute of device file nodes must have the value DEVICE. The CAIS predefines the following special device files: magnetic tape drive files and three kinds of terminal files. The DEVICE_KIND attribute of nodes for these CAIS predefined device files must have one or more of the values MAGNETIC_TAPE_DRIVE, SCROLL_TERMINAL, PAGE_TERMINAL and FORM_TERMINAL, respectively.

   A *terminal file* in the CAIS represents an interactive terminal device. Three kinds of terminal devices are distinguished in the CAIS: scroll, page and form terminals. These are distinguished because they have different characteristics which require specialized interfaces. The functionality of these interfaces is derived from [ANSI 79]. Scroll and page terminals may be represented either by a single terminal file for input and output or by two terminal files, one for input and one for output. The implementation determines, for each physical terminal, whether it will be represented by one or two terminal files. A form terminal is represented by a single terminal file for both input and output. Interfaces must be provided outside of the CAIS for the creation of terminal file nodes. Scroll terminal files are described in more detail in Section 5.3.8, page 284. Page terminal files are described in more detail in Section 5.3.9, page 319. Form terminal files are further described in Section 5.3.10, page 362. Terminal

Downloaded from http://www.everyspec.com

file nodes have a value of TEXT for the predefined attribute ACCESS_METHOD; this indicates that the package CAIS_TEXT_IO may be used to operate upon the contents of terminal file nodes.

A *magnetic tape drive file* in the CAIS represents a magnetic tape drive. Operations on magnetic tape drive files can affect either the magnetic tape or the drive. Interfaces must be provided outside of the CAIS for the creation of magnetic tape drive file nodes. Operations on magnetic tape drive files are defined by the interfaces in the package CAIS_MAGNETIC_TAPE_IO. Magnetic tape drive file nodes have a value of TEXT for the predefined attribute ACCESS_METHOD; this indicates that the package CAIS_TEXT_IO may be used to operate upon the contents of magnetic tape drive file nodes. Magnetic tape drive files are further described in Section 5.3.11, page 390.

The above discussion is summarized in Table XII.

Downloaded from http://www.everyspec.com

TABLE XII. File Node Predefined Entities

The attributes and attribute values listed below are applicable to the file kinds listed to the right.	SECONDARY_ STORAGE	QUEUE	DEVICE
ACCESS_METHOD	Attribute	Attribute	Attribute
SEQUENTIAL	Value	Value	N/A
DIRECT	Value	N/A	N/A
TEXT	Value	Value	Value
CURRENT_FILE_SIZE	Attribute	N/A	N/A
DEVICE_KIND	N/A	N/A	Attribute
SCROLL_TERMINAL	N/A	N/A	Value
PAGE_TERMINAL	N/A	N/A	Value
FORM_TERMINAL	N/A	N/A	Value
MAGNETIC_TAPE_DRIVE	N/A	N/A	Value
FILE_KIND	Attribute	Attribute	Attribute
SECONDARY_STORAGE	Value	N/A	N/A
QUEUE	N/A	Value	N/A
DEVICE	N/A	N/A	Value
HIGHEST_CLASSIFICATION	Attribute	Attribute	Attribute
LOWEST_CLASSIFICATION	Attribute	Attribute	Attribute
MAXIMUM_FILE_SIZE	Attribute	N/A	N/A
OBJECT_CLASSIFICATION	Attribute	Attribute	Attribute
TIME_ATTRIBUTE_WRITTEN	Attribute	Attribute	Attribute
TIME_CONTENTS_WRITTEN	Attribute	Attribute	Attribute
TIME_CREATED	Attribute	Attribute	Attribute
TIME_RELATIONSHIP_ WRITTEN	Attribute	Attribute	Attribute

Downloaded from http://www.everyspec.com

5.3.1 Package CAIS_DEVICES

This package defines certain types associated with input and output; these types allow a CAIS implementation to provide additional packages in support of additional devices in a way that is compatible with the CAIS standard.

```
type ACCESS_METHOD_KIND is (DIRECT, SEQUENTIAL, TEXT,
              implementation_defined);
```

ACCESS_METHOD_KIND is the enumeration of the kinds of access methods. A CAIS implementation is permitted to add additional enumeration literals to this type declaration to denote other access methods. If a CAIS implementation adds additional enumeration literals to this type declaration, then it must document these extensions in Appendix F.

```
type DEVICE_KIND_TYPE is (SCROLL_TERMINAL, PAGE_TERMINAL,
         FORM_TERMINAL, MAGNETIC_TAPE_DRIVE, implementation_defined);
```

DEVICE_KIND_TYPE is the enumeration of the possible kinds of devices. A CAIS implementation is permitted to add additional enumeration literals to this type declaration to denote other device kinds. If a CAIS implementation adds additional enumeration literals to this type declaration, then it must document these extensions in Appendix F.

Notes:

If a CAIS implementation adds additional enumeration literals to these types, then applying a USE clause may cause visibility problems regarding homographs of the added enumeration literals. Therefore, USE clauses applied to package CAIS_DEVICES should be avoided.

Downloaded from http://www.everyspec.com

5.3.2 Package CAIS_IO_DEFINITIONS

This package defines certain types, subtypes, constants and exceptions associated with file nodes.

```
type FILE_KIND is (SECONDARY_STORAGE, QUEUE, DEVICE);
```

FILE_KIND is the enumeration of file kinds.

```
type QUEUE_KIND is
          (SYNCHRONOUS_SOLO, NONSYNCHRONOUS_SOLO,
           NONSYNCHRONOUS_COPY, NONSYNCHRONOUS_MIMIC);
```

QUEUE_KIND is the enumeration of queue kinds.

```
type DEVICE_KIND_ARRAY is array (CAIS_POSITIVE range <>)
                      of CAIS_DEVICES.DEVICE_KIND_TYPE;
```

DEVICE_KIND_ARRAY is an array type whose elements are the possible kinds of devices.

```
IN_INTENT:      constant INTENT_ARRAY := (1=>READ_CONTENTS);
INOUT_INTENT:   constant INTENT_ARRAY := (READ_CONTENTS, WRITE_CONTENTS);
OUT_INTENT:     constant INTENT_ARRAY := (1=>WRITE_CONTENTS);
APPEND_INTENT:  constant INTENT_ARRAY := (1=>APPEND_CONTENTS);
```

The intents defined above correspond to the intents required for each of the modes associated with file nodes.

```
UNBOUNDED_FILE_SIZE:  constant CAIS_NATURAL := 0;
UNBOUNDED_QUEUE_SIZE: constant CAIS_NATURAL := 0;
```

UNBOUNDED_FILE_SIZE and UNBOUNDED_QUEUE_SIZE indicate, upon creation of a file or queue, respectively, that the size of the file or queue is to have an unrestricted limit.

```
DATA_ERROR:                 exception;
END_ERROR:                  exception;
FILE_KIND_ERROR:            exception;
FORM_STATUS_ERROR:          exception;
FUNCTION_KEY_STATUS_ERROR:  exception;
LAYOUT_ERROR:               exception;
MODE_ERROR:                 exception;
TERMINAL_POSITION_ERROR:    exception;
```

DATA_ERROR may be raised on input if the data read cannot be properly interpreted due to syntax or type errors.

END_ERROR is raised if an attempt is made to skip or to read past the end of a file or if no more elements can be read from a queue file and no process has the associated queue node open with the intent to write contents.

FILE_KIND_ERROR is raised if the value of any of the file attributes FILE_KIND, ACCESS_METHOD or DEVICE_KIND is incorrect for the operation.

Downloaded from http://www.everyspec.com

FORM_STATUS_ERROR is raised if a form already exists (was created) when it should not exist or if a form does not exist (has not been created) when it should exist.

FUNCTION_KEY_STATUS_ERROR is raised if a non-existent function key is referenced.

LAYOUT_ERROR is raised if a subprogram attempts to exceed various limits on output or if an interface returns a value which exceeds the range of its subtype.

MODE_ERROR is raised if an input or output operation is attempted that is in conflict with the mode specified upon opening or resetting the respective file handle.

TERMINAL_POSITION_ERROR is raised if more rows or columns are specified than exist following the terminal active position or if the active position is inappropriate for the operation.

The exception CAPACITY_ERROR of the package CAIS_PRAGMATICS (see Section 5.7, page 509) is raised if MAXIMUM_FILE_SIZE or MAXIMUM_QUEUE_SIZE is exceeded. This may occur either during a write operation or a close operation.

Downloaded from http://www.everyspec.com

5.3.3 Package CAIS_IO_ATTRIBUTES

This package provides facilities for obtaining the values of the following predefined file node attributes:

ACCESS_METHOD
FILE_KIND
QUEUE_KIND
DEVICE_KIND
CURRENT_FILE_SIZE
MAXIMUM_FILE_SIZE
CURRENT_QUEUE_SIZE
MAXIMUM_QUEUE_SIZE

The exceptions raised by all subprograms in this package are defined in the package CAIS_ DEFINITIONS.

The attributes ACCESS_METHOD, FILE_KIND, DEVICE_KIND, CURRENT_FILE_SIZE and MAXIMUM_FILE_SIZE are described in the introduction to CAIS Input and Output (see Section 5.3, page 208 and following). The attributes QUEUE_KIND, CURRENT_ QUEUE_SIZE and MAXIMUM_QUEUE_SIZE are described in the introduction to the discussion of Package CAIS_QUEUE_MANAGEMENT (see Section 5.3.7, page 253 and following).

Downloaded from http://www.everyspec.com

5.3.3.1 Determining the access method

```
function ACCESS_METHOD (NODE: in NODE_TYPE)
    return CAIS_DEVICES.ACCESS_METHOD_KIND;
```

Purpose:

This function returns the value of the predefined node attribute ACCESS_METHOD. The node is identified by the open node handle NODE.

Parameter:

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

Exceptions:

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

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a file node.

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

Additional Interface:

```
function ACCESS_METHOD (NAME: in PATHNAME)
    return CAIS_DEVICES.ACCESS_METHOD_KIND
is
    NODE: NODE_TYPE;
    RESULT: CAIS_DEVICES.ACCESS_METHOD_KIND;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := ACCESS_METHOD (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end ACCESS_METHOD;
```

Downloaded from http://www.everyspec.com

5.3.3.2 Determining the file kind

```
function KIND_OF_FILE (NODE: in NODE_TYPE)
    return FILE_KIND;
```

Purpose:

This function returns the value of the predefined node attribute FILE_KIND. The node is identified by the open node handle NODE.

Parameter:

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

Exceptions:

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

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a file node.

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

Additional Interface:

```
function KIND_OF_FILE (NAME: in PATHNAME)
    return FILE_KIND
is
    NODE: NODE_TYPE;
    RESULT: FILE_KIND;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := KIND_OF_FILE (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end KIND_OF_FILE;
```

Downloaded from http://www.everyspec.com

5.3.3.3 Determining the queue kind

```
function KIND_OF_QUEUE (NODE: in NODE_TYPE)
    return QUEUE_KIND;
```

Purpose:

This function returns the value of the predefined node attribute QUEUE_KIND. The node is identified by the open node handle NODE.

Parameter:

NODE is an open node handle identifying the queue node whose attribute QUEUE_KIND is being queried.

Exceptions:

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

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
is raised if the value of the predefined attribute FILE_KIND on the node identified by NODE is not QUEUE.

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

Additional Interface:

```
function KIND_OF_QUEUE (NAME: in PATHNAME)
    return QUEUE_KIND
is
    NODE: NODE_TYPE;
    RESULT: QUEUE_KIND;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := KIND_OF_QUEUE (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end KIND_OF_QUEUE;
```

Downloaded from http://www.everyspec.com

5.3.3.4 Determining the device kind

```
function KIND_OF_DEVICE (NODE: in NODE_TYPE)
    return DEVICE_KIND_ARRAY;
```

Purpose:

This function returns the value of the predefined node attribute DEVICE_KIND. The node is identified by the open node handle NODE. The result is an array of type DEVICE_KIND_ARRAY.

Parameter:

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

Exceptions:

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

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
: is raised if the value of the predefined attribute FILE_KIND on the node identified by NODE is not DEVICE.

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

Additional Interface:

```
function KIND_OF_DEVICE (NAME: in PATHNAME)
    return DEVICE_KIND_ARRAY
is
    NODE:   NODE_TYPE;
    RESULT: DEVICE_KIND_ARRAY;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := KIND_OF_DEVICE (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end KIND_OF_DEVICE;
```

Downloaded from http://www.everyspec.com

5.3.3.5 Determining the current file size

```
function CURRENT_FILE_SIZE (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns the value of the predefined node attribute CURRENT_FILE_SIZE. The node is identified by the open node handle NODE. The value is expressed in multiples of FILE_STORAGE_UNIT_SIZE defined in the package CAIS_PRAGMATICS.

Parameter:

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

Exceptions:

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

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
: is raised if the value of the predefined attribute FILE_KIND of the node identified by NODE is not SECONDARY_STORAGE.

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

Additional Interface:

```
function CURRENT_FILE_SIZE (NAME: in PATHNAME)
    return CAIS_NATURAL
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_NATURAL;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := CURRENT_FILE_SIZE (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end CURRENT_FILE_SIZE;
```

Downloaded from http://www.everyspec.com

5.3.3.6 Determining the maximum file size

```
function MAXIMUM_FILE_SIZE (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns the value of the predefined node attribute MAXIMUM_FILE_SIZE. The node is identified by the open node handle NODE. The value is expressed in multiples of FILE_STORAGE_UNIT_SIZE defined in the package CAIS_PRAGMATICS.

Parameter:

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

Exceptions:

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

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
is raised if the value of the predefined attribute FILE_KIND of the node identified by NODE is not SECONDARY_STORAGE.

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

Additional Interface:

```
function MAXIMUM_FILE_SIZE (NAME: in PATHNAME)
    return CAIS_NATURAL
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_NATURAL;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := MAXIMUM_FILE_SIZE (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end MAXIMUM_FILE_SIZE;
```

Downloaded from http://www.everyspec.com

5.3.3.7 Determining the current queue size

```
function CURRENT_QUEUE_SIZE (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns the value of the predefined node attribute CURRENT_QUEUE_SIZE. The node is identified by the open node handle NODE. The value is expressed in multiples of QUEUE_STORAGE_UNIT_SIZE defined in the package CAIS_PRAGMATICS.

Parameter:

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

Exceptions:

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

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
: is raised if the value of the predefined attribute FILE_KIND of the node identified by NODE is not QUEUE or if the value of the predefined attribute QUEUE_KIND of the node identified by NODE is not NONSYNCHRONOUS_SOLO, NONSYNCHRONOUS_COPY or NONSYNCHRONOUS_MIMIC.

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

Additional Interface:

```
function CURRENT_QUEUE_SIZE (NAME: in PATHNAME)
    return CAIS_NATURAL
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_NATURAL;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := CURRENT_QUEUE_SIZE (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end CURRENT_QUEUE_SIZE;
```

Downloaded from http://www.everyspec.com

5.3.3.8 Determining the maximum queue size

```
function MAXIMUM_QUEUE_SIZE (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns the value of the predefined node attribute MAXIMUM_QUEUE_SIZE. The node is identified by the open node handle NODE. The value is expressed in multiples of QUEUE_STORAGE_UNIT_SIZE defined in the package CAIS_PRAGMATICS.

Parameter:

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

Exceptions:

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

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
is raised if the value of the predefined attribute FILE_KIND of the node identified by NODE is not QUEUE or if the value of the predefined attribute QUEUE_KIND of the node identified by NODE is not NONSYNCHRONOUS_SOLO, NONSYNCHRONOUS_COPY or NONSYNCHRONOUS_MIMIC.

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

Additional Interface:

```
function MAXIMUM_QUEUE_SIZE (NAME: in PATHNAME)
    return CAIS_NATURAL
is
    NODE:   NODE_TYPE;
    RESULT: CAIS_NATURAL;
begin
    OPEN (NODE, NAME, (1=>READ_ATTRIBUTES));
    RESULT := MAXIMUM_QUEUE_SIZE (NODE);
    CLOSE (NODE);
    return RESULT;
exception
    when others =>
        CLOSE (NODE);
        raise;
end MAXIMUM_QUEUE_SIZE;
```

Downloaded from http://www.everyspec.com

5.3.4 Package CAIS_DIRECT_IO

This package provides facilities for directly accessing data elements in CAIS files.

Files written with CAIS_DIRECT_IO are also readable by CAIS_SEQUENTIAL_IO if the two generic packages are instantiated with the same data type. The package specification and semantics of CAIS_DIRECT_IO are comparable to those of the [1815A] package DIRECT_IO.

The subprograms of CAIS_DIRECT_IO correspond to the subprograms in [1815A] DIRECT_IO as follows:

CREATE replaces [1815A] DIRECT_IO.CREATE.

OPEN replaces [1815A] DIRECT_IO.OPEN.

CLOSE replaces [1815A] DIRECT_IO.CLOSE.

DELETE does not exist in the package CAIS_DIRECT_IO.

RESET replaces [1815A] DIRECT_IO.RESET.

NAME does not exist in the package CAIS_DIRECT_IO.

FORM does not exist in the package CAIS_DIRECT_IO.

SYNCHRONIZE
is an additional subprogram that does not exist in [1815A] DIRECT_IO.

All other subprograms in [1815A] DIRECT_IO are also in CAIS_DIRECT_IO and have the same syntax and semantics, except that all types and subtypes are CAIS-defined types and subtypes and that additional semantics apply to input and output operations on queues (see Section 5.3.7, page 253).

The exceptions raised by all subprograms in CAIS_DIRECT_IO are defined in CAIS_DEFINITIONS and CAIS_IO_DEFINITIONS.

Downloaded from http://www.everyspec.com

5.3.4.1 Definition of Types

```
type FILE_TYPE is limited private;

type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
```

FILE_TYPE describes the type for file handles for all direct input and output operations.

FILE_MODE indicates whether input operations, output operations or both can be performed on the direct file handle. The values for FILE_MODE are the same as [1815A] and correspond respectively to the following cases:

a. IN_FILE corresponds to the case where only reading is to be performed.

b. INOUT_FILE corresponds to the case where both reading and writing are to be performed.

c. OUT_FILE corresponds to the case where only writing is to be performed.

Downloaded from http://www.everyspec.com

5.3.4.2 Creating a direct file

```
procedure CREATE
   (NODE:                    in out NODE_TYPE;
    FILE:                    in out FILE_TYPE;
    BASE:                    in     NODE_TYPE;
    KEY:                     in     RELATIONSHIP_KEY := LATEST_KEY;
    RELATION:                in     RELATION_NAME := DEFAULT_RELATION;
    INTENT:                  in     INTENT_ARRAY := INOUT_INTENT;
    MODE:                    in     FILE_MODE := INOUT_FILE;
    ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
    MAXIMUM_FILE_SIZE:       in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
    DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
```

Purpose:

This procedure creates a file and its file node; the file node is identified by the BASE, KEY and RELATION parameters. It also installs the primary relationship to the node NODE as well as the corresponding secondary relationship of the predefined relation PARENT from this node to the node identified by BASE. Each element of the file is directly addressable by an index. [1815A] defines what constitutes an *element*. The predefined attributes NODE_KIND, FILE_KIND, and ACCESS_METHOD are assigned the values FILE, SECONDARY_STORAGE, and DIRECT, respectively, as part of the creation.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node. The MAXIMUM_FILE_SIZE parameter provides the value for the predefined attribute MAXIMUM_FILE_SIZE with a value of zero indicating unrestricted size. The DISCRETIONARY_ACCESS parameter specifies initial access control information to be established for the created node (see Section 4.4.2 for details).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

Parameters:

NODE	is a node handle, initially closed, to be opened to the newly created node.
FILE	is a file handle, initially closed, to be opened.
BASE	is an open node handle to the node which will be the source node of the primary relationship to the new node.
KEY	is the relationship key designator of the primary relationship to be created.
RELATION	is the relation name of the primary relationship to be created.

Downloaded from http://www.everyspec.com

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

MODE indicates the mode under which the file handle is to be opened.

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

MAXIMUM_FILE_SIZE
defines the value for the predefined attribute MAXIMUM_FILE_SIZE.

DISCRETIONARY_ACCESS
is the initial access control information associated with the created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given by BASE, KEY and RELATION.

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

Downloaded from http://www.everyspec.com

STATUS_ERROR
is raised if BASE is not an open node handle, if FILE is an open file handle at the time of the call or if NODE is an open node handle at the time of the call.

INTENT_VIOLATION
is raised if BASE was not opened with an intent establishing the right to create relationships or if the INTENT given is incompatible with the MODE according to Table X, page 209.

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:                   in out NODE_TYPE;
    FILE:                   in out FILE_TYPE;
    NAME:                   in     PATHNAME;
    INTENT:                 in     INTENT_ARRAY := INOUT_INTENT;
    MODE:                   in     FILE_MODE := INOUT_FILE;
    ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
    MAXIMUM_FILE_SIZE:      in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
    DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                               CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST)
is
     BASE: NODE_TYPE;
begin
     OPEN (BASE, BASE_PATH(NAME), (1=>APPEND_RELATIONSHIPS));
     CREATE (NODE, FILE, BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
             INTENT, MODE, ATTRIBUTES, MAXIMUM_FILE_SIZE,
             DISCRETIONARY_ACCESS, MANDATORY_ACCESS);
     CLOSE (BASE);
exception
     when others =>
          CLOSE (BASE);
          raise;
end CREATE;
```

Downloaded from http://www.everyspec.com

5.3.4.3 Opening a direct file handle

```
procedure OPEN (FILE: in out FILE_TYPE;
                NODE: in     NODE_TYPE;
                MODE: in     FILE_MODE);
```

Purpose:

This procedure opens a file handle on a direct file, given an open node handle on the associated file node. Each element of the file is directly addressable by an index.

Parameters:

FILE	is a file handle, initially closed, to be opened.
NODE	is an open node handle to the file node.
MODE	indicates the mode under which the file handle is to be opened.

Exceptions:

STATUS_ERROR
: is raised if FILE is an open file handle at the time of the call on OPEN or if NODE is not an open node handle.

INTENT_VIOLATION
: is raised if NODE was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

NODE_KIND_ERROR
: is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
: is raised if the values of the predefined file node attributes FILE_KIND, ACCESS_METHOD and DEVICE_KIND are not appropriate for the package containing this procedure according to Table XI, page 210.

USE_ERROR
: is raised if an open file handle identifies the same file node contents and the CAIS implementation does not support the existence of multiple file handles identifying the same file node contents. Any such restriction must be documented in Appendix F. An implementation is allowed to raise this exception only if it is based on operating system support that does not provide this capability.

Notes:

Closing an open node handle also closes any open file handles which may be associated with it.

Downloaded from http://www.everyspec.com

5.3.4.4 Closing a direct file handle

```
procedure CLOSE (FILE: in out FILE_TYPE);
```

Purpose:

This procedure severs the association between the internal file identified by the file handle FILE and its associated node contents. It also severs any association between the file handle FILE and its associated node handle. Closing an already closed file handle has no effect.

Parameter:

FILE	is a file handle, initially open, to be closed.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.4.5 Resetting a direct file handle

```
procedure RESET (FILE: in out FILE_TYPE;
                 MODE: in     FILE_MODE);
```

Purpose:

This procedure sets the current mode of the file handle FILE to the mode given by the MODE parameter.

It also positions the given internal file so that reading from or writing to its elements can be restarted from the beginning of the internal file. The current index is set to one.

Parameters:

FILE — is an open file handle identifying the internal file to be reset.

MODE — indicates the new mode under which the file handle is to be reset.

Exceptions:

STATUS_ERROR
: is raised if FILE is not an open file handle.

USE_ERROR
: is raised if the CAIS implementation does not support resetting the file handle to the specified mode.

INTENT_VIOLATION
: is raised if the file node handle associated with the file handle FILE was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

Downloaded from http://www.everyspec.com

5.3.4.6 Synchronizing the internal file with file node contents

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

Purpose:

This procedure forces all data that has been written using the file handle FILE to be transmitted to the contents of the file node with which it is associated.

Parameter:

FILE is an open file handle identifying the internal file to be synchronized.

Exceptions:

MODE_ERROR is raised if the file handle FILE is of mode IN_FILE.

STATUS_ERROR
is raised if FILE is not an open file handle.

Notes:

For all write operations in the CAIS, the conditions upon which data are transferred from an internal file to the contents of a file node are implementation-dependent. Data in the internal file of a process are inaccessible to other processes. This procedure ensures that the data in the internal file and the data in the contents of the file node coincide.

Downloaded from http://www.everyspec.com

5.3.5 Package CAIS_SEQUENTIAL_IO

This package provides facilities for sequentially accessing data elements in CAIS files. [1815A] defines what constitutes an element.

The package specification and semantics of CAIS_SEQUENTIAL_IO are comparable to those of the [1815A] package SEQUENTIAL_IO.

The subprograms of CAIS_SEQUENTIAL_IO correspond to the subprograms in [1815A] SEQUENTIAL_IO as follows:

CREATE	replaces [1815A] SEQUENTIAL_IO.CREATE.
OPEN	replaces [1815A] SEQUENTIAL_IO.OPEN.
CLOSE	replaces [1815A] SEQUENTIAL_IO.CLOSE.
DELETE	does not exist in the package CAIS_SEQUENTIAL_IO.
RESET	replaces [1815A] SEQUENTIAL_IO.RESET.
NAME	does not exist in the package CAIS_SEQUENTIAL_IO.
FORM	does not exist in the package CAIS_SEQUENTIAL_IO.
SYNCHRONIZE	is an additional subprogram that does not exist in [1815A] SEQUENTIAL_IO.

All other subprograms in [1815A] SEQUENTIAL_IO are also in CAIS_SEQUENTIAL_IO and have the same syntax and semantics, except that all types and subtypes are CAIS-defined types and subtypes, that all operations on file handles of mode APPEND_FILE should be the same as those of mode OUT_FILE and that additional semantics apply to input and output operations on queues (see Section 5.3.7, page 253).

The exceptions raised by all subprograms in CAIS_SEQUENTIAL_IO are defined in CAIS_DEFINITIONS and CAIS_IO_DEFINITIONS.

Downloaded from http://www.everyspec.com

5.3.5.1 Definition of types

```
type FILE_TYPE is limited private;

type FILE_MODE is (IN_FILE, OUT_FILE, APPEND_FILE);
```

FILE_TYPE describes the type for file handles for all sequential input and output operations.

FILE_MODE indicates whether input operations or output operations can be performed on the sequential file handle. A mode of APPEND_FILE causes any elements that are written to the specified file handle to be appended to the elements that are already in the file. The values for FILE_MODE, except for APPEND_FILE, are the same as [1815A] and correspond respectively to the following cases:

a. IN_FILE corresponds to the case where only reading is to be performed.

b. OUT_FILE corresponds to the case where only writing is to be performed.

c. APPEND_FILE corresponds to the case where only writing (beginning at the end of the file) is to be performed. The file terminator is deleted at the time the file is opened.

Downloaded from http://www.everyspec.com

5.3.5.2 Creating a sequential file

```
procedure CREATE
    (NODE:                     in out NODE_TYPE;
     FILE:                     in out FILE_TYPE;
     BASE:                     in     NODE_TYPE;
     KEY:                      in     RELATIONSHIP_KEY := LATEST_KEY;
     RELATION:                 in     RELATION_NAME := DEFAULT_RELATION;
     INTENT:                   in     INTENT_ARRAY := OUT_INTENT;
     MODE:                     in     FILE_MODE := OUT_FILE;
     ATTRIBUTES:               in     ATTRIBUTE_LIST := EMPTY_LIST;
     MAXIMUM_FILE_SIZE:        in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
     DISCRETIONARY_ACCESS:     in     DISCRETIONARY_ACCESS_LIST :=
                                  CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:         in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
```

Purpose:

This procedure creates a file and its file node; the file node is identified by the BASE, KEY and RELATION parameters. It also installs the primary relationship to the node NODE as well as the corresponding secondary relationship of the predefined relation PARENT from this node to the node identified by BASE. Each element of the file is sequentially accessible. The predefined node attributes NODE_KIND, FILE_KIND, and ACCESS_METHOD are assigned the values FILE, SECONDARY_STORAGE, and SEQUENTIAL, respectively, as part of the creation.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node. The MAXIMUM_FILE_SIZE parameter provides the value for the predefined node attribute MAXIMUM_FILE_SIZE with a value of zero indicating unrestricted size. The DISCRETIONARY_ACCESS parameter specifies initial access control information to be established for the created node (see Section 4.4.2, page 36 for details).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

Parameters:

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

FILE — is a file handle, initially closed, to be opened.

BASE — is an open node handle to the node which will be the source node of the primary relationship to the new node.

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

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

Downloaded from http://www.everyspec.com

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

MODE indicates the mode under which the file handle is to be opened.

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

MAXIMUM_FILE_SIZE
defines the value for the predefined attribute MAXIMUM_FILE_SIZE.

DISCRETIONARY_ACCESS
is the initial access control information associated with the created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given by BASE, KEY and RELATION.

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

Downloaded from http://www.everyspec.com

STATUS_ERROR
: is raised if BASE is not an open node handle, if FILE is an open file handle at the time of the call, or if NODE is an open node handle at the time of the call.

INTENT_VIOLATION
: is raised if BASE was not opened with an intent establishing the right to append relationships or if the INTENT given is incompatible with the MODE according to Table X, page 209.

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:                      in out NODE_TYPE;
      FILE:                      in out FILE_TYPE;
      NAME:                      in     PATHNAME;
      INTENT:                    in     INTENT_ARRAY := OUT_INTENT;
      MODE:                      in     FILE_MODE := OUT_FILE;
      ATTRIBUTES:                in     ATTRIBUTE_LIST := EMPTY_LIST;
      MAXIMUM_FILE_SIZE:         in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
      DISCRETIONARY_ACCESS:      in     DISCRETIONARY_ACCESS_LIST :=
                                     CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:          in     MANDATORY_ACCESS_LIST := EMPTY_LIST)
is
     BASE: NODE_TYPE;
begin
     OPEN (BASE, BASE_PATH(NAME), (1=>APPEND_RELATIONSHIPS));
     CREATE (NODE, FILE, BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
             INTENT, MODE, ATTRIBUTES, MAXIMUM_FILE_SIZE,
             DISCRETIONARY_ACCESS, MANDATORY_ACCESS);
     CLOSE (BASE);
exception
     when others =>
          CLOSE (BASE);
          raise;
end CREATE;
```

Downloaded from http://www.everyspec.com

5.3.5.3 Opening a sequential file handle

```
procedure OPEN (FILE:   in out FILE_TYPE;
                NODE:   in     NODE_TYPE;
                MODE:   in     FILE_MODE);
```

Purpose:

This procedure opens a file handle on a sequential file, given an open node handle on the associated file node. Each element of the file is sequentially accessible.

Parameters:

FILE is a file handle, initially closed, to be opened.

NODE is an open node handle to the file node.

MODE indicates the mode under which the file handle is to be opened.

Exceptions:

STATUS_ERROR
is raised if FILE is an open file handle at the time of the call on OPEN or if NODE is not an open node handle.

INTENT_VIOLATION
is raised if NODE was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
is raised if the values of the predefined file node attributes FILE_KIND, ACCESS_METHOD and DEVICE_KIND are not appropriate for the package containing this procedure according to Table XI, page 210.

USE_ERROR is raised if an open file handle identifies the same file node contents and the CAIS implementation does not support the existence of multiple file handles identifying the same file node contents. Any such restriction must be documented in Appendix F. An implementation is allowed to raise this exception only if it is based on operating system support that does not provide this capability.

Notes:

Closing an open node handle also closes any open file handles which may be associated with it.

Downloaded from http://www.everyspec.com

5.3.5.4 Closing a sequential file handle

```
procedure CLOSE (FILE: in out FILE_TYPE);
```

Purpose:

This procedure severs the association between the internal file identified by the file handle FILE and its associated node contents. It also severs any association between the file handle FILE and its associated node handle. Closing an already closed file handle has no effect.

Parameter:

FILE	is a file handle, initially open, to be closed.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.5.5 Resetting a sequential file handle

```
procedure RESET (FILE: in out FILE_TYPE;
                 MODE: in     FILE_MODE);
```

Purpose:

This procedure sets the current mode of the file handle FILE to the mode given by the MODE parameter.

If the new mode is IN_FILE or OUT_FILE, this procedure positions the given internal file so that reading from or writing to its elements can be restarted from the beginning of the internal file. If the new mode is APPEND_FILE, this procedure positions the given internal file so that writing to its elements can be restarted at the end of the internal file.

Parameters:

FILE — is an open file handle identifying the internal file to be reset.

MODE — indicates the new mode under which the file handle is to be reset.

Exceptions:

STATUS_ERROR
is raised if FILE is not an open file handle.

USE_ERROR — is raised if the CAIS implementation does not support resetting the file handle to the specified mode.

INTENT_VIOLATION
is raised if the file node handle associated with the file handle FILE was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

Downloaded from http://www.everyspec.com

5.3.5.6 Synchronizing the internal file with file node contents

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

Purpose:

This procedure forces all data that has been written using the file handle FILE to be transmitted to the contents of the file node with which it is associated.

Parameters:

FILE is an open file handle identifying the internal file to be synchronized.

Exceptions:

MODE_ERROR is raised if the file handle identified by FILE is of mode IN_FILE.

STATUS_ERROR
is raised if FILE is not an open file handle.

Notes:

For all write operations in the CAIS the conditions upon which data are transferred from an internal file to the contents of a file node are implementation-dependent. Data in the internal file of a process are inaccessible to other processes. This procedure ensures that the data in the internal file and the data in the contents of the file node coincide.

Downloaded from http://www.everyspec.com

5.3.6 Package CAIS_TEXT_IO

This package provides facilities for accessing textual data elements in CAIS files. [1815A] defines what constitutes an *element*.

The package specification and semantics of CAIS_TEXT_IO are comparable to those of the [1815A] package TEXT_IO.

The subprograms of CAIS_TEXT_IO correspond to the subprograms in [1815A] TEXT_IO as follows:

CREATE	replaces [1815A] TEXT_IO.CREATE.
OPEN	replaces [1815A] TEXT_IO.OPEN.
CLOSE	replaces [1815A] TEXT_IO.CLOSE.
DELETE	does not exist in the package CAIS_TEXT_IO.
RESET	replaces [1815A] TEXT_IO.RESET.
NAME	does not exist in the package CAIS_TEXT_IO.
FORM	does not exist in the package CAIS_TEXT_IO.
STANDARD_INPUT	does not exist in the package CAIS_TEXT_IO.
STANDARD_OUTPUT	does not exist in the package CAIS_TEXT_IO.
SYNCHRONIZE	is an additional subprogram that does not exist in [1815A] TEXT_IO.

All other subprograms in [1815A] TEXT_IO are also in CAIS_TEXT_IO and have the same syntax and semantics, except that all types and subtypes are CAIS-defined types and subtypes, that all operations on file handles of mode APPEND_FILE should be the same as those of mode OUT_FILE and that additional semantics apply to input and output operations on queues (see Section 5.3.7, page 253).

The exceptions raised by all subprograms in CAIS_TEXT_IO are defined in CAIS_DEFINITIONS and CAIS_IO_DEFINITIONS.

Downloaded from http://www.everyspec.com

5.3.6.1 Definition of types

```
type FILE_TYPE is limited private;

type FILE_MODE is (IN_FILE, OUT_FILE, APPEND_FILE);
```

FILE_TYPE describes the type for file handles for all text input and output operations.

FILE_MODE indicates whether input operations or output operations can be performed on the text file handle. A mode of APPEND_FILE causes any text written to the specified file handle to be appended to the text that is already in the file. The values for FILE_MODE, except for APPEND_FILE, are the same as [1815A] and correspond respectively to the following cases:

a. IN_FILE corresponds to the case where only reading is to be performed.

b. OUT_FILE corresponds to the case where only writing is to be performed.

c. APPEND_FILE corresponds to the case where only writing (beginning at the end of the file) is to be performed. The file terminator is deleted at the time the file handle is opened.

Downloaded from http://www.everyspec.com

5.3.6.2 Creating a text file

```
procedure CREATE
   (NODE:                    in out NODE_TYPE;
    FILE:                    in out FILE_TYPE;
    BASE:                    in     NODE_TYPE;
    KEY:                     in     RELATIONSHIP_KEY := LATEST_KEY;
    RELATION:                in     RELATION_NAME := DEFAULT_RELATION;
    INTENT:                  in     INTENT_ARRAY := OUT_INTENT;
    MODE:                    in     FILE_MODE := OUT_FILE;
    ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
    MAXIMUM_FILE_SIZE:       in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
    DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
```

Purpose:

This procedure creates a file and its file node; the file node is identified by the BASE, KEY and RELATION parameters. It also installs the primary relationship to the node NODE as well as the corresponding secondary relationship of the predefined relation PARENT from this node to the node identified by BASE. The file is textual. The attributes NODE_KIND, FILE_KIND and ACCESS_METHOD are assigned the values FILE, SECONDARY_STORAGE and TEXT, respectively, as part of the creation.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node. The MAXIMUM_FILE_SIZE provides the value for the predefined node attribute MAXIMUM_FILE_SIZE with a value of zero indicating unrestricted size. The DISCRETIONARY_ACCESS parameter specifies initial access control information to be established for the created node (see Section 4.4.2, page 36 for details).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

Parameters:

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

FILE — is a file handle, initially closed, to be opened.

BASE — is an open node handle to the node which will be the source node of the primary relationship to the new node.

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

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

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

Downloaded from http://www.everyspec.com

MODE indicates the mode under which the file handle is to be opened.

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

MAXIMUM_FILE_SIZE
defines the value for the predefined node attribute MAXIMUM_FILE_SIZE.

DISCRETIONARY_ACCESS
is the initial access control information associated with the created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by KEY and RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given by BASE, KEY and RELATION.

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

STATUS_ERROR
is raised if BASE is not an open node handle, if FILE is an open file handle at the time of the call, or if NODE is an open node handle at the time of the call.

Downloaded from http://www.everyspec.com

INTENT_VIOLATION
is raised if BASE was not opened with an intent establishing the right to append relationships or if the INTENT given is incompatible with the MODE according to Table X, page 209.

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:                  in out NODE_TYPE;
     FILE:                  in out FILE_TYPE;
     NAME:                  in     PATHNAME;
     INTENT:                in     INTENT_ARRAY := OUT_INTENT;
     MODE:                  in     FILE_MODE := OUT_FILE;
     ATTRIBUTES:            in     ATTRIBUTE_LIST := EMPTY_LIST;
     MAXIMUM_FILE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
     DISCRETIONARY_ACCESS:  in     DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:      in     MANDATORY_ACCESS_LIST := EMPTY_LIST)
is
     BASE: NODE_TYPE;
begin
     OPEN (BASE, BASE_PATH(NAME), (1=>APPEND_RELATIONSHIPS));
     CREATE (NODE, FILE, BASE, LAST_KEY(NAME), LAST_RELATION(NAME),
             INTENT, MODE, ATTRIBUTES, MAXIMUM_FILE_SIZE,
             DISCRETIONARY_ACCESS, MANDATORY_ACCESS);
     CLOSE (BASE);
exception
     when others =>
          CLOSE (BASE);
          raise;
end CREATE;
```

Downloaded from http://www.everyspec.com

5.3.6.3 Opening a text file handle

```
procedure OPEN (FILE: in out FILE_TYPE;
                NODE: in     NODE_TYPE;
                MODE: in     FILE_MODE);
```

Purpose:

This procedure opens a file handle on a file that has textual contents, given an open node handle on the associated file node.

Parameters:

FILE is a file handle, initially closed, to be opened.

NODE is an open node handle to the file node.

MODE indicates the mode under which the file handle is to be opened.

Exceptions:

STATUS_ERROR
is raised if FILE is an open file handle at the time of the call on OPEN or if NODE is not an open node handle.

INTENT_VIOLATION
is raised if NODE was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
is raised if the values of the predefined file node attributes FILE_KIND, ACCESS_METHOD and DEVICE_KIND are not appropriate for the package containing this procedure according to Table XI, page 210.

USE_ERROR is raised if an open file handle identifies the same file node contents and the CAIS implementation does not support the existence of multiple file handles identifying the same file node contents. Any such restriction must be documented in Appendix F. An implementation is allowed to raise this exception only if it is based on operating system support that does not provide this capability.

Notes:

Closing an open node handle also closes any open file handles which may be associated with it.

Downloaded from http://www.everyspec.com

5.3.6.4 Closing a text file handle

```
procedure CLOSE (FILE: in out FILE_TYPE);
```

Purpose:

This procedure severs the association between the internal file identified by the file handle FILE and its associated node contents. It also severs any association between the file handle FILE and its associated node handle. Closing an already closed file handle has no effect.

Parameter:

FILE	is a file handle, initially open, to be closed.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.6.5 Resetting a text file handle

```
procedure RESET (FILE: in out FILE_TYPE;
                 MODE: in     FILE_MODE);
```

Purpose:

This procedure sets the current mode of the file handle FILE to the mode given by the MODE parameter. If the new mode for the file handle is OUT_FILE or APPEND_FILE, the page and line lengths are unbounded. For all modes, the current column, line and page numbers are set to one.

If the file handle FILE has the current mode OUT_FILE or APPEND_FILE, this procedure has the effect of calling NEW_PAGE, unless the current page is already terminated; then outputs a file terminator.

Parameters:

FILE — is an open file handle identifying the internal file to be reset.

MODE — indicates the new mode under which the file handle is to be reset.

Exceptions:

STATUS_ERROR
is raised if FILE is not an open file handle.

USE_ERROR — is raised if the CAIS implementation does not support resetting the file handle to the specified mode.

INTENT_VIOLATION
is raised if the file node handle associated with the file handle FILE was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

Downloaded from http://www.everyspec.com

5.3.6.6 Synchronizing the internal file with file node contents

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

Purpose:

This procedure forces all data that has been written to the internal file identified by FILE to be transmitted to the contents of the file node with which it is associated.

Parameters:

FILE is an open file handle identifying the internal file to be synchronized.

Exceptions:

MODE_ERROR is raised if the file handle FILE is of mode IN_FILE.

STATUS_ERROR
is raised if FILE is not an open file handle.

Notes:

For all write operations in the CAIS the conditions upon which data are transferred from an internal file to the contents of the file node are implementation-dependent. Data in the internal file of a process are inaccessible to other processes. This procedure ensures that the data in the internal file and the data in the contents of the file node coincide.

Downloaded from http://www.everyspec.com

5.3.7 Package CAIS_QUEUE_MANAGEMENT

This package provides facilities for creating queue file nodes. Queue file nodes may be used for interprocess communication or the sharing of a single data file among several processes. The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_IO_DEFINITIONS.

A queue file in the CAIS represents a sequence of elements that is accessed in a first-in, first-out manner (i.e., elements are read from a *queue* in the same order as they are written). Each element of a queue may be read only once (destructive read) from that queue. Elements are written to and read from queue files by using the packages CAIS_TEXT_IO (see Section 5.3.6) and CAIS_SEQUENTIAL_IO (see Section 5.3.5).

Queue file nodes have a predefined attribute QUEUE_KIND that determines the functionality of read or write operations upon the contents of that queue file node. The predefined values for the attribute QUEUE_KIND are SYNCHRONOUS_SOLO, NONSYNCHRONOUS_SOLO, NONSYNCHRONOUS_COPY and NONSYNCHRONOUS_MIMIC.

There are three kinds of CAIS queue files: solo queue, copy queue and mimic queue. The queue kinds differ in their initial contents and the effect of write operations.

a. A *solo queue* file operates like a simple queue, initially empty, in which all writes append information to the end and all reads are destructive. A write operation on a solo queue file handle affects only the solo queue file. The values for the predefined attribute QUEUE_KIND of a solo queue file node are SYNCHRONOUS_SOLO or NONSYNCHRONOUS_SOLO.

b. A *copy queue* file is initialized from the contents of another secondary storage file node (containing either text or sequential elements). After the creation of the copy queue file, the two files are independent of each other. The value for the predefined attribute QUEUE_KIND of a copy queue file node is NONSYNCHRONOUS_COPY.

c. A *mimic queue* file is initialized from the contents of another secondary storage file node called a *coupled file* node (containing either text or sequential elements). After the creation of the mimic queue file, the mimic queue file and its coupled file are mutually dependent. This means that elements written to a mimic queue file handle are appended to its coupled file (at an implementation-dependent time no later than when the mimic queue file handle is closed). Opening a mimic queue file handle with a mode of OUT_FILE or APPEND_FILE implies opening the coupled file node with intent to append contents. There is no effect on the contents of the mimic queue file node of writing or appending directly to the contents of its coupled file node. A relationship of the predefined relation MIMIC_FILE is established from the mimic queue file node to its coupled file node. The value for the predefined attribute QUEUE_KIND of a mimic queue file node is NONSYNCHRONOUS_MIMIC.

When a write operation is completed on a queue file handle the elements written are immediately available to be read from the queue file.

Downloaded from http://www.everyspec.com

Queue files may be either synchronous or nonsynchronous. A *synchronous queue* file has no elements. A write operation on a synchronous queue file handle is not completed until a corresponding read operation on the same queue file has been completed. Only a solo queue file can be synchronous. A *nonsynchronous queue* file permits an implementation-dependent number of write operations to occur (dependent upon the MAXIMUM_QUEUE_SIZE established at creation time) independent of any read operations on the queue file.

Every nonsynchronous queue file node has two predefined attributes related to the number of queue storage units (see QUEUE_STORAGE_UNIT_SIZE, Section 5.7, page 513) allocated to the contents of the queue file node. The predefined attribute MAXIMUM_QUEUE_SIZE indicates the maximum number of queue storage units that may be allocated to the contents of the nonsynchronous queue file node. A value of zero indicates that the number of queue storage units is unrestricted. The predefined attribute CURRENT_QUEUE_SIZE indicates the number of queue storage units that are allocated to the contents of a nonsynchronous queue file node. The exception CAPACITY_ERROR is raised when an attempt is made to create a nonsynchronous queue file node if copying of the contents of the coupled file node would exceed the permitted maximum number of allocated queue storage units for the queue file node contents to be created.

The above discussion is summarized in Table XIII.

Downloaded from http://www.everyspec.com

TABLE XIII. Queue File Node Predefined Entities

The attributes, attribute values and relations listed below are applicable to the queue kinds listed to the right.	Synchronous Solo	Nonsynchronous Solo	Nonsynchronous Copy	Nonsynchronous Mimic
ACCESS_METHOD	Attribute	Attribute	Attribute	Attribute
SEQUENTIAL	Value	Value	Value	Value
DIRECT	Value	N/A	N/A	N/A
TEXT	Value	Value	Value	Value
CURRENT_QUEUE_SIZE	N/A	Attribute	Attribute	Attribute
HIGHEST_CLASSIFICATION	Attribute	Attribute	Attribute	Attribute
LOWEST_CLASSIFICATION	Attribute	Attribute	Attribute	Attribute
MAXIMUM_QUEUE_SIZE	N/A	Attribute	Attribute	Attribute
MIMIC_FILE	N/A	N/A	N/A	Relation
OBJECT_CLASSIFICATION	Attribute	Attribute	Attribute	Attribute
QUEUE_KIND	Attribute	Attribute	Attribute	Attribute
SYNCHRONOUS_SOLO	Value	N/A	N/A	N/A
NONSYNCHRONOUS_SOLO	N/A	Value	N/A	N/A
NONSYNCHRONOUS_COPY	N/A	N/A	Value	N/A
NONSYNCHRONOUS_MIMIC	N/A	N/A	N/A	Value
TIME_ATTRIBUTE_WRITTEN	Attribute	Attribute	Attribute	Attribute
TIME_CONTENTS_WRITTEN	Attribute	Attribute	Attribute	Attribute
TIME_CREATED	Attribute	Attribute	Attribute	Attribute
TIME_RELATIONSHIP_WRITTEN	Attribute	Attribute	Attribute	Attribute

Copy queue files and mimic queue files are created by the interfaces CREATE_NONSYNCHRONOUS_COPY_QUEUE (see Section 5.3.7.1, page 257) and CREATE_NONSYNCHRONOUS_MIMIC_QUEUE (see Section 5.3.7.2, page 262), respectively. The value of the predefined attribute ACCESS_METHOD on the file node from which the contents of the copy queue file or mimic queue file are initialized determines whether the copy queue or the mimic queue can be operated on by the interfaces of the package CAIS_TEXT_IO (see Section 5.3.6, page 244) or the package CAIS_SEQUENTIAL_IO (see Section 5.3.5, page 235). Solo queue files are created by the following interfaces:

a. CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE (see Section 5.3.7.5, page 274) creates a synchronous solo queue that can only be operated on by the interfaces of the package CAIS_TEXT_IO (see Section 5.3.6, page 244).

b. CREATE_SYNCHRONOUS_SOLO_SEQUENTIAL_QUEUE (see Section 5.3.7.6, page 278) creates a synchronous solo queue that can only be operated on by the interfaces of the package CAIS_SEQUENTIAL_IO (see Section 5.3.5, page 235).

Downloaded from http://www.everyspec.com

c. CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE (see Section 5.3.7.3, page 267) creates a nonsynchronous solo queue that can only be operated on by the interfaces of the package CAIS_TEXT_IO (see Section 5.3.6, page 244).

d. CREATE_NONSYNCHRONOUS_SOLO_SEQUENTIAL_QUEUE (see Section 5.3.7.4, page 271) creates a nonsynchronous solo queue that can only be operated on by the interfaces of the package CAIS_SEQUENTIAL_IO (see Section 5.3.5, page 235).

Downloaded from http://www.everyspec.com

5.3.7.1 Creating a nonsynchronous copy queue node

```
procedure CREATE_NONSYNCHRONOUS_COPY_QUEUE
     (QUEUE_NODE:              in out NODE_TYPE;
      FILE_NODE:               in     NODE_TYPE;
      QUEUE_BASE:              in     NODE_TYPE;
      QUEUE_KEY:               in     RELATIONSHIP_KEY := LATEST_KEY;
      QUEUE_RELATION:          in     RELATION_NAME := DEFAULT_RELATION;
      INTENT:                  in     INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                   CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:      in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
```

Purpose:

This procedure creates a nonsynchronous copy queue file node and installs the primary relationship to it. The newly created nonsynchronous copy queue file node is identified by the QUEUE_BASE, QUEUE_KEY and QUEUE_RELATION parameters. It also installs the primary relationship to the node QUEUE_NODE as well as the corresponding secondary relationship of the predefined relation PARENT from this node to the node identified by QUEUE_BASE. An open node handle to the newly created node is returned in QUEUE_NODE.

The predefined attributes NODE_KIND and FILE_KIND are assigned the values FILE and QUEUE, respectively, as part of the creation. The predefined attribute ACCESS_METHOD is assigned the value of the predefined attribute ACCESS_METHOD of the node identified by FILE_NODE, or SEQUENTIAL if the latter value is DIRECT. The predefined attribute QUEUE_KIND is assigned the value NONSYNCHRONOUS_COPY.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

The MAXIMUM_QUEUE_SIZE parameter provides the value for the predefined node attribute MAXIMUM_QUEUE_SIZE with a value of zero indicating unrestricted size.

Upon completion of the call to this interface, the contents of the newly created nonsynchronous copy queue file node are initialized from the contents of the node identified by the node handle FILE_NODE.

Downloaded from http://www.everyspec.com

Parameters:

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

FILE_NODE is an open node handle identifying the file node whose contents will be used to initialize the contents of the queue file node.

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

QUEUE_KEY is the relationship key designator of the primary relationship to be created.

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

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

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

DISCRETIONARY_ACCESS
is the initial access control information associated with the newly created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

MAXIMUM_QUEUE_SIZE
defines the maximum size to which the queue may grow in terms of queue storage units (see Section 5.7, page 513).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by QUEUE_KEY and QUEUE_RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given.

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

Downloaded from http://www.everyspec.com

PREDEFINED_RELATION_ERROR
is raised if QUEUE_RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

FILE_KIND_ERROR
is raised if the value of the predefined attribute FILE_KIND on the node identified by FILE_NODE is not SECONDARY_STORAGE.

STATUS_ERROR
is raised if QUEUE_BASE or FILE_NODE are not open node handles or if QUEUE_NODE is an open node handle at the time of the call.

INTENT_VIOLATION
is raised if QUEUE_BASE was not opened with an intent establishing the right to append relationships or if FILE_NODE was not opened with an intent establishing the right to read contents.

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.

Downloaded from http://www.everyspec.com

Additional Interfaces:

```
procedure CREATE_NONSYNCHRONOUS_COPY_QUEUE
     (QUEUE_NODE:            in out NODE_TYPE;
      FILE_NODE:             in     NODE_TYPE;
      QUEUE_NAME:            in     PATHNAME;
      INTENT:                in     INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:            in     ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:  in     DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:      in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:    in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE)
is
     QUEUE_BASE: NODE_TYPE;
begin
     OPEN (QUEUE_BASE, BASE_PATH (QUEUE_NAME),
            (1=>APPEND_RELATIONSHIPS));
     CREATE_NONSYNCHRONOUS_COPY_QUEUE
               (QUEUE_NODE, FILE_NODE, QUEUE_BASE,
                LAST_KEY (QUEUE_NAME), LAST_RELATION (QUEUE_NAME),
                INTENT, ATTRIBUTES, DISCRETIONARY_ACCESS,
                MANDATORY_ACCESS, MAXIMUM_QUEUE_SIZE);
     CLOSE (QUEUE_BASE);
exception
     when others =>
          CLOSE (QUEUE_BASE);
          raise;
end CREATE_NONSYNCHRONOUS_COPY_QUEUE;

procedure CREATE_NONSYNCHRONOUS_COPY_QUEUE
     (FILE_NODE:             in NODE_TYPE;
      QUEUE_BASE:            in NODE_TYPE;
      QUEUE_KEY:             in RELATIONSHIP_KEY := LATEST_KEY;
      QUEUE_RELATION:        in RELATION_NAME := DEFAULT_RELATION;
      INTENT:                in INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:            in ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:  in DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:      in MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:    in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE)
is
     QUEUE_NODE: NODE_TYPE;
begin
     CREATE_NONSYNCHRONOUS_COPY_QUEUE
               (QUEUE_NODE, FILE_NODE, QUEUE_BASE,
                QUEUE_KEY, QUEUE_RELATION, INTENT, ATTRIBUTES,
                DISCRETIONARY_ACCESS, MANDATORY_ACCESS,
                MAXIMUM_QUEUE_SIZE);
     CLOSE (QUEUE_NODE);
exception
     when others =>
          CLOSE (QUEUE_NODE);
          raise;
end CREATE_NONSYNCHRONOUS_COPY_QUEUE;
```

Downloaded from http://www.everyspec.com


```
procedure CREATE_NONSYNCHRONOUS_COPY_QUEUE
     (FILE_NODE:              in NODE_TYPE;
      QUEUE_NAME:             in PATHNAME;
      INTENT:                 in INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:   in DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:     in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE)
is
     QUEUE_NODE: NODE_TYPE;
begin
     CREATE_NONSYNCHRONOUS_COPY_QUEUE
               (QUEUE_NODE, FILE_NODE, QUEUE_NAME, INTENT,
                ATTRIBUTES, DISCRETIONARY_ACCESS, MANDATORY_ACCESS,
                MAXIMUM_QUEUE_SIZE);
     CLOSE (QUEUE_NODE);
end CREATE_NONSYNCHRONOUS_COPY_QUEUE;
```

Notes:

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

Downloaded from http://www.everyspec.com

5.3.7.2 Creating a nonsynchronous mimic queue node

```
procedure CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
    (QUEUE_NODE:            in out NODE_TYPE;
     FILE_NODE:             in     NODE_TYPE;
     QUEUE_BASE:            in     NODE_TYPE;
     QUEUE_KEY:             in     RELATIONSHIP_KEY := LATEST_KEY;
     QUEUE_RELATION:        in     RELATION_NAME := DEFAULT_RELATION;
     INTENT:                in     INTENT_ARRAY := IN_INTENT;
     ATTRIBUTES:            in     ATTRIBUTE_LIST := EMPTY_LIST;
     DISCRETIONARY_ACCESS:  in     DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:      in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
     MAXIMUM_QUEUE_SIZE:    in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
```

Purpose:

This procedure creates a nonsynchronous mimic queue file node and installs the primary relationship to it. The newly created nonsynchronous mimic queue file node is identified by the QUEUE_BASE, QUEUE_KEY and QUEUE_RELATION parameters. It also installs the primary relationship to the node QUEUE_NODE as well as the corresponding secondary relationship of the predefined relation PARENT from this node to the node identified by QUEUE_BASE. An open node handle to the newly created node is returned in QUEUE_NODE.

The predefined attributes NODE_KIND and FILE_KIND are assigned the values FILE and QUEUE, respectively, as part of the creation. The predefined attribute ACCESS_METHOD is assigned the value of the predefined attribute ACCESS_METHOD of the node identified by FILE_NODE, or SEQUENTIAL if the latter value is DIRECT. The predefined attribute QUEUE_KIND is assigned the value NONSYNCHRONOUS_MIMIC.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

The MAXIMUM_QUEUE_SIZE parameter provides the value for the predefined node attribute MAXIMUM_QUEUE_SIZE with a value of zero indicating unrestricted size.

A relationship of the predefined relation MIMIC_FILE with an empty relationship key is created from the newly created nonsynchronous mimic queue file node to the node identified by the node handle FILE_NODE.

Downloaded from http://www.everyspec.com

Upon completion of the call to this interface, the contents of the newly created nonsynchronous mimic queue file node are initialized from the contents of the node identified by the node handle FILE_NODE.

Parameters:

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

FILE_NODE is an open node handle identifying the file node with which the mimic queue file node is to be coupled.

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

QUEUE_KEY is the relationship key designator of the primary relationship to be created.

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

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

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

DISCRETIONARY_ACCESS
is the initial access control information associated with the newly created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

MAXIMUM_QUEUE_SIZE
defines the maximum size to which the queue may grow in terms of queue storage units (see Section 5.7, page 513).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by QUEUE_KEY and QUEUE_RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given.

Downloaded from http://www.everyspec.com

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if QUEUE_RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

FILE_KIND_ERROR
is raised if the value of the predefined attribute *FILE_KIND* on the node identified by FILE_NODE is not SECONDARY_STORAGE.

STATUS_ERROR
is raised if QUEUE_BASE or FILE_NODE are not open node handles or if QUEUE_NODE is an open node handle at the time of the call.

INTENT_VIOLATION
is raised if QUEUE_BASE was not opened with an intent establishing the right to append relationships or if FILE_NODE was not opened with an intent establishing the right to read contents.

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.

Downloaded from http://www.everyspec.com

Additional Interfaces:

```
procedure CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
     (QUEUE_NODE:              in out NODE_TYPE;
      FILE_NODE:               in     NODE_TYPE;
      QUEUE_NAME:              in     PATHNAME;
      INTENT:                  in     INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                   CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:      in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE)
is
     QUEUE_BASE: NODE_TYPE;
begin
     OPEN (QUEUE_BASE, BASE_PATH (QUEUE_NAME),
          (1=>APPEND_RELATIONSHIPS));
     CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
            (QUEUE_NODE, FILE_NODE, QUEUE_BASE,
             LAST_KEY (QUEUE_NAME), LAST_RELATION (QUEUE_NAME),
             INTENT, ATTRIBUTES, DISCRETIONARY_ACCESS,
             MANDATORY_ACCESS, MAXIMUM_QUEUE_SIZE);
     CLOSE (QUEUE_BASE);
exception
     when others =>
          CLOSE (QUEUE_BASE);
          raise;
end CREATE_NONSYNCHRONOUS_MIMIC_QUEUE;

procedure CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
     (FILE_NODE:               in NODE_TYPE;
      QUEUE_BASE:              in NODE_TYPE;
      QUEUE_KEY:               in RELATIONSHIP_KEY := LATEST_KEY;
      QUEUE_RELATION:          in RELATION_NAME := DEFAULT_RELATION;
      INTENT:                  in INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:              in ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:    in DISCRETIONARY_ACCESS_LIST :=
                                   CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:        in MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:      in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE)
is
     QUEUE_NODE: NODE_TYPE;
begin
     CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
            (QUEUE_NODE, FILE_NODE, QUEUE_BASE,
             QUEUE_KEY, QUEUE_RELATION, INTENT, ATTRIBUTES,
             DISCRETIONARY_ACCESS, MANDATORY_ACCESS,
             MAXIMUM_QUEUE_SIZE);
     CLOSE (QUEUE_NODE);
exception
     when others =>
          CLOSE (QUEUE_NODE);
          raise;
end CREATE_NONSYNCHRONOUS_MIMIC_QUEUE;
```

Downloaded from http://www.everyspec.com


```
procedure CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
     (FILE_NODE:              in NODE_TYPE;
      QUEUE_NAME:             in PATHNAME;
      INTENT:                 in INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:   in DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:     in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE)
is
     QUEUE_NODE: NODE_TYPE;
begin
     CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
              (QUEUE_NODE, FILE_NODE, QUEUE_NAME,
               INTENT, ATTRIBUTES, DISCRETIONARY_ACCESS,
               MANDATORY_ACCESS, MAXIMUM_QUEUE_SIZE);
     CLOSE (QUEUE_NODE);
end CREATE_NONSYNCHRONOUS_MIMIC_QUEUE;
```

Notes:

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

Downloaded from http://www.everyspec.com

5.3.7.3 Creating a nonsynchronous solo text queue node

```
procedure CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
      (QUEUE_NODE:              in out NODE_TYPE;
       QUEUE_BASE:              in     NODE_TYPE;
       QUEUE_KEY:               in     RELATIONSHIP_KEY := LATEST_KEY;
       QUEUE_RELATION:          in     RELATION_NAME := DEFAULT_RELATION;
       INTENT:                  in     INTENT_ARRAY := IN_INTENT;
       ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
       DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                   CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
       MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
       MAXIMUM_QUEUE_SIZE:      in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
```

Purpose:

This procedure creates a nonsynchronous solo text queue file node and installs the primary relationship to it. The newly created nonsynchrononous solo text queue file node is identified by the QUEUE_BASE, QUEUE_KEY and QUEUE_RELATION parameters. It also installs the primary relationship to the node QUEUE_NODE as well as the corresponding secondary relationship of the predefined relation PARENT from this node to the node identified by QUEUE_BASE. An open node handle to the newly created node is returned in QUEUE_NODE.

The predefined attributes NODE_KIND and FILE_KIND are assigned the values FILE and QUEUE, respectively, as part of the creation. The predefined attribute ACCESS_METHOD is assigned the value TEXT. The predefined attribute QUEUE_KIND is assigned the value NONSYNCHRONOUS_SOLO.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

The MAXIMUM_QUEUE_SIZE parameter provides the value for the predefined node attribute MAXIMUM_QUEUE_SIZE with a value of zero indicating unrestricted size.

Parameters:

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

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

Downloaded from http://www.everyspec.com

QUEUE_KEY is the relationship key designator of the primary relationship to be created.

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

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

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

DISCRETIONARY_ACCESS
is the initial access control information associated with the newly created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

MAXIMUM_QUEUE_SIZE
defines the maximum size to which the queue may grow in terms of queue storage units (see Section 5.7, page 513).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by QUEUE_KEY and QUEUE_RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given.

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if QUEUE_RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

Downloaded from http://www.everyspec.com

STATUS_ERROR
is raised if QUEUE_BASE is not an open node handle or if QUEUE_NODE is an open node handle at the time of 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 access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Additional Interfaces:

```
procedure CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
     (QUEUE_NODE:             in out NODE_TYPE;
      QUEUE_NAME:             in     PATHNAME;
      INTENT:                 in     INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE)
is
     QUEUE_BASE: NODE_TYPE;
begin
     OPEN (QUEUE_BASE, BASE_PATH (QUEUE_NAME),
            (1=>APPEND_RELATIONSHIPS));
     CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
              (QUEUE_NODE, QUEUE_BASE,
               LAST_KEY (QUEUE_NAME), LAST_RELATION (QUEUE_NAME),
               INTENT, ATTRIBUTES, DISCRETIONARY_ACCESS,
               MANDATORY_ACCESS, MAXIMUM_QUEUE_SIZE);
     CLOSE (QUEUE_BASE);
exception
     when others =>
          CLOSE (QUEUE_BASE);
          raise;
end CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE;
```

Downloaded from http://www.everyspec.com


```
procedure CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
     (QUEUE_BASE:            in NODE_TYPE;
      QUEUE_KEY:             in RELATIONSHIP_KEY := LATEST_KEY;
      QUEUE_RELATION:        in RELATION_NAME := DEFAULT_RELATION;
      INTENT:                in INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:            in ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:  in DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:      in MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:    in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE)
is
     QUEUE_NODE: NODE_TYPE;
begin
     CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
             (QUEUE_NODE, QUEUE_BASE,
              QUEUE_KEY, QUEUE_RELATION, INTENT, ATTRIBUTES,
              DISCRETIONARY_ACCESS, MANDATORY_ACCESS,
              MAXIMUM_QUEUE_SIZE);
     CLOSE (QUEUE_NODE);
exception
     when others =>
          CLOSE (QUEUE_NODE);
          raise;
end CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE;

procedure CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
     (QUEUE_NAME:            in PATHNAME;
      INTENT:                in INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:            in ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:  in DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:      in MANDATORY_ACCESS_LIST := EMPTY_LIST;
      MAXIMUM_QUEUE_SIZE:    in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE)
is
     QUEUE_NODE: NODE_TYPE;
begin
     CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
             (QUEUE_NODE, QUEUE_NAME, INTENT,
              ATTRIBUTES, DISCRETIONARY_ACCESS, MANDATORY_ACCESS,
              MAXIMUM_QUEUE_SIZE);
     CLOSE (QUEUE_NODE);
end CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE;
```

Notes:

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

Downloaded from http://www.everyspec.com

5.3.7.4 Creating a nonsynchronous solo sequential queue node

```
generic
    type ELEMENT_TYPE is private;
procedure CREATE_NONSYNCHRONOUS_SOLO_SEQUENTIAL_QUEUE
    (QUEUE_NODE:              in out NODE_TYPE;
     QUEUE_BASE:              in     NODE_TYPE;
     QUEUE_KEY:               in     RELATIONSHIP_KEY := LATEST_KEY;
     QUEUE_RELATION:          in     RELATION_NAME := DEFAULT_RELATION;
     INTENT:                  in     INTENT_ARRAY := IN_INTENT;
     ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
     DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
     MAXIMUM_QUEUE_SIZE:      in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
```

Purpose:

This procedure creates a nonsynchronous solo sequential queue file node and installs the primary relationship to it. The newly created nonsynchronous solo sequential queue file node is identified by the QUEUE_BASE, QUEUE_KEY and QUEUE_RELATION parameters. It also installs the primary relationship to the node QUEUE_NODE as well as the corresponding secondary relationship of the predefined relation PARENT from this node to the node identified by QUEUE_BASE. An open node handle to the newly created node is returned in QUEUE_NODE.

The predefined attributes NODE_KIND and FILE_KIND are assigned the values FILE and QUEUE, respectively, as part of the creation. The predefined attribute ACCESS_METHOD is assigned the value SEQUENTIAL. The predefined attribute QUEUE_KIND is assigned the value NONSYNCHRONOUS_SOLO.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

The MAXIMUM_QUEUE_SIZE parameter provides the value for the predefined node attribute MAXIMUM_QUEUE_SIZE with a value of zero indicating unrestricted size.

The generic formal type ELEMENT_TYPE is the type for file elements that are to be read from or written to the queue file.

Downloaded from http://www.everyspec.com

Parameters:

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

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

QUEUE_KEY is the relationship key designator of the primary relationship to be created.

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

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

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

DISCRETIONARY_ACCESS
is the initial access control information associated with the newly created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

MAXIMUM_QUEUE_SIZE
defines the maximum size to which the queue may grow in terms of queue storage units (see Section 5.7, page 513).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by QUEUE_KEY and QUEUE_RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given.

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if QUEUE_RELATION is the name of a predefined relation that cannot be created by the user.

Downloaded from http://www.everyspec.com

PREDEFINED_ATTRIBUTE_ERROR
: is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR
: is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

STATUS_ERROR
: is raised if QUEUE_BASE is not an open node handle or if QUEUE_NODE is an open node handle at the time of 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 access controls. SECURITY_VIOLATION is raised only if the conditions for other exceptions are not present.

Downloaded from http://www.everyspec.com

5.3.7.5 Creating a synchronous solo text queue node

```
procedure CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
     (QUEUE_NODE:              in out NODE_TYPE;
      QUEUE_BASE:              in     NODE_TYPE;
      QUEUE_KEY:               in     RELATIONSHIP_KEY := LATEST_KEY;
      QUEUE_RELATION:          in     RELATION_NAME := DEFAULT_RELATION;
      INTENT:                  in     INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                  CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
```

Purpose:

This procedure creates a synchronous solo text queue file node and installs the primary relationship to it. The newly created synchronous solo text queue file node is identified by the QUEUE_BASE, QUEUE_KEY and QUEUE_RELATION parameters. It also installs the primary relationship to the node QUEUE_NODE as well as the corresponding secondary relationship of the predefined relation PARENT from this node to the node identified by QUEUE_BASE. An open node handle to the newly created node is returned in QUEUE_NODE.

The predefined attributes NODE_KIND and FILE_KIND are assigned the values FILE and QUEUE, respectively, as part of the creation. The predefined attribute ACCESS_METHOD is assigned the value TEXT. The predefined attribute QUEUE_KIND is assigned the value SYNCHRONOUS_SOLO.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

Parameters:

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

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

QUEUE_KEY is the relationship key designator of the primary relationship to be created.

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

Downloaded from http://www.everyspec.com

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

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

DISCRETIONARY_ACCESS
is the initial access control information associated with the newly created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by QUEUE_KEY and QUEUE_RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given.

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if QUEUE_RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

STATUS_ERROR
is raised if QUEUE_BASE is not an open node handle or if QUEUE_NODE is an open node handle at the time of the call.

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

Downloaded from http://www.everyspec.com

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_SYNCHRONOUS_SOLO_TEXT_QUEUE
     (QUEUE_NODE:              in out NODE_TYPE;
      QUEUE_NAME:              in     PATHNAME;
      INTENT:                  in     INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST)
is
     QUEUE_BASE: NODE_TYPE;
begin
     OPEN (QUEUE_BASE, BASE_PATH (QUEUE_NAME),
           (1=>APPEND_RELATIONSHIPS));
     CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
           (QUEUE_NODE, QUEUE_BASE,
            LAST_KEY (QUEUE_NAME), LAST_RELATION (QUEUE_NAME),
            INTENT, ATTRIBUTES, DISCRETIONARY_ACCESS,
            MANDATORY_ACCESS);
     CLOSE (QUEUE_BASE);
exception
     when others =>
          CLOSE (QUEUE_BASE);
          raise;
end CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE;

procedure CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
     (QUEUE_BASE:              in NODE_TYPE;
      QUEUE_KEY:               in RELATIONSHIP_KEY := LATEST_KEY;
      QUEUE_RELATION:          in RELATION_NAME := DEFAULT_RELATION;
      INTENT:                  in INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:              in ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:    in DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:        in MANDATORY_ACCESS_LIST := EMPTY_LIST)
is
     QUEUE_NODE: NODE_TYPE;
begin
     CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
           (QUEUE_NODE, QUEUE_BASE,
            QUEUE_KEY, QUEUE_RELATION, INTENT, ATTRIBUTES,
            DISCRETIONARY_ACCESS, MANDATORY_ACCESS);
     CLOSE (QUEUE_NODE);
exception
     when others =>
          CLOSE (QUEUE_NODE);
          raise;
end CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE;
```

Downloaded from http://www.everyspec.com


```
procedure CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
     (QUEUE_NAME:              in PATHNAME;
      INTENT:                  in INTENT_ARRAY := IN_INTENT;
      ATTRIBUTES:              in ATTRIBUTE_LIST := EMPTY_LIST;
      DISCRETIONARY_ACCESS:    in DISCRETIONARY_ACCESS_LIST :=
                                  CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:        in MANDATORY_ACCESS_LIST := EMPTY_LIST)
is
     QUEUE_NODE: NODE_TYPE;
begin
     CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
            (QUEUE_NODE, QUEUE_NAME, INTENT,
             ATTRIBUTES, DISCRETIONARY_ACCESS, MANDATORY_ACCESS);
     CLOSE (QUEUE_NODE);
end CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE;
```

Notes:

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

Downloaded from http://www.everyspec.com

5.3.7.6 Creating a synchronous solo sequential queue node

```
generic
    type ELEMENT_TYPE is private;
procedure CREATE_SYNCHRONOUS_SOLO_SEQUENTIAL_QUEUE
    (QUEUE_NODE:              in out NODE_TYPE;
     QUEUE_BASE:              in     NODE_TYPE;
     QUEUE_KEY:               in     RELATIONSHIP_KEY := LATEST_KEY;
     QUEUE_RELATION:          in     RELATION_NAME := DEFAULT_RELATION;
     INTENT:                  in     INTENT_ARRAY := IN_INTENT;
     ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
     DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
```

Purpose:

This procedure creates a synchronous solo sequential queue file node and installs the primary relationship to it. The newly created synchronous solo sequential queue file node is identified by the QUEUE_BASE, QUEUE_KEY and QUEUE_RELATION parameters. It also installs the primary relationship to the node QUEUE_NODE as well as the corresponding secondary relationship of the predefined relation PARENT from this node to the node identified by QUEUE_BASE. An open node handle to the newly created node is returned in QUEUE_NODE.

The predefined attributes NODE_KIND and FILE_KIND are assigned the values FILE and QUEUE, respectively, as part of the creation. The predefined attribute ACCESS_METHOD is assigned the value SEQUENTIAL. The predefined attribute QUEUE_KIND is assigned the value SYNCHRONOUS_SOLO.

The ATTRIBUTES parameter defines and provides initial values for attributes of the node.

The DISCRETIONARY_ACCESS parameter specifies the initial access control information to be established between the created node and the default group node of the current user (see Section 4.4).

The MANDATORY_ACCESS parameter specifies the object classification labels with which the node is to be created. If its value is the empty list, the node inherits the subject classification of the creating process as its object classification. Otherwise, it must be an unnamed list consisting of an identifier item and, optionally, an unnamed list of identifier items (see Table IV, page 51).

The generic type ELEMENT_TYPE describes the type for file elements that are to be read from or written to the queue file.

Parameters:

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

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

Downloaded from http://www.everyspec.com

QUEUE_KEY is the relationship key designator of the primary relationship to be created.

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

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

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

DISCRETIONARY_ACCESS
is the initial access control information associated with the newly created node; it is the value of the GRANT attribute of the access relationship to the user's default group node (see Section 4.4.2.3, page 40).

MANDATORY_ACCESS
is a list defining the classification label for the created node (see Table IV, page 51).

Exceptions:

PATHNAME_SYNTAX_ERROR
is raised if the node identification given by QUEUE_KEY and QUEUE_RELATION is syntactically illegal (see Table I, page 32).

EXISTING_NODE_ERROR
is raised if a node already exists with the identification given.

SYNTAX_ERROR
is raised if the ATTRIBUTES parameter (see description above), the DISCRETIONARY_ACCESS parameter (see Section 4.4.2.3) or the MANDATORY_ACCESS parameter (see Table IV, page 51) is syntactically illegal.

PREDEFINED_RELATION_ERROR
is raised if QUEUE_RELATION is the name of a predefined relation that cannot be created by the user.

PREDEFINED_ATTRIBUTE_ERROR
is raised if any attribute name given by the ATTRIBUTES parameter is the name of a predefined attribute that cannot be created by the user.

USE_ERROR is raised if the value for the DISCRETIONARY_ACCESS or MANDATORY_ACCESS parameter is semantically illegal.

STATUS_ERROR
is raised if QUEUE_BASE is not an open node handle or if QUEUE_NODE is an open node handle at the time of the call.

Downloaded from http://www.everyspec.com

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

Downloaded from http://www.everyspec.com

5.3.7.7 Opening a queue file node handle

A queue file node handle is opened by calling OPEN (see Section 5.1.2.1, page 63).

5.3.7.8 Closing a queue file node handle

A queue file node handle is closed by calling CLOSE (see Section 5.1.2.2, page 66). If the queue file node handle that is being closed is associated with a mimic queue node and if there is an open node handle to the coupled file node and if there is an open file handle associated with the coupled file node handle, then closing the queue file node handle also closes the open coupled node handle and the open coupled file handle.

5.3.7.9 Opening a queue file handle

A queue file handle is opened by calling the OPEN interface of CAIS_SEQUENTIAL_IO (see Section 5.3.5.3, page 240) or by calling the OPEN interface of CAIS_TEXT_IO (see Section 5.3.6.3, page 249) with an open queue file node handle as parameter. In addition to the exceptions raised by these OPEN interfaces, SECURITY_VIOLATION may be raised if the operation represents a violation of mandatory access controls. If the value of the queue file node attribute QUEUE_KIND is NONSYNCHRONOUS_MIMIC and the mode is OUT_FILE or APPEND_FILE, then the call on the OPEN interface must achieve the additional semantic effect of the following code fragment.

```
procedure OPEN (FILE: in out FILE_TYPE;
                NODE: in     NODE_TYPE;
                MODE: in     FILE_MODE)
is
    COUPLED_NODE: NODE_TYPE;
    COUPLED_FILE: FILE_TYPE;
    .
    .
    .
begin
    .
    .
    .
    CAIS_NODE_MANAGEMENT.OPEN (COUPLED_NODE, NODE, "",
                               "MIMIC_FILE", (1=>APPEND_CONTENTS));
    begin
        OPEN (COUPLED_FILE, COUPLED_NODE, APPEND_FILE);
    exception
        when others =>
            CLOSE (COUPLED_FILE);
            raise;
    end;
    .
    .
    .
end OPEN;
```

Downloaded from http://www.everyspec.com

5.3.7.10 Closing a queue file handle

A queue file handle is closed by calling the CLOSE interface of CAIS_SEQUENTIAL_IO (see Section 5.3.5.4, page 241) or by calling the CLOSE interface of CAIS_TEXT_IO (see Section 5.3.6.4, page 250). If the value of the attribute QUEUE_KIND of the queue file node associated with the queue file handle is NONSYNCHRONOUS_MIMIC and the mode of the queue file handle is OUT_FILE or APPEND_FILE, then the coupled file handle and the coupled file node handle opened by the OPEN call described in Section 5.3.7.9 are implicitly closed.

5.3.7.11 Reading elements from a queue file

The READ procedure in CAIS_SEQUENTIAL_IO and the GET procedures in CAIS_TEXT_IO (including GET_LINE) are used to read elements from a queue file.

Exception:

END_ERROR is raised if no more elements can be read from the given queue file and no process has the associated queue node open with the intent to write contents.

5.3.7.12 Writing elements to a queue file

The WRITE procedure in CAIS_SEQUENTIAL_IO is used to append elements to a queue file.

The PUT, PUT_LINE, NEW_LINE and NEW_PAGE procedures in CAIS_TEXT_IO are used to append characters, line terminators and page terminators to a queue file.

Exceptions:

CAPACITY_ERROR
is raised if the maximum queue size of a nonsynchronous queue is exceeded by the respective operation.

5.3.7.13 Resetting a queue file handle

The RESET procedures in CAIS_SEQUENTIAL_IO and CAIS_TEXT_IO are used to reset a queue file handle. Resetting a queue file handle has no effect other than changing the mode of the file handle.

Exceptions:

ACCESS_VIOLATION
is raised if the current process does not have sufficient access rights to append to the contents of the coupled file node pointed to by the relationship of the predefined relation MIMIC_FILE when the mimic queue file handle is reset to mode OUT_FILE or APPEND_FILE.

NAME_ERROR is raised if the coupled file node that is associated with a mimic queue file handle reset to mode OUT_FILE or APPEND_FILE is unobtainable or inaccessible.

Downloaded from http://www.everyspec.com

5.3.7.14 Determining end of file of a queue file

The END_OF_FILE functions in CAIS_SEQUENTIAL_IO and CAIS_TEXT_IO are used to determine the end of file of a queue file.

Purpose:

For the package CAIS_SEQUENTIAL_IO, this function returns TRUE if no more elements can be read from the given queue file and no process has the associated queue node open with intent to write contents; otherwise, it returns FALSE.

For the package CAIS_TEXT_IO, this function returns TRUE if no more characters, line terminators or page terminators can be read from the given queue file and no process has the associated queue node open with intent to write contents; otherwise, it returns FALSE.

Notes:

The value returned by this function varies depending upon the reading and writing activity upon the contents of the queue file node. Programs should not be written to depend on the value of END_OF_FILE for a file handle being constant once TRUE has been returned.

Downloaded from http://www.everyspec.com

5.3.8 Package CAIS_SCROLL_TERMINAL_IO

This package provides subprograms for communicating with a scroll terminal. The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_IO_DEFINITIONS.

The functionality of this package is built upon a subset of the operations defined in [ANSI 79]. The physical devices with which this package is intended to be used are typified by interactive printing terminals (i.e., interactive terminals that have a keyboard for sending characters to the program using the CAIS_SCROLL_TERMINAL_IO package and a printer for displaying the output from the CAIS_SCROLL_TERMINAL_IO package). The display device may also be what is commonly referred to as a "glass TTY".

Communication using the CAIS_SCROLL_TERMINAL_IO package consists of reading characters and/or function keys from the scroll terminal, modifying the scroll terminal display, and querying characteristics of the scroll terminal.

Data read from a scroll terminal are either Ada characters or function key identification numbers. These data are read using the GET functions. The data returned from a GET operation consist of a string of Ada characters and, optionally, a list of function key identification numbers. No function key identification numbers are returned if the function keys have been disabled. Instead, each function key identification number is translated into an implementation-dependent sequence of Ada characters that are returned in GET operations. The number of function key identification numbers and the string representation of the names of the function key identification numbers are implementation (and scroll terminal) dependent.

The display device for a scroll 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 positive row number and a positive column number. A display device for a scroll terminal has a fixed number of columns and rows. The rows are incrementally indexed starting with one after performing the NEW_PAGE (see Section 5.3.8.29, page 314) operation. The columns are incrementally indexed starting with one at the left side of the output device.

The *active position* on the output device of a scroll terminal is the position at which the next operation will be performed. The active position is said to *advance* if (1) the row number of the new position is greater than the row number of the old position or (2) the row number of the new position is the same as the row number of the old position and the new position has a greater column number. Similarly, a position is said to *precede* the active position if (1) the row number of the position is less than the row number of the active position or (2) the row number of the position is the same as the row number of the active position and the column number of the position is smaller than the column number of the active position.

When accessing a particular scroll terminal it is important to know several aspects of the terminal. Some of the information about the terminal that can be obtained by using subprograms in this package are the characters from the Ada character set that cannot be read from or written to the terminal (see INTERCEPTED_INPUT_CHARACTERS, Section 5.3.8.6, page 291 and INTERCEPTED_OUTPUT_CHARACTERS, Section 5.3.8.7, page 292), the size of the terminal (see PAGE_SIZE, Section 5.3.8.12, page 297), and the number

Downloaded from http://www.everyspec.com

of function keys for the terminal (see FUNCTION_KEY_COUNT, Section 5.3.8.23, page 308).

For all write operations in the CAIS the condition(s) upon which data are transferred from an internal file to the contents of a terminal file node are implementation-dependent. Data in the internal file of a process are inaccessible to other processes. *Synchronization* of a scroll terminal file handle forces all data written to the internal file identified by the file handle to be transmitted to the contents of the file node with which it is associated. Synchronization ensures that the data in the internal file and the data in the contents of the file node coincide.

Downloaded from http://www.everyspec.com

5.3.8.1 Types and subtypes

```
type FILE_TYPE is limited private;

type FILE_MODE is (IN_FILE, OUT_FILE, INOUT_FILE);
```

FILE_TYPE describes the type for file handles. FILE_MODE describes whether a file handle is to be used for input, output, or both.

```
type CHARACTER_ARRAY is array (CHARACTER) of BOOLEAN;

type FUNCTION_KEY_DESCRIPTOR is limited private;

subtype FUNCTION_KEY_NAME is STRING;
```

CHARACTER_ARRAY is used to determine the characters that are intercepted due to the characteristics of the underlying system and the individual terminal. FUNCTION_KEY_DESCRIPTOR is used to obtain information about function keys read from a terminal. FUNCTION_KEY_NAME is used to identify function keys by string representations.

```
type TERMINAL_POSITION_TYPE is
     record
          ROW:    CAIS_POSITIVE;
          COLUMN: CAIS_POSITIVE;
     end record;

type TAB_STOP_KIND is (HORIZONTAL, VERTICAL);
```

TERMINAL_POSITION_TYPE describes the type for a position on a terminal. TAB_STOP_KIND is used to specify the kind of tab stop to be set or cleared.

Downloaded from http://www.everyspec.com

5.3.8.2 Opening a scroll terminal file handle

```
procedure OPEN (TERMINAL: in out FILE_TYPE;
                NODE:     in     NODE_TYPE;
                MODE:     in     FILE_MODE);
```

Purpose:

This procedure returns an open file handle in TERMINAL to the file identified by the open node handle NODE.

Parameters:

TERMINAL is a file handle, initially closed, to be opened.

NODE is an open node handle to the file node.

MODE indicates the mode under which the file handle is opened.

Exceptions:

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
is raised if the values of the predefined file node attributes FILE_KIND, ACCESS_METHOD and DEVICE_KIND are not appropriate for the package containing this procedure according to Table XI, page 210.

STATUS_ERROR
is raised if the file handle TERMINAL is open at the time of the call or if the node handle NODE is not open.

USE_ERROR is raised if an open file handle identifies the same file node contents and the CAIS implementation does not support the existence of multiple file handles identifying the same file node contents. Any such restriction must be documented in Appendix F. An implementation is allowed to raise this exception only if it is based on operating system support that does not provide this capability.

INTENT_VIOLATION
is raised if NODE was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

Notes:

Closing the node handle associated with the file handle TERMINAL closes the file handle.

Downloaded from http://www.everyspec.com

5.3.8.3 Closing a scroll terminal file handle

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

Purpose:

This procedure severs any association between the internal file identified by the file handle TERMINAL and its associated node contents. It also severs any association between the file handle TERMINAL and its associated node handle. Closing an already closed file handle has no effect.

Parameter:

TERMINAL is a file handle to be closed.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.8.4 Determining whether a file handle is open

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

Purpose:

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

Parameter:

TERMINAL   is a file handle.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.8.5 Determining the number of function keys

```
function NUMBER_OF_FUNCTION_KEYS (TERMINAL: in FILE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns the number of function keys defined for the terminal associated with the internal file identifiedy by the file handle TERMINAL.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

Downloaded from http://www.everyspec.com

5.3.8.6 Determining intercepted input characters

```
function INTERCEPTED_INPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
```

Purpose:

This function returns an array of type CHARACTER_ARRAY that indicates the input characters that can never appear in the terminal file identified by TERMINAL due to characteristics of the underlying system and the individual terminal for the mode under which the file handle TERMINAL was opened. A value of FALSE indicates that the input character can appear; a value of TRUE indicates that it cannot appear.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

Notes:

The input characters intercepted by an underlying system or an individual terminal may differ with the mode (IN_FILE or INOUT_FILE) in which the file handle TERMINAL is being accessed. The input characters being intercepted may also be affected by whether or not function keys are enabled (see ENABLE_FUNCTION_KEYS, Section 5.3.8.8, page 293).

Downloaded from http://www.everyspec.com

5.3.8.7 Determining intercepted output characters

```
function INTERCEPTED_OUTPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
```

Purpose:

This function returns an array of type CHARACTER_ARRAY that indicates the output characters that can never appear in the terminal file identified by TERMINAL due to characteristics of the underlying system and the individual terminal for the mode under which the file handle TERMINAL was opened. A value of FALSE indicates that the output character can appear; a value of TRUE indicates that it cannot appear.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exception:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Notes:

The output characters intercepted by an underlying system or an individual terminal may differ with the mode (OUT_FILE or INOUT_FILE) in which the file handle TERMINAL is being accessed.

Downloaded from http://www.everyspec.com

5.3.8.8 Enabling and disabling function key usage

```
procedure ENABLE_FUNCTION_KEYS (TERMINAL: in FILE_TYPE;
                                ENABLE:   in BOOLEAN);
```

Purpose:

This procedure establishes whether function keys are read as a sequence of CHARACTERs or as a function key number. A value of TRUE for ENABLE designates that function keys should be read as function key numbers. A value of FALSE for ENABLE designates that function keys should be read as CHARACTERs.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

ENABLE indicates how function keys are to appear.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

Notes:

The characters being intercepted may also be affected by whether or not function keys are enabled. Intercepted characters can be part of a function key sequence when function keys are enabled.

Downloaded from http://www.everyspec.com

5.3.8.9 Determining function key usage

```
function FUNCTION_KEYS_ARE_ENABLED (TERMINAL: in FILE_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if the function keys are enabled; otherwise, it returns FALSE.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

Downloaded from http://www.everyspec.com

5.3.8.10 Setting the active position

```
procedure SET_ACTIVE_POSITION (TERMINAL: in FILE_TYPE;
                               POSITION: in TERMINAL_POSITION_TYPE);
```

Purpose:

This procedure advances the active position to the specified POSITION on the internal file identified by the output terminal file handle TERMINAL.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

POSITION is the new active position in the output terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

TERMINAL_POSITION_ERROR
is raised if POSITION does not exist on the terminal or POSITION precedes the active position.

Downloaded from http://www.everyspec.com

5.3.8.11 Determining the active position

```
function ACTIVE_POSITION (TERMINAL: in FILE_TYPE)
    return TERMINAL_POSITION_TYPE;
```

Purpose:

This function returns the active position of the internal file identified by the output terminal file handle TERMINAL.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.8.12 Determining the size of the terminal

```
function PAGE_SIZE (TERMINAL: in FILE_TYPE)
    return TERMINAL_POSITION_TYPE;
```

Purpose:

This function returns the maximum row and maximum column of the internal file identified by the output terminal file handle TERMINAL.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.8.13 Setting a tab stop

```
procedure SET_TAB_STOP (TERMINAL: in FILE_TYPE;
                        KIND:     in TAB_STOP_KIND := HORIZONTAL);
```

Purpose:

This procedure establishes a horizontal tab stop at the column of the active position if KIND is HORIZONTAL or a vertical tab stop at the row of the active position if KIND is VERTICAL.

Parameters:

TERMINAL	is an open file handle identifying the internal file associated with the terminal file.
KIND	is the kind of tab stop to be set.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.8.14 Clearing a tab stop

```
procedure CLEAR_TAB_STOP (TERMINAL: in FILE_TYPE;
                          KIND:     in TAB_STOP_KIND := HORIZONTAL);
```

Purpose:

This procedure removes a horizontal tab stop from the column of the active position if KIND is HORIZONTAL or a vertical tab stop from the row of the active position if KIND is VERTICAL. Removing a tab stop from a position that does not have a tab stop has no effect.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

KIND is the kind of tab stop to be removed.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.8.15 Clearing all tab stops

```
procedure CLEAR_ALL_TAB_STOPS (TERMINAL: in FILE_TYPE;
                               KIND:     in TAB_STOP_KIND := HORIZONTAL);
```

Purpose:

This procedure removes all horizontal tab stops if KIND is HORIZONTAL or all vertical tab stops if KIND is VERTICAL. Removing a tab stop from a position that does not have a tab stop has no effect.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

KIND is the kind of tab stops to be removed.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.8.16 Advancing to the next tab position

```
procedure TAB (TERMINAL: in FILE_TYPE;
               COUNT:    in CAIS_POSITIVE := 1;
               KIND:     in TAB_STOP_KIND := HORIZONTAL);
```

Purpose:

This procedure advances the active position COUNT tab stops. Horizontal advancement causes a change in only the column number of the active position. Vertical advancement causes a change in only the row number of the active position.

If there are fewer than COUNT tab stops following the active position, the active position is advanced to the column of the maximum column (HORIZONTAL) or to the row of the maximum row (VERTICAL).

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

COUNT is the number of tab stops the active position is to advance.

KIND is the kind of tab stop to which the active position will be advanced.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.8.17 Sounding a terminal bell

```
procedure SOUND_BELL (TERMINAL: in FILE_TYPE);
```

Purpose:

This procedure sounds the bell (beeper) on the internal file identified by the output terminal file TERMINAL.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.8.18 Writing to the terminal

```
procedure PUT (TERMINAL: in FILE_TYPE;
               ITEM:     in CHARACTER);
```

Purpose:

This procedure writes a single character to the internal file identified by the output file handle TERMINAL and advances the active position by one column. After a character is written in the maximum column of a row, the active position is the first column of the next row. After a character is written in the maximum column of the maximum row, the active position is the first column of a new page.

Parameters:

TERMINAL — is an open file handle identifying the internal file associated with the terminal file.

ITEM — is the character to be written.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Additional Interfaces:

```
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;
```

Notes:

Positioning to a new page constitutes advancing the active position an implementation-dependent number of rows.

Downloaded from http://www.everyspec.com

5.3.8.19 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 a single function key identification number into KEYS from the internal file identified by the input file handle TERMINAL. If no character is available at the time of the call the interface does not complete until a character becomes available.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

ITEM is the character that was read.

KEYS is the description of the function key that was read.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

FUNCTION_KEY_STATUS_ERROR
is raised if KEYS has not been previously created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.8.21, page 306) and the value of FUNCTION_KEYS_ARE_ENABLED (see Section 5.3.8.9, page 294) is TRUE.

Notes:

This procedure will only return function key identification numbers in KEYS if function keys have been enabled (see Section 5.3.8.9, page 294). Otherwise the characters in the ASCII character sequence representing the function key will appear one at a time in ITEM. Use FUNCTION_KEY_COUNT (see section 5.3.8.23, page 308) to determine whether a character or function key was read.

Downloaded from http://www.everyspec.com

5.3.8.20 Reading all available characters from a terminal

```
procedure GET (TERMINAL: in     FILE_TYPE;
               ITEM:        out STRING;
               LAST:        out CAIS_NATURAL;
               KEYS:     in out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:

This procedure successively reads characters and function key identification numbers into ITEM and KEYS, respectively, until either all positions of ITEM or KEYS are filled or there are no more characters available in the internal file identified by the input file handle TERMINAL. Upon completion, LAST contains the index of the last position in ITEM to contain a character that has been read. If there are no elements available for reading from the internal file, then LAST has a value one less than ITEM'FIRST and FUNCTION_KEY_COUNT(KEYS) (see section 5.3.8.23, page 308) is equal to zero.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

ITEM is the string of characters that were read.

LAST is the position of the last character read in ITEM.

KEYS is a description of the function keys that were read.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

FUNCTION_KEY_STATUS_ERROR
is raised if KEYS has not been previously created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.8.21, page 306) and the value of FUNCTION_KEYS_ARE_ENABLED (see Section 5.3.8.9, page 294) is TRUE.

Notes:

This procedure will only return function key identification numbers in KEYS if function keys have been enabled (see the interface FUNCTION_KEYS_ARE_ENABLED, Section 5.3.8.9, page 294). Otherwise, the characters in the ASCII character sequence representing the function key will appear in ITEM.

Downloaded from http://www.everyspec.com

5.3.8.21 Creating a function key descriptor

```
procedure CREATE_FUNCTION_KEY_DESCRIPTOR
                    (KEYS:          in out FUNCTION_KEY_DESCRIPTOR;
                     MAXIMUM_COUNT: in     CAIS_POSITIVE);
```

Purpose:

This procedure establishes a function key descriptor KEYS with capacity for MAXIMUM_COUNT function key descriptions.

Parameters:

KEYS is the function key descriptor returned.

MAXIMUM_COUNT
is the maximum number of function key descriptions that may be read into KEYS.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.8.22 Deleting a function key descriptor

```
procedure DELETE_FUNCTION_KEY_DESCRIPTOR
                (KEYS: in out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:

This procedure deletes a function key descriptor. The value of its parameter after the call is as if it were never created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.8.21, page 306). Deleting a function key descriptor that has already been deleted or that has never been created has no effect.

Parameter:

KEYS is a function key descriptor.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.8.23 Determining the number of function keys that were read

```
function FUNCTION_KEY_COUNT (KEYS: in FUNCTION_KEY_DESCRIPTOR)
    return CAIS_NATURAL;
```

Purpose:

This function returns the number of function keys described in KEYS.

Parameter:

KEYS — is the function key descriptor being queried.

Exception:

FUNCTION_KEY_STATUS_ERROR
is raised if KEYS has not been previously created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.8.21, page 306).

Downloaded from http://www.everyspec.com

5.3.8.24 Determining function key usage

```
procedure GET_FUNCTION_KEY
           (KEYS:           in    FUNCTION_KEY_DESCRIPTOR;
            INDEX:          in    CAIS_POSITIVE;
            KEY_IDENTIFIER:    out CAIS_POSITIVE;
            POSITION:          out CAIS_NATURAL);
```

Purpose:

This procedure returns the identification number of a function key. If KEYS was obtained by GET (see Section 5.3.8.20, page 305) this procedure returns the position in the string (read at the same time as the function keys) of the character following the function key. If KEYS was obtained by GET (see Section 5.3.8.19, page 304) this procedure sets POSITION to zero.

Parameters:

KEYS is the description of the function keys that were read.

INDEX is the index in KEYS of the function key to be queried.

KEY_IDENTIFIER
is the identification number of a function key.

POSITION is the position of the character read after the function key.

Exceptions:

FUNCTION_KEY_STATUS_ERROR
is raised if KEYS has not been previously created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.8.21, page 306).

CONSTRAINT_ERROR
is raised if INDEX is greater than FUNCTION_KEY_COUNT(KEYS).

Notes:

See FUNCTION_KEY_IDENTIFICATION, Section 5.3.8.25, page 310, to get a string representation of the function key identification number returned in KEY_IDENTIFIER.

Downloaded from http://www.everyspec.com

5.3.8.25 Determining the identification of a function key

```
function FUNCTION_KEY_IDENTIFICATION
                        (TERMINAL:       in FILE_TYPE;
                         KEY_IDENTIFIER: in CAIS_POSITIVE)
    return FUNCTION_KEY_NAME;
```

Purpose:

This function returns the string identification of the function key designated by KEY_IDENTIFIER.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

KEY_IDENTIFIER
is the identification number of a function key.

Exception:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

FUNCTION_KEY_STATUS_ERROR
is raised if the value of KEY_IDENTIFIER is greater than NUMBER_OF_FUNCTION_KEYS(TERMINAL).

Notes:

Function key names are implementation-dependent.

Downloaded from http://www.everyspec.com

5.3.8.26 Determining the mode of a terminal

```
function MODE (TERMINAL: in FILE_TYPE)
    return FILE_MODE;
```

Purpose:

This function returns the mode under which the file handle TERMINAL is opened.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exception:

STATUS_ERROR is raised if the file handle TERMINAL is not open.

Downloaded from http://www.everyspec.com

5.3.8.27 Backspacing the active position

```
procedure BACKSPACE (TERMINAL: in FILE_TYPE;
                     COUNT:    in CAIS_POSITIVE := 1);
```

Purpose:

This procedure sets the active position to the column COUNT columns toward the beginning of the active row. If COUNT is greater than or equal to the column number of the active position, the active position is set to the first column.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

COUNT is the number of columns to backspace.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Notes:

The CAIS does not define for a scroll terminal the results of writing a character at a position where a character has already been written. It may be replaced or overstruck.

Downloaded from http://www.everyspec.com

5.3.8.28 Advancing the active position to the next line

```
procedure NEW_LINE (TERMINAL: in FILE_TYPE;
                    COUNT:    in CAIS_POSITIVE := 1);
```

Purpose:

This procedure advances the active position in the internal file identified by the output terminal file handle TERMINAL to column one, COUNT rows after the active position.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

COUNT is the number of rows to advance.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Notes:

The next row after the maximum row of a page is the first row of a new page.

Downloaded from http://www.everyspec.com

5.3.8.29 Advancing the active position to the next page

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

Purpose:

This procedure advances the active position in the internal file identified by the output terminal file handle TERMINAL to the first column of the first row of a new page.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Notes:

Positioning to a new page constitutes advancing the active position an implementation-dependent number of rows.

Downloaded from http://www.everyspec.com

5.3.8.30 Resetting a scroll terminal file handle

```
procedure RESET (TERMINAL: in out FILE_TYPE;
                 MODE:     in     FILE_MODE);
```

Purpose:

This procedure sets the current mode of the file handle TERMINAL to the mode given by the MODE parameter.

Parameters:

TERMINAL is an open file handle identifying the internal file to be reset.

MODE indicates the new mode under which the file handle is to be reset.

Exceptions:

STATUS_ERROR
is raised if TERMINAL is not an open file handle.

INTENT_VIOLATION
is raised if the file node handle associated with the file handle TERMINAL was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

USE_ERROR is raised if the CAIS implementation does not support resetting the file handle to the specified mode.

Downloaded from http://www.everyspec.com

5.3.8.31 Synchronizing the internal file with file node contents

```
procedure SYNCHRONIZE (TERMINAL: in FILE_TYPE);
```

Purpose:

This procedure forces all data written to the internal file identified by TERMINAL to be transmitted to the contents of the file node with which it is associated.

Parameter:

TERMINAL is an open file handle identifying the internal file to be synchronized.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.8.32 Setting terminal file handle synchronization

```
procedure ENABLE_SYNCHRONIZATION (TERMINAL: in FILE_TYPE;
                                  ENABLE:   in BOOLEAN);
```

Purpose:

This procedure establishes operations on the file handle TERMINAL to be synchronized if ENABLE is TRUE; otherwise, synchronization is implementation-dependent.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

ENABLE indicates whether or not the file handle is to be enabled for synchronization.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file identified by TERMINAL is of mode IN_FILE.

Notes:

When SYNCHRONIZATION_IS_ENABLED (see Section 5.3.8.33, page 318) returns FALSE for a file handle, the effect of synchronization for the file handle can be achieved by (1) preceding each read operation on the file handle immediately by a call to SYNCHRONIZE (see Section 5.3.8.31, page 316) on the file handle and (2) following each write operation on the file handle immediately by a call to SYNCHRONIZE on the file handle.

Downloaded from http://www.everyspec.com

5.3.8.33 Determining the synchronization of a terminal file handle

```
function SYNCHRONIZATION_IS_ENABLED (TERMINAL: in FILE_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if the file handle is enabled for synchronization; otherwise, it returns FALSE.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file identified by TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9 Package CAIS_PAGE_TERMINAL_IO

This package provides subprograms for communicating with a page terminal. The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_IO_DEFINITIONS.

The functionality of this package is built upon a subset of the operations defined in [ANSI 79]. The physical devices with which this package is intended to be used are typified by interactive video display terminals (i.e., interactive terminals that have a keyboard for sending characters to the program using the CAIS_PAGE_TERMINAL_IO package and a video screen for displaying the output from the CAIS_PAGE_TERMINAL_IO package).

Communication using the CAIS_PAGE_TERMINAL_IO package consists of reading characters and/or function keys from the page terminal, modifying the page terminal display, and querying characteristics of the page terminal.

Data read from a page terminal are either Ada characters or function key identification numbers. These data are read using the GET functions. The data returned from a GET operation consist of a string of Ada characters and, optionally, a list of function key identification numbers. No function key identification numbers are returned if the function keys have been disabled. Instead, each function key identification number is translated into an implementation-dependent sequence of Ada characters that are returned in GET operations. The number of function key identification numbers and the string representation of the names of the function key identification numbers are implementation (and page terminal) dependent.

The display device 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 positive row number and a positive column number. A display device for a page terminal has a fixed number of columns and rows. The rows are incrementally indexed starting with one at the top of the output device. The columns are incrementally indexed starting with one at the left side of the output device.

The *active position* on the output device of a page terminal is the position at which the next operation will be performed. The active position is said to *advance* if (1) the row number of the new position is greater than the row number of the old position or (2) the row number of the new position is the same as the row number of the old position and the new position has a greater column number. Similarly, a position is said to *precede* the active position if (1) the row number of the position is less than the row number of the active position or (2) the row number of the position is the same as the row number of the active position and the column number of the position is smaller than the column number of the active position.

When accessing a particular page terminal it is important to know several aspects of the terminal. Some of the information about the terminal that can be obtained by using subprograms in this package are the characters from the Ada character set that cannot be read from or written to the terminal (see INTERCEPTED_INPUT_CHARACTERS, Section 5.3.9.6, page 327) and INTERCEPTED_OUTPUT_CHARACTERS, Section 5.3.9.7, page 328), the size of the terminal (see PAGE_SIZE, Section 5.3.9.12, page 333), the number of function keys for the terminal (see FUNCTION_KEY_COUNT, Section 5.3.9.23, page 344),

Downloaded from http://www.everyspec.com

the characteristics of writing a character into the last position on the output device (see END_POSITION_SUPPORT, Section 5.3.9.36, page 357), and the graphic renditions supported by the individual terminal (see GRAPHIC_RENDITION_IS_SUPPORTED, Section 5.3.9.34, page 355).

For all write operations in the CAIS the condition(s) upon which data are transferred from an internal file to the contents of a terminal file node are implementation-dependent. Data in the internal file of a process are inaccessible to other processes. *Synchronization* of a page terminal file handle forces all data written to the internal file identified by the file handle to be transmitted to the contents of the file node with which it is associated. Synchronization ensures that the data in the internal file and the data in the contents of the file node coincide.

Downloaded from http://www.everyspec.com

5.3.9.1 Types, subtypes and constants

```
type FILE_TYPE is limited private;

type FILE_MODE is (IN_FILE, OUT_FILE, INOUT_FILE);
```

FILE_TYPE describes the type for file handles. FILE_MODE describes whether a file handle is to be used for input, output, or both.

```
type CHARACTER_ARRAY is array (CHARACTER) of BOOLEAN;

type FUNCTION_KEY_DESCRIPTOR is limited private;

subtype FUNCTION_KEY_NAME is STRING;
```

CHARACTER_ARRAY is used to determine the characters that are intercepted due to the characteristics of the underlying system and the individual terminal. FUNCTION_KEY_DESCRIPTOR is used to obtain information about function keys read from a terminal. FUNCTION_KEY_NAME is used to identify function keys by string representations.

```
type TERMINAL_POSITION_TYPE is
    record
        ROW:     CAIS_POSITIVE;
        COLUMN:  CAIS_POSITIVE;
    end record;

type TAB_STOP_KIND is (HORIZONTAL, VERTICAL);

type SELECT_RANGE_KIND is
    (FROM_ACTIVE_POSITION_TO_END,
     FROM_START_TO_ACTIVE_POSITION,
     ALL_POSITIONS);

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

type GRAPHIC_RENDITION_ARRAY is array (GRAPHIC_RENDITION_KIND)
    of BOOLEAN;
```

TERMINAL_POSITION_TYPE describes the type for a position on a terminal. TAB_STOP_KIND is used to specify the kind of tab stop to be set. SELECT_RANGE_KIND is used in ERASE_IN_DISPLAY (see Section 5.3.9.30, page 351) and ERASE_IN_LINE (see Section 5.3.9.31, page 352) to determine the portion of the display or line to be erased. GRAPHIC_RENDITION_KIND and GRAPHIC_RENDITION_ARRAY are used to determine display characteristics of printable characters.

Downloaded from http://www.everyspec.com


```
DEFAULT_GRAPHIC_RENDITION: constant GRAPHIC_RENDITION_ARRAY :=
        (PRIMARY_RENDITION => TRUE, BOLD..REVERSE_IMAGE => FALSE);
```

DEFAULT_GRAPHIC_RENDITION is a constant used to determine display characteristics of printable characters.

Downloaded from http://www.everyspec.com

5.3.9.2 Opening a page terminal file handle

```
procedure OPEN (TERMINAL: in out FILE_TYPE;
                NODE:     in     NODE_TYPE;
                MODE:     in     FILE_MODE);
```

Purpose:

This procedure returns an open file handle in TERMINAL to the node identified by the the node handle NODE.

Parameters:

TERMINAL — is a file handle, initially closed, to be opened.

NODE — is an open node handle to the file node.

MODE — indicates the mode under which the file handle is opened.

Exceptions:

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
is raised if the values of the predefined file node attributes FILE_KIND, ACCESS_METHOD and DEVICE_KIND are not appropriate for the package containing this procedure according to Table XI, page 210.

STATUS_ERROR
is raised if the file handle TERMINAL is open at the time of the call or if the node handle NODE is not open.

USE_ERROR
is raised if an open file handle identifies the same file node contents and the CAIS implementation does not support the existence of multiple file handles identifying the same file node contents. Any such restriction must be documented in Appendix F. An implementation is allowed to raise this exception only if it is based on operating system support that does not provide this capability.

INTENT_VIOLATION
is raised if NODE was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

Notes:

Closing the node handle associated with the file handle TERMINAL closes the file handle.

Downloaded from http://www.everyspec.com

5.3.9.3 Closing a page terminal file handle

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

Purpose:

This procedure severs any association between the internal file identified by the file handle TERMINAL and its associated node contents. It also severs any association between the file handle TERMINAL and its associated node handle. Closing an already closed file handle has no effect.

Parameter:

TERMINAL   is a file handle to be closed.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.9.4 Determining whether a file handle is open

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

Purpose:

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

Parameter:

TERMINAL is a file handle.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.9.5 Determining the number of function keys

```
function NUMBER_OF_FUNCTION_KEYS (TERMINAL: in FILE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns the number of function keys defined for the terminal associated with the internal file identified by the file handle TERMINAL.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

Downloaded from http://www.everyspec.com

5.3.9.6 Determining intercepted input characters

```
function INTERCEPTED_INPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
```

Purpose:

This function returns an array of type CHARACTER_ARRAY that indicates the input characters that can never appear in the terminal file identified by TERMINAL due to characteristics of the underlying system and the individual terminal for the mode under which the file handle TERMINAL was opened. A value of FALSE indicates that the input character can appear; a value of TRUE indicates that it cannot appear.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

Notes:

The input characters intercepted by an underlying system or an individual terminal may differ with the mode (IN_FILE or INOUT_FILE) in which the file handle TERMINAL is being accessed. The input characters being intercepted may also be affected by whether or not function keys are enabled (see ENABLE_FUNCTION_KEYS, Section 5.3.9.8, page 329).

Downloaded from http://www.everyspec.com

5.3.9.7 Determining intercepted output characters

```
function INTERCEPTED_OUTPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
```

Purpose:

This function returns an array of type CHARACTER_ARRAY that indicates the output characters that can never appear in the terminal file identified by TERMINAL due to characteristics of the underlying system and the individual terminal for the mode under which the file handle TERMINAL was opened. A value of FALSE indicates that the output character can appear; a value of TRUE indicates that it cannot appear.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exception:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Notes:

The output characters intercepted by an underlying system or an individual terminal may differ with the mode (OUT_FILE or INOUT_FILE) in which the file handle TERMINAL is being accessed.

Downloaded from http://www.everyspec.com

5.3.9.8 Enabling and disabling function key usage

```
procedure ENABLE_FUNCTION_KEYS (TERMINAL: in FILE_TYPE;
                                ENABLE:   in BOOLEAN);
```

Purpose:

This procedure establishes whether function keys are read as a sequence of CHARACTERs or as a function key number. A value of TRUE for ENABLE designates that function keys should be read as function key numbers. A value of FALSE for ENABLE designates that function keys should be read as CHARACTERs.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

ENABLE indicates how function keys are to appear.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

Notes:

The characters being intercepted are also affected by whether or not function keys are enabled. Intercepted characters can be part of a function key sequence when function keys are enabled.

Downloaded from http://www.everyspec.com

5.3.9.9 Determining function key usage

```
function FUNCTION_KEYS_ARE_ENABLED (TERMINAL: in FILE_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if the function keys are enabled; otherwise, it returns FALSE.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

Downloaded from http://www.everyspec.com

5.3.9.10 Setting the active position

```
procedure SET_ACTIVE_POSITION (TERMINAL: in FILE_TYPE;
                               POSITION: in TERMINAL_POSITION_TYPE);
```

Purpose:

This procedure advances the active position to the specified POSITION on the internal file identified by the output terminal file handle TERMINAL.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

POSITION is the new active position in the output terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

TERMINAL_POSITION_ERROR
is raised if POSITION does not exist on the terminal.

Downloaded from http://www.everyspec.com

5.3.9.11 Determining the active position

```
function ACTIVE_POSITION (TERMINAL: in FILE_TYPE)
    return TERMINAL_POSITION_TYPE;
```

Purpose:

This function returns the active position of the internal file identified by the output terminal file identified by TERMINAL.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.12 Determining the size of the terminal

```
function PAGE_SIZE (TERMINAL: in FILE_TYPE)
    return TERMINAL_POSITION_TYPE;
```

Purpose:

This function returns the maximum row and maximum column of the internal file identified by the output terminal file TERMINAL.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.13 Setting a tab stop

```
procedure SET_TAB_STOP (TERMINAL: in FILE_TYPE;
                        KIND:     in TAB_STOP_KIND := HORIZONTAL);
```

Purpose:

This procedure establishes a horizontal tab stop at the column of the active position if KIND is HORIZONTAL or a vertical tab stop at the row of the active position if KIND is VERTICAL.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

KIND is the kind of tab stop to be set.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.14 Clearing a tab stop

```
procedure CLEAR_TAB_STOP (TERMINAL: in FILE_TYPE;
                          KIND:     in TAB_STOP_KIND := HORIZONTAL);
```

Purpose:

This procedure removes a horizontal tab stop from the column of the active position if KIND is HORIZONTAL or a vertical tab stop from the row of the active position if KIND is VERTICAL. Removing a tab stop from a position that does not have a tab stop has no effect.

Parameters:

TERMINAL — is an open file handle identifying the internal file associated with the terminal file.

KIND — is the kind of tab stop to be removed.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.15 Clearing all tab stops

```
procedure CLEAR_ALL_TAB_STOPS (TERMINAL: in FILE_TYPE;
                               KIND:     in TAB_STOP_KIND := HORIZONTAL);
```

Purpose:

This procedure removes all horizontal tab stops if KIND is HORIZONTAL or all vertical tab stops if KIND is VERTICAL. Removing a tab stop from a position that does not have a tab stop has no effect.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

KIND is the kind of tab stops to be removed.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.16 Advancing to the next tab position

```
procedure TAB (TERMINAL: in FILE_TYPE;
               COUNT:    in CAIS_POSITIVE := 1;
               KIND:     in TAB_STOP_KIND := HORIZONTAL);
```

Purpose:

This procedure advances the active position COUNT tab stops. Horizontal advancement causes a change in only the column number of the active position. Vertical advancement causes a change in only the row number of the active position.

If there are fewer than COUNT tab stops following the active position the active position is advanced to the column of the maximum column (HORIZONTAL) or to the row of the maximum row (VERTICAL).

Parameters:

TERMINAL	is an open file handle identifying the internal file associated with the terminal file.
COUNT	is the number of tab stops the active position is to advance.
KIND	is the kind of tab stop to which the active position will be advanced.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.17 Sounding a terminal bell

```
procedure SOUND_BELL (TERMINAL: in FILE_TYPE);
```

Purpose:

This procedure sounds the bell (beeper) on the internal file identified by the output terminal file identified by TERMINAL.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.18 Writing to the terminal

```
procedure PUT (TERMINAL: in FILE_TYPE;
               ITEM:     in CHARACTER);
```

Purpose:

This procedure writes a single character to the internal file identified by the output file handle TERMINAL and advances the active position by one column. After a character is written in the maximum column of a row, the active position is the first column of the next row. The effect of writing to the last position of a page is terminal-dependent, but may be (partially) determined by using the function END_POSITION_SUPPORT (see Section 5.3.9.36, page 357).

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

ITEM is the character to be written.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Additional Interface:

```
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;
```

Downloaded from http://www.everyspec.com

5.3.9.19 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 a single function key identification number into KEYS from the internal file identified by the input file handle TERMINAL. If no character is available the interface does not complete until one becomes available.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

ITEM is the character that was read.

KEYS is the description of the function key that was read.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

FUNCTION_KEY_STATUS_ERROR
is raised if KEYS has not been previously created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.9.21, page 342) and the value of FUNCTION_KEYS_ARE_ENABLED (see Section 5.3.9.9, page 330) is TRUE.

Notes:

This procedure will only return function key identification numbers in KEYS if function keys have been enabled (see Section 5.3.9.9, page 330). Otherwise the characters in the ASCII character sequence representing the function key will appear one at a time in ITEM. Use FUNCTION_KEY_COUNT (see Section 5.3.9.23, page 344) to determine whether a character or function key was read.

Downloaded from http://www.everyspec.com

5.3.9.20 Reading all available characters from a terminal

```
procedure GET (TERMINAL: in     FILE_TYPE;
               ITEM:        out STRING;
               LAST:        out CAIS_NATURAL;
               KEYS:     in out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:

This procedure successively reads characters and function key identification numbers into ITEM and KEYS, respectively, until either all positions of ITEM or KEYS are filled or there are no more characters available in the internal file identified by the input file handle TERMINAL. Upon completion, LAST contains the index of the last position in ITEM to contain a character that has been read. If there are no elements available for reading from the input terminal file, then LAST has a value one less than ITEM'FIRST and FUNCTION_KEY_COUNT(KEYS) (see Section 5.3.9.23, page 344) is equal to zero.

Parameters:

TERMINAL	is an open file handle identifying the internal file associated with the terminal file.
ITEM	is the string of characters that were read.
LAST	is the position of the last character read in ITEM.
KEYS	is the description of the function keys that were read.

Exceptions:

STATUS_ERROR
: is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

FUNCTION_KEY_STATUS_ERROR
: is raised if KEYS has not been previously created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.9.21, page 342) and the value of FUNCTION_KEYS_ARE_ENABLED (see Section 5.3.9.9, page 330) is TRUE.

Notes:

This procedure will only return function key identification numbers in KEYS if function keys have been enabled (see Section 5.3.9.9, page 330). Otherwise, the characters in the ASCII character sequence representing the function key will appear in ITEM.

Downloaded from http://www.everyspec.com

5.3.9.21 Creating a function key descriptor

```
procedure CREATE_FUNCTION_KEY_DESCRIPTOR
                    (KEYS:          in out FUNCTION_KEY_DESCRIPTOR;
                     MAXIMUM_COUNT: in     CAIS_POSITIVE);
```

Purpose:

This procedure establishes a function key descriptor KEYS with capacity for MAXIMUM_COUNT function key descriptions.

Parameters:

KEYS — is the function key descriptor returned.

MAXIMUM_COUNT — is the maximum number of function key descriptions that may be read into KEYS.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.9.22 Deleting a function key descriptor

```
procedure DELETE_FUNCTION_KEY_DESCRIPTOR
              (KEYS: in out FUNCTION_KEY_DESCRIPTOR);
```

Purpose:

This procedure deletes a function key descriptor. The value of its parameter after the call is as if it were never created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.9.21, page 342). Deleting a function key descriptor that has already been deleted or that has never been created has no effect.

Parameter:

KEYS is a function key descriptor.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.9.23 Determining the number of function keys that were read

```
function FUNCTION_KEY_COUNT (KEYS: in FUNCTION_KEY_DESCRIPTOR)
    return CAIS_NATURAL;
```

Purpose:

This function returns the number of function keys described in KEYS.

Parameter:

KEYS is the function key descriptor being queried.

Exception:

FUNCTION_KEY_STATUS_ERROR
is raised if KEYS has not been previously created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.9.21, page 342).

Downloaded from http://www.everyspec.com

5.3.9.24 Determining function key usage

```
procedure GET_FUNCTION_KEY
          (KEYS:            in    FUNCTION_KEY_DESCRIPTOR;
           INDEX:           in    CAIS_POSITIVE;
           KEY_IDENTIFIER:  out   CAIS_POSITIVE;
           POSITION:        out   CAIS_NATURAL);
```

Purpose:

This procedure returns the identification number of a function key. If KEYS was obtained by GET (see Section 5.3.9.20, page 341), this procedure returns the position in the string (read at the same time as the function keys) of the character following the function key. If KEYS was obtained by GET (see Section 5.3.9.19, page 340), this procedure sets POSITION to zero.

Parameters:

KEYS is the description of the function keys that were read.

INDEX is the index in KEYS of the function key to be queried.

KEY_IDENTIFIER
is the identification number of a function key.

POSITION is the position of the character read after the function key.

Exceptions:

FUNCTION_KEY_STATUS_ERROR
is raised if KEYS has not been previously created by the procedure CREATE_FUNCTION_KEY_DESCRIPTOR (see Section 5.3.9.21, page 342).

CONSTRAINT_ERROR
is raised if INDEX is greater than FUNCTION_KEY_COUNT(KEYS).

Downloaded from http://www.everyspec.com

5.3.9.25 Determining the identification of a function key

```
function FUNCTION_KEY_IDENTIFICATION
                         (TERMINAL:       in FILE_TYPE;
                          KEY_IDENTIFIER: in CAIS_POSITIVE)
    return FUNCTION_KEY_NAME;
```

Purpose:

This function returns the string identification of the function key designated by KEY_IDENTIFIER.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

KEY_IDENTIFIER
is the identification number of a function key.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode OUT_FILE.

FUNCTION_KEY_STATUS_ERROR
is raised if the value of KEY_IDENTIFIER is greater than NUMBER_OF_FUNCTION_KEYS(TERMINAL).

Notes:

Function key names are implementation-dependent.

Downloaded from http://www.everyspec.com

5.3.9.26 Determining the mode of a terminal

```
function MODE (TERMINAL: in FILE_TYPE)
    return FILE_MODE;
```

Purpose:

This function returns the mode under which the file handle TERMINAL is opened.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exception:

STATUS_ERROR is raised if the file handle TERMINAL is not open.

Downloaded from http://www.everyspec.com

5.3.9.27 Deleting characters

```
procedure DELETE_CHARACTER (TERMINAL: in FILE_TYPE;
                            COUNT:    in CAIS_POSITIVE := 1);
```

Purpose:

This procedure deletes COUNT characters on the active row starting at the active position and advancing toward the maximum column. Adjacent characters following the deleted characters are shifted toward the active position. Open space at the end of the row is filled with space characters. The active position is not changed. If the value of COUNT is greater than the number of columns in the active line between the active position and the maximum column (inclusive), all positions on the active row from the active column to the maximum column (inclusive) are replaced with space characters.

Parameters:

TERMINAL	is an open file handle identifying the internal file associated with the terminal file.
COUNT	is the number of characters to be deleted.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.28 Deleting lines

```
procedure DELETE_LINE (TERMINAL: in FILE_TYPE;
                       COUNT:    in CAIS_POSITIVE := 1);
```

Purpose:

This procedure deletes COUNT lines starting at the active row and advancing toward the maximum row. Lines following the deleted lines are shifted toward the active position. Open space at the end of the page is filled with erased lines. The active position is not changed. If the value of COUNT is greater than the number of rows between the active row and the maximum row (inclusive), the rows from the active to the maximum row (inclusive) are replaced with erased lines.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

COUNT is the number of lines to be deleted.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.29 Replacing characters in a line with space characters

```
procedure ERASE_CHARACTER (TERMINAL: in FILE_TYPE;
                           COUNT:    in CAIS_POSITIVE := 1);
```

Purpose:

This procedure replaces COUNT characters on the active line with space characters starting at the active position and advancing toward the maximum column. The active position is not changed. If the value of COUNT is greater than the number of columns in the active line between the active position and the maximum column (inclusive), all positions on the active row from the active column to the maximum column (inclusive) are replaced with space characters.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

COUNT is the number of characters to be erased.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.30 Erasing characters in a display

```
procedure ERASE_IN_DISPLAY (TERMINAL:  in FILE_TYPE;
                            SELECTION: in SELECT_RANGE_KIND);
```

Purpose:

This procedure erases the characters in the display as determined by the active position and the given SELECTION (including the active position). After erasure, erased positions have space characters. The active position is not changed.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

SELECTION is the portion of the display to be erased.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.31 Erasing characters in a line

```
procedure ERASE_IN_LINE (TERMINAL:  in FILE_TYPE;
                         SELECTION: in SELECT_RANGE_KIND);
```

Purpose:

This procedure erases the characters in the active line as determined by the active position and the given SELECTION (including the active position). After erasure erased positions have space characters. The active position is not changed.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

SELECTION is the portion of the line to be erased.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.32 Inserting space characters in a line

```
procedure INSERT_SPACE (TERMINAL: in FILE_TYPE;
                        COUNT:    in CAIS_POSITIVE := 1);
```

Purpose:

This procedure inserts COUNT space characters into the active line at the active position. The character at the active position and following characters on the active row are shifted toward the maximum column. The COUNT last characters on the row are lost. The active position is not changed.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

COUNT is the number of space characters to be inserted.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.33 Inserting blank lines in the output terminal file

```
procedure INSERT_LINE (TERMINAL: in FILE_TYPE;
                       COUNT:    in CAIS_POSITIVE := 1);
```

Purpose:

This procedure inserts COUNT erased lines into the output terminal file at the active line. The lines at the active position and following rows are shifted toward the maximum row. The COUNT last lines of the display are lost. The active row is not changed. The active column is changed to one.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

COUNT is the number of blank lines to be inserted.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.34 Determining graphic rendition support

```
function GRAPHIC_RENDITION_IS_SUPPORTED
                        (TERMINAL:  in FILE_TYPE;
                         RENDITION: in GRAPHIC_RENDITION_ARRAY)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if the combined graphic renditions RENDITION are supported by the internal file identified by the output file handle TERMINAL; otherwise, it returns FALSE.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

RENDITION is a combination of graphic renditions.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.35 Selecting the graphic rendition

```
procedure SELECT_GRAPHIC_RENDITION
                    (TERMINAL:  in FILE_TYPE;
                     RENDITION: in GRAPHIC_RENDITION_ARRAY :=
                                   DEFAULT_GRAPHIC_RENDITION);
```

Purpose:

This procedure sets the graphic rendition for subsequent characters to be output to the internal file identified by the output file handle TERMINAL.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

RENDITION is the graphic rendition to be used in subsequent output operations.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.36 Determining the effect of writing to the end position

```
function END_POSITION_SUPPORT (TERMINAL: in FILE_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if, after writing a character to the position at the maximum row and maximum column of the internal file identified by the file handle TERMINAL, the only changes to the internal file are that (1) the character is graphically displayed in the maximum row and maximum column position and (2) the active position is set to the first row and first column of the terminal file; otherwise, it returns FALSE.

Parameter:

TERMINAL   is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
    is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.37 Resetting a page terminal file handle

```
procedure RESET (TERMINAL: in out FILE_TYPE;
                 MODE:     in     FILE_MODE);
```

Purpose:

This procedure sets the current mode of the file handle TERMINAL to the mode given by the MODE parameter.

Parameters:

TERMINAL is an open file handle identifying the terminal file handle to be reset.

MODE indicates the new mode under which the file handle is to be reset.

Exceptions:

STATUS_ERROR
is raised if TERMINAL is not an open file handle.

INTENT_VIOLATION
is raised if the file node handle associated with the file handle TERMINAL was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

USE_ERROR is raised if the CAIS implementation does not support resetting the file handle to the specified mode.

Downloaded from http://www.everyspec.com

5.3.9.38 Synchronizing the internal file with file node contents

```
procedure SYNCHRONIZE (TERMINAL: in FILE_TYPE);
```

Purpose:

This procedure forces all data written using the file handle TERMINAL to be transmitted to the contents of the file node with which it is associated.

Parameter:

TERMINAL is an open file handle identifying the file to be synchronized.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.9.39 Setting terminal file handle synchronization

```
procedure ENABLE_SYNCHRONIZATION (TERMINAL: in FILE_TYPE;
                                  ENABLE:   in BOOLEAN);
```

Purpose:

This procedure establishes operations on the file handle TERMINAL to be synchronized if ENABLE is TRUE; otherwise, synchronization is implementation-dependent.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

ENABLE *indicates whether or not the file handle is to be enabled for* synchronization.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Notes:

When SYNCHRONIZATION_IS_ENABLED (see Section 5.3.9.40, page 361) returns FALSE for a file handle, the effect of synchronization for the file handle can be achieved by (1) preceding each read operation on the file handle immediately by a call to SYNCHRONIZE (see Section 5.3.9.38, page 359) on the file handle and (2) following each write operation on the file handle immediately by a call to SYNCHRONIZE on the file handle.

Downloaded from http://www.everyspec.com

5.3.9.40 Determining the synchronization of a terminal file handle

```
function SYNCHRONIZATION_IS_ENABLED (TERMINAL: in FILE_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if the file handle is enabled for synchronization; otherwise, it returns FALSE.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

MODE_ERROR is raised if the file handle TERMINAL is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.10 Package CAIS_FORM_TERMINAL_IO

This package provides the functionality of a form terminal. A form terminal consists of a single device (inasmuch as a programmer is concerned). The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_IO_DEFINITIONS.

The scenario for usage of a form terminal has two active agents: a process and a user. Each interaction with the form terminal consists of a three-step sequence. First, the process creates and writes a form to the terminal. Second, the user modifies the form. Third, the process reads the modified form.

A *form* is a two-dimensional matrix of character *positions*, i.e., places on a form where printable ASCII characters may be displayed. The rows of a form are indexed by positive numbers starting with row one at the top of the display. The columns of a form are indexed by positive numbers starting with column one at the left side of the form. The position identified by row one, column one, is called the *start position* of the form. The position with the highest row and column index is called the *end position* of the form.

The *active position* on a form is the position at which the next operation will be performed. The active position is said to *advance* if (1) the row number of the new position is greater than the row number of the old position or (2) the row number of the new position is the same as the row number of the old position and the new position has a greater column number. Similarly, a position is said to *precede* the active position if (1) the row number of the position is less than the row number of the active position or (2) the row number of the position is the same as the row number of the active position and the column number of the position is smaller than the column number of the active position.

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 either at the end position of the form or at the position preceding the next area qualifier toward the end of the form. Qualification of positions preceding the first area qualifier on a form is implementation-dependent. The area qualifier at the beginning of an area defines the set of characteristics for that area. 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.

Downloaded from http://www.everyspec.com

5.3.10.1 Types and subtypes

```
type FILE_TYPE is limited private;

type FORM_TYPE is limited private;
```

FILE_TYPE describes the type for file handles. FORM_TYPE describes characteristics of forms.

```
type CHARACTER_ARRAY is array (CHARACTER) of BOOLEAN;

subtype FUNCTION_KEY_NAME is STRING;
```

CHARACTER_ARRAY is used to determine the characters that are intercepted due to the characteristics of the underlying operating system and the individual terminal. FUNCTION_KEY_NAME is used to identify function keys by string representations.

```
type TERMINAL_POSITION_TYPE is
    record
        ROW:      CAIS_POSITIVE;
        COLUMN:   CAIS_POSITIVE;
    end record;

type AREA_INTENSITY_KIND is
    (NONE,
     NORMAL,
     HIGH);

type AREA_PROTECTION_KIND is
    (UNPROTECTED,
     PROTECTED);

type AREA_INPUT_KIND is
    (GRAPHIC_CHARACTERS,
     NUMERICS,
     ALPHABETICS);

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

TERMINAL_POSITION_TYPE describes the type of a position on a terminal. AREA_INTENSITY_KIND indicates the intensity at which the characters in the area should be displayed; NONE indicates that characters are not displayed. AREA_PROTECTION_KIND specifies whether the user can modify the contents of the area when the form has been activated. AREA_INPUT_KIND specifies the valid characters that may be entered by the user; GRAPHIC_CHARACTERS indicates that any printable character may be entered. AREA_VALUE_KIND indicates the initial value that the area should have when activated; NO_FILL indicates that the value will be specified by a PUT statement.

```
subtype PRINTABLE_CHARACTER is CHARACTER range ' ' .. ASCII.TILDE;
```

PRINTABLE_CHARACTER describes the characters that can be output to a form terminal.

Downloaded from http://www.everyspec.com

5.3.10.2 Opening a form terminal file handle

```
procedure OPEN (TERMINAL: in out FILE_TYPE;
                NODE:     in     NODE_TYPE);
```

Purpose:

This procedure opens a file handle on a terminal file, given an open node handle on the associated terminal file node.

Parameters:

TERMINAL is a file handle, initially closed, to be opened.

NODE is an open node handle to the file node.

Exceptions:

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
is raised if the values of the predefined file node attributes FILE_KIND, ACCESS_METHOD and DEVICE_KIND are not appropriate for the package containing this procedure according to Table XI, page 210.

STATUS_ERROR
is raised if the file handle TERMINAL is open at the time of the call or if the node handle NODE is not open.

USE_ERROR is raised if an open file handle identifies the same file node contents and the CAIS implementation does not support the existence of multiple file handles identifying the same file node contents. Any such restriction must be documented in Appendix F. An implementation is allowed to raise this exception only if it is based on operating system support that does not provide this capability.

INTENT_VIOLATION
is raised if NODE was not opened with an intent specification including at least the intents required for reading and writing contents.

Notes:

Closing the node handle associated with the file handle TERMINAL closes the file handle.

Downloaded from http://www.everyspec.com

5.3.10.3 Closing a form terminal file handle

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

Purpose:

This procedure severs any association between the internal file identified by the file handle TERMINAL and its associated node contents. It also severs any association between the file handle TERMINAL and its associated node handle. Closing an already closed file handle has no effect.

Parameter:

TERMINAL is a file handle to be closed.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.10.4 Determining whether a file handle is open

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

Purpose:

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

Parameter:

TERMINAL is a file handle.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.10.5 Determining the number of function keys

```
function NUMBER_OF_FUNCTION_KEYS (TERMINAL: in FILE_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns the number of function keys defined for the terminal associated with the internal file identified by the file handle TERMINAL.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exception:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

Downloaded from http://www.everyspec.com

5.3.10.6 Determining intercepted input characters

```
function INTERCEPTED_INPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
```

Purpose:

This function returns an array of type CHARACTER_ARRAY that indicates the input characters that can never appear in a form activated on the terminal file handle TERMINAL due to characteristics of the underlying system and the individual terminal. A value of FALSE indicates that the input character can appear; a value of TRUE indicates that it cannot appear.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exception:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

Downloaded from http://www.everyspec.com

5.3.10.7 Determining intercepted output characters

```
function INTERCEPTED_OUTPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
```

Purpose:

This function returns an array of type CHARACTER_ARRAY that indicates the output characters that can never appear in a form activated on the terminal file handle TERMINAL due to characteristics of the underlying system and the individual terminal. A value of FALSE indicates that the output character can appear; a value of TRUE indicates that it cannot appear.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exception:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

Downloaded from http://www.everyspec.com

5.3.10.8 Creating a form

```
procedure CREATE_FORM
                  (FORM:                             in out FORM_TYPE;
                   ROWS:                             in     CAIS_POSITIVE;
                   COLUMNS:                          in     CAIS_POSITIVE;
                   AREA_QUALIFIER_REQUIRES_SPACE:    in     BOOLEAN);
```

Purpose:

This procedure creates a form. The form is erased (see Section 5.3.10.18, page 380).

Parameters:

FORM — is the form created.

ROWS — is the number of rows in the form.

COLUMNS — is the number of columns in the form.

AREA_QUALIFIER_REQUIRES_SPACE
: indicates whether or not an area qualifier requires space on the form.

Exception:

FORM_STATUS_ERROR
: is raised if the form FORM exists (was created) at the time of the call.

Downloaded from http://www.everyspec.com

5.3.10.9 Deleting a form

```
procedure DELETE_FORM (FORM: in out FORM_TYPE);
```

Purpose:

This procedure deletes a form. The value of its parameter after the call is as if it were never created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370). Deleting an already deleted form or a form that was never created has no effect.

Parameter:

FORM is a form.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.10.10 Copying a form

```
procedure COPY_FORM (FROM: in     FORM_TYPE;
                     TO:   in out FORM_TYPE);
```

Purpose:

This procedure copies the value of the form FROM to the form TO.

Parameters:

FROM	is the form to be copied.
TO	is the form receiving the value.

Exception:

FORM_STATUS_ERROR
is raised if FROM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Notes:

If the form TO does not exist at the time of the call, it is created with the same parameters with which the form FROM was created.

Downloaded from http://www.everyspec.com

5.3.10.11 Defining a qualified area

```
procedure DEFINE_QUALIFIED_AREA
                (FORM:        in out FORM_TYPE;
                 INTENSITY:   in     AREA_INTENSITY_KIND := NORMAL;
                 PROTECTION:  in     AREA_PROTECTION_KIND := PROTECTED;
                 INPUT:       in     AREA_INPUT_KIND := GRAPHIC_CHARACTERS;
                 VALUE:       in     AREA_VALUE_KIND := NO_FILL);
```

Purpose:

This procedure places an area qualifier with the designated attributes (INTENSITY, PROTECTION, INPUT, VALUE) at the active position of the form FORM. A qualified area consists of the character positions between two area qualifiers. The area is qualified by the area qualifier that precedes the area. A qualified area may or may not include the position of its area qualifier (see Section 5.3.10.26, page 388, and Section 5.3.10.27, page 389).

Parameters:

FORM	is the form on which the qualified area is being defined.
INTENSITY	is the intensity at which the qualified area is to be displayed.
PROTECTION	is the protection for the qualified area.
INPUT	is the set of permissible input characters for the qualified area.
VALUE	is the initial value of the qualified area.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Notes:

The characteristics (intensity, protection, input, value) of positions that precede the first area qualifier on a form are implementation-defined.

Downloaded from http://www.everyspec.com

5.3.10.12 Removing an area qualifier

```
procedure REMOVE_AREA_QUALIFIER (FORM: in out FORM_TYPE);
```

Purpose:

This procedure removes an area qualifier from the active position of the form. Removing an area qualifier from a position that does not have an area qualifier has no effect.

Parameter:

FORM is the form from which the qualified area is to be removed.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Notes:

The positions of the area from which the area qualifier is being removed become part of the preceding qualified area. If there is no preceding qualified area the set of characteristics for the positions are implementation-dependent.

Downloaded from http://www.everyspec.com

5.3.10.13 Changing the active position

```
procedure SET_ACTIVE_POSITION (FORM:     in out FORM_TYPE;
                               POSITION: in     TERMINAL_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:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

TERMINAL_POSITION_ERROR
is raised if POSITION does not identify a position in FORM.

Downloaded from http://www.everyspec.com

5.3.10.14 Querying the active position

```
function ACTIVE_POSITION (FORM: in FORM_TYPE)
    return TERMINAL_POSITION_TYPE;
```

Purpose:

This function returns the active position of the form FORM.

Parameter:

FORM is the form to be queried.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Downloaded from http://www.everyspec.com

5.3.10.15 Advancing forward to qualified area

```
procedure ADVANCE_TO_QUALIFIED_AREA (FORM:  in out FORM_TYPE;
                                     COUNT: in     CAIS_POSITIVE := 1);
```

Purpose:

This procedure advances the active position COUNT area qualifiers toward the end of the form. If there are fewer than COUNT area qualifiers between the active position and the end of the FORM the active position is set to the position of the last area qualifier on the form. If there are no area qualifiers between the active position and the end of the FORM the active position is not changed.

Parameters:

FORM — is the form on which the active position is being advanced.

COUNT — is the number of qualified areas the active position is to be advanced.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Downloaded from http://www.everyspec.com

5.3.10.16 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 set to the start position.

Parameters:

FORM is the form being written.

ITEM is the character to be written to the form.

Exceptions:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

USE_ERROR is raised if the active position contains an area qualifier and AREA_QUALIFIER_REQUIRES_SPACE(FORM) was set to TRUE.

Additional interface:

```
procedure PUT (FORM: in out FORM_TYPE;
               ITEM: in     STRING)
is
begin
    for INDEX in ITEM'FIRST .. ITEM'LAST loop
        PUT (FORM, ITEM(INDEX));
    end loop;
end PUT;
```

Downloaded from http://www.everyspec.com

5.3.10.17 Erasing a qualified area

```
procedure ERASE_AREA (FORM: in out FORM_TYPE);
```

Purpose:

This procedure places space characters in all positions of the area in which the active position of the form is located. If the active position is not in a qualified area, this procedure has no effect.

Parameter:

FORM is the form on which the qualified area is being erased.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Downloaded from http://www.everyspec.com

5.3.10.18 Erasing a form

```
procedure ERASE_FORM (FORM: in out FORM_TYPE);
```

Purpose:

This procedure removes all area qualifiers and places space characters in all positions of the form. The active position is set to the start position. The form is established as not updated (see Section 5.3.10.21, page 383). The termination key is established as the normal termination key (a value of zero) (see Section 5.3.10.23, page 385).

Parameter:

FORM is the form to be erased.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Downloaded from http://www.everyspec.com

5.3.10.19 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 file associated with the internal file identified by the file handle TERMINAL. The contents of the terminal file are modified to reflect the contents of the form. When the user of the terminal enters a termination key, the modified contents of the terminal file are copied back to the form and returned. This operation may not result in the modification of protected areas. Qualification of positions preceding the first area qualifier on a form is implementation-dependent.

Parameters:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

FORM is the form to be activated.

Exceptions:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

USE_ERROR is raised if FORM_SIZE(FORM) is not equal to TERMINAL_SIZE (TERMINAL) or if AREA_QUALIFIER_REQUIRES_SPACE(FORM) is not equal to AREA_QUALIFIER_REQUIRES_SPACE(TERMINAL).

Downloaded from http://www.everyspec.com

5.3.10.20 Reading from a form

```
procedure GET (FORM: in out FORM_TYPE;
               ITEM:    out PRINTABLE_CHARACTER);
```

Purpose:

This procedure reads a character from FORM at the active position and advances the active position one position toward the end position. If the active position is the end position, the active position is set to the start position. An area qualifier on a form on which the area qualifier requires space is read as the space character.

Parameters:

FORM	is the form to be read.
ITEM	is the character that was read.

Exception:

FORM_STATUS_ERROR

is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Additional Interface:

```
procedure GET (FORM: in out FORM_TYPE;
               ITEM:    out STRING)
is
begin
    for INDEX in ITEM'FIRST .. ITEM'LAST loop
         GET (FORM, ITEM(INDEX));
    end loop;
end GET;
```

Downloaded from http://www.everyspec.com

5.3.10.21 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.

Parameter:

FORM is the form to be queried.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Downloaded from http://www.everyspec.com

5.3.10.22 Determining the identification of a function key

```
function FUNCTION_KEY_IDENTIFICATION
                         (TERMINAL:       in FILE_TYPE;
                          KEY_IDENTIFIER: in CAIS_POSITIVE)
    return FUNCTION_KEY_NAME;
```

Purpose:

This function returns the string identification of the function key designated by KEY_IDENTIFIER.

Parameters:

TERMINAL — is an open file handle identifying the internal file associated with the terminal file.

KEY_IDENTIFIER — is the identification number of a function key.

Exceptions:

STATUS_ERROR — is raised if the file handle TERMINAL is not open.

FUNCTION_KEY_STATUS_ERROR — is raised if the value of KEY_IDENTIFIER is greater than NUMBER_OF_FUNCTION_KEYS(TERMINAL).

Downloaded from http://www.everyspec.com

5.3.10.23 Determining the termination key

```
function TERMINATION_KEY (FORM: in FORM_TYPE)
    return CAIS_NATURAL;
```

Purpose:

This function returns a number that indicates which (implementation-dependent) key terminated the ACTIVATE procedure (see Section 5.3.10.19, page 381) for the form FORM. A value of zero indicates the normal termination key (e.g., the ENTER key). A positive value indicates a function key.

Parameter:

FORM is the form to be queried.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Downloaded from http://www.everyspec.com

5.3.10.24 Determining the size of a form

```
function FORM_SIZE (FORM: in FORM_TYPE)
    return TERMINAL_POSITION_TYPE;
```

Purpose:

This function returns the position of the maximum column of the maximum row of the form.

Parameter:

FORM is the form to be queried.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Downloaded from http://www.everyspec.com

5.3.10.25 Determining the size of the terminal

```
function TERMINAL_SIZE (TERMINAL: in FILE_TYPE)
    return TERMINAL_POSITION_TYPE;
```

Purpose:

This function returns the position of the maximum column of the maximum row of the internal file identified by the file handle TERMINAL.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exception:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

Downloaded from http://www.everyspec.com

5.3.10.26 Determining if the area qualifier requires space in the form

```
function AREA_QUALIFIER_REQUIRES_SPACE (FORM: in FORM_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if the area qualifier requires space in the form FORM; otherwise it returns FALSE.

Parameter:

FORM is the form to be queried.

Exception:

FORM_STATUS_ERROR
is raised if FORM has not been previously created by the procedure CREATE_FORM (see Section 5.3.10.8, page 370).

Downloaded from http://www.everyspec.com

5.3.10.27 Determining if the area qualifier requires space on a terminal

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

Purpose:

This function returns TRUE if the area qualifier requires space on the internal file identified by the file handle TERMINAL; otherwise it returns FALSE.

Parameter:

TERMINAL is an open file handle identifying the internal file associated with the terminal file.

Exception:

STATUS_ERROR
is raised if the file handle TERMINAL is not open.

Downloaded from http://www.everyspec.com

5.3.11 Package CAIS_MAGNETIC_TAPE_IO

This package provides interfaces for the support of input and output operations on magnetic tapes. For purposes of interoperability the interfaces defined in this section should be used in accordance with level II of [ANSI 78]. The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_IO_DEFINITIONS.

To use a tape drive, a file handle on the file representing the tape drive must be obtained (see OPEN in Section 5.3.11.2).

When information transfer is completed, the tape is unloaded and dismounted using the UNLOAD (see Section 5.3.11.8, page 401) and REQUEST_DISMOUNT (see Section 5.3.11.9, page 402) procedures.

Once a tape is dismounted, another tape may be mounted. When the user is finished utilizing the drive, the file handle on the file representing the tape on the drive should be closed (see Section 5.3.11.3, page 396).

Magnetic tape drive file nodes can only be created by the implementation. Implementation-defined file characteristics must be supported by the implementation and will include the densities and block sizes supported by the tape drive, whether or not a tape is mounted on the drive.

Character data is transferred to and from magnetic tapes in fixed length records. Each logical record is also a physical block.

When transferring an Ada text file to or from a magnetic tape, the text file must be read or written as blocks of Ada characters. Table XIV identifies the mapping that is to be used between the contents of an Ada text file and a file on magnetic tape containing an Ada text file.

Downloaded from http://www.everyspec.com

TABLE XIV. Allowed Magnetic Tape Characters

ALLOWED CHARACTERS	REPRESENTATION OF CHARACTERS
All printable characters	CHARACTER(' ') .. ASCII.TILDE
Horizontal Tab	ASCII.HT
Vertical Tab	ASCII.VT
Carriage Return	ASCII.CR
Line Terminator	ASCII.LF
Page Terminator	ASCII.FF
Fill Character	ASCII.NUL
File Terminator	Zero or more fill characters followed by a tape mark.

Use of other characters is not defined. Each block of a file may be terminated by zero or more fill characters.

All tape read and write operations use odd parity.

For magnetic tape, the exception DEVICE_ERROR may be raised for many implementation-dependent reasons, e.g., a write ring not on the tape when attempting to write to it.

Downloaded from http://www.everyspec.com

5.3.11.1 Types, subtypes and exceptions

```
type FILE_TYPE is limited private;

type FILE_MODE is (IN_FILE, OUT_FILE);
```

FILE_TYPE defines the type for file handles, which are used for controlling all operations on tape drives. FILE_MODE describes whether a file handle is to be used for input or output; it can never be used for both.

```
subtype TAPE_NAME is STRING;

subtype TAPE_BLOCK is STRING;
```

TAPE_NAME defines a subtype for the name of a tape to be mounted. TAPE_BLOCK defines a subtype for buffers for the reading and writing of blocks of characters.

```
type TAPE_DRIVE_STATUS_KIND is
      (OPENED,
       MOUNT_REQUESTED,
       MOUNTED,
       LOADED,
       CLOSED);

type TAPE_POSITION_KIND is
      (BEGINNING_OF_VOLUME,
       END_OF_VOLUME,
       END_OF_TAPE,
       AFTER_TAPE_MARK,
       OTHER);

type TAPE_RECORDING_METHOD_KIND is
      (NON_RETURN_TO_ZERO_INVERTED,
       PHASE_ENCODED,
       GROUP_CODED_RECORDING);
```

TAPE_DRIVE_STATUS_KIND defines the states of an internal magnetic tape drive file. Figure 11 shows the state transitions that may occur. The success (MOUNTED) or failure (OPENED) state of a mount request is determined by a particular CAIS implementation. The time for the actual change of state from OPENED to MOUNT_REQUESTED for the REQUEST_MOUNT interface and the time for the actual change from MOUNTED to OPENED for the REQUEST_DISMOUNT interface are implementation-defined. In all other situations the time of the transition from one state to another is the completion of the interface call. The function STATUS(TAPE_DRIVE) should be used to determine the state of an internal magnetic tape drive file.

TAPE_POSITION_KIND describes the position of the tape on the tape drive; a value of AFTER_TAPE_MARK means that the tape is positioned just after a tape mark. That is, a read in this position will read the next block.

TAPE_RECORDING_METHOD_KIND identifies the particular tape recording method and tape recording density being used. NON_RETURN_TO_ZERO_INVERTED indicates conformance to ANSI X3.22 [ANSI 73a] or ISO 1863 [ISO 76]. PHASE_ENCODED indicates conformance to ANSI X3.39 [ANSI 73b] or ISO 3788 [ISO 76b]. GROUP_

Downloaded from http://www.everyspec.com

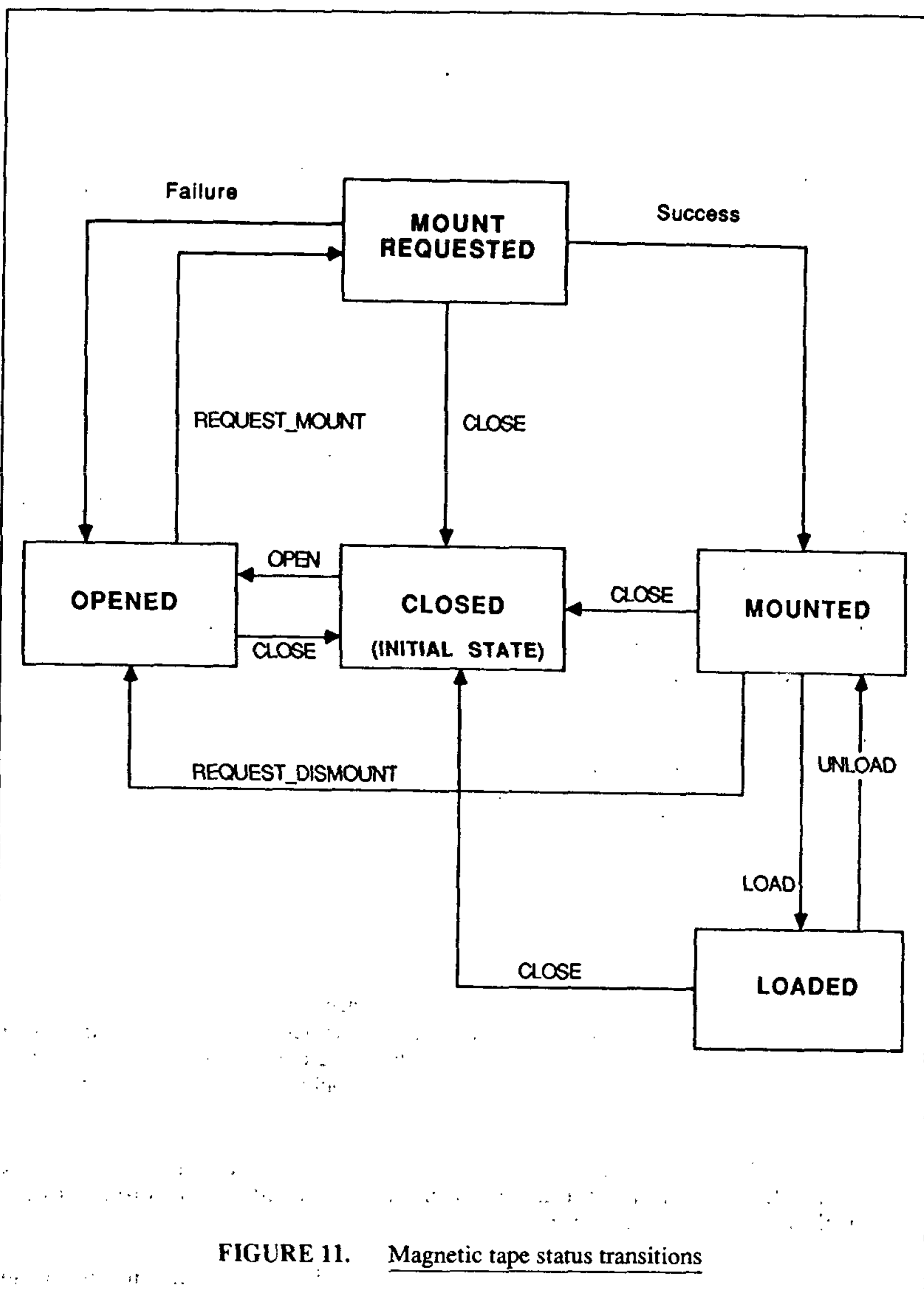

FIGURE 11. Magnetic tape status transitions

Downloaded from http://www.everyspec.com

CODED_RECORDING indicates conformance to ANSI X3.54 [ANSI 76] or ISO 5652 [ISO 84].

```
TAPE_STATUS_ERROR: exception;
```

TAPE_STATUS_ERROR is raised if the mounted or loaded state of a tape drive is incorrect for the operation.

Downloaded from http://www.everyspec.com

5.3.11.2 Opening a tape drive file handle

```
procedure OPEN (TAPE_DRIVE: in out FILE_TYPE;
                NODE:       in     NODE_TYPE;
                MODE:       in     FILE_MODE);
```

Purpose:

This procedure opens a file handle on a magnetic tape drive file, given an open node handle on the associated magnetic tape drive file node.

Parameters:

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

NODE is an open node handle to a file node.

MODE indicates the mode under which the file handle is to be opened.

Exceptions:

NODE_KIND_ERROR
is raised if the node identified by NODE is not a file node.

FILE_KIND_ERROR
is raised if the values of the predefined file node attributes FILE_KIND, ACCESS_METHOD and DEVICE_KIND are not appropriate for the package containing this procedure according to Table XI, page 210.

STATUS_ERROR
is raised if the file handle TAPE_DRIVE is already open at the time of the call on OPEN or if NODE is not an open node handle.

USE_ERROR is raised if an open file handle identifies the same file node contents and the CAIS implementation does not support the existence of multiple file handles identifying the same file node contents. Any such restriction must be documented in Appendix F. An implementation is allowed to raise this exception only if it is based on operating system support that does not provide this capability.

INTENT_VIOLATION
is raised if NODE was not opened with an intent specification including at least the intents required for MODE, as specified in Table X, page 209.

Notes:

Closing the node handle associated with the file handle TAPE_DRIVE closes the file handle.

Downloaded from http://www.everyspec.com

5.3.11.3 Closing a tape drive file handle

```
procedure CLOSE (TAPE_DRIVE: in out FILE_TYPE);
```

Purpose:

This procedure severs any association between the internal file identified by the file handle TAPE_DRIVE and its associated node contents. It also severs any association between the file handle TAPE_DRIVE and its associated node handle. Closing an already closed file handle has no effect.

If the state of the file handle TAPE_DRIVE is LOADED, the tape represented by the file handle is unloaded and dismounted before closing. If the state of the file handle is MOUNTED or MOUNT_REQUESTED, the tape represented by the file handle is dismounted before closing.

Parameter:

TAPE_DRIVE is a file handle, initially open, to be closed.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.11.4 Determining whether a file handle is open

```
function IS_OPEN (TAPE_DRIVE: in FILE_TYPE)
    return BOOLEAN;
```

Purpose:

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

Parameter:

TAPE_DRIVE   is a file handle.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.3.11.5 Determining the mode of a magnetic tape drive

```
function MODE (TAPE_DRIVE: in FILE_TYPE)
    return FILE_MODE;
```

Purpose:

This function returns the mode under which TAPE_DRIVE is opened.

Parameter:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

Exception:

STATUS_ERROR
is raised if the file handle TAPE_DRIVE is not open.

Downloaded from http://www.everyspec.com

5.3.11.6 Requesting the mounting of a tape

```
procedure REQUEST_MOUNT (TAPE_DRIVE:         in FILE_TYPE;
                         NAME:               in TAPE_NAME;
                         RECORDING_METHOD:   in TAPE_RECORDING_METHOD_KIND;
                         INSTALL_WRITE_RING: in BOOLEAN := FALSE);
```

Purpose:

This procedure generates an implementation-defined request that the tape whose external name is NAME be mounted on the tape drive associated with the internal file identified by the file handle TAPE_DRIVE, the tape drive recording method be set to RECORDING_METHOD, and, if INSTALL_WRITE_RING is TRUE, that a write ring be installed on the tape before mounting.

Following completion of this procedure until the request is fulfilled or denied, STATUS (TAPE_DRIVE) returns MOUNT_REQUESTED. If the request is fulfilled, the function STATUS(TAPE_DRIVE) returns MOUNTED. If the request is denied, the function STATUS(TAPE_DRIVE) returns OPENED.

Parameters:

TAPE_DRIVE
: is an open file handle identifying the internal file associated with the tape drive file.

NAME
: is an external name which identifies the tape to be mounted on the tape drive.

RECORDING_METHOD
: is the recording method to be used when writing to the tape.

INSTALL_WRITE_RING
: indicates whether or not a write ring is to be installed on the tape.

Exception:

STATUS_ERROR
: is raised if TAPE_DRIVE is not an open file handle.

Downloaded from http://www.everyspec.com

5.3.11.7 Loading a tape

```
procedure LOAD (TAPE_DRIVE: in FILE_TYPE;
                BLOCK_SIZE: in CAIS_POSITIVE);
```

Purpose:

This procedure loads the tape on the tape drive represented by the file associated with the internal file identified by TAPE_DRIVE. The tape is positioned after the beginning of tape mark. Following completion of this procedure, the function STATUS(TAPE_DRIVE) returns LOADED.

Parameters:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

BLOCK_SIZE is the number of bytes to be read or written during input or output operations.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not MOUNTED.

USE_ERROR is raised if BLOCK_SIZE is less than MINIMUM_TAPE_BLOCK_LENGTH (see Section 5.7, page 513) or greater than MAXIMUM_TAPE_BLOCK_LENGTH (see Section 5.7, page 513).

Downloaded from http://www.everyspec.com

5.3.11.8 Unloading a tape

```
procedure UNLOAD (TAPE_DRIVE: in FILE_TYPE);
```

Purpose:

This procedure unloads the tape on the tape drive file associated with the internal file identified by the file handle TAPE_DRIVE. Following completion of this procedure, the tape is positioned after the beginning of tape mark, the function STATUS(TAPE_DRIVE) will return MOUNTED, and the function POSITION(TAPE_DRIVE) will return BEGINNING_OF_VOLUME. Unloading a tape in which STATUS(TAPE_DRIVE) is MOUNTED has no effect.

Parameter:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is neither MOUNTED nor LOADED.

Downloaded from http://www.everyspec.com

5.3.11.9 Requesting the dismounting of a tape

```
procedure REQUEST_DISMOUNT (TAPE_DRIVE: in FILE_TYPE);
```

Purpose:

This procedure generates an implementation-defined request that the tape on the tape drive represented by the file associated with the internal file identified by file handle TAPE_DRIVE be removed from the drive. It makes the tape available for removal. Following the completion of this procedure, the function STATUS(TAPE_DRIVE) will return OPENED.

Parameter:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not MOUNTED.

Downloaded from http://www.everyspec.com

5.3.11.10 Determining the position of the tape

```
function POSITION (TAPE_DRIVE: in FILE_TYPE)
    return TAPE_POSITION_KIND;
```

Purpose:

This function returns the current position of the internal file identified by the file handle TAPE_DRIVE.

Parameter:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not LOADED.

Downloaded from http://www.everyspec.com

5.3.11.11 Rewinding the tape

```
procedure REWIND_TAPE (TAPE_DRIVE: in FILE_TYPE);
```

Purpose:

This procedure positions the internal file identified by the file handle TAPE_DRIVE after the beginning of tape mark. Following completion of this procedure, the function POSITION(TAPE_DRIVE) returns BEGINNING_OF_VOLUME.

Parameter:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not LOADED.

Downloaded from http://www.everyspec.com

5.3.11.12 Skipping tape marks

```
procedure SKIP_TAPE_MARK (TAPE_DRIVE: in FILE_TYPE;
                          COUNT:      in CAIS_POSITIVE := 1);
```

Purpose:

This procedure skips COUNT tape marks on the internal file identified by the file handle TAPE_DRIVE.

Following a call to SKIP_TAPE_MARK, the tape is positioned immediately following the appropriate tape mark. If the end of tape mark is encountered during this operation, the function POSITION(TAPE_DRIVE) returns END_OF_TAPE; otherwise POSITION(TAPE_DRIVE) returns AFTER_TAPE_MARK.

If two consecutive tape marks are encountered during this operation, the tape is positioned between the two tape marks and the function POSITION(TAPE_DRIVE) returns END_OF_VOLUME. If at the time of the call the tape is positioned between two consecutive tape marks, the position of the tape is not changed and the function POSITION(TAPE_DRIVE) returns END_OF_VOLUME.

Parameters:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

COUNT is the number of tape marks to skip.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not LOADED.

MODE_ERROR is raised if the file handle TAPE_DRIVE is of mode OUT_FILE.

Downloaded from http://www.everyspec.com

5.3.11.13 Writing a tape mark

```
procedure WRITE_TAPE_MARK (TAPE_DRIVE: in FILE_TYPE);
```

Purpose:

This procedure writes a tape mark on the internal file identified by the file handle TAPE_DRIVE. If the end of tape mark is encountered during this operation, the function POSITION(TAPE_DRIVE) returns END_OF_TAPE; otherwise, POSITION (TAPE_DRIVE) returns AFTER_TAPE_MARK.

Parameter:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not LOADED.

MODE_ERROR is raised if the file handle TAPE_DRIVE is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.11.14 Determining the status of a magnetic tape drive

```
function STATUS (TAPE_DRIVE: in FILE_TYPE)
    return TAPE_DRIVE_STATUS_KIND;
```

Purpose:

This function returns the tape drive status of the file handle TAPE_DRIVE.

Parameter:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

Exception:

STATUS_ERROR
is raised if the file handle TAPE_DRIVE is not open.

Downloaded from http://www.everyspec.com

5.3.11.15 Determining the recording method of a magnetic tape

```
function RECORDING_METHOD (TAPE_DRIVE: in FILE_TYPE)
    return TAPE_RECORDING_METHOD_KIND;
```

Purpose:

This function returns the value specified for RECORDING_METHOD during the most recent request to mount a tape on the tape drive associated with the internal file identified by the file handle TAPE_DRIVE.

Parameter:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not MOUNTED.

Downloaded from http://www.everyspec.com

5.3.11.16 Determining whether a write ring is installed

```
function IS_WRITE_RING_INSTALLED (TAPE_DRIVE: in FILE_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns the value specified for INSTALL_WRITE_RING during the most recent request to mount a tape on the tape drive associated with the internal file identified by the file handle TAPE_DRIVE.

Parameter:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is neither MOUNTED nor LOADED.

Downloaded from http://www.everyspec.com

5.3.11.17 Skipping blocks in a magnetic tape file

```
procedure SKIP_BLOCK (TAPE_DRIVE: in FILE_TYPE;
                      COUNT:      in CAIS_POSITIVE := 1);
```

Purpose:

This procedure skips COUNT blocks on the internal file identified by the file handle TAPE_DRIVE.

The tape is positioned COUNT blocks toward the end of the tape. If a tape mark is encountered during this operation, the tape is positioned after the tape mark and POSITION(TAPE_DRIVE) returns AFTER_TAPE_MARK. If during or after this operation the tape is positioned after the end of tape mark, POSITION(TAPE_DRIVE) returns END_OF_TAPE even if a tape mark was encountered.

Parameters:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

COUNT is the number of blocks to skip.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not LOADED.

MODE_ERROR is raised if the file handle TAPE_DRIVE is of mode OUT_FILE.

Downloaded from http://www.everyspec.com

5.3.11.18 Reading a block from a magnetic tape file

```
procedure READ_BLOCK (TAPE_DRIVE:      in    FILE_TYPE;
                      BLOCK:           out   TAPE_BLOCK;
                      LAST:            out   CAIS_NATURAL;
                      BLOCK_OVERFLOW:  out   CAIS_NATURAL);
```

Purpose:

This procedure reads a block of characters from the internal file identified by the file handle TAPE_DRIVE into the string BLOCK. The number LAST identifies the index into BLOCK of the last character read. Characters are read sequentially into BLOCK starting at BLOCK'FIRST.

If the block of characters in the internal file is greater than BLOCK'LENGTH, the characters in the block after BLOCK'LENGTH characters are lost. The number of characters that are lost is returned in BLOCK_OVERFLOW.

If during or after this operation the internal file is positioned after the end of tape mark, POSITION(TAPE_DRIVE) returns END_OF_TAPE; otherwise, POSITION(TAPE_DRIVE) returns OTHER. If during this operation a tape mark is read, POSITION (TAPE_DRIVE) returns AFTER_TAPE_MARK and LAST is equal to CAIS_NATURAL'PRED(BLOCK'FIRST).

Parameters:

TAPE_DRIVE
: is an open file handle identifying the internal file associated with the tape drive file.

BLOCK
: is a string to receive the characters that are read.

LAST
: is the index into BLOCK of the last character read.

BLOCK_OVERFLOW
: is the number of characters lost as a result of the block of characters on tape being greater than BLOCK'LENGTH.

Exceptions:

STATUS_ERROR
: is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
: is raised if STATUS(TAPE_DRIVE) is not LOADED.

MODE_ERROR is raised if the file handle TAPE_DRIVE is of mode OUT_FILE.

Downloaded from http://www.everyspec.com

5.3.11.19 Writing a block to a magnetic tape file

```
procedure WRITE_BLOCK (TAPE_DRIVE: in FILE_TYPE;
                       BLOCK:      in TAPE_BLOCK);
```

Purpose:

This procedure writes the block of characters BLOCK to the internal file identified by the file handle TAPE_DRIVE.

If during or after this operation the tape is positioned after the end of tape mark, POSITION(TAPE_DRIVE) returns END_OF_TAPE; otherwise POSITION(TAPE_DRIVE) returns OTHER.

Parameters:

TAPE_DRIVE is an open file handle identifying the internal file associated with the tape drive file.

BLOCK is the string to be written.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

USE_ERROR is raised if BLOCK'LENGTH is less than MINIMUM_TAPE_BLOCK_LENGTH (see Section 5.7, page 513) or greater than MAXIMUM_TAPE_BLOCK_LENGTH (see Section 5.7, page 513) or not equal to the block size specified when the tape was loaded.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not LOADED.

MODE_ERROR is raised if the file handle TAPE_DRIVE is of mode IN_FILE.

Downloaded from http://www.everyspec.com

5.3.11.20 Resetting a magnetic tape file handle

```
procedure RESET (TAPE_DRIVE: in out FILE_TYPE;
                 MODE:       in     FILE_MODE);
```

Purpose:

This procedure sets the current mode of the file handle TAPE_DRIVE to the mode given by the MODE parameter.

It also positions the tape after the beginning of tape mark. Following completion of this procedure, the function POSITION(TAPE_DRIVE) returns BEGINNING_OF_VOLUME.

Parameters:

TAPE_DRIVE is an open file handle identifying the tape drive file handle to be reset.

MODE indicates the new mode under which the file handle is to be reset.

Exceptions:

STATUS_ERROR
is raised if TAPE_DRIVE is not an open file handle.

TAPE_STATUS_ERROR
is raised if STATUS(TAPE_DRIVE) is not LOADED.

INTENT_VIOLATION
is raised if the file node handle associated with the file handle TAPE_DRIVE was not opened with an intent specification including at least the intents required for the MODE, as specified in Table X, page 209.

USE_ERROR is raised if the CAIS implementation does not support resetting the file handle to the specified mode.

Notes:

Since a magnetic tape drive file can only have the modes IN_FILE or OUT_FILE, this procedure can be used to read a tape that has just been written by changing the mode from IN_FILE to OUT_FILE.

Downloaded from http://www.everyspec.com

5.3.12 Package CAIS_IMPORT_EXPORT

The CAIS allows a particular CAIS implementation to maintain files separately from files maintained by the host file system. This package provides the capability to transfer files between these two systems. The exceptions raised by all subprograms in this package are defined in the packages CAIS_DEFINITIONS and CAIS_IO_DEFINITIONS.

Downloaded from http://www.everyspec.com

5.3.12.1 Importing a file

```
procedure IMPORT_CONTENTS (FROM:            in STRING;
                           TO:              in NODE_TYPE;
                           CHARACTERISTICS: in LIST_TYPE := EMPTY_LIST);
```

Purpose:

This procedure copies the file identified by FROM in the host file system into the contents of the file node associated with TO. The contents of the file identified by FROM in the host file system replace the contents of the file node associated with TO.

Parameters:

FROM
: is the name of the host file to be copied.

TO
: is an open node handle to the node receiving the contents.

CHARACTERISTICS
: is a list (see Section 5.4, page 419) of implementation-dependent information to be used for copying the host file.

Exceptions:

USE_ERROR
: is raised if the parameters FROM or CHARACTERISTICS have (implementation-dependent) values that do not permit copying the host file.

STATUS_ERROR
: is raised if TO is not an open node handle.

NODE_KIND_ERROR
: is raised if the node identified by TO is not a file node.

FILE_KIND_ERROR
: is raised if the value of the predefined file node attribute FILE_KIND is not SECONDARY_STORAGE.

INTENT_VIOLATION
: is raised if TO was not opened with an intent establishing the right to write contents. INTENT_VIOLATION is not raised if the conditions for other exceptions (excluding SECURITY_VIOLATION) are present.

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

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure IMPORT_CONTENTS (FROM:            in STRING;
                           TO:              in PATHNAME;
                           CHARACTERISTICS: in LIST_TYPE := EMPTY_LIST)
is
     TO_NODE: NODE_TYPE;
begin
     OPEN (TO_NODE, TO, (1=>WRITE_CONTENTS));
     IMPORT_CONTENTS (FROM, TO_NODE, CHARACTERISTICS);
     CLOSE (TO_NODE);
exception
     when others =>
          CLOSE (TO_NODE);
          raise;
end IMPORT_CONTENTS;
```

Downloaded from http://www.everyspec.com

5.3.12.2 Exporting a file

```
procedure EXPORT_CONTENTS (FROM:            in NODE_TYPE;
                           TO:              in STRING;
                           CHARACTERISTICS: in LIST_TYPE := EMPTY_LIST);
```

Purpose:

This procedure copies the contents of the file node identified by the open node handle FROM to the file identified by TO in the host file system. The CHARACTERISTICS parameter provides implementation-dependent information for copying the file node contents.

Parameters:

FROM
: is an open node handle to the node whose contents are to be exported.

TO
: is the name of the host file receiving the contents.

CHARACTERISTICS
: is a list (see Section 5.4, page 419) of implementation-dependent information to be used for copying the contents.

Exceptions:

USE_ERROR
: is raised if the parameters TO or CHARACTERISTICS have (implementation-dependent) values that do not permit copying the contents.

STATUS_ERROR
: is raised if FROM is not an open node handle.

NODE_KIND_ERROR
: is raised if the node identified by FROM is not a file node.

FILE_KIND_ERROR
: is raised if the value of the predefined file node attribute FILE_KIND is not SECONDARY_STORAGE.

INTENT_VIOLATION
: is raised if FROM was not opened with an intent establishing the right to read attributes and contents. INTENT_VIOLATION is not raised if the conditions for other exceptions (excluding SECURITY_VIOLATION) are present.

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

Downloaded from http://www.everyspec.com

Additional Interface:

```
procedure EXPORT_CONTENTS (FROM:             in PATHNAME;
                           TO:               in STRING;
                           CHARACTERISTICS:  in LIST_TYPE := EMPTY_LIST)
is
     FROM_NODE: NODE_TYPE;
begin
     OPEN (FROM_NODE, FROM, (READ_ATTRIBUTES,READ_CONTENTS));
     EXPORT_CONTENTS (FROM_NODE, TO, CHARACTERISTICS);
     CLOSE (FROM_NODE);
exception
     when others =>
          CLOSE (FROM_NODE);
          raise;
end EXPORT_CONTENTS;
```

Downloaded from http://www.everyspec.com

5.4 CAIS List Management

This section describes the CAIS interfaces for the manipulation of lists. Lists are used as the values of node and relationship attributes. The interfaces described in package CAIS_LIST_MANAGEMENT are intended for use by other CAIS interfaces. The exceptions raised by all subprograms in this package are defined in the packages CAIS_LIST_MANAGEMENT and CAIS_PRAGMATICS.

A *linear list* is a linearly ordered set of data elements called *list items*. Each list item has an *item value*. A list item may also have an *item name*, in which case it is called a *named item*; if a list item has no item name, it is called an *unnamed item*.

There are three kinds of linear lists: empty lists, named lists, and unnamed lists. An *empty list* is a linear list that contains no items. Such a list is not considered to be either named or unnamed. A *named list* is a non-empty linear list that contains only named items; the names of distinct items in a named list must be distinct. An *unnamed list* is a non-empty linear list that contains only unnamed items. The values of type LIST_KIND enumerate these three kinds of lists.

There are five kinds of list items, discriminated on the kind of values they can have: strings, integers, floating point numbers, identifiers and linear lists. The values of type ITEM_KIND enumerate these five kinds of list items. The actual, internal types of list item values are not specified, although certain properties of these types are specified in the package CAIS_PRAGMATICS (Section 5.7, page 509). However, the means to manipulate these internal values are provided by explicit, external types (or generic type parameters) used by the subprograms in CAIS_LIST_MANAGEMENT. These external types have been chosen so as to maximize the likelihood that they will be related very efficiently to the internal types used by a given implementation of CAIS_LIST_MANAGEMENT; in some cases, the external types can even be the same as the internal types.

Although a CAIS implementation may choose a representation for internal values different from that of the external values at the interfaces, it must preserve certain properties. The process of inserting values into a list (for example by inserting list items or by replacing the values of existing list items) and then retrieving the inserted values must either preserve the values (or, in the case of floating point values, nearly preserve the values) or result in CAPACITY_ERROR upon insertion and CONSTRAINT_ERROR upon retrieval, in case the constraints of the respective internal or external types are violated.

Because of the conversions between external, instantiated types and internal types, the storage and retrieval cycle for floating point numbers may not preserve the values precisely; the degree of error is completely contained within that provided by doing standard Ada type conversions.

Some interfaces, such as those which determine an item's position by value, require a value provided at the interface to be compared with an internal value in a list; these operations behave as if the value provided at the interface is converted to internal representation prior to comparison.

The principles of external values, internal values and comparison of values are realized in distinct ways for values of each of the five kinds of list items:

Downloaded from http://www.everyspec.com

a. String values of list items are represented by the external type STRING at the interfaces. String equality is defined according to the predefined equality for the Ada type STRING.

b. Integer values of list items are represented at the interfaces by external integer types defined in the programs using the interfaces; operations to manipulate these values are placed in generic packages that can be instantiated for user-defined types.

   Internal integer values include exactly the values of the CAIS type CAIS_INTEGER (see Section 5.1.1, 54); integer equality is defined according to the predefined equality for the CAIS type CAIS_INTEGER.

c. Floating point values of list items are represented at the interfaces by external floating point types defined in the programs using the interfaces; operations to manipulate these values are placed in generic packages that can be instantiated for user-defined types.

   Floating point internal values preserve at least CAIS_PRAGMATICS.LIST_MAXIMUM_DIGITS accuracy. Floating point equality of two values is defined according to predefined equality for the two values when converted to a floating point type of CAIS_PRAGMATICS.LIST_MAXIMUM_DIGITS accuracy.

   Upon insertion and subsequent extraction of floating point values, it is implementation-dependent whether or not changes to the physical representation of the value are made. If such changes are made, they must be consistent with the accuracy of the types used in the respective instantiations of the package CAIS_FLOAT_ITEM, but may affect the meaning of equality between the retrieved value and the inserted value.

   Users should be aware of the accuracy issues for relational operations between floating point values explained in [1815A], Section 4.5.7, in order to avoid erroneous assumptions about the equality of float items and lists involving such items.

d. Identifier values of list items occur in two different forms which are represented by two different external types at the interfaces: a *token* form represented by a limited private type (TOKEN_TYPE) and an *identifier text* form represented by STRING (restricted to strings with the syntax of Ada identifiers, indicated by the use of the subtype IDENTIFIER_TEXT). Identifier values can be manipulated using either form; there are interfaces to transform the external representation of identifier values from identifier text form to token form and vice versa.

   The value of every TOKEN_TYPE variable is initially a distinguished *undefined token*; the variable is given a valid token value by interfaces that copy tokens, transform identifier text forms into tokens, or otherwise produce tokens. There is no text form for an undefined token. No interface can produce the undefined token; no interface can use an undefined token as a legitimate input value. The undefined token can only be used as the value of an **in out** parameter in an interface; in such a case, the purpose of the interface is to update the parameter value but not use it as input. All interfaces guarantee that attempts to use an undefined token in any other way result in raising the TOKEN_ERROR exception.

Downloaded from http://www.everyspec.com

If an identifier text form is transformed to token form and then transformed back, the resulting identifier text will be the original identifier text with any originally lower case letters transformed to upper case.

Identifier equality holds among tokens and identifier text forms when the tokens and/or identifier text forms designate the same identifier value. Identifier equality follows the Ada rule when applied to the identifier text form: two identifier text forms which differ only in the case of alphabetic characters designate the same identifier value and therefore satisfy identifier equality. A token obtained from an identifier text form designates the same identifier value as the identifier text form and is therefore identifier-equal to the identifier text form.

Identifiers are not only used as item values; they also serve as item names. Interfaces that refer to item names are provided with both forms of external types, and the rule of identifier equality applies. Therefore, no two distinct items in a linear list may have names that are identifier-equal.

e. The value of a list item may be itself a linear list. Whether or not such a list item is named, its linear list value may be named, unnamed or empty. The linear list value is called a *nested sublist* of the linear list containing the list item. Any linear list containing a list item can be viewed as a *nested list structure* consisting of the linear list with all of its nested sublists (and all of their nested sublists, recursively including all the nested sublists).

The only way to refer to linear list values at the interfaces is through the use of the limited private type LIST_TYPE. Every LIST_TYPE value includes a single, outermost linear list, which may constitute a nested list structure. By convention, the list kind of a LIST_TYPE value's outermost linear list is said to be the list kind of the LIST_TYPE value itself.

Many list manipulations provided by the CAIS_LIST_MANAGEMENT interfaces require designating a list item by its position within its linear list. To provide the ability to apply such list manipulations at all levels of nesting of a nested list structure, every LIST_TYPE value includes not only a list structure but also a designation of the *current linear list* within the list structure, to which the list manipulations implicitly refer. Exactly one designation of a current linear list is associated with each LIST_TYPE value; a nested sublist within that value does not have its own designated current linear list.

The initial value of every Ada object of type LIST_TYPE represents the empty list; that is, the empty list is the Ada object's outermost linear list and is also designated as its current linear list.

CAIS_LIST_MANAGEMENT interfaces provide several operations upon the current linear list of LIST_TYPE values. These include such linear list manipulations as:

1. extracting values of items in a linear list,
2. extracting contiguous sequences of items from a linear list,
3. replacing or changing values of items in a linear list, and
4. inserting new items into a linear list.

They also include operations which change the designation of the current linear list of a LIST_TYPE value, such as:

Downloaded from http://www.everyspec.com

1. making a nested sublist of the current linear list the (new) current linear list, and
2. making the linear list containing the current linear list the (new) current linear list.

These two kinds of operations (linear list manipulations and current linear list designations) are independent in the following sense: linear list manipulations do not change the designation of the current linear list of a LIST_TYPE value and current linear list designations applied to a LIST_TYPE value do not change the nested list structure being represented.

For convenience, a STRING-valued external representation (restricted to strings with the syntax given in Table XV) is defined for linear lists.

TABLE XV. List External Representation BNF

```
list            ::= named_list
                    | unnamed_list
                    | empty_list
named_list      ::= ( named_item { , named_item } )
unnamed_list    ::= ( item_value { , item_value } )
empty_list      ::= ()
named_item      ::= item_name => item_value
item_name       ::= identifier
item_value      ::= list
                    | quoted_string
                    | integer_number
                    | float_number
                    | identifier
integer_number  ::= [-] integer
float_number    ::= [-] decimal_literal
quoted_string   ::= string_literal
```

See Appendix D for a description of the notation used.

It is legal for blanks, format effectors and/or non-printing characters to occur between the syntactic constituents of LIST_TEXT representations of list values.

There are interfaces to transform the external representation of a linear list from list form to text form and vice versa. If a list representation is transformed to list text form the result is a canonical list text representation. The *canonical list text representation* of a list consists of the list text of its list items, composed

Downloaded from http://www.everyspec.com

according to the syntax of Table XV, and in addition adheres to the following rules:

1. There are no blanks, format effectors or non-printing characters between the syntactic constituents.
2. For an integer value of a list item, the list text representation is the decimal representation of its numeric value without leading zeroes. Negative values have a leading minus sign; positive values are unsigned.
3. For a floating point value of a list item, the list text representation is the string image of its numeric value in decimal notation with a format as obtained under default settings of the FORE, AFT, and EXP parameters in PUT operations of Ada TEXT_IO (see [1815A] 14.3.8), except that the value of 'DIGITS to be assumed for AFT is LIST_MAXIMUM_DIGITS (see CAIS_PRAGMATICS, Section 5.7, page 514) and the FORE field does not contain any leading spaces.
4. For a string value of a list item, the list text representation is the string literal representing the string value (i.e., the string value enclosed by quotation characters and with inner quotation characters doubled). The replacement character "%" may be used uniformly instead of the quotation character as described in [1815A] 2.10.
5. For an identifier value of a list item or the name of a list item, the list text representation is the identifier string (in upper case characters, without enclosing quotation characters).
6. For a linear list value of a list item, the list text representation is (recursively) the list text of the list value.

If a list text representation of a linear list is transformed to list form, the result is a LIST_TYPE value whose current (and outermost) linear list is the linear list described in the list text representation.

List equality is defined according to the following rule:

Two linear lists are equal if and only if:

1. both lists are of the same kind (i.e., named, unnamed or empty), and
2. both lists contain the same number of list items, and
3. in the case of named lists, for each position in the list, the names of the list items at this position are equal under identifier-equality, and
4. for each position in the list, the values of the list items at this position are of the same kind and are equal according to the appropriate form of equality (as described above), i.e.:
   - (a) for identifier items, identifier equality;
   - (b) for string items, string equality;
   - (c) for integer items, integer equality;
   - (d) for floating point items, floating point equality;
   - (e) for list items (whose values are in turn lists), list equality as defined in this rule.

Downloaded from http://www.everyspec.com

Equality of lists involving floating point items should be applied with considerable caution and awareness of all the issues documented in [1815A], Section 4.5.7, regarding the accuracy of relational operations with real operands and the discussion above defining floating point equality for floating point list items.

Downloaded from http://www.everyspec.com

5.4.1 Package CAIS_LIST_MANAGEMENT

This package defines types, subtypes, constants, exceptions and general list manipulation interfaces. The latter are supplemented by generic subpackages for the manipulation of list items of numeric type.

5.4.1.1 Types, subtypes, constants and exceptions

```
type     LIST_TYPE        is limited private;

subtype  LIST_TEXT        is STRING;

type     LIST_SIZE        is range 0 .. CAIS_PRAGMATICS.LIST_LENGTH;
subtype  POSITION_COUNT   is LIST_SIZE range 1 .. LIST_SIZE'LAST;
subtype  INSERT_COUNT     is LIST_SIZE range 0 .. LIST_SIZE'LAST - 1;

type     LIST_KIND        is (UNNAMED, NAMED, EMPTY);

type     ITEM_KIND        is (LIST_ITEM_KIND, STRING_ITEM_KIND,
                              INTEGER_ITEM_KIND, FLOAT_ITEM_KIND,
                              IDENTIFIER_ITEM_KIND);
```

LIST_TYPE describes the values used for lists at the list manipulation interfaces. LIST_TEXT describes the values used for the text representation of lists. The interfaces enforce the syntax of Table XV, page 422, for such text values. LIST_SIZE describes the values that can be used to indicate the number of items in a linear list. POSITION_COUNT describes the values that can be used to indicate the position of an item in a non-empty linear list. INSERT_COUNT describes the values that can be used to indicate the position in a linear list after which items are to be inserted. LIST_KIND enumerates the kinds of lists. ITEM_KIND enumerates the kinds of list items.

```
type     TOKEN_TYPE         is limited private;
subtype  IDENTIFIER_TEXT    is STRING;
```

TOKEN_TYPE describes the token values used at the interfaces to designate identifiers. IDENTIFIER_TEXT describes the text values used at the interfaces to designate identifiers. The interfaces enforce the syntax of Ada identifiers for such text values.

```
EMPTY_LIST:   constant   LIST_TYPE;
```

EMPTY_LIST is a deferred constant denoting the value of an empty list. The value of the function IS_EQUAL(EMPTY_LIST,X) is TRUE for any object X of type LIST_TYPE whose value is an empty list.

```
ITEM_KIND_ERROR:       exception;
LIST_KIND_ERROR:       exception;
LIST_POSITION_ERROR:   exception;
NAMED_LIST_ERROR:      exception;
SEARCH_ERROR:          exception;
SYNTAX_ERROR:          exception;
TOKEN_ERROR:           exception;
```

ITEM_KIND_ERROR is raised if the kind of item is incorrect for the operation being attempted.

Downloaded from http://www.everyspec.com

LIST_KIND_ERROR is raised if the kind of list is incorrect for the operation being attempted.

LIST_POSITION_ERROR is raised if an attempt is made to specify a list item's position larger than the list's length, a start position larger than the list's length or an end position less than the specified start position.

NAMED_LIST_ERROR is raised if an attempt is made to specify an item by name in an unnamed list or to construct a linear list with more than one item with the same name.

SEARCH_ERROR is raised if a search for an item fails because the item is not present in a non-empty list.

SYNTAX_ERROR is raised if an attempt is made to use IDENTIFIER_TEXT values that do not satisfy the syntax of an Ada identifier or LIST_TEXT values that do not satisfy the syntax defined in Table XV, page 422.

TOKEN_ERROR is raised if an undefined token value is used where a valid token is required.

Downloaded from http://www.everyspec.com

5.4.1.2 Copying a list

```
procedure COPY_LIST (FROM_LIST: in      LIST_TYPE;
                     TO_LIST:   in out  LIST_TYPE);
```

Purpose:

This procedure returns in the parameter TO_LIST a copy of the current linear list value of the parameter FROM_LIST. In the newly copied TO_LIST, the outermost list is the current linear list. Subsequent modifications of either list do not affect the other list.

Parameters:

FROM_LIST	is the list whose current linear list is to be copied.
TO_LIST	is the list returned as a copy of the current linear list of FROM_LIST.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.4.1.3 Making a list empty

```
procedure SET_TO_EMPTY_LIST (LIST: in out LIST_TYPE);
```

Purpose:

This procedure resets the parameter LIST so that its current (and outermost) linear list is the empty list.

Parameter:

LIST	is the list to be made empty.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.4.1.4 Converting from text to list form

```
procedure CONVERT_TEXT_TO_LIST (LIST_STRING: in      LIST_TEXT;
                                LIST:        in out  LIST_TYPE);
```

Purpose:

This procedure converts the text representation of a list into the private list representation. It establishes the current (and outermost) linear list of LIST to be a named, unnamed, or empty list. The individual list items are classified according to their text representation. For a numeric item value, the item is classified as an integer item if the numeric value can be interpreted as a (possibly negated) literal of universal_integer type; otherwise, the numeric item is classified as a floating point item. Blanks, format effectors and non-printing characters are allowed between syntactic elements in the value of the parameter LIST_STRING.

Parameters:

LIST_STRING is the text form to be interpreted as a linear list value.

LIST is the list whose current and outermost linear list is built and returned according to the contents of LIST_STRING.

Exceptions:

SYNTAX_ERROR
is raised if the value of the parameter LIST_STRING does not conform to the syntax of Table XV, page 422.

CAPACITY_ERROR
is raised if the length of the LIST_STRING parameter exceeds the value of the constant CAIS_PRAGMATICS.LIST_TEXT_LENGTH or if its value contains any name or value which cannot be represented in the LIST result due to exceeding implementation limits for the particular CAIS implementation. (See CAIS_PRAGMATICS, Section 5.7, page 514.)

NAMED_LIST_ERROR
is raised if the LIST_STRING parameter designates a named linear (sub)list with two or more items of the same name.

Downloaded from http://www.everyspec.com

5.4.1.5 Converting a list to its text representation

```
function TEXT_FORM (LIST: in LIST_TYPE)
    return LIST_TEXT;
```

Purpose:

This function returns the text representation of the value of the current linear list of LIST. The result is in the canonical list text representation defined in Section 5.4.

Parameter:

LIST — is the list whose current linear list is to be converted to its text representation.

Exception:

CAPACITY_ERROR
is raised if the length of the canonical list text representation to be returned exceeds the value of the constant CAIS_PRAGMATICS.LIST_TEXT_LENGTH. (See CAIS_PRAGMATICS, Section 5.7, page 514.)

Downloaded from http://www.everyspec.com

5.4.1.6 Determining the equality of two lists

```
function IS_EQUAL (LIST1: in LIST_TYPE;
                   LIST2: in LIST_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if the values of the two current linear lists of LIST1 and LIST2 are equal according to list equality (see Section 5.4); otherwise, it returns FALSE.

Parameters:

LIST1, LIST2 are the lists containing the current linear lists whose equality is to be determined.

Exceptions:

None.

Notes:

Equality of lists involving floating point items should be applied with considerable caution and awareness of all the issues documented in [1815A], Section 4.5.7, regarding the accuracy of relational operations with real operands, and the discussion in Section 5.4 defining floating point equality for floating point list items.

Downloaded from http://www.everyspec.com

5.4.1.7 Deleting an item from a linear list

```
procedure DELETE (LIST:          in out LIST_TYPE;
                  ITEM_POSITION: in     POSITION_COUNT);

procedure DELETE (LIST:          in out LIST_TYPE;
                  ITEM_NAME:     in     IDENTIFIER_TEXT);

procedure DELETE (LIST:          in out LIST_TYPE;
                  ITEM_NAME:     in     TOKEN_TYPE);
```

Purpose:

This procedure deletes the item specified by ITEM_POSITION or ITEM_NAME from the current linear list of LIST. If this was the only item in the linear list, the kind of the linear list changes to EMPTY.

Parameters:

LIST — is the list containing the current linear list from which the item will be deleted.

ITEM_POSITION — is the position within the current linear list of the item to be deleted.

ITEM_NAME — is the name of the list item to be deleted.

Exceptions:

LIST_KIND_ERROR — is raised if the current linear list of LIST is empty.

LIST_POSITION_ERROR — is raised if ITEM_POSITION has a value larger than the current length of the current linear list of LIST.

SYNTAX_ERROR — is raised if the identifier-text value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR — is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR — is raised if the parameter ITEM_NAME is used and the current linear list of LIST is unnamed.

SEARCH_ERROR — is raised if there is no item in the current linear list of LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.8 Determining the kind of list

```
function KIND_OF_LIST (LIST: in LIST_TYPE)
    return LIST_KIND;
```

Purpose:

This function returns the kind of the current linear list of LIST; the value returned is either UNNAMED, NAMED or EMPTY.

Parameter:

LIST is the list whose current linear list is of interest.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.4.1.9 Determining the kind of list item

```
function KIND_OF_ITEM (LIST:          in LIST_TYPE;
                       ITEM_POSITION: in POSITION_COUNT)
     return ITEM_KIND;

function KIND_OF_ITEM (LIST:          in LIST_TYPE;
                       ITEM_NAME:     in IDENTIFIER_TEXT)
     return ITEM_KIND;

function KIND_OF_ITEM (LIST:          in LIST_TYPE;
                       ITEM_NAME:     in TOKEN_TYPE)
     return ITEM_KIND;
```

Purpose:

This function returns the kind of an item in the current linear list of LIST.

Parameters:

LIST — is the list of interest.

ITEM_POSITION — is the position of the item of interest within the current linear list of LIST.

ITEM_NAME — is the name of the list item of interest within the current linear list of LIST.

Exceptions:

LIST_KIND_ERROR — is raised if the current linear list of LIST is empty.

LIST_POSITION_ERROR — is raised if ITEM_POSITION has a value larger than the current length of the current linear list of LIST.

SYNTAX_ERROR — is raised if the identifier-text value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR — is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR — is raised if the parameter ITEM_NAME is used and the current linear list of LIST is unnamed.

SEARCH_ERROR — is raised if there is no item in the current linear list of LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.10 Inserting a sequence of items into a linear list

```
procedure SPLICE (LIST:         in out LIST_TYPE;
                  POSITION:     in     INSERT_COUNT;
                  SOURCE_LIST:  in     LIST_TYPE);
```

Purpose:

This procedure allows the items of the current linear list of one list to be inserted into the current linear list of another list. Copies of the items in the linear list to be inserted will become items in the resulting linear list. The copied items will appear, in order, immediately following the item position designated by POSITION. Subsequent modifications to the value of LIST or to the value of SOURCE_LIST do not affect the other list.

Parameters:

LIST
: is the list into whose current linear list the copied items are to be inserted.

POSITION
: is the position in LIST's current linear list after which the new items will be inserted.

SOURCE_LIST
: is the list whose current linear list supplies the items to be inserted.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear lists of LIST and SOURCE_LIST are not of the same kind and neither of them is an empty list.

LIST_POSITION_ERROR
: is raised if POSITION has a value larger than the current length of the current linear list of LIST.

CAPACITY_ERROR
: is raised if the number of items in the resulting linear list would exceed the value of the constant CAIS_PRAGMATICS.LIST_LENGTH. (See CAIS_PRAGMATICS, Section 5.7, page 514.)

NAMED_LIST_ERROR
: is raised if the current linear lists of LIST and SOURCE_LIST are both named and contain an item of the same name.

Downloaded from http://www.everyspec.com

5.4.1.11 Concatenating two linear lists

```
procedure CONCATENATE_LISTS (FRONT:  in     LIST_TYPE;
                             BACK:   in     LIST_TYPE;
                             RESULT: in out LIST_TYPE);
```

Purpose:

This procedure returns in RESULT a list constructed by concatenating the current linear list of BACK to the end of the current linear list of FRONT. The current linear lists of FRONT and BACK must be of the same kind or one must be an empty list. The values of FRONT and BACK are not affected. Subsequent modifications to the value of FRONT or of BACK or to the value of the returned RESULT list do not affect either of the other (unmodified) lists.

Parameters:

FRONT is the first list whose current linear list is to be concatenated.

BACK is the second list whose current linear list is to be concatenated.

RESULT is the list produced by the concatenation; its outermost linear list has as its initial items the items in the current linear list of FRONT and as the rest of its items the items in the current linear list of BACK.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear lists of FRONT and BACK are not of the same kind and neither of them is an empty list.

CAPACITY_ERROR
is raised if the number of items in the resulting linear list would exceed the value of the constant CAIS_PRAGMATICS.LIST_LENGTH. (See CAIS_PRAGMATICS, Section 5.7, page 514.)

NAMED_LIST_ERROR
is raised if the current linear lists of FRONT and BACK are both named and contain an item of the same name.

Downloaded from http://www.everyspec.com

5.4.1.12 Extracting a sequence of items from a linear list

```
procedure EXTRACT_LIST (LIST:           in     LIST_TYPE;
                        START_POSITION: in     POSITION_COUNT;
                        END_POSITION:   in     POSITION_COUNT;
                        RESULT_LIST:    in out LIST_TYPE);
```

Purpose:

This procedure extracts a sequence of items from the current linear list of a list, forming a new list from them. The items to be extracted are those in the positions from START_POSITION through END_POSITION inclusive. Copies of the extracted items form the current (and outermost) linear list of RESULT_LIST.

Parameters:

LIST — is the list whose current linear list supplies the items to be extracted.

START_POSITION — is the position (within LIST's current linear list) of the first in the sequence of items to be extracted.

END_POSITION — is the position (within LIST's current linear list) of the last in the sequence of items to be extracted.

RESULT_LIST — is the list constructed from copies of the extracted items.

Exception:

LIST_POSITION_ERROR — is raised if START_POSITION is greater than END_POSITION or if either START_POSITION or END_POSITION is greater than the number of items in the current linear list of LIST.

Downloaded from http://www.everyspec.com

5.4.1.13 Determining the length of a linear list

```
function NUMBER_OF_ITEMS (LIST: in LIST_TYPE)
    return LIST_SIZE;
```

Purpose:

This function returns a count of the number of items in the current linear list of LIST. If the current linear list is empty, zero is returned.

Parameter:

LIST	is the list whose current linear list is being measured.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.4.1.14 Determining the position of the current linear list

```
function POSITION_OF_CURRENT_LIST (LIST: in LIST_TYPE)
    return POSITION_COUNT;
```

Purpose:

This function returns the position (within the innermost linear list containing the current linear list) of the item whose value is the current linear list.

Parameter:

LIST — is the list whose current linear list is the value of some list item.

Exception:

LIST_POSITION_ERROR
is raised if the current linear list of LIST is the outermost linear list of LIST.

Downloaded from http://www.everyspec.com

5.4.1.15 Determining whether the current linear list is outermost

```
function CURRENT_LIST_IS_OUTERMOST (LIST: in LIST_TYPE)
    return BOOLEAN;
```

Purpose:

This function returns TRUE if the current linear list of LIST is the outermost linear list of LIST; otherwise it returns FALSE.

Parameter:

LIST is the list whose current linear list may either be the outermost linear list or the value of some list item.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.4.1.16 Making the next outer linear list current

```
procedure MAKE_CONTAINING_LIST_CURRENT (IN_LIST: in out LIST_TYPE);
```

Purpose:

This procedure causes the innermost list containing the current linear list to become the (new) current linear list.

Parameter:

IN_LIST is the list whose current linear list is of interest.

Exception:

LIST_POSITION_ERROR
is raised if the current linear list of LIST is the outermost linear list of LIST.

Downloaded from http://www.everyspec.com

5.4.1.17 Making a nested sublist the current linear list

```
procedure MAKE_THIS_ITEM_CURRENT (IN_LIST:       in out LIST_TYPE;
                                  ITEM_POSITION: in     POSITION_COUNT);

procedure MAKE_THIS_ITEM_CURRENT (IN_LIST:       in out LIST_TYPE;
                                  ITEM_NAME:     in     IDENTIFIER_TEXT);

procedure MAKE_THIS_ITEM_CURRENT (IN_LIST:       in out LIST_TYPE;
                                  ITEM_NAME:     in     TOKEN_TYPE);
```

Purpose:

This procedure causes the list value of an item in the current linear list of LIST to become the (new) current linear list.

Parameters:

IN_LIST
: is the list whose current linear list is of interest.

ITEM_POSITION
: is the position (in the current linear list) of the list-kind item whose value is to become the (new) current linear list.

ITEM_NAME
: is the name of the list-kind item (in the current linear list) whose value is to become the (new) current linear list.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of IN_LIST is empty.

LIST_POSITION_ERROR
: is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
: is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
: is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR
: is raised if the parameter ITEM_NAME is used and the current linear list of IN_LIST is unnamed.

ITEM_KIND_ERROR
: is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not a list.

Downloaded from http://www.everyspec.com

SEARCH_ERROR
is raised if there is no item in the current linear list of IN_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.18 Determining the length of the text form of a list or a list item

```
function TEXT_LENGTH (LIST:           in LIST_TYPE)
    return CAIS_POSITIVE;

function TEXT_LENGTH (LIST:           in LIST_TYPE;
                      ITEM_POSITION: in POSITION_COUNT)
    return CAIS_POSITIVE;

function TEXT_LENGTH (LIST:           in LIST_TYPE;
                      ITEM_NAME:     in IDENTIFIER_TEXT)
    return CAIS_POSITIVE;

function TEXT_LENGTH (LIST:           in LIST_TYPE;
                      ITEM_NAME:     in TOKEN_TYPE)
    return CAIS_POSITIVE;
```

Purpose:

This function returns the length of the text form of the current linear list of LIST [first interface], or the length of the text form of the value of a list item (of the current linear list) identified by ITEM_POSITION or ITEM_NAME [last three interfaces].

Parameters:

LIST is the list whose current linear list is of interest.

ITEM_POSITION
is the position within the current linear list that identifies the item of interest.

ITEM_NAME is the name of the list item of interest.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear list of LIST is empty (last three interfaces).

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of LIST.

SYNTAX_ERROR
is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of LIST is unnamed.

Downloaded from http://www.everyspec.com

SEARCH_ERROR
: is raised if there is no item in the current linear list of LIST with the name ITEM_NAME.

Notes:

The text form of every value is non-null, and therefore the result is always positive. This is even true of the empty list, which is represented by the string value designated by the Ada string literal ''()'' and therefore has length 2.

Downloaded from http://www.everyspec.com

5.4.1.19 Determining the name of a named item

```
procedure GET_ITEM_NAME (LIST:          in     LIST_TYPE;
                         ITEM_POSITION: in     POSITION_COUNT;
                         NAME:          in out TOKEN_TYPE);
```

Purpose:

This procedure returns, in NAME, the token form of the name of item that is in the position indicated by ITEM_POSITION in the (named) current linear list of the LIST.

Parameters:

LIST
: is the list whose current linear list is of interest.

ITEM_POSITION
: is the position within the current linear list that identifies the item.

NAME
: is the token representation of the name of the item in the named current linear list.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of LIST is not a named list.

LIST_POSITION_ERROR
: is raised if ITEM_POSITION has a value larger than the current length of the current linear list of LIST.

Downloaded from http://www.everyspec.com

5.4.1.20 Determining the position of a named item

```
function POSITION_BY_NAME (LIST:      in LIST_TYPE;
                           ITEM_NAME: in IDENTIFIER_TEXT)
     return POSITION_COUNT;

function POSITION_BY_NAME (LIST:      in LIST_TYPE;
                           ITEM_NAME: in TOKEN_TYPE)
     return POSITION_COUNT;
```

Purpose:

This function returns the position at which an item with the given name ITEM_NAME is located in the current linear list of LIST.

Parameters:

LIST — is the list in whose current linear list the named item is to be located.

ITEM_NAME — is the name of the item to be located.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear list of LIST is empty.

SYNTAX_ERROR
is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of LIST is unnamed.

SEARCH_ERROR
is raised if there is no item in the current linear list of LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.21 Package CAIS_LIST_ITEM

This is a package for manipulating list items whose values are list items. The exceptions raised by all subprograms in this package are defined in the packages CAIS_LIST_MANAGEMENT and CAIS_PRAGMATICS.

Downloaded from http://www.everyspec.com

5.4.1.21.1 Extracting a list value from a list item

```
procedure EXTRACT_VALUE (FROM_LIST:     in     LIST_TYPE;
                         ITEM_POSITION: in     POSITION_COUNT;
                         VALUE:         in out LIST_TYPE);

procedure EXTRACT_VALUE (FROM_LIST:     in     LIST_TYPE;
                         ITEM_NAME:     in     IDENTIFIER_TEXT;
                         VALUE:         in out LIST_TYPE);

procedure EXTRACT_VALUE (FROM_LIST:     in     LIST_TYPE;
                         ITEM_NAME:     in     TOKEN_TYPE;
                         VALUE:         in out LIST_TYPE);
```

Purpose:

This procedure locates a list-valued item in the current linear list of FROM_LIST and returns a copy of the list value as the current (and outermost) linear list of VALUE. Subsequent modification to the value of FROM_LIST or to the value returned in VALUE does not affect the other value.

Parameters:

FROM_LIST is the list whose current linear list contains the item to be extracted.

ITEM_POSITION
is the position within the current linear list that identifies the item whose value is to be extracted.

ITEM_NAME is the name of the item whose value is to be extracted.

VALUE is the list value extracted from the designated item.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear list of FROM_LIST is empty.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of FROM_LIST.

SYNTAX_ERROR
is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of FROM_LIST is unnamed.

Downloaded from http://www.everyspec.com

ITEM_KIND_ERROR
is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not a list.

SEARCH_ERROR
is raised if there is no item in the current linear list of FROM_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.21.2 Replacing a list value in a list item

```
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_POSITION:  in     POSITION_COUNT;
                   VALUE:          in     LIST_TYPE);

procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     IDENTIFIER_TEXT;
                   VALUE:          in     LIST_TYPE);

procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     TOKEN_TYPE;
                   VALUE:          in     LIST_TYPE);
```

Purpose:

This procedure replaces the value of a list-valued item in the current linear list of IN_LIST with the current linear list of VALUE. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value.

Parameters:

IN_LIST
: is the list whose current linear list contains the item whose value is to be replaced.

ITEM_POSITION
: is the position within the current linear list that identifies the item whose value is to be replaced.

ITEM_NAME
: is the name of the item whose value is to be replaced.

VALUE
: is the list value whose current linear list is to become the new value of the designated item.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of IN_LIST is empty.

LIST_POSITION_ERROR
: is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
: is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
: is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

Downloaded from http://www.everyspec.com

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of IN_LIST is unnamed.

ITEM_KIND_ERROR
is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not a list.

SEARCH_ERROR
is raised if there is no item in the current linear list of IN_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.21.3 Inserting a list-valued item into a linear list

```
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  VALUE:     in     LIST_TYPE);

procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     IDENTIFIER_TEXT;
                  VALUE:     in     LIST_TYPE);

procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     TOKEN_TYPE;
                  VALUE:     in     LIST_TYPE);
```

Purpose:

This procedure inserts a list-valued item into the current linear list of IN_LIST; the new list item will be positioned after the list item specified by POSITION. The list value of the item to be inserted is the current linear list of the VALUE parameter. A value of zero in POSITION specifies a position at the head of the current linear list. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value. If the current linear list of IN_LIST is empty, it will be a named list after the successful completion of the call, if the second or third interface is used, and an unnamed list otherwise.

Parameters:

IN_LIST — is the list into whose current linear list the item will be inserted.

POSITION — is the position in the current linear list after which the item is to be inserted.

NAME — is the name of the new item to be inserted.

VALUE — is the list value whose current linear list is the value of the new item to be inserted.

Exceptions:

LIST_KIND_ERROR
is raised if an attempt is made to insert an item by NAME into an unnamed list or, conversely, if an attempt is made to insert an item without NAME into a named list.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
is raised if the value of the parameter NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

Downloaded from http://www.everyspec.com

TOKEN_ERROR
is raised if the NAME of type TOKEN_TYPE is an undefined token.

CAPACITY_ERROR
is raised if the number of items in the resulting linear list would exceed the value of the constant CAIS_PRAGMATICS.LIST_LENGTH. (See CAIS_PRAGMATICS, Section 5.7, page 514.)

NAMED_LIST_ERROR
is raised if the current linear list of IN_LIST is a named list that already contains an item with the name given by NAME.

Downloaded from http://www.everyspec.com

5.4.1.21.4 Locating a list-valued item by value within a linear list

```
function POSITION_BY_VALUE
            (LIST:           in LIST_TYPE;
             VALUE:          in LIST_TYPE;
             START_POSITION: in POSITION_COUNT := POSITION_COUNT'FIRST;
             END_POSITION:   in POSITION_COUNT := POSITION_COUNT'LAST)
    return POSITION_COUNT;
```

Purpose:

This function returns the position in the current linear list of LIST of the next list-valued item whose value equals that of the current linear list of the VALUE parameter under list equality (see Section 5.4, page 423). The search begins at the START_POSITION and ends when either an item whose value equals the current linear list of VALUE is found, the last item of the list has been examined, or the item at the END_POSITION has been examined, whichever comes first.

Parameters:

LIST — is the list in whose current linear list the item of interest is to be located.

VALUE — is the list value whose current linear list is the value of interest.

START_POSITION
: is the position of the first item in the current linear list of LIST to be considered in the search.

END_POSITION
: is the position beyond which the search will not proceed; the search may terminate prior to reaching END_POSITION should the sought item be found or should the last element of the list be considered.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of LIST is empty.

LIST_POSITION_ERROR
: is raised if START_POSITION specifies a value larger than the current length of the current linear list of LIST, or if END_POSITION is less than START_POSITION.

SEARCH_ERROR
: is raised if there is no item in the current linear list of LIST within the region specified by START_POSITION and END_POSITION that has the value specified by VALUE.

Downloaded from http://www.everyspec.com

Notes:

Determining the position by value of a list involving floating point items should be applied with considerable caution and awareness of all the issues documented in [1815A], Section 4.5.7, regarding the accuracy of relational operations with real operands and the discussion in Section 5.4, page 420, defining floating point equality for floating point list items.

Downloaded from http://www.everyspec.com

5.4.1.22 Package CAIS_IDENTIFIER_ITEM

This package provides interfaces for the manipulation of identifier values and list items whose values are identifiers. The exceptions raised by all subprograms in this package are defined in the packages CAIS_LIST_MANAGEMENT and CAIS_PRAGMATICS.

Identifier values are represented at the interfaces as values of the types TOKEN_TYPE or IDENTIFIER_TEXT.

Downloaded from http://www.everyspec.com

5.4.1.22.1 Copying a token

```
procedure COPY_TOKEN (FROM_TOKEN: in     TOKEN_TYPE;
                      TO_TOKEN:   in out TOKEN_TYPE);
```

Purpose:

This procedure returns in TO_TOKEN a copy of the token in FROM_TOKEN.

Parameters:

FROM_TOKEN is the token to be copied.

TO_TOKEN is the copied token to be returned.

Exception:

TOKEN_ERROR
is raised if FROM_TOKEN is an undefined token.

Downloaded from http://www.everyspec.com

5.4.1.22.2 Converting an identifier from text to token form

```
procedure CONVERT_TEXT_TO_TOKEN (IDENTIFIER: in     IDENTIFIER_TEXT;
                                 TOKEN:      in out TOKEN_TYPE);
```

Purpose:

This procedure converts the text representation of an identifier into the corresponding token representation.

Parameters:

IDENTIFIER is the text to be converted to a token.

TOKEN is the token corresponding to the value of IDENTIFIER.

Exceptions:

SYNTAX_ERROR
is raised if the value of the parameter IDENTIFIER does not conform to the syntax of an Ada identifier.

CAPACITY_ERROR
is raised if the length of the IDENTIFIER parameter exceeds the value of the constant CAIS_PRAGMATICS.IDENTIFIER_ITEM_LENGTH (see CAIS_PRAGMATICS, Section 5.7, page 513).

Downloaded from http://www.everyspec.com

5.4.1.22.3 Converting an identifier from token to text form

```
function TEXT_FORM (TOKEN: in TOKEN_TYPE)
    return IDENTIFIER_TEXT;
```

Purpose:

This function returns the text representation of the token value of the TOKEN parameter. The result has the syntax of an Ada identifier.

Parameter:

TOKEN is the identifier expressed as a token.

Exception:

TOKEN_ERROR
is raised if the TOKEN is an undefined token.

Notes:

The result does not contain any lower-case characters.

Downloaded from http://www.everyspec.com

5.4.1.22.4 Determining the equality of two identifier tokens

```
function IS_EQUAL (TOKEN1: in TOKEN_TYPE;
                   TOKEN2: in TOKEN_TYPE)
     return BOOLEAN;
```

Purpose:

This function returns TRUE if the two identifier tokens TOKEN1 and TOKEN2 are identifier-equal; otherwise, it returns FALSE.

Parameters:

TOKEN1, TOKEN2
: are the identifier tokens whose equality is to be determined.

Exception:

TOKEN_ERROR
: is raised if either TOKEN1 or TOKEN2 is an undefined token.

Downloaded from http://www.everyspec.com

5.4.1.22.5 Extracting an identifier value from a list item

```
procedure EXTRACT_VALUE (FROM_LIST:     in     LIST_TYPE;
                         ITEM_POSITION: in     POSITION_COUNT;
                         VALUE:         in out TOKEN_TYPE);

procedure EXTRACT_VALUE (FROM_LIST:     in     LIST_TYPE;
                         ITEM_NAME:     in     IDENTIFIER_TEXT;
                         VALUE:         in out TOKEN_TYPE);

procedure EXTRACT_VALUE (FROM_LIST:     in     LIST_TYPE;
                         ITEM_NAME:     in     TOKEN_TYPE;
                         VALUE:         in out TOKEN_TYPE);
```

Purpose:

This procedure locates an identifier-valued item in the current linear list of FROM_LIST and returns a copy of the identifier value (in token form) in VALUE.

Parameters:

FROM_LIST is the list whose current linear list contains the item to be extracted.

ITEM_POSITION
is the position within the current linear list that identifies the item whose value is to be extracted.

ITEM_NAME is the name of the item whose value is to be extracted.

VALUE is the identifier-value (in token form) extracted from the designated item.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear list of FROM_LIST is empty.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of FROM_LIST.

SYNTAX_ERROR
is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of FROM_LIST is unnamed.

Downloaded from http://www.everyspec.com

ITEM_KIND_ERROR
is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not an identifier.

SEARCH_ERROR
is raised if there is no item in the current linear list of FROM_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.22.6 Replacing an identifier value in a list item

```
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_POSITION: in     POSITION_COUNT;
                   VALUE:         in     TOKEN_TYPE);

procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:     in     IDENTIFIER_TEXT;
                   VALUE:         in     TOKEN_TYPE);

procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:     in     TOKEN_TYPE;
                   VALUE:         in     TOKEN_TYPE);
```

Purpose:

This procedure replaces the value of an identifier-valued item in the current linear list of IN_LIST with VALUE. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value.

Parameters:

IN_LIST — is the list whose current linear list contains the item whose value is to be replaced.

ITEM_POSITION — is the position within the current linear list that identifies the item whose value is to be replaced.

ITEM_NAME — is the name of the item whose value is to be replaced.

VALUE — is the new identifier value (in token form) of the designated item.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear list of IN_LIST is empty.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token or if VALUE is an undefined token.

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of IN_LIST is unnamed.

Downloaded from http://www.everyspec.com

ITEM_KIND_ERROR
is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not an identifier.

SEARCH_ERROR
is raised if there is no item in the current linear list of IN_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.22.7 Inserting an identifier-valued item into a linear list

```
procedure INSERT (IN_LIST:  in out LIST_TYPE;
                  POSITION: in     INSERT_COUNT;
                  VALUE:    in     TOKEN_TYPE);

procedure INSERT (IN_LIST:  in out LIST_TYPE;
                  POSITION: in     INSERT_COUNT;
                  NAME:     in     IDENTIFIER_TEXT;
                  VALUE:    in     TOKEN_TYPE);

procedure INSERT (IN_LIST:  in out LIST_TYPE;
                  POSITION: in     INSERT_COUNT;
                  NAME:     in     TOKEN_TYPE;
                  VALUE:    in     TOKEN_TYPE);
```

Purpose:

This procedure inserts a identifier-valued item into the current linear list of IN_LIST; the new list item will be positioned after the list item specified by POSITION. A value of zero in POSITION specifies a position at the head of the current linear list. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value. If the current linear list of IN_LIST is empty, it will be a named list after the successful completion of the call, if the second or third interface is used, and an unnamed list otherwise.

Parameters:

IN_LIST — is the list into whose current linear list the item will be inserted.

POSITION — is the position in the current linear list after which the item is to be inserted.

NAME — is the name of the new item to be inserted.

VALUE — is the identifier value (in token form) of the new item to be inserted.

Exceptions:

LIST_KIND_ERROR
is raised if an attempt is made to insert an item by NAME into an unnamed list or, conversely, if an attempt is made to insert an item without NAME into a named list.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
is raised if the value of the parameter NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

Downloaded from http://www.everyspec.com

TOKEN_ERROR
is raised if the NAME of type TOKEN_TYPE is an undefined token or if VALUE is an undefined token.

CAPACITY_ERROR
is raised if the number of items in the resulting linear list would exceed the value of the constant CAIS_PRAGMATICS.LIST_LENGTH. (See CAIS_PRAGMATICS, Section 5.7, page 514.)

NAMED_LIST_ERROR
is raised if the current linear list of IN_LIST is a named list that already contains an item with the name given by NAME.

Downloaded from http://www.everyspec.com

5.4.1.22.8 Locating an identifier-valued item by value within a linear list

```
function POSITION_BY_VALUE
            (LIST:           in LIST_TYPE;
             VALUE:          in TOKEN_TYPE;
             START_POSITION: in POSITION_COUNT := POSITION_COUNT'FIRST;
             END_POSITION:   in POSITION_COUNT := POSITION_COUNT'LAST)
    return POSITION_COUNT;
```

Purpose:

This function returns the position in the current linear list of LIST of the next identifier-valued item whose value equals that of the VALUE parameter under identifier equality (see Section 5.4, page 421). The search begins at the START_POSITION and ends when either an item whose value equals VALUE is found, the last item of the list has been examined, or the item at the END_POSITION has been examined, whichever comes first.

Parameters:

LIST
: is the list in whose current linear list the item of interest is to be located.

VALUE
: is the identifier value of interest.

START_POSITION
: is the position of the first item in the current linear list to be considered in the search.

END_POSITION
: is the position beyond which the search will not proceed; the search may terminate prior to reaching END_POSITION should the sought item be found or should the last element of the list be considered.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of LIST is empty.

LIST_POSITION_ERROR
: is raised if START_POSITION specifies a value larger than the current length of the current linear list of LIST, or if END_POSITION is less than START_POSITION.

TOKEN_ERROR
: is raised if VALUE is an undefined token.

SEARCH_ERROR
: is raised if there is no item in the current linear list of LIST within the region specified by START_POSITION and END_POSITION that has the value specified by VALUE.

Downloaded from http://www.everyspec.com

5.4.1.23 Generic package CAIS_INTEGER_ITEM

This is a generic package for manipulating list items whose values are integers. This package must be instantiated for the appropriate integer type (indicated by NUMBER in the specification). The exceptions raised by all subprograms in this package are defined in the packages CAIS_LIST_MANAGEMENT and CAIS_PRAGMATICS.

```
generic
    type NUMBER is range <>;
package CAIS_INTEGER_ITEM is
    -- Specifications of subprograms for this generic package.
end CAIS_INTEGER_ITEM;
```

Downloaded from http://www.everyspec.com

5.4.1.23.1 Converting an integer value to its canonical text representation

```
function TEXT_FORM (INTEGER_VALUE: in NUMBER)
    return STRING;
```

Purpose:

This function returns the canonical text form representation of the value of the INTEGER_VALUE parameter. The canonical text form representation is the string representation defined in Section 5.4.

Parameter:

INTEGER_VALUE
is the integer value whose external representation is to be returned.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.4.1.23.2 Extracting an integer value from a list item

```
function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_POSITION:  in POSITION_COUNT)
    return NUMBER;

function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_NAME:      in IDENTIFIER_TEXT)
    return NUMBER;

function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_NAME:      in TOKEN_TYPE)
    return NUMBER;
```

Purpose:

This function locates an integer-valued item in the current linear list of FROM_LIST and returns a copy of its numeric value.

Parameters:

FROM_LIST is the list whose current linear list contains the item whose value is to be extracted.

ITEM_POSITION
is the position within the current linear list that identifies the item whose value is to be extracted.

ITEM_NAME is the name of the item whose value is to be extracted.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear list of FROM_LIST is empty.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of FROM_LIST.

SYNTAX_ERROR
is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of FROM_LIST is unnamed.

Downloaded from http://www.everyspec.com

ITEM_KIND_ERROR
is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not an integer.

CONSTRAINT_ERROR
is raised if the value to be extracted violates the constraints of the type designated by NUMBER.

SEARCH_ERROR
is raised if there is no item in the current linear list of FROM_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.23.3 Replacing an integer value in a list item

```
procedure REPLACE (IN_LIST:       in out LIST_TYPE;
                   ITEM_POSITION: in     POSITION_COUNT;
                   VALUE:         in     NUMBER);

procedure REPLACE (IN_LIST:       in out LIST_TYPE;
                   ITEM_NAME:     in     IDENTIFIER_TEXT;
                   VALUE:         in     NUMBER);

procedure REPLACE (IN_LIST:       in out LIST_TYPE;
                   ITEM_NAME:     in     TOKEN_TYPE;
                   VALUE:         in     NUMBER);
```

Purpose:

This procedure replaces the value of an integer-valued item in the current linear list of IN_LIST with VALUE. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value.

Parameters:

IN_LIST
: is the list whose current linear list contains the item whose value is to be replaced.

ITEM_POSITION
: is the position within the current linear list that identifies the item whose value is to be replaced.

ITEM_NAME
: is the name of the item whose value is to be replaced.

VALUE
: is the new integer value of the designated item.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of IN_LIST is empty.

LIST_POSITION_ERROR
: is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
: is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
: is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

Downloaded from http://www.everyspec.com

CAPACITY_ERROR
is raised if the VALUE violates the constraints defined by the type CAIS_INTEGER (see Section 5.1.1, page 54).

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of IN_LIST is unnamed.

ITEM_KIND_ERROR
is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not an integer.

SEARCH_ERROR
is raised if there is no item in the current linear list of IN_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.23.4 Inserting an integer-valued item into a linear list

```
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  VALUE:     in     NUMBER);

procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     IDENTIFIER_TEXT;
                  VALUE:     in     NUMBER);

procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     TOKEN_TYPE;
                  VALUE:     in     NUMBER);
```

Purpose:

This procedure inserts an integer-valued item into the current linear list of IN_LIST; the new list item will be positioned after the list item specified by POSITION. A value of zero in POSITION specifies a position at the head of the current linear list. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value. If the current linear list of IN_LIST is empty, it will be a named list after the successful completion of the call, if the second or third interface is used, and an unnamed list otherwise.

Parameters:

IN_LIST is the list into whose current linear list the item will be inserted.

POSITION is the position in the current linear list after which the item is to be inserted.

NAME is the name of the new item to be inserted.

VALUE is the integer value of the new item to be inserted.

Exceptions:

LIST_KIND_ERROR
is raised if an attempt is made to insert an item by NAME into an unnamed list or, conversely, if an attempt is made to insert an item without NAME into a named list.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
is raised if the value of the parameter NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the NAME of type TOKEN_TYPE is an undefined token.

Downloaded from http://www.everyspec.com

CAPACITY_ERROR
is raised if the number of items in the resulting linear list would exceed the value of the constant CAIS_PRAGMATICS.LIST_LENGTH or if the VALUE to be inserted violates the constraints defined by the type CAIS_INTEGER (see Section 5.1.1, page 54).

NAMED_LIST_ERROR
is raised if the current linear list of IN_LIST is a named list that already contains an item with the name given by NAME.

Downloaded from http://www.everyspec.com

5.4.1.23.5 Locating an integer-valued item by value within a linear list

```
function POSITION_BY_VALUE
         (LIST:           in LIST_TYPE;
          VALUE:          in NUMBER;
          START_POSITION: in POSITION_COUNT := POSITION_COUNT'FIRST;
          END_POSITION:   in POSITION_COUNT := POSITION_COUNT'LAST).
    return POSITION_COUNT;
```

Purpose:

This function returns the position in the current linear list of LIST of the next integer-valued item whose value equals that of the VALUE parameter under integer equality (see Section 5.4, page 420). The search begins at the START_POSITION and ends when either an item whose value equals VALUE is found, the last item of the list has been examined, or the item at the END_POSITION has been examined, whichever comes first.

Parameters:

LIST
: is the list in whose current linear list the item of interest is to be located.

VALUE
: is the integer value of interest.

START_POSITION
: is the position of the first item in the current linear list to be considered in the search.

END_POSITION
: is the position beyond which the search will not proceed; the search may terminate prior to reaching END_POSITION should the sought item be found or should the last element of the list be considered.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of LIST is empty.

LIST_POSITION_ERROR
: is raised if START_POSITION specifies a value larger than the current length of the current linear list of LIST, or if END_POSITION is less than START_POSITION.

CAPACITY_ERROR
: is raised if the VALUE violates the constraints defined by the type CAIS_INTEGER (see Section 5.1.1, page 54).

SEARCH_ERROR
: is raised if there is no item in the current linear list of LIST within the region specified by START_POSITION and END_POSITION that has the value specified by VALUE.

Downloaded from http://www.everyspec.com

5.4.1.24 Generic package CAIS_FLOAT_ITEM

This is a generic package for manipulating list items whose values are floating point numbers. This package must be instantiated for the appropriate type (indicated by NUMBER in the specification). The exceptions raised by all subprograms in this package are defined in the packages CAIS_LIST_MANAGEMENT and CAIS_PRAGMATICS.

```
generic
    type NUMBER is digits <>;
package CAIS_FLOAT_ITEM is
    -- Specifications of subprograms for this generic package.
end CAIS_FLOAT_ITEM;
```

Use of floating point values in lists has certain adverse consequences for the meaning of list equality as detailed in Section 5.4. Users should be aware of the accuracy issues for relational operations between floating point values explained in [1815A], Section 4.5.7, in order to avoid erroneous assumptions about the equality of float items and lists involving such items.

See also the discussion of floating point values in Section 5.4, page 420, for further cautions.

Downloaded from http://www.everyspec.com

5.4.1.24.1 Converting a floating point value to its canonical text form

```
function TEXT_FORM (FLOAT_VALUE: in NUMBER)
    return STRING;
```

Purpose:

This function returns the canonical text form representation of the value of the FLOAT_VALUE parameter. The canonical text form representation is the string representation defined in Section 5.4.

Parameter:

FLOAT_VALUE
is the floating point item whose external representation is to be returned.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.4.1.24.2 Extracting a floating point value from a list item

```
function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_POSITION: in POSITION_COUNT)
    return NUMBER;

function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_NAME:      in IDENTIFIER_TEXT)
    return NUMBER;

function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_NAME:      in TOKEN_TYPE)
    return NUMBER;
```

Purpose:

This function locates a floating point-valued item in the current linear list of FROM_LIST and returns a copy of its numeric value.

Parameters:

FROM_LIST
: is the list whose current linear list contains the item whose value is to be extracted.

ITEM_POSITION
: is the position within the current linear list that identifies the item whose value is to be extracted.

ITEM_NAME
: is the name of the item whose value is to be extracted.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of FROM_LIST is empty.

LIST_POSITION_ERROR
: is raised if ITEM_POSITION has a value larger than the current length of the current linear list of FROM_LIST.

SYNTAX_ERROR
: is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
: is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR
: is raised if the parameter ITEM_NAME is used and the current linear list of FROM_LIST is unnamed.

Downloaded from http://www.everyspec.com

ITEM_KIND_ERROR
: is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not an floating point number.

CONSTRAINT_ERROR
: is raised if the value to be extracted violates the range constraints of the type designated by NUMBER.

SEARCH_ERROR
: is raised if there is no item in the current linear list of FROM_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.24.3 Replacing a floating point value in a list item

```
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_POSITION:  in     POSITION_COUNT;
                   VALUE:          in     NUMBER);

procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     IDENTIFIER_TEXT;
                   VALUE:          in     NUMBER);

procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     TOKEN_TYPE;
                   VALUE:          in     NUMBER);
```

Purpose:

This procedure replaces the value of a floating point-valued item in the current linear list of IN_LIST with VALUE. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value.

Parameters:

IN_LIST — is the list whose current linear list contains the item whose value is to be replaced.

ITEM_POSITION — is the position within the current linear list that identifies the item whose value is to be replaced.

ITEM_NAME — is the name of the item whose value is to be replaced.

VALUE — is the new floating point value of the designated item.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear list of IN_LIST is empty.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

CAPACITY_ERROR
is raised if the VALUE (see note below) violates the range constraints of the CAIS implementation.

Downloaded from http://www.everyspec.com

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of IN_LIST is unnamed.

ITEM_KIND_ERROR
is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not a floating point number.

SEARCH_ERROR
is raised if there is no item in the current linear list of IN_LIST with the name ITEM_NAME.

Notes:

The most restrictive range constraints applicable to VALUE can be inferred from CAIS_PRAGMATICS.LIST_MAXIMUM_DIGITS according to rules defined in [1815A] 3.5.9, 4.5.7 and 4.6. (See CAIS_PRAGMATICS, Section 5.7, page 514.)

Downloaded from http://www.everyspec.com

5.4.1.24.4 Inserting a floating point-valued item into a linear list

```
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  VALUE:     in     NUMBER);

procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     IDENTIFIER_TEXT;
                  VALUE:     in     NUMBER);

procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     TOKEN_TYPE;
                  VALUE:     in     NUMBER);
```

Purpose:

This procedure inserts an floating point-valued item into the current linear list of IN_LIST; the new list item will be positioned after the list item specified by POSITION. A value of zero in POSITION specifies a position at the head of the current linear list. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value. If the current linear list of IN_LIST is empty, it will be a named list after the successful completion of the call, if the second or third interface is used, and an unnamed list otherwise.

Parameters:

IN_LIST — is the list into whose current linear list the item will be inserted.

POSITION — is the position in the current linear list after which the item is to be inserted.

NAME — is the name of the new item to be inserted.

VALUE — is the floating point value of the new item to be inserted.

Exceptions:

LIST_KIND_ERROR
is raised if an attempt is made to insert an item by NAME into an unnamed list or, conversely, if an attempt is made to insert an item without NAME into a named list.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
is raised if the value of the parameter NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the NAME of type TOKEN_TYPE is an undefined token.

Downloaded from http://www.everyspec.com

CAPACITY_ERROR
: is raised if the number of items in the resulting linear list would exceed the value of the constant CAIS_PRAGMATICS.LIST_LENGTH or if the VALUE to be inserted violates the range constraints of the CAIS implementation. (See the note below.)

NAMED_LIST_ERROR
: is raised if the current linear list of IN_LIST is a named list that already contains an item with the name given by NAME.

Notes:

The most restrictive range constraints applicable to VALUE can be inferred from the value of the constant CAIS_PRAGMATICS.LIST_MAXIMUM_DIGITS according to rules defined in [1815A] 3.5.9, 4.5.7 and 4.6. (See CAIS_PRAGMATICS, Section 5.7, page 514.)

Downloaded from http://www.everyspec.com

5.4.1.24.5 Locating a floating point-valued item by value within a linear list

```
function POSITION_BY_VALUE
              (LIST:            in LIST_TYPE;
               VALUE:           in NUMBER;
               START_POSITION:  in POSITION_COUNT := POSITION_COUNT'FIRST;
               END_POSITION:    in POSITION_COUNT := POSITION_COUNT'LAST)
    return POSITION_COUNT;
```

Purpose:

This function returns the position in the current linear list of LIST of the next floating point-valued item whose value equals that of the VALUE parameter under floating point equality (see Section 5.4, page 420). The search begins at the START_POSITION and ends when either an item whose value equals VALUE is found, the last item of the list has been examined, or the item at the END_POSITION has been examined, whichever comes first.

Parameters:

LIST
: is the list in whose current linear list the item of interest is to be located.

VALUE
: is the floating point value of interest.

START_POSITION
: is the position of the first item in the current linear list to be considered in the search.

END_POSITION
: is the position beyond which the search will not proceed; the search may terminate prior to reaching END_POSITION should the sought item be found or should the last element of the list be considered.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of LIST is empty.

LIST_POSITION_ERROR
: is raised if START_POSITION specifies a value larger than the current length of the current linear list of LIST, or if END_POSITION is less than START_POSITION.

CAPACITY_ERROR
: is raised if the VALUE violates the range constraints of the CAIS implementation. (See the note below.)

SEARCH_ERROR
: is raised if there is no item in the current linear list of LIST within the region specified by START_POSITION and END_POSITION that has the value specified by VALUE.

Downloaded from http://www.everyspec.com

Notes:

Determining the position by value of floating point items should be applied with considerable caution and awareness of all the issues documented in [1815A], Section 4.5.7, regarding the accuracy of relational operations with real operands, and the discussion, in Section 5.4, page 420, defining floating point equality for floating point list items.

The most restrictive range constraints applicable to VALUE can be inferred from the value of the constant CAIS_PRAGMATICS.LIST_MAXIMUM_DIGITS according to rules defined in [1815A] 3.5.9, 4.5.7 and 4.6. (See CAIS_PRAGMATICS, Section 5.7, page 514.)

Downloaded from http://www.everyspec.com

5.4.1.25 Package CAIS_STRING_ITEM

This is a package for manipulating list items whose values are strings. The exceptions raised by all subprograms in this package are defined in the packages CAIS_LIST_MANAGEMENT and CAIS_PRAGMATICS.

Downloaded from http://www.everyspec.com

5.4.1.25.1 Extracting a string value from a list item

```
function EXTRACTED_VALUE (FROM_LIST:     in LIST_TYPE;
                          ITEM_POSITION: in POSITION_COUNT)
    return STRING;

function EXTRACTED_VALUE (FROM_LIST:     in LIST_TYPE;
                          ITEM_NAME:     in IDENTIFIER_TEXT)
    return STRING;

function EXTRACTED_VALUE (FROM_LIST:     in LIST_TYPE;
                          ITEM_NAME:     in TOKEN_TYPE)
    return STRING;
```

Purpose:

This function locates a string-valued item in the current linear list of FROM_LIST and returns a copy of its string value.

Parameters:

FROM_LIST is the list whose current linear list contains the item to be extracted.

ITEM_POSITION
is the position within the current linear list that identifies the item whose value is to be extracted.

ITEM_NAME is the name of the item whose value is to be extracted.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear list of FROM_LIST is empty.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of FROM_LIST.

SYNTAX_ERROR
is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

NAMED_LIST_ERROR
is raised if the parameter ITEM_NAME is used and the current linear list of FROM_LIST is unnamed.

SEARCH_ERROR
is raised if there is no item in the current linear list of FROM_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.25.2 Replacing a string value in a list item

```
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_POSITION:  in     POSITION_COUNT;
                   VALUE:          in     STRING);

procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     IDENTIFIER_TEXT;
                   VALUE:          in     STRING);

procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     TOKEN_TYPE;
                   VALUE:          in     STRING);
```

Purpose:

This procedure replaces the value of a string-valued item in the current linear list of IN_LIST with VALUE. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value.

Parameters:

IN_LIST
: is the list whose current linear list contains the item whose value is to be replaced.

ITEM_POSITION
: is the position within the current linear list that identifies the item whose value is to be replaced.

ITEM_NAME
: is the name of the item whose value is to be replaced.

VALUE
: is the new string value of the designated item.

Exceptions:

LIST_KIND_ERROR
: is raised if the current linear list of IN_LIST is empty.

LIST_POSITION_ERROR
: is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
: is raised if the value of the parameter ITEM_NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
: is raised if the ITEM_NAME of type TOKEN_TYPE is an undefined token.

Downloaded from http://www.everyspec.com

CAPACITY_ERROR
: is raised if the string value of VALUE is longer than the value of the constant CAIS_PRAGMATICS.STRING_ITEM_LENGTH (see Section 5.7, page 514).

NAMED_LIST_ERROR
: is raised if the parameter ITEM_NAME is used and the current linear list of IN_LIST is unnamed.

ITEM_KIND_ERROR
: is raised if ITEM_POSITION or ITEM_NAME specifies an item whose value is not a string.

SEARCH_ERROR
: is raised if there is no item in the current linear list of IN_LIST with the name ITEM_NAME.

Downloaded from http://www.everyspec.com

5.4.1.25.3 Inserting a string-valued item into a linear list

```
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  VALUE:     in     STRING);

procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     IDENTIFIER_TEXT;
                  VALUE:     in     STRING);

procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     TOKEN_TYPE;
                  VALUE:     in     STRING);
```

Purpose:

This procedure inserts an string-valued item into the current linear list of IN_LIST; the new list item will be positioned after the list item specified by POSITION. A value of zero in POSITION specifies a position at the head of the current linear list. Subsequent modification to the value of IN_LIST or of VALUE does not affect the other value. If the current linear list of IN_LIST is empty, it will be a named list after the successful completion of the call, if the second or third interface is used, and an unnamed list otherwise.

Parameters:

IN_LIST — is the list into whose current linear list the item will be inserted.

POSITION — is the position in the current linear list after which the item is to be inserted.

NAME — is the name of the new item to be inserted.

VALUE — is the string value of the new item to be inserted.

Exceptions:

LIST_KIND_ERROR
is raised if an attempt is made to insert an item by NAME into an unnamed list or, conversely, if an attempt is made to insert an item without NAME into a named list.

LIST_POSITION_ERROR
is raised if ITEM_POSITION has a value larger than the current length of the current linear list of IN_LIST.

SYNTAX_ERROR
is raised if the value of the parameter NAME of type IDENTIFIER_TEXT does not conform to the syntax of an Ada identifier.

TOKEN_ERROR
is raised if the NAME of type TOKEN_TYPE is an undefined token.

Downloaded from http://www.everyspec.com

CAPACITY_ERROR
is raised if the string value of VALUE to be inserted is longer than the value of the constant CAIS_PRAGMATICS.STRING_ITEM_LENGTH or if the number of items in the resulting linear list would exceed the value of the constant CAIS_PRAGMATICS.LIST_LENGTH (see Section 5.7, page 514).

NAMED_LIST_ERROR
is raised if the current linear list of IN_LIST is a named list that already contains an item with the name given by NAME.

Downloaded from http://www.everyspec.com

5.4.1.25.4 Locating a string-valued item by value within a linear list

```
function POSITION_BY_VALUE
          (LIST:           in LIST_TYPE;
           VALUE:          in STRING;
           START_POSITION: in POSITION_COUNT := POSITION_COUNT'FIRST;
           END_POSITION:   in POSITION_COUNT := POSITION_COUNT'LAST)
    return POSITION_COUNT;
```

Purpose:

This function returns the position in the current linear list of LIST of the next string-valued item whose value equals that of the VALUE parameter under string equality (see Section 5.4, page 420). The search begins at the START_POSITION and ends when either an item whose value equals VALUE is found, the last item of the list has been examined, or the item at the END_POSITION has been examined, whichever comes first.

Parameters:

LIST is the list in whose current linear list the item of interest is to be located.

VALUE is the string value of interest.

START_POSITION
is the position of the first item in the current linear list to be considered in the search.

END_POSITION
is the position beyond which the search will not proceed; the search may terminate prior to reaching END_POSITION should the sought item be found or should the last element of the list be considered.

Exceptions:

LIST_KIND_ERROR
is raised if the current linear list of LIST is empty.

LIST_POSITION_ERROR
is raised if START_POSITION specifies a value larger than the current length of the current linear list of LIST, or if END_POSITION is less than START_POSITION.

CAPACITY_ERROR
is raised if the string value of VALUE is longer than the value of the constant CAIS_PRAGMATICS.STRING_ITEM_LENGTH (see Section 5.7, page 514).

SEARCH_ERROR
is raised if there is no item in the current linear list of LIST within the region specified by START_POSITION and END_POSITION that has the value specified by VALUE.

Downloaded from http://www.everyspec.com

5.5 Package CAIS_STANDARD

This package contains certain scalar types predefined in the CAIS. The intent of providing this package is to make these types reasonably independent of any predefined types in the Ada language, whose characteristics may vary among compilers.

```
type    CAIS_INTEGER  is range
    CAIS_PRAGMATICS.MINIMUM_INTEGER .. CAIS_PRAGMATICS.MAXIMUM_INTEGER;
subtype CAIS_NATURAL  is CAIS_INTEGER range 0..CAIS_INTEGER'LAST;
subtype CAIS_POSITIVE is CAIS_INTEGER range 1..CAIS_INTEGER'LAST;
```

CAIS_INTEGER is a CAIS-defined type in analogy to the Ada type INTEGER. CAIS_NATURAL and CAIS_POSITIVE are CAIS-defined subtypes in analogy to the Ada subtypes NATURAL and POSITIVE, respectively.

```
type CAIS_DURATION is delta implementation_defined;
for CAIS_DURATION'SMALL use CAIS_PRAGMATICS.SMALL_FOR_CAIS_DURATION;
```

For the CAIS, an implementation of the type CAIS_DURATION must allow representation of durations (both positive and negative) up to at least 86400 seconds (one day). The smallest representable duration, CAIS_DURATION'SMALL must equal CAIS_PRAGMATICS.SMALL_FOR_CAIS_DURATION.

Downloaded from http://www.everyspec.com

5.6 Package CAIS_CALENDAR

This package provides facilities for accessing a system clock and interpreting its values. It is semantically almost identical to the package CALENDAR in [1815A], Section 9.6.. The differences relate to the use of the types CAIS_INTEGER and CAIS_DURATION in lieu of the corresponding Ada predefined types.

For the CAIS, an implementation of the type CAIS_DURATION must allow representation of durations (both positive and negative) up to at least 86400 seconds (one day). The smallest representable duration, CAIS_DURATION'SMALL, must equal CAIS_PRAGMATICS.SMALL_FOR_CAIS_DURATION (see Section 5.7, page 514).

The meaning of the values of type CAIS_CALENDAR.TIME is implementation-dependent, as these values will usually be obtained from the underlying system clock. In particular, the values need not be synchronized to some standard time, such as Greenwich Mean Time. Prior to comparing two values of type CAIS_CALENDAR.TIME obtained by two different calls on CAIS_CALENDAR.CLOCK, the user should consult Appendix F of the respective CAIS implementations to determine and account for the implementation dependencies.

Downloaded from http://www.everyspec.com

5.6.1 Definition of types, subtypes and exceptions

```
type TIME is private;
```

TIME is the type for the implementation-dependent time; values of type TIME must be able to be decomposed into values of the subtypes YEAR_NUMBER, MONTH_NUMBER, DAY_NUMBER and DAY_DURATION.

```
subtype YEAR_NUMBER  is CAIS_INTEGER range 1901 .. 2099;
subtype MONTH_NUMBER is CAIS_INTEGER range 1 .. 12;
subtype DAY_NUMBER   is CAIS_INTEGER range 1 .. 31;
subtype DAY_DURATION is CAIS_DURATION range 0.0 .. 86_400.0;
```

YEAR_NUMBER, MONTH_NUMBER and DAY_NUMBER are subtypes for the year, month and day, respectively, of a time. DAY_DURATION is the type which identifies the second within the day.

A proper time is a time that is formed from these types, in particular, the year number must be in the range of the subtype YEAR_NUMBER.

```
TIME_ERROR: exception;
```

TIME_ERROR is raised if a proper time cannot be formed by the functions defined in this package or the operator "-" cannot return a result that is in the range of the type CAIS_DURATION.

Downloaded from http://www.everyspec.com

5.6.2 Getting the current time

```
function CLOCK
    return TIME;
```

Purpose:

This function returns a value of CAIS_CALENDAR.TIME, representing the implementation-dependent time at which the interface was called.

Parameters:

None.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.6.3 *Getting the year part of the time*

```
function YEAR (DATE: in TIME)
    return YEAR_NUMBER;
```

Purpose:

This function returns the value of the year component within the time DATE for a given value of the type CAIS_CALENDAR.TIME.

Parameter:

DATE — is the time from which to extract the value of the year.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.6.4 Getting the month part of the time

```
function MONTH (DATE: in TIME)
    return MONTH_NUMBER;
```

Purpose:

This function returns the value of the month component within the time DATE for a given value of the type CAIS_CALENDAR.TIME.

Parameter:

DATE is the time from which to extract the value of the month.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.6.5 Getting the day part of the time

```
function DAY (DATE: in TIME)
    return DAY_NUMBER;
```

Purpose:

This function returns the value of the day component within the time DATE for a given value of the type CAIS_CALENDAR.TIME.

Parameter:

DATE   is the time from which to extract the value of the day.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.6.6 Getting the seconds part of the time

```
function SECONDS (DATE: in TIME)
    return DAY_DURATION;
```

Purpose:

This function returns the value of the seconds component within the time DATE for a given value of the type CAIS_CALENDAR.TIME.

Parameter:

DATE is the time from which to extract the value of the seconds.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.6.7 Splitting time into its components

```
procedure SPLIT (DATE:      in    TIME;
                 YEAR:      out   YEAR_NUMBER;
                 MONTH:     out   MONTH_NUMBER;
                 DAY:       out   DAY_NUMBER;
                 SECONDS:   out   DAY_DURATION);
```

Purpose:

This procedure returns all four component values (year, month, day and seconds) for the time DATE.

Parameters:

DATE	is the value of type CAIS_CALENDAR.TIME that will be split into its components.
YEAR	is the year component returned.
MONTH	is the month component returned.
DAY	is the day component returned.
SECONDS	is the seconds component returned.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.6.8 Combining components of time

```
function TIME_OF (YEAR:    in YEAR_NUMBER;
                  MONTH:   in MONTH_NUMBER;
                  DAY:     in DAY_NUMBER;
                  SECONDS: in DAY_DURATION)
        return TIME;
```

Purpose:

This function combines a year number, a month number, a day number and a seconds number into a value of the type CAIS_CALENDAR.TIME.

Parameters:

YEAR	is the value of the year component.
MONTH	is the value of the month component.
DAY	is the value of the day component.
SECONDS	is the value of the seconds component.

Exception:

TIME_ERROR	is raised if the actual parameters do not form a proper time.

Downloaded from http://www.everyspec.com

5.6.9 Adding time and duration

```
function "+" (LEFT:  in TIME;
              RIGHT: in CAIS_DURATION)
    return TIME;

function "+" (LEFT:  in CAIS_DURATION;
              RIGHT: in TIME)
    return TIME;
```

Purpose:

This function performs the operation of addition of times and durations.

Parameters:

LEFT, RIGHT are the values of type CAIS_CALENDAR.TIME and CAIS_DURATION to be added together.

Exception:

TIME_ERROR is raised if, for the given operands, the operator cannot return a time whose year number is in the range of the corresponding subtype.

Downloaded from http://www.everyspec.com

5.6.10 Subtracting time and duration

```
function "-" (LEFT:  in TIME;
              RIGHT: in CAIS_DURATION)
    return TIME;

function "-" (LEFT:  in TIME;
              RIGHT: in TIME)
    return CAIS_DURATION;
```

Purpose:

This function performs the operation of subtraction of times and durations.

Parameters:

LEFT, RIGHT are the values of type CAIS_CALENDAR.TIME and CAIS_DURATION to be subtracted from each other.

Exception:

TIME_ERROR is raised if, for the given operands, the operator cannot return a time whose year number is in the range of the corresponding subtype or if the result is not in the range of the type CAIS_DURATION.

Downloaded from http://www.everyspec.com

5.6.11 Comparing two values of time

```
function "<"  (LEFT:  in TIME;
               RIGHT: in TIME)
    return BOOLEAN;

function "<=" (LEFT:  in TIME;
               RIGHT: in TIME)
    return BOOLEAN;

function ">"  (LEFT:  in TIME;
               RIGHT: in TIME)
    return BOOLEAN;

function ">=" (LEFT:  in TIME;
               RIGHT: in TIME)
    return BOOLEAN;
```

Purpose:

This function performs the operation of the relational operators for times and has the conventional mathematical meanings.

Parameters:

LEFT, RIGHT are the values of type CAIS_CALENDAR.TIME to be compared.

Exceptions:

None.

Downloaded from http://www.everyspec.com

5.7 CAIS Pragmatics

Pragmatics are constraints imposed by an implementation that are not defined by the syntax or semantics of the CAIS. This section delineates the minimum capacities a conforming CAIS implementation must support. For most pragmatic limitations, two constants are defined in this package CAIS_PRAGMATICS. One is prefixed with "CAIS_"; it is the minimum value that any CAIS implementation must support. The other one without the prefix specifies an implementation-defined limit equal to or beyond the minimum required.

Each CAIS implementation must supply this package with the actual values filled in. All implementation-defined exceptions will be declared in Package CAIS_PRAGMATICS. An implementation can raise these implementation-defined exceptions in any of the interfaces as long as the semantics given in this document are maintained.

```
CAPACITY_ERROR: exception;
RESOURCE_ERROR: exception;
```

CAPACITY_ERROR is raised if a call on a CAIS interface detects a violation of the implementation-dependent maxima for the pragmatic limitations specified in this package. RESOURCE_ERROR is raised if a call on a CAIS interface exceeds resource limitations imposed by the underlying implementation. This exception is raised only if the conditions for CAPACITY_ERROR are not present.

```
UNRESTRICTED: constant := implementation_defined;
```

UNRESTRICTED is a very large *universal_integer* constant, usable only in universal expressions (see [1815A] 4.10).

```
CAIS_PATHNAME_LENGTH: constant := 255;
PATHNAME_LENGTH:      constant := implementation_defined;
```

CAIS_PATHNAME_LENGTH and PATHNAME_LENGTH are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of characters in a pathname and the actual upper limit imposed by a particular implementation.

```
CAIS_IDENTIFIER_LENGTH: constant := 80;
IDENTIFIER_LENGTH:      constant := implementation_defined;
```

CAIS_IDENTIFIER_LENGTH and IDENTIFIER_LENGTH are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of characters in an identifier and the actual upper limit imposed by a particular implementation.

```
CAIS_NODE_HANDLES_PER_PROCESS: constant := 255;
NODE_HANDLES_PER_PROCESS:      constant := implementation_defined;
```

CAIS_NODE_HANDLES_PER_PROCESS and NODE_HANDLES_PER_PROCESS are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of open node handles a process can have at one time and the actual upper limit imposed by a particular implementation.

Downloaded from http://www.everyspec.com


```
CAIS_NODES_IN_COPY_TREE: constant := 2 ** 15 - 1;
NODES_IN_COPY_TREE:      constant := implementation_defined;
```

CAIS_NODES_IN_COPY_TREE and NODES_IN_COPY_TREE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of nodes that can be copied with a single call of COPY_TREE and the actual upper limit imposed by a particular implementation.

```
CAIS_NODES_IN_DELETE_TREE: constant := 2 ** 15 - 1;
NODES_IN_DELETE_TREE:      constant := implementation_defined;
```

CAIS_NODES_IN_DELETE_TREE and NODES_IN_DELETE_TREE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of nodes that can be deleted with a single call of DELETE_TREE and the actual upper limit imposed by a particular implementation.

```
CAIS_EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE: constant := 2 ** 10 - 1;
EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE: constant := implementation_defined;
```

CAIS_EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE and EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of primary relationships that can emanate from a single node at one time and the actual upper limit imposed by a particular implementation.

```
CAIS_EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE: constant := 2 ** 10 - 1;
EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE: constant := implementation_defined;
```

CAIS_EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE and EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of secondary relationships that can emanate from a single node at one time and the actual upper limit imposed by a particular implementation.

```
CAIS_ELEMENTS_OF_NODE_ITERATOR: constant := 2 ** 11 -2;
ELEMENTS_OF_NODE_ITERATOR:      constant := implementation_defined;
```

CAIS_ELEMENTS_OF_NODE_ITERATOR and ELEMENTS_OF_NODE_ITERATOR are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of elements that can be contained in a node iterator at one time and the actual upper limit imposed by a particular implementation. The constant ELEMENTS_OF_NODE_ITERATOR must be at least as large as the sum of the two constants EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE and EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE.

```
CAIS_ELEMENTS_OF_ATTRIBUTE_ITERATOR: constant := 255;
ELEMENTS_OF_ATTRIBUTE_ITERATOR:      constant := implementation_defined;
```

CAIS_ELEMENTS_OF_ATTRIBUTE_ITERATOR and ELEMENTS_OF_ATTRIBUTE_ITERATOR are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of elements that can be contained in an attribute iterator at one time and the actual upper limit imposed by a particular implementation.

Downloaded from http://www.everyspec.com


```
CAIS_ATTRIBUTES_PER_NODE: constant := 255;
ATTRIBUTES_PER_NODE:      constant := implementation_defined;
```

CAIS_ATTRIBUTES_PER_NODE and ATTRIBUTES_PER_NODE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of attributes that can be associated with a single node at one time and the actual upper limit imposed by a particular implementation.

```
CAIS_ATTRIBUTES_PER_RELATIONSHIP: constant := 255;
ATTRIBUTES_PER_RELATIONSHIP:      constant := implementation_defined;
```

CAIS_ATTRIBUTES_PER_RELATIONSHIP and ATTRIBUTES_PER_RELATIONSHIP are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of attributes that can be associated with a single relationship at one time and the actual upper limit imposed by a particular implementation.

```
CAIS_ACCESS_RELATIONSHIPS_OF_OBJECT: constant := 255;
ACCESS_RELATIONSHIPS_OF_OBJECT:      constant := implementation_defined;
```

CAIS_ACCESS_RELATIONSHIPS_OF_OBJECT and ACCESS_RELATIONSHIPS_OF_OBJECT are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of relationships of the predefined relation ACCESS that can emanate from a node at one time and the actual upper limit imposed by a particular implementation.

```
CAIS_GRANT_ITEMS_ON_GRANT_ATTRIBUTE: constant := 15;
GRANT_ITEMS_ON_GRANT_ATTRIBUTE:      constant := implementation_defined;
```

CAIS_GRANT_ITEMS_ON_GRANT_ATTRIBUTE and GRANT_ITEMS_ON_GRANT_ATTRIBUTE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of grant items that can be contained in the value of a GRANT attribute at one time and the actual upper limit imposed by a particular implementation.

```
CAIS_GROUP_NODES: constant := 255;
GROUP_NODES:      constant := implementation_defined;
```

CAIS_GROUP_NODES and GROUP_NODES are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of group nodes that can be contained in a given CAIS implementation and the actual upper limit imposed by a particular implementation.

```
CAIS_ADOPTED_ROLES_OF_PROCESS: constant := 7;
ADOPTED_ROLES_OF_PROCESS:      constant := implementation_defined;
```

CAIS_ADOPTED_ROLES_OF_PROCESS and ADOPTED_ROLES_OF_PROCESS are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of roles which a single process can have adopted at one time and the actual upper limit imposed by a particular implementation.

```
CAIS_NUMBER_OF_NODES: constant := UNRESTRICTED;
```

CAIS_NUMBER_OF_NODES is a constant specifying the smallest upper limit which can be imposed by any CAIS implementation on the number of nodes that can be contained in a given CAIS implementation at one time.

Downloaded from http://www.everyspec.com


```
CAIS_LENGTH_OF_PRIMARY_PATH: constant := PATHNAME_LENGTH/2;
```

CAIS_LENGTH_OF_PRIMARY_PATH is a constant specifying the smallest upper limit which can be imposed by any CAIS implementation on the number of path elements in a primary pathname.

```
CAIS_DIRECT_IO_RECORD_SIZE: constant := 2 ** 15 - 1;
DIRECT_IO_RECORD_SIZE:      constant := implementation_defined;
```

CAIS_DIRECT_IO_RECORD_SIZE and DIRECT_IO_RECORD_SIZE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of bits in a direct input or output record and the actual upper limit imposed by a particular implementation.

```
CAIS_SEQUENTIAL_IO_RECORD_SIZE: constant := 2 ** 15 - 1;
SEQUENTIAL_IO_RECORD_SIZE:      constant := implementation_defined;
```

CAIS_SEQUENTIAL_IO_RECORD_SIZE and SEQUENTIAL_IO_RECORD_SIZE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of bits in a sequential input or output record and the actual upper limit imposed by a particular implementation. The constant SEQUENTIAL_IO_RECORD_SIZE must be at least as large as the constant DIRECT_IO_RECORD_SIZE.

```
CAIS_DIRECT_IO_INDEX_RANGE_UPPER_BOUND: constant := 2 ** 15 - 1;
DIRECT_IO_INDEX_RANGE_UPPER_BOUND:      constant := implementation_defined;
```

CAIS_DIRECT_IO_INDEX_RANGE_UPPER_BOUND and DIRECT_IO_INDEX_RANGE_UPPER_BOUND are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the type COUNT in the package CAIS_DIRECT_IO and the actual upper limit imposed by a particular implementation.

```
CAIS_SEQUENTIAL_IO_FILE_SIZE: constant := 2 ** 15 - 1;
SEQUENTIAL_IO_FILE_SIZE:      constant := implementation_defined;
```

CAIS_SEQUENTIAL_IO_FILE_SIZE and SEQUENTIAL_IO_FILE_SIZE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of CAIS_SEQUENTIAL_IO.WRITE operations that can be performed on a sequential file and the actual upper limit imposed by a particular implementation. SEQUENTIAL_IO_FILE_SIZE must be at least as large as DIRECT_IO_INDEX_RANGE_UPPER_BOUND.

```
CAIS_TEXT_IO_LINES_PER_FILE: constant := 2 ** 15 - 1;
TEXT_IO_LINES_PER_FILE:      constant := implementation_defined;
```

CAIS_TEXT_IO_LINES_PER_FILE and TEXT_IO_LINES_PER_FILE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of lines in a text input or output file and the actual upper limit imposed by a particular implementation.

```
CAIS_TEXT_IO_LINES_PER_PAGE: constant := 2 ** 15 - 1;
TEXT_IO_LINES_PER_PAGE:      constant := implementation_defined;
```

CAIS_TEXT_IO_LINES_PER_PAGE and TEXT_IO_LINES_PER_PAGE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of lines per page in a text input or output file and the actual upper limit imposed by a particular implementation.

Downloaded from http://www.everyspec.com


```
CAIS_TEXT_IO_COLUMNS_PER_LINE: constant := 255;
TEXT_IO_COLUMNS_PER_LINE:      constant := implementation_defined;
```

CAIS_TEXT_IO_COLUMNS_PER_LINE and TEXT_IO_COLUMNS_PER_LINE are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of columns per line of a text input or output file and the actual upper limit imposed by a particular implementation.

```
CAIS_MINIMUM_TAPE_BLOCK_LENGTH: constant := 18;
MINIMUM_TAPE_BLOCK_LENGTH:      constant := implementation_defined;
```

CAIS_MINIMUM_TAPE_BLOCK_LENGTH and MINIMUM_TAPE_BLOCK_LENGTH are constants, specifying, respectively, the largest lower limit which can be imposed by any CAIS implementation on the number of characters written to a magnetic tape in a single block and the actual lower limit imposed by a particular implementation.

```
CAIS_MAXIMUM_TAPE_BLOCK_LENGTH: constant := 2048;
MAXIMUM_TAPE_BLOCK_LENGTH:      constant := implementation_defined;
```

CAIS_MAXIMUM_TAPE_BLOCK_LENGTH and MAXIMUM_TAPE_BLOCK_LENGTH are constants, specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of characters written to a magnetic tape in a single block and the actual upper limit imposed by a particular implementation.

```
CAIS_FILE_HANDLES_PER_PROCESS: constant := 15;
FILE_HANDLES_PER_PROCESS:      constant := implementation_defined;
```

CAIS_FILE_HANDLES_PER_PROCESS and FILE_HANDLES_PER_PROCESS are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of open file handles a process can have at one time and the actual upper limit imposed by a particular implementation.

```
FILE_STORAGE_UNIT_SIZE: constant := implementation_defined;
```

FILE_STORAGE_UNIT_SIZE is a constant specifying the number of bits per file storage unit in sequential files and direct files for a particular CAIS implementation.

```
MEMORY_STORAGE_UNIT_SIZE: constant := implementation_defined;
```

MEMORY_STORAGE_UNIT_SIZE is a constant specifying the number of bits per memory storage unit for a particular CAIS implementation.

```
QUEUE_STORAGE_UNIT_SIZE: constant := implementation_defined;
```

QUEUE_STORAGE_UNIT_SIZE is a constant specifying the number of bits per queue storage unit in nonsynchronous queue files for a particular CAIS implementation.

```
CAIS_IDENTIFIER_ITEM_LENGTH: constant := CAIS_IDENTIFIER_LENGTH;
IDENTIFIER_ITEM_LENGTH:      constant := implementation_defined;
```

CAIS_IDENTIFIER_ITEM_LENGTH and IDENTIFIER_ITEM_LENGTH are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of characters in an identifier item or the value of a token and the actual upper limit imposed by a particular implementation.

Downloaded from http://www.everyspec.com


```
CAIS_LIST_LENGTH: constant := 255;
LIST_LENGTH:      constant := implementation_defined;
```

CAIS_LIST_LENGTH and LIST_LENGTH are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of items in a list and the actual upper limit imposed by a particular implementation.

```
CAIS_STRING_ITEM_LENGTH: constant := CAIS_PATHNAME_LENGTH;
STRING_ITEM_LENGTH:      constant := implementation_defined;
```

CAIS_STRING_ITEM_LENGTH and STRING_ITEM_LENGTH are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of characters in a string item and the actual upper limit imposed by a particular implementation. STRING_ITEM_LENGTH must be at least as large as the value of PATHNAME_LENGTH.

```
CAIS_LIST_TEXT_LENGTH: constant := 2 ** 10;
LIST_TEXT_LENGTH:      constant := implementation_defined;
```

CAIS_LIST_TEXT_LENGTH and LIST_TEXT_LENGTH are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of characters in the external representation of a list and the actual upper limit imposed by a particular implementation.

```
CAIS_MINIMUM_INTEGER: constant := - (2 ** 15 - 1);
MINIMUM_INTEGER:      constant := implementation_defined;
```

CAIS_MINIMUM_INTEGER and MINIMUM_INTEGER are constants specifying, respectively, the largest lower limit which can be imposed by any CAIS implementation on the smallest (most negative) value of the type CAIS_INTEGER and the actual lower limit imposed by a particular implementation.

```
CAIS_MAXIMUM_INTEGER: constant := 2 ** 15 - 1;
MAXIMUM_INTEGER:      constant := implementation_defined;
```

CAIS_MAXIMUM_INTEGER and MAXIMUM_INTEGER are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the largest (most positive) value of the type CAIS_INTEGER and the actual upper limit imposed by a particular implementation.

```
CAIS_LIST_MAXIMUM_DIGITS: constant := 6;
LIST_MAXIMUM_DIGITS:      constant := implementation_defined;
```

CAIS_LIST_MAXIMUM_DIGITS and LIST_MAXIMUM_DIGITS are constants specifying, respectively, the smallest upper limit which can be imposed by any CAIS implementation on the number of significant decimal digits of the floating point type used to contain floating point item values and the actual upper limit imposed by a particular CAIS implementation.

```
CAIS_SMALL_FOR_CAIS_DURATION: constant := 0.015625; -- 1/64
SMALL_FOR_CAIS_DURATION:      constant := implementation_defined;
```

CAIS_SMALL_FOR_CAIS_DURATION and SMALL_FOR_CAIS_DURATION are constants specifying, respectively, the maximum for the smallest representable duration, i.e., for the value of CAIS_DURATION'SMALL, and the actual smallest duration supported by a particular CAIS implementation.

Downloaded from http://www.everyspec.com

6. NOTES

6.1 Keywords

The following list represents the keywords applicable to this standard. These keywords may be used to categorize the concepts presented within this standard and assist in automatic retrieval of appropriate data used in automated document retrieval systems.

```
Ada
Ada Programming Support Environment
APSE
CAIS
Common APSE Interface Set
computer file system
KAPSE
Kernel Ada Programming Support Environment
high level languages
interfaces
interoperability
operating system
portability
programming support environment
software engineering environment
transportability
virtual operating system
```

Downloaded from http://www.everyspec.com

Appendix A
Predefined Relations, Attributes and Attribute Values

The material contained in this appendix is not a mandatory part of the standard.

10. Predefined Relations:

ACCESS: designates a secondary relationship from an object node to a group node representing a role; the access rights that are granted to adopters of the role are given in the GRANT attribute of this relationship.

ADOPTED_ROLE:
designates a secondary relationship from a subject (process) node to a group node representing a role; indicates that the process has adopted the role represented by the group node.

CURRENT_JOB:
designates a secondary relationship from a process node to the root process node of the tree which contains the process node.

CURRENT_NODE:
designates a secondary relationship from a process node to the node representing the current focus of attention or context for activities of that process.

CURRENT_USER:
designates a secondary relationship from a process node to a top-level node representing the user on whose behalf the process was initiated.

DEFAULT_ROLE:
designates a secondary relationship from a top-level user node to a group node; there must be exactly one such relationship from a user node. Also, designates a secondary relationship from a file node that contains an executable image of a process to a group node; there can only be one such relationship from the file node.

DEVICE: designates a secondary relationship from a process node to a top-level node representing a device to which the process has access. Also designates a primary relationship from the system-level node to a node representing a device.

DOT: designates the default relation name to be used when none is provided. Special rules apply for pathname abbreviations in the presence of path elements whose relation name is DOT. Also, the CAIS discretionary access control model associates specific semantics with relationships of the DOT relation among group nodes in determining the role under which a process executes. No other semantics or restrictions are associated with DOT.

EXECUTABLE_IMAGE:
designates a secondary relationship from a process node to the node containing the executable image of the process.

Downloaded from http://www.everyspec.com

GROUP: designates a secondary relationship from a process node to a top-level group node. Also designates a primary relationship from the system-level node to a top-level group node.

JOB: designates a primary relationship from the top-level node of a user to the root process node of a job.

MIMIC_FILE: designates a secondary relationship from a node representing a mimic queue file to the node representing that file's coupled file; indicates that the queue file and the other file are coupled; this means that the contents of the file are the initial contents of the queue file and subsequent writes to the queue file are appended to the other file as well.

PARENT: designates the secondary relationship from a given node to the node which is the source node of the unique primary relationship pointing to the given node.

POTENTIAL_MEMBER: designates a secondary relationship from a group node to another group node representing a potential member of the group.

STANDARD_ERROR: designates a secondary relationship from a process node to a file node representing the file to which error messages are to be written by default.

STANDARD_INPUT: designates a secondary relationship from a process node to a file node representing the file which is the initial default source of process inputs.

STANDARD_OUTPUT: designates a secondary relationship from a process node to a file node representing the file to which outputs are initially being directed by default.

USER: designates a secondary relationship from a process node to a top-level user node. Also designates a primary relationship from the system-level node to a top-level node representing a user.

Downloaded from http://www.everyspec.com

20. Predefined Attributes:

ACCESS_METHOD:
applies to file nodes; designates the kind of access which can be used on the node's contents; the predefined attribute values are SEQUENTIAL, DIRECT and TEXT.

CURRENT_FILE_SIZE:
applies to file nodes with a FILE_KIND attribute value of SECONDARY_STORAGE; designates the current size of a file.

CURRENT_QUEUE_SIZE:
applies to file nodes with a FILE_KIND attribute value of QUEUE and a QUEUE_KIND attribute value of NONSYNCHRONOUS_SOLO, NONSYNCHRONOUS_COPY or NONSYNCHRONOUS_MIMIC; designates the current number of elements of a queue.

CURRENT_STATUS:
applies to process nodes; designates the current status of the node's contents; possible values are READY, SUSPENDED, ABORTED or TERMINATED.

DEVICE_KIND: applies to file nodes with a FILE_KIND attribute value of DEVICE; designates the kinds of devices which are represented by the node's contents; the predefined attribute values are SCROLL_TERMINAL, PAGE_TERMINAL, FORM_TERMINAL, MAGNETIC_TAPE_DRIVE or combinations thereof.

FILE_KIND: applies to file nodes; designates the kind of file that is the node's contents; possible values are SECONDARY_STORAGE, QUEUE or DEVICE.

GRANT: applies to relationships of the predefined relation ACCESS; designates the access rights which are granted by means of the access relationship; values are lists of grant items as specified in Table II, page 42.

HIGHEST_CLASSIFICATION:
applies to file nodes; designates the highest allowable object classification label that may be assigned to the node; values are implementation-defined.

INHERITABLE: applies to all relationships; designates whether or not the relationship is inheritable. Possible values are TRUE and FALSE. For primary relationships the attribute value is always FALSE.

IO_UNIT_COUNT:
applies to process nodes; designates the number of GET and PUT operations that have been performed by the node's process.

NODE_KIND: applies to all relationships; designates the kind of the target node; possible values are STRUCTURAL, PROCESS or FILE.

Downloaded from http://www.everyspec.com

LOWEST_CLASSIFICATION:
applies to file nodes; designates the lowest allowable object classification label that may be assigned to the node; values are implementation-defined.

MACHINE_TIME:
applies to process nodes; designates the length of time the process was active on the logical processor, if the process has terminated or aborted, or zero, if the process has not terminated or aborted.

MAXIMUM_FILE_SIZE:
applies to file nodes with a FILE_KIND attribute value of SECONDARY_STORAGE; designates the maximum allowable size for a file.

MAXIMUM_QUEUE_SIZE:
applies to file nodes with a FILE_KIND attribute value of QUEUE and a QUEUE_KIND attribute value of NONSYNCHRONOUS_SOLO, NONSYNCHRONOUS_COPY or NONSYNCHRONOUS_MIMIC; designates the maximum allowable size for a queue.

OBJECT_CLASSIFICATION:
applies to all nodes; designates the node's classification as an object; values are implementation-defined.

OPEN_NODE_HANDLE_COUNT:
applies to process nodes; designates the number of node handles the node's process currently has opened.

PARAMETERS: applies to process nodes; designates the parameters with which the process was initiated.

PROCESS_SIZE:
applies to process nodes; designates the amount of memory currently in use by the process.

QUEUE_KIND: applies to file nodes with a FILE_KIND attribute value of QUEUE; designates the kind of queue file; possible values are SYNCHRONOUS_SOLO, NONSYNCHRONOUS_SOLO, NONSYNCHRONOUS_MIMIC or NONSYNCHRONOUS_COPY.

RESULTS: applies to process nodes; designates the intermediate results of the process; values are user-defined.

SUBJECT_CLASSIFICATION:
applies to process nodes; designates the classification of the node's process as a subject; values are implementation-defined.

Downloaded from http://www.everyspec.com

TIME_ATTRIBUTE_WRITTEN:
applies to all nodes; designates the most recent implementation-defined time at which any attribute was modified (i.e., attribute value changed by the user, new attribute added or existing attribute deleted) by a call on a CAIS interface; changes to attributes that are made implicitly by the implementation are not reflected in TIME_ATTRIBUTE_WRITTEN.

TIME_CONTENTS_WRITTEN:
applies to file nodes; designates the most recent implementation-defined time at which the file contents have been modified (i.e., written).

TIME_CREATED:
applies to all nodes; designates the implementation-defined time at which the node was created.

TIME_FINISHED:
applies to process nodes; designates the implementation-defined time at which the process terminated or aborted.

TIME_RELATIONSHIP_WRITTEN:
applies to all nodes; designates the most recent implementation-defined time at which any relationship was modified (i.e., a new relationship added or an existing relationship deleted) or at which any attributes of any relationship emanating from the node were modified (i.e., attribute value of an attribute of the relationship changed by the user, a new attribute of the relationship added or an existing attribute of the relationship deleted); changes to relationships that are maintained by the implementation and cannot be set using CAIS interfaces are not reflected in TIME_RELATIONSHIP_WRITTEN.

TIME_STARTED:
applies to process nodes; designates the implementation-defined time of activation of the process.

Downloaded from http://www.everyspec.com

30. Predefined Attribute Values:

ABORTED
APPEND
APPEND_ATTRIBUTES
APPEND_CONTENTS
APPEND_RELATIONSHIPS
CONTROL
DEVICE
DIRECT
EXECUTE
EXISTENCE
FALSE
FILE
FORM_TERMINAL
MAGNETIC_TAPE_DRIVE
NONSYNCHRONOUS_COPY
NONSYNCHRONOUS_MIMIC
NONSYNCHRONOUS_SOLO
PAGE_TERMINAL
PROCESS
QUEUE
READ
READ_ATTRIBUTES
READ_CONTENTS
READ_RELATIONSHIPS
READY
SCROLL_TERMINAL
SECONDARY_STORAGE
SEQUENTIAL
STRUCTURAL
SUSPENDED
SYNCHRONOUS_SOLO
TERMINATED
TEXT
TRUE
WRITE
WRITE_ATTRIBUTES
WRITE_CONTENTS
WRITE_RELATIONSHIPS

Downloaded from http://www.everyspec.com

40. Predefined Contents, Attributes, and Relationships

The following tables show the predefined contents, attributes, and relationships for the system-level node and for each kind of node (STRUCTURAL, PROCESS, and FILE).

System-Level Node			
Node Contents	Node Attributes	Node as Source of Relationship	Node as Target of Relationship
None	None	DEVICE GROUP USER	PARENT

Structural Nodes			
Node Contents	Node Attribute	Node as Source of Relationship	Node as Target of Relationship
None	OBJECT_ CLASSIFICATION TIME_ATTRIBUTE_ WRITTEN TIME_CREATED TIME_ RELATIONSHIP_ WRITTEN	ACCESS DEFAULT_ROLE (if top-level user node) DOT JOB (if top-level user node) PARENT POTENTIAL_MEMBER (if group)	ACCESS (if group node) ADOPTED_ROLE (if group node) CURRENT_NODE CURRENT_USER (if top-level user node) DEFAULT_ROLE (if group node) DOT GROUP (if group node) PARENT POTENTIAL_MEMBER (if group node) USER (if top-level user node)

Downloaded from http://www.everyspec.com

Process Nodes			
Node Contents	Node Attributes	Node as Source of Relationship	Node as Target of Relationship
Represents Execution of an Ada program	CURRENT_STATUS IO_UNIT_COUNT MACHINE_TIME OBJECT_CLASSIFICATION OPEN_NODE_HANDLE_COUNT PARAMETERS PROCESS_SIZE RESULTS SUBJECT_CLASSIFICATION TIME_ATTRIBUTE_WRITTEN TIME_CREATED TIME_FINISHED TIME_RELATIONSHIP_WRITTEN TIME_STARTED	ACCESS ADOPTED_ROLE CURRENT_JOB CURRENT_NODE CURRENT_USER DEVICE DOT EXECUTABLE_IMAGE GROUP PARENT STANDARD_ERROR STANDARD_INPUT STANDARD_OUTPUT USER	CURRENT_JOB (if root process node) CURRENT_NODE DOT JOB (if root process node) PARENT

Downloaded from http://www.everyspec.com

File Nodes			
Node Contents	Node Attributes	Node as Source of Relationship	Node as Target of Relationship
Ada external file	ACCESS_METHOD CURRENT_FILE_SIZE (if SECONDARY_STORAGE File Node) CURRENT_QUEUE_SIZE (if QUEUE File Node) DEVICE_KIND (if DEVICE File Node) FILE_KIND HIGHEST_CLASSIFICATION LOWEST_CLASSIFICATION MAXIMUM_FILE_SIZE (if SECONDARY_STORAGE File Node) MAXIMUM_QUEUE_SIZE (if QUEUE File Node) OBJECT_CLASSIFICATION QUEUE_KIND (if QUEUE File Node) TIME_ATTRIBUTE_WRITTEN TIME_CONTENTS_WRITTEN TIME_CREATED TIME_RELATIONSHIP_WRITTEN	ACCESS DEFAULT_ROLE (if File Node has Executable Contents) DOT MIMIC_FILE (if QUEUE File Node) PARENT	CURRENT_NODE DEVICE (if top-level device node) DOT MIMIC_FILE (if QUEUE File Node) PARENT STANDARD_ERROR STANDARD_INPUT STANDARD_OUTPUT

Downloaded from http://www.everyspec.com

Appendix B
CAIS Specification

The material contained in this appendix is a mandatory part of the standard.

This appendix contains a set of Ada package specifications of the CAIS interfaces in their canonical form (see Section 4.2, page 19 and Section 4.2.1, page 20). 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.

```
package CAIS_PRAGMATICS is

    CAPACITY_ERROR: exception;
    RESOURCE_ERROR: exception;

    UNRESTRICTED: constant := implementation_defined;

    CAIS_PATHNAME_LENGTH: constant := 255;
    PATHNAME_LENGTH:      constant := implementation_defined;

    CAIS_IDENTIFIER_LENGTH: constant := 80;
    IDENTIFIER_LENGTH:      constant := implementation_defined;

    CAIS_NODE_HANDLES_PER_PROCESS: constant := 255;
    NODE_HANDLES_PER_PROCESS:      constant := implementation_defined;

    CAIS_NODES_IN_COPY_TREE: constant := 2 ** 15 - 1;
    NODES_IN_COPY_TREE:      constant := implementation_defined;

    CAIS_NODES_IN_DELETE_TREE: constant := 2 ** 15 - 1;
    NODES_IN_DELETE_TREE:      constant := implementation_defined;

    CAIS_EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE: constant := 2 ** 10 - 1;
    EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE: constant := implementation_defined;

    CAIS_EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE: constant := 2 ** 10 - 1;
    EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE: constant := implementation_defined;

    CAIS_ELEMENTS_OF_NODE_ITERATOR: constant := 2 ** 11 -2;
    ELEMENTS_OF_NODE_ITERATOR:      constant := implementation_defined;

    CAIS_ELEMENTS_OF_ATTRIBUTE_ITERATOR: constant := 255;
    ELEMENTS_OF_ATTRIBUTE_ITERATOR:      constant := implementation_defined;

    CAIS_ATTRIBUTES_PER_NODE: constant := 255;
    ATTRIBUTES_PER_NODE:      constant := implementation_defined;

    CAIS_ATTRIBUTES_PER_RELATIONSHIP: constant := 255;
    ATTRIBUTES_PER_RELATIONSHIP:      constant := implementation_defined;

    CAIS_ACCESS_RELATIONSHIPS_OF_OBJECT: constant := 255;
    ACCESS_RELATIONSHIPS_OF_OBJECT:      constant := implementation_defined;

    CAIS_GRANT_ITEMS_ON_GRANT_ATTRIBUTE: constant := 15;
    GRANT_ITEMS_ON_GRANT_ATTRIBUTE:      constant := implementation_defined;
```

Downloaded from http://www.everyspec.com


```
CAIS_GROUP_NODES: constant := 255;
GROUP_NODES:      constant := implementation_defined;

CAIS_ADOPTED_ROLES_OF_PROCESS: constant := 7;
ADOPTED_ROLES_OF_PROCESS:      constant := implementation_defined;

CAIS_NUMBER_OF_NODES: constant := UNRESTRICTED;

CAIS_LENGTH_OF_PRIMARY_PATH: constant := PATHNAME_LENGTH/2;

CAIS_DIRECT_IO_RECORD_SIZE: constant := 2 ** 15 - 1;
DIRECT_IO_RECORD_SIZE:      constant := implementation_defined;

CAIS_SEQUENTIAL_IO_RECORD_SIZE: constant := 2 ** 15 - 1;
SEQUENTIAL_IO_RECORD_SIZE:      constant := implementation_defined;

CAIS_DIRECT_IO_INDEX_RANGE_UPPER_BOUND: constant := 2 ** 15 - 1;
DIRECT_IO_INDEX_RANGE_UPPER_BOUND:      constant := implementation_defined;

CAIS_SEQUENTIAL_IO_FILE_SIZE: constant := 2 ** 15 - 1;
SEQUENTIAL_IO_FILE_SIZE:      constant := implementation_defined;

CAIS_TEXT_IO_LINES_PER_FILE: constant := 2 ** 15 - 1;
TEXT_IO_LINES_PER_FILE:      constant := implementation_defined;

CAIS_TEXT_IO_LINES_PER_PAGE: constant := 2 ** 15 - 1;
TEXT_IO_LINES_PER_PAGE:      constant := implementation_defined;

CAIS_TEXT_IO_COLUMNS_PER_LINE: constant := 255;
TEXT_IO_COLUMNS_PER_LINE:      constant := implementation_defined;

CAIS_MINIMUM_TAPE_BLOCK_LENGTH: constant := 18;
MINIMUM_TAPE_BLOCK_LENGTH:      constant := implementation_defined;

CAIS_MAXIMUM_TAPE_BLOCK_LENGTH: constant := 2048;
MAXIMUM_TAPE_BLOCK_LENGTH:      constant := implementation_defined;

CAIS_FILE_HANDLES_PER_PROCESS: constant := 15;
FILE_HANDLES_PER_PROCESS:      constant := implementation_defined;

FILE_STORAGE_UNIT_SIZE: constant := implementation_defined;

MEMORY_STORAGE_UNIT_SIZE: constant := implementation_defined;

QUEUE_STORAGE_UNIT_SIZE: constant := implementation_defined;

CAIS_IDENTIFIER_ITEM_LENGTH: constant := CAIS_IDENTIFIER_LENGTH;
IDENTIFIER_ITEM_LENGTH:      constant := implementation_defined;

CAIS_LIST_LENGTH: constant := 255;
LIST_LENGTH:      constant := implementation_defined;

CAIS_STRING_ITEM_LENGTH: constant := CAIS_PATHNAME_LENGTH;
STRING_ITEM_LENGTH:      constant := implementation_defined;

CAIS_LIST_TEXT_LENGTH: constant := 2 ** 10;
LIST_TEXT_LENGTH:      constant := implementation_defined;
```

Downloaded from http://www.everyspec.com


```
    CAIS_MINIMUM_INTEGER: constant := - (2 ** 15 - 1);
    MINIMUM_INTEGER:      constant := implementation_defined;

    CAIS_MAXIMUM_INTEGER: constant := 2 ** 15 - 1;
    MAXIMUM_INTEGER:      constant := implementation_defined;

    CAIS_LIST_MAXIMUM_DIGITS: constant := 6;
    LIST_MAXIMUM_DIGITS:      constant := implementation_defined;

    CAIS_SMALL_FOR_CAIS_DURATION: constant := 0.015625; -- 1/64
    SMALL_FOR_CAIS_DURATION:      constant := implementation_defined;

end CAIS_PRAGMATICS;


with CAIS_PRAGMATICS;
package CAIS_STANDARD is

    type    CAIS_INTEGER  is range
        CAIS_PRAGMATICS.MINIMUM_INTEGER .. CAIS_PRAGMATICS.MAXIMUM_INTEGER;
    subtype CAIS_NATURAL  is CAIS_INTEGER range 0..CAIS_INTEGER'LAST;
    subtype CAIS_POSITIVE is CAIS_INTEGER range 1..CAIS_INTEGER'LAST;

    type CAIS_DURATION is delta implementation_defined;
    for CAIS_DURATION'SMALL use CAIS_PRAGMATICS.SMALL_FOR_CAIS_DURATION;

end CAIS_STANDARD;


with CAIS_STANDARD;
with CAIS_PRAGMATICS;
package CAIS_LIST_MANAGEMENT is

    use CAIS_STANDARD;

    type    LIST_TYPE       is limited private;

    subtype LIST_TEXT       is STRING;

    type    LIST_SIZE       is range 0 .. CAIS_PRAGMATICS.LIST_LENGTH;
    subtype POSITION_COUNT  is LIST_SIZE range 1 .. LIST_SIZE'LAST;
    subtype INSERT_COUNT    is LIST_SIZE range 0 .. LIST_SIZE'LAST - 1;

    type    LIST_KIND       is (UNNAMED, NAMED, EMPTY);

    type    ITEM_KIND       is (LIST_ITEM_KIND, STRING_ITEM_KIND,
                                INTEGER_ITEM_KIND, FLOAT_ITEM_KIND,
                                IDENTIFIER_ITEM_KIND);

    type    TOKEN_TYPE        is limited private;
    subtype IDENTIFIER_TEXT   is STRING;

    EMPTY_LIST:  constant  LIST_TYPE;

    ITEM_KIND_ERROR:       exception;
```

Downloaded from http://www.everyspec.com


```
LIST_KIND_ERROR:      exception;
LIST_POSITION_ERROR:  exception;
NAMED_LIST_ERROR:     exception;
SEARCH_ERROR:         exception;
SYNTAX_ERROR:         exception;
TOKEN_ERROR:          exception;

procedure COPY_LIST (FROM_LIST: in     LIST_TYPE;
                     TO_LIST:   in out LIST_TYPE);
procedure SET_TO_EMPTY_LIST (LIST: in out LIST_TYPE);
procedure CONVERT_TEXT_TO_LIST (LIST_STRING: in     LIST_TEXT;
                                LIST:        in out LIST_TYPE);
function TEXT_FORM (LIST: in LIST_TYPE)
    return LIST_TEXT;
function IS_EQUAL (LIST1: in LIST_TYPE;
                   LIST2: in LIST_TYPE)
    return BOOLEAN;
procedure DELETE (LIST:          in out LIST_TYPE;
                  ITEM_POSITION: in     POSITION_COUNT);
procedure DELETE (LIST:          in out LIST_TYPE;
                  ITEM_NAME:     in     IDENTIFIER_TEXT);
procedure DELETE (LIST:          in out LIST_TYPE;
                  ITEM_NAME:     in     TOKEN_TYPE);
function KIND_OF_LIST (LIST: in LIST_TYPE)
    return LIST_KIND;
function KIND_OF_ITEM (LIST:          in LIST_TYPE;
                       ITEM_POSITION: in POSITION_COUNT)
    return ITEM_KIND;
function KIND_OF_ITEM (LIST:          in LIST_TYPE;
                       ITEM_NAME:     in IDENTIFIER_TEXT)
    return ITEM_KIND;
function KIND_OF_ITEM (LIST:          in LIST_TYPE;
                       ITEM_NAME:     in TOKEN_TYPE)
    return ITEM_KIND;
procedure SPLICE (LIST:        in out LIST_TYPE;
                  POSITION:    in     INSERT_COUNT;
                  SOURCE_LIST: in     LIST_TYPE);
procedure CONCATENATE_LISTS (FRONT:  in     LIST_TYPE;
                             BACK:   in     LIST_TYPE;
                             RESULT: in out LIST_TYPE);
procedure EXTRACT_LIST (LIST:           in     LIST_TYPE;
                        START_POSITION: in     POSITION_COUNT;
                        END_POSITION:   in     POSITION_COUNT;
                        RESULT_LIST:    in out LIST_TYPE);
function NUMBER_OF_ITEMS (LIST: in LIST_TYPE)
    return LIST_SIZE;
function POSITION_OF_CURRENT_LIST (LIST: in LIST_TYPE)
    return POSITION_COUNT;
function CURRENT_LIST_IS_OUTERMOST (LIST: in LIST_TYPE)
    return BOOLEAN;
procedure MAKE_CONTAINING_LIST_CURRENT (IN_LIST: in out LIST_TYPE);
procedure MAKE_THIS_ITEM_CURRENT (IN_LIST:       in out LIST_TYPE;
                                  ITEM_POSITION: in     POSITION_COUNT);
procedure MAKE_THIS_ITEM_CURRENT (IN_LIST:       in out LIST_TYPE;
                                  ITEM_NAME:     in     IDENTIFIER_TEXT);
procedure MAKE_THIS_ITEM_CURRENT (IN_LIST:       in out LIST_TYPE;
                                  ITEM_NAME:     in     TOKEN_TYPE);
```

Downloaded from http://www.everyspec.com


```
function TEXT_LENGTH (LIST:           in LIST_TYPE)
    return CAIS_POSITIVE;
function TEXT_LENGTH (LIST:           in LIST_TYPE;
                      ITEM_POSITION:  in POSITION_COUNT)
    return CAIS_POSITIVE;
function TEXT_LENGTH (LIST:           in LIST_TYPE;
                      ITEM_NAME:      in IDENTIFIER_TEXT)
    return CAIS_POSITIVE;
function TEXT_LENGTH (LIST:           in LIST_TYPE;
                      ITEM_NAME:      in TOKEN_TYPE)
    return CAIS_POSITIVE;
procedure GET_ITEM_NAME (LIST:           in     LIST_TYPE;
                         ITEM_POSITION:  in     POSITION_COUNT;
                         NAME:           in out TOKEN_TYPE);
function POSITION_BY_NAME (LIST:       in LIST_TYPE;
                           ITEM_NAME:  in IDENTIFIER_TEXT)
    return POSITION_COUNT;
function POSITION_BY_NAME (LIST:       in LIST_TYPE;
                           ITEM_NAME:  in TOKEN_TYPE)
    return POSITION_COUNT;


package CAIS_LIST_ITEM is

procedure EXTRACT_VALUE (FROM_LIST:      in     LIST_TYPE;
                         ITEM_POSITION:  in     POSITION_COUNT;
                         VALUE:          in out LIST_TYPE);
procedure EXTRACT_VALUE (FROM_LIST:      in     LIST_TYPE;
                         ITEM_NAME:      in     IDENTIFIER_TEXT;
                         VALUE:          in out LIST_TYPE);
procedure EXTRACT_VALUE (FROM_LIST:      in     LIST_TYPE;
                         ITEM_NAME:      in     TOKEN_TYPE;
                         VALUE:          in out LIST_TYPE);
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_POSITION:  in     POSITION_COUNT;
                   VALUE:          in     LIST_TYPE);
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     IDENTIFIER_TEXT;
                   VALUE:          in     LIST_TYPE);
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     TOKEN_TYPE;
                   VALUE:          in     LIST_TYPE);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  VALUE:     in     LIST_TYPE);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     IDENTIFIER_TEXT;
                  VALUE:     in     LIST_TYPE);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     TOKEN_TYPE;
                  VALUE:     in     LIST_TYPE);
function POSITION_BY_VALUE
         (LIST:           in LIST_TYPE;
          VALUE:          in LIST_TYPE;
```

Downloaded from http://www.everyspec.com


```
             START_POSITION: in POSITION_COUNT := POSITION_COUNT'FIRST;
             END_POSITION:   in POSITION_COUNT := POSITION_COUNT'LAST)
        return POSITION_COUNT;

    end CAIS_LIST_ITEM;


    package CAIS_IDENTIFIER_ITEM is

    procedure COPY_TOKEN (FROM_TOKEN: in     TOKEN_TYPE;
                          TO_TOKEN:   in out TOKEN_TYPE);
    procedure CONVERT_TEXT_TO_TOKEN (IDENTIFIER: in     IDENTIFIER_TEXT;
                                     TOKEN:      in out TOKEN_TYPE);
    function TEXT_FORM (TOKEN: in TOKEN_TYPE)
        return IDENTIFIER_TEXT;
    function IS_EQUAL (TOKEN1: in TOKEN_TYPE;
                       TOKEN2: in TOKEN_TYPE)
        return BOOLEAN;
    procedure EXTRACT_VALUE (FROM_LIST:     in     LIST_TYPE;
                             ITEM_POSITION: in     POSITION_COUNT;
                             VALUE:         in out TOKEN_TYPE);
    procedure EXTRACT_VALUE (FROM_LIST:     in     LIST_TYPE;
                             ITEM_NAME:     in     IDENTIFIER_TEXT;
                             VALUE:         in out TOKEN_TYPE);
    procedure EXTRACT_VALUE (FROM_LIST:     in     LIST_TYPE;
                             ITEM_NAME:     in     TOKEN_TYPE;
                             VALUE:         in out TOKEN_TYPE);
    procedure REPLACE (IN_LIST:       in out LIST_TYPE;
                       ITEM_POSITION: in     POSITION_COUNT;
                       VALUE:         in     TOKEN_TYPE);
    procedure REPLACE (IN_LIST:       in out LIST_TYPE;
                       ITEM_NAME:     in     IDENTIFIER_TEXT;
                       VALUE:         in     TOKEN_TYPE);
    procedure REPLACE (IN_LIST:       in out LIST_TYPE;
                       ITEM_NAME:     in     TOKEN_TYPE;
                       VALUE:         in     TOKEN_TYPE);
    procedure INSERT (IN_LIST:  in out LIST_TYPE;
                      POSITION: in     INSERT_COUNT;
                      VALUE:    in     TOKEN_TYPE);
    procedure INSERT (IN_LIST:  in out LIST_TYPE;
                      POSITION: in     INSERT_COUNT;
                      NAME:     in     IDENTIFIER_TEXT;
                      VALUE:    in     TOKEN_TYPE);
    procedure INSERT (IN_LIST:  in out LIST_TYPE;
                      POSITION: in     INSERT_COUNT;
                      NAME:     in     TOKEN_TYPE;
                      VALUE:    in     TOKEN_TYPE);
    function POSITION_BY_VALUE
             (LIST:           in LIST_TYPE;
              VALUE:          in TOKEN_TYPE;
              START_POSITION: in POSITION_COUNT := POSITION_COUNT'FIRST;
              END_POSITION:   in POSITION_COUNT := POSITION_COUNT'LAST)
        return POSITION_COUNT;

    end CAIS_IDENTIFIER_ITEM;
```

Downloaded from http://www.everyspec.com


```
generic
     type NUMBER is range <>;
package CAIS_INTEGER_ITEM is

function TEXT_FORM (INTEGER_VALUE: in NUMBER)
     return STRING;
function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_POSITION: in POSITION_COUNT)
     return NUMBER;
function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_NAME:      in IDENTIFIER_TEXT)
     return NUMBER;
function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_NAME:      in TOKEN_TYPE)
     return NUMBER;
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_POSITION: in     POSITION_COUNT;
                   VALUE:          in     NUMBER);
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     IDENTIFIER_TEXT;
                   VALUE:          in     NUMBER);
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     TOKEN_TYPE;
                   VALUE:          in     NUMBER);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION: in     INSERT_COUNT;
                  VALUE:     in     NUMBER);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION: in     INSERT_COUNT;
                  NAME:      in     IDENTIFIER_TEXT;
                  VALUE:     in     NUMBER);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION: in     INSERT_COUNT;
                  NAME:      in     TOKEN_TYPE;
                  VALUE:     in     NUMBER);
function POSITION_BY_VALUE
          (LIST:            in LIST_TYPE;
           VALUE:           in NUMBER;
           START_POSITION: in POSITION_COUNT := POSITION_COUNT'FIRST;
           END_POSITION:   in POSITION_COUNT := POSITION_COUNT'LAST)
     return POSITION_COUNT;

end CAIS_INTEGER_ITEM;


generic
     type NUMBER is digits <>;
package CAIS_FLOAT_ITEM is

function TEXT_FORM (FLOAT_VALUE: in NUMBER)
     return STRING;
function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_POSITION: in POSITION_COUNT)
     return NUMBER;
function EXTRACTED_VALUE (FROM_LIST:      in LIST_TYPE;
                          ITEM_NAME:      in IDENTIFIER_TEXT)
     return NUMBER;
```

Downloaded from http://www.everyspec.com


```
function EXTRACTED_VALUE (FROM_LIST:     in LIST_TYPE;
                          ITEM_NAME:     in TOKEN_TYPE)
    return NUMBER;
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_POSITION:  in     POSITION_COUNT;
                   VALUE:          in     NUMBER);
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     IDENTIFIER_TEXT;
                   VALUE:          in     NUMBER);
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     TOKEN_TYPE;
                   VALUE:          in     NUMBER);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  VALUE:     in     NUMBER);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     IDENTIFIER_TEXT;
                  VALUE:     in     NUMBER);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  NAME:      in     TOKEN_TYPE;
                  VALUE:     in     NUMBER);
function POSITION_BY_VALUE
          (LIST:            in LIST_TYPE;
           VALUE:           in NUMBER;
           START_POSITION:  in POSITION_COUNT := POSITION_COUNT'FIRST;
           END_POSITION:    in POSITION_COUNT := POSITION_COUNT'LAST)
    return POSITION_COUNT;

end CAIS_FLOAT_ITEM;


package CAIS_STRING_ITEM is

function EXTRACTED_VALUE (FROM_LIST:     in LIST_TYPE;
                          ITEM_POSITION: in POSITION_COUNT)
    return STRING;
function EXTRACTED_VALUE (FROM_LIST:     in LIST_TYPE;
                          ITEM_NAME:     in IDENTIFIER_TEXT)
    return STRING;
function EXTRACTED_VALUE (FROM_LIST:     in LIST_TYPE;
                          ITEM_NAME:     in TOKEN_TYPE)
    return STRING;
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_POSITION:  in     POSITION_COUNT;
                   VALUE:          in     STRING);
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     IDENTIFIER_TEXT;
                   VALUE:          in     STRING);
procedure REPLACE (IN_LIST:        in out LIST_TYPE;
                   ITEM_NAME:      in     TOKEN_TYPE;
                   VALUE:          in     STRING);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
                  POSITION:  in     INSERT_COUNT;
                  VALUE:     in     STRING);
procedure INSERT (IN_LIST:   in out LIST_TYPE;
```

Downloaded from http://www.everyspec.com


```
                    POSITION:  in      INSERT_COUNT;
                    NAME:      in      IDENTIFIER_TEXT;
                    VALUE:     in      STRING);
procedure INSERT (IN_LIST:     in out LIST_TYPE;
                    POSITION:  in      INSERT_COUNT;
                    NAME:      in      TOKEN_TYPE;
                    VALUE:     in      STRING);
function POSITION_BY_VALUE
          (LIST:             in LIST_TYPE;
           VALUE:            in STRING;
           START_POSITION:   in POSITION_COUNT := POSITION_COUNT'FIRST;
           END_POSITION:     in POSITION_COUNT := POSITION_COUNT'LAST)
    return POSITION_COUNT;

end CAIS_STRING_ITEM;

private
    type LIST_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
    type TOKEN_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
    EMPTY_LIST: constant LIST_TYPE := IMPLEMENTATION_DEFINED;
    -- This constant should be defined by the implementer.
end CAIS_LIST_MANAGEMENT;


with CAIS_STANDARD;
with CAIS_LIST_MANAGEMENT;
package CAIS_DEFINITIONS is

    use CAIS_STANDARD;

    type NODE_TYPE is limited private;

    type NODE_KIND is (FILE, STRUCTURAL, PROCESS);

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

    type INTENT_ARRAY is array (CAIS_POSITIVE range <>) of INTENT_SPECIFICATION;

    subtype PATHNAME          is STRING;
    subtype RELATIONSHIP_KEY  is STRING;
    subtype RELATION_NAME     is STRING;

    subtype ATTRIBUTE_NAME is STRING;
```

Downloaded from http://www.everyspec.com


```
    subtype ATTRIBUTE_LIST              is CAIS_LIST_MANAGEMENT.LIST_TYPE;
    subtype DISCRETIONARY_ACCESS_LIST is CAIS_LIST_MANAGEMENT.LIST_TYPE;
    subtype MANDATORY_ACCESS_LIST      is CAIS_LIST_MANAGEMENT.LIST_TYPE;

    CURRENT_USER:     constant PATHNAME         := "'CURRENT_USER";
    CURRENT_NODE:     constant PATHNAME         := "'CURRENT_NODE";
    CURRENT_PROCESS:  constant PATHNAME         := ":";
    LATEST_KEY:       constant RELATIONSHIP_KEY := "#";
    DEFAULT_RELATION: constant RELATION_NAME    := "DOT";
    LONG_DELAY:       constant CAIS_DURATION    := CAIS_DURATION'LAST;

    ACCESS_VIOLATION:             exception;
    ATTRIBUTE_ERROR:              exception;
    DEVICE_ERROR:                 exception;
    EXISTING_NODE_ERROR:          exception;
    INTENT_VIOLATION:             exception;
    ITERATOR_ERROR:               exception;
    LOCK_ERROR:                   exception;
    NAME_ERROR:                   exception;
    NODE_KIND_ERROR:              exception;
    PATHNAME_SYNTAX_ERROR:        exception;
    PREDEFINED_ATTRIBUTE_ERROR:   exception;
    PREDEFINED_RELATION_ERROR:    exception;
    RELATIONSHIP_ERROR:           exception;
    SECURITY_VIOLATION:           exception;
    STATUS_ERROR:                 exception;
    SYNTAX_ERROR:                 exception;
    USE_ERROR:                    exception;

private
    type NODE_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_DEFINITIONS;


with CAIS_STANDARD;
package CAIS_CALENDAR is

    use CAIS_STANDARD;

    type TIME is private;

    subtype YEAR_NUMBER  is CAIS_INTEGER range 1901 .. 2099;
    subtype MONTH_NUMBER is CAIS_INTEGER range 1 .. 12;
    subtype DAY_NUMBER   is CAIS_INTEGER range 1 .. 31;
    subtype DAY_DURATION is CAIS_DURATION range 0.0 .. 86_400.0;

    TIME_ERROR: exception;

    function CLOCK
        return TIME;
    function YEAR  (DATE: in TIME)
        return YEAR_NUMBER;
    function MONTH (DATE: in TIME)
        return MONTH_NUMBER;
    function DAY  (DATE: in TIME)
```

Downloaded from http://www.everyspec.com


```
        return DAY_NUMBER;

    function SECONDS (DATE: in TIME)
        return DAY_DURATION;
    procedure SPLIT (DATE:     in     TIME;
                     YEAR:        out YEAR_NUMBER;
                     MONTH:       out MONTH_NUMBER;
                     DAY:         out DAY_NUMBER;
                     SECONDS:     out DAY_DURATION);
    function TIME_OF (YEAR:    in YEAR_NUMBER;
                      MONTH:   in MONTH_NUMBER;
                      DAY:     in DAY_NUMBER;
                      SECONDS: in DAY_DURATION)
        return TIME;
    function "+" (LEFT:  in TIME;
                  RIGHT: in CAIS_DURATION)
        return TIME;
    function "+" (LEFT:  in CAIS_DURATION;
                  RIGHT: in TIME)
        return TIME;
    function "-" (LEFT:  in TIME;
                  RIGHT: in CAIS_DURATION)
        return TIME;
    function "-" (LEFT:  in TIME;
                  RIGHT: in TIME)
        return CAIS_DURATION;
    function "<" (LEFT:  in TIME;
                  RIGHT: in TIME)
        return BOOLEAN;
    function "<=" (LEFT:  in TIME;
                   RIGHT: in TIME)
        return BOOLEAN;
    function ">" (LEFT:  in TIME;
                  RIGHT: in TIME)
        return BOOLEAN;
    function ">=" (LEFT:  in TIME;
                   RIGHT: in TIME)
        return BOOLEAN;

private
    type TIME is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_CALENDAR;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_CALENDAR;
with CAIS_LIST_MANAGEMENT;
package CAIS_NODE_MANAGEMENT is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;
    use CAIS_CALENDAR;
    use CAIS_LIST_MANAGEMENT;
```

Downloaded from http://www.everyspec.com


```
procedure OPEN (NODE:        in out NODE_TYPE;
                NAME:        in     PATHNAME;
                INTENT:      in     INTENT_ARRAY;
                TIME_LIMIT:  in     CAIS_DURATION := LONG_DELAY);
procedure OPEN (NODE:        in out NODE_TYPE;
                BASE:        in     NODE_TYPE;
                KEY:         in     RELATIONSHIP_KEY;
                RELATION:    in     RELATION_NAME := DEFAULT_RELATION;
                INTENT:      in     INTENT_ARRAY;
                TIME_LIMIT:  in     CAIS_DURATION := LONG_DELAY);
procedure OPEN (NODE:         in out NODE_TYPE;
                NAME:         in     PATHNAME;
                INTENT:       in     INTENT_SPECIFICATION := READ;
                TIME_LIMIT:   in     CAIS_DURATION := LONG_DELAY);
procedure OPEN (NODE:         in out NODE_TYPE;
                BASE:         in     NODE_TYPE;
                KEY:          in     RELATIONSHIP_KEY;
                RELATION:     in     RELATION_NAME := DEFAULT_RELATION;
                INTENT:       in     INTENT_SPECIFICATION := READ;
                TIME_LIMIT:   in     CAIS_DURATION := LONG_DELAY);
procedure CLOSE (NODE: in out NODE_TYPE);
procedure CHANGE_INTENT (NODE:         in out NODE_TYPE;
                         INTENT:       in     INTENT_ARRAY;
                         TIME_LIMIT:   in     CAIS_DURATION := LONG_DELAY);
procedure CHANGE_INTENT (NODE:         in out NODE_TYPE;
                         INTENT:       in     INTENT_SPECIFICATION;
                         TIME_LIMIT:   in     CAIS_DURATION := LONG_DELAY);
function IS_OPEN (NODE: in NODE_TYPE)
    return BOOLEAN;
function INTENT (NODE: in NODE_TYPE)
    return INTENT_ARRAY;
function KIND_OF_NODE (NODE: in NODE_TYPE)
    return NODE_KIND;
function OPEN_FILE_HANDLE_COUNT (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
function PRIMARY_NAME (NODE: in NODE_TYPE)
    return PATHNAME;
function PRIMARY_KEY (NODE: in NODE_TYPE)
    return RELATIONSHIP_KEY;
function PRIMARY_RELATION (NODE: in NODE_TYPE)
    return RELATION_NAME;
function PATH_KEY (NODE: in NODE_TYPE)
    return RELATIONSHIP_KEY;
function PATH_RELATION (NODE: in NODE_TYPE)
    return RELATION_NAME;
function BASE_PATH (NAME: in PATHNAME)
    return PATHNAME;
function LAST_RELATION (NAME: in PATHNAME)
    return RELATION_NAME;
function LAST_KEY (NAME: in PATHNAME)
    return RELATIONSHIP_KEY;
function IS_OBTAINABLE (NODE: in NODE_TYPE)
    return BOOLEAN;
function IS_OBTAINABLE (NAME: in PATHNAME)
    return BOOLEAN;
function IS_OBTAINABLE (BASE:    in NODE_TYPE;
                        KEY:     in RELATIONSHIP_KEY;
```

Downloaded from http://www.everyspec.com


```
                         RELATION: in RELATION_NAME := DEFAULT_RELATION)
    return BOOLEAN;
function IS_SAME (NODE1: in NODE_TYPE;
                  NODE2: in NODE_TYPE)
    return BOOLEAN;
function IS_SAME (NAME1: in PATHNAME;
                  NAME2: in PATHNAME)
    return BOOLEAN;
function INDEX (NODE:   in NODE_TYPE;
                MODULO: in CAIS_POSITIVE)
    return CAIS_NATURAL;
procedure OPEN_PARENT (PARENT:     in out NODE_TYPE;
                       NODE:       in     NODE_TYPE;
                       INTENT:     in     INTENT_ARRAY;
                       TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY);
procedure OPEN_PARENT (PARENT:     in out NODE_TYPE;
                       NODE:       in     NODE_TYPE;
                       INTENT:     in     INTENT_SPECIFICATION := READ;
                       TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY);
procedure COPY_NODE (FROM:        in NODE_TYPE;
                     TO_BASE:     in NODE_TYPE;
                     TO_KEY:      in RELATIONSHIP_KEY;
                     TO_RELATION: in RELATION_NAME := DEFAULT_RELATION);
procedure COPY_NODE (FROM: in NODE_TYPE;
                     TO:   in PATHNAME);
procedure COPY_TREE (FROM:        in NODE_TYPE;
                     TO_BASE:     in NODE_TYPE;
                     TO_KEY:      in RELATIONSHIP_KEY;
                     TO_RELATION: in RELATION_NAME := DEFAULT_RELATION);
procedure COPY_TREE (FROM: in NODE_TYPE;
                     TO:   in PATHNAME);
procedure RENAME (NODE:         in NODE_TYPE;
                  NEW_BASE:     in NODE_TYPE;
                  NEW_KEY:      in RELATIONSHIP_KEY;
                  NEW_RELATION: in RELATION_NAME := DEFAULT_RELATION);
procedure RENAME (NODE:     in NODE_TYPE;
                  NEW_NAME: in PATHNAME);
procedure DELETE_NODE (NODE:       in out NODE_TYPE;
                       TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY);
procedure DELETE_NODE (NAME: in PATHNAME);
procedure DELETE_TREE (NODE: in out NODE_TYPE);
procedure DELETE_TREE (NAME: in PATHNAME);
procedure CREATE_SECONDARY_RELATIONSHIP
               (TARGET_NODE:  in NODE_TYPE;
                SOURCE_BASE:  in NODE_TYPE;
                NEW_KEY:      in RELATIONSHIP_KEY;
                NEW_RELATION: in RELATION_NAME := DEFAULT_RELATION;
                INHERITABLE:  in BOOLEAN := FALSE);
procedure CREATE_SECONDARY_RELATIONSHIP
                       (TARGET_NODE: in NODE_TYPE;
                        NEW_NAME:    in PATHNAME;
                        INHERITABLE: in BOOLEAN := FALSE);
procedure DELETE_SECONDARY_RELATIONSHIP
                        (BASE:     in NODE_TYPE;
                         KEY:      in RELATIONSHIP_KEY;
                         RELATION: in RELATION_NAME := DEFAULT_RELATION);
procedure DELETE_SECONDARY_RELATIONSHIP (NAME: in PATHNAME);
```

Downloaded from http://www.everyspec.com


```
procedure SET_INHERITANCE
                    (BASE:          in NODE_TYPE;
                     KEY:           in RELATIONSHIP_KEY;
                     RELATION:      in RELATION_NAME := DEFAULT_RELATION;
                     INHERITABLE:   in BOOLEAN);
procedure SET_INHERITANCE (NAME:          in PATHNAME;
                           INHERITABLE:   in BOOLEAN);
function IS_INHERITABLE (BASE:       in NODE_TYPE;
                         KEY:        in RELATIONSHIP_KEY;
                         RELATION:   in RELATION_NAME := DEFAULT_RELATION)
    return BOOLEAN;
function IS_INHERITABLE (NAME: in PATHNAME)
    return BOOLEAN;

type NODE_ITERATOR is limited private;

subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
subtype RELATION_NAME_PATTERN    is RELATION_NAME;

type RELATIONSHIP_KIND is (PRIMARY, SECONDARY, BOTH);
type NODE_KIND_ARRAY   is array (CAIS_NATURAL range <>)
                          of NODE_KIND;

procedure CREATE_ITERATOR
    (ITERATOR:          in out NODE_ITERATOR;
     NODE:              in     NODE_TYPE;
     KIND:              in     NODE_KIND_ARRAY := (FILE, STRUCTURAL);
     KEY:               in     RELATIONSHIP_KEY_PATTERN := "*";
     RELATION:          in     RELATION_NAME_PATTERN := DEFAULT_RELATION;
     KIND_OF_RELATION:  in     RELATIONSHIP_KIND := PRIMARY);
procedure CREATE_ITERATOR
    (ITERATOR:          in out NODE_ITERATOR;
     NAME:              in     PATHNAME;
     KIND:              in     NODE_KIND_ARRAY := (FILE, STRUCTURAL);
     KEY:               in     RELATIONSHIP_KEY_PATTERN := "*";
     RELATION:          in     RELATION_NAME_PATTERN := DEFAULT_RELATION;
     KIND_OF_RELATION:  in     RELATIONSHIP_KIND := PRIMARY);
function MORE (ITERATOR: in NODE_ITERATOR)
    return BOOLEAN;
function APPROXIMATE_SIZE (ITERATOR: in NODE_ITERATOR)
    return CAIS_NATURAL;
procedure GET_NEXT (ITERATOR:    in out  NODE_ITERATOR;
                    NEXT_NODE:   in out  NODE_TYPE;
                    INTENT:      in      INTENT_ARRAY;
                    TIME_LIMIT:  in      CAIS_DURATION := LONG_DELAY);
procedure GET_NEXT (ITERATOR:    in out NODE_ITERATOR;
                    NEXT_NODE:   in out NODE_TYPE;
                    INTENT:      in     INTENT_SPECIFICATION := NO_ACCESS;
                    TIME_LIMIT:  in     CAIS_DURATION := LONG_DELAY);
procedure SKIP_NEXT (ITERATOR: in out NODE_ITERATOR);
function NEXT_NAME (ITERATOR: in NODE_ITERATOR)
    return PATHNAME;
procedure DELETE_ITERATOR (ITERATOR: in out NODE_ITERATOR);
procedure SET_CURRENT_NODE (NODE:         in NODE_TYPE;
                            TIME_LIMIT:   in CAIS_DURATION := LONG_DELAY);
procedure SET_CURRENT_NODE (NAME:         in PATHNAME;
                            TIME_LIMIT:   in CAIS_DURATION := LONG_DELAY);
```

Downloaded from http://www.everyspec.com


```
    procedure GET_CURRENT_NODE
                        (NODE:       in out NODE_TYPE;
                         INTENT:     in     INTENT_ARRAY;
                         TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY);
    procedure GET_CURRENT_NODE
                        (NODE:       in out NODE_TYPE;
                         INTENT:     in     INTENT_SPECIFICATION := NO_ACCESS;
                         TIME_LIMIT: in     CAIS_DURATION := LONG_DELAY);
    function TIME_CREATED (NODE: in NODE_TYPE)
        return CAIS_CALENDAR.TIME;
    function TIME_CREATED (NAME: in PATHNAME)
        return CAIS_CALENDAR.TIME;
    function TIME_RELATIONSHIP_WRITTEN (NODE: in NODE_TYPE)
        return CAIS_CALENDAR.TIME;
    function TIME_RELATIONSHIP_WRITTEN (NAME: in PATHNAME)
        return CAIS_CALENDAR.TIME;
    function TIME_CONTENTS_WRITTEN (NODE: in NODE_TYPE)
        return CAIS_CALENDAR.TIME;
    function TIME_CONTENTS_WRITTEN (NAME: in PATHNAME)
        return CAIS_CALENDAR.TIME;
    function TIME_ATTRIBUTE_WRITTEN (NODE: in NODE_TYPE)
        return CAIS_CALENDAR.TIME;
    function TIME_ATTRIBUTE_WRITTEN (NAME: in PATHNAME)
        return CAIS_CALENDAR.TIME;

private
    type NODE_ITERATOR is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_NODE_MANAGEMENT;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_LIST_MANAGEMENT;
package CAIS_ATTRIBUTE_MANAGEMENT is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;
    use CAIS_LIST_MANAGEMENT;

    procedure CREATE_NODE_ATTRIBUTE (NODE:      in NODE_TYPE;
                                     ATTRIBUTE: in ATTRIBUTE_NAME;
                                     VALUE:     in LIST_TYPE);
    procedure CREATE_NODE_ATTRIBUTE (NAME:      in PATHNAME;
                                     ATTRIBUTE: in ATTRIBUTE_NAME;
                                     VALUE:     in LIST_TYPE);
    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);
    procedure CREATE_PATH_ATTRIBUTE (NAME:      in PATHNAME;
                                     ATTRIBUTE: in ATTRIBUTE_NAME;
                                     VALUE:     in LIST_TYPE);
    procedure DELETE_NODE_ATTRIBUTE (NODE:      in NODE_TYPE;
```

Downloaded from http://www.everyspec.com


```
                                   ATTRIBUTE: in ATTRIBUTE_NAME);

procedure DELETE_NODE_ATTRIBUTE (NAME:      in PATHNAME;
                                   ATTRIBUTE: in ATTRIBUTE_NAME);
procedure DELETE_PATH_ATTRIBUTE
                           (BASE:       in NODE_TYPE;
                            KEY:        in RELATIONSHIP_KEY;
                            RELATION:   in RELATION_NAME := DEFAULT_RELATION;
                            ATTRIBUTE:  in ATTRIBUTE_NAME);
procedure DELETE_PATH_ATTRIBUTE (NAME:      in PATHNAME;
                                   ATTRIBUTE: in ATTRIBUTE_NAME);
procedure SET_NODE_ATTRIBUTE (NODE:      in NODE_TYPE;
                               ATTRIBUTE: in ATTRIBUTE_NAME;
                               VALUE:     in LIST_TYPE);
procedure SET_NODE_ATTRIBUTE (NAME:      in PATHNAME;
                               ATTRIBUTE: in ATTRIBUTE_NAME;
                               VALUE:     in LIST_TYPE);
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);
procedure SET_PATH_ATTRIBUTE (NAME:      in PATHNAME;
                               ATTRIBUTE: in ATTRIBUTE_NAME;
                               VALUE:     in LIST_TYPE);
procedure GET_NODE_ATTRIBUTE (NODE:      in     NODE_TYPE;
                               ATTRIBUTE: in     ATTRIBUTE_NAME;
                               VALUE:     in out LIST_TYPE);
procedure GET_NODE_ATTRIBUTE (NAME:      in     PATHNAME;
                               ATTRIBUTE: in     ATTRIBUTE_NAME;
                               VALUE:     in out LIST_TYPE);
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);
procedure GET_PATH_ATTRIBUTE (NAME:      in     PATHNAME;
                               ATTRIBUTE: in     ATTRIBUTE_NAME;
                               VALUE:     in out LIST_TYPE);

type    ATTRIBUTE_ITERATOR     is limited private;
subtype ATTRIBUTE_NAME_PATTERN is STRING;

procedure CREATE_NODE_ATTRIBUTE_ITERATOR
                          (ITERATOR: in out ATTRIBUTE_ITERATOR;
                           NODE:     in     NODE_TYPE;
                           PATTERN:  in     ATTRIBUTE_NAME_PATTERN := "*");
procedure CREATE_NODE_ATTRIBUTE_ITERATOR
                          (ITERATOR: in out ATTRIBUTE_ITERATOR;
                           NAME:     in     PATHNAME;
                           PATTERN:  in     ATTRIBUTE_NAME_PATTERN := "*");
procedure CREATE_PATH_ATTRIBUTE_ITERATOR
                         (ITERATOR: in out ATTRIBUTE_ITERATOR;
                          BASE:     in     NODE_TYPE;
                          KEY:      in     RELATIONSHIP_KEY;
```

Downloaded from http://www.everyspec.com


```
                         RELATION: in     RELATION_NAME := DEFAULT_RELATION;
                         PATTERN:  in     ATTRIBUTE_NAME_PATTERN := "*");
    procedure CREATE_PATH_ATTRIBUTE_ITERATOR
                        (ITERATOR: in out ATTRIBUTE_ITERATOR;
                         NAME:     in     PATHNAME;
                         PATTERN:  in     ATTRIBUTE_NAME_PATTERN := "*");
    function MORE (ITERATOR: in ATTRIBUTE_ITERATOR)
        return BOOLEAN;
    function APPROXIMATE_SIZE (ITERATOR: in ATTRIBUTE_ITERATOR)
        return CAIS_NATURAL;
    function NEXT_NAME (ITERATOR: in ATTRIBUTE_ITERATOR)
        return ATTRIBUTE_NAME;
    procedure GET_NEXT_VALUE (ITERATOR: in out ATTRIBUTE_ITERATOR;
                              VALUE:    in out LIST_TYPE);
    procedure SKIP_NEXT (ITERATOR: in ATTRIBUTE_ITERATOR);
    procedure DELETE_ITERATOR (ITERATOR: in out ATTRIBUTE_ITERATOR);

private
    type ATTRIBUTE_ITERATOR is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_ATTRIBUTE_MANAGEMENT;



with CAIS_DEFINITIONS;
with CAIS_LIST_MANAGEMENT;
package CAIS_ACCESS_CONTROL_MANAGEMENT is

    use CAIS_DEFINITIONS;

    subtype GRANT_VALUE   is CAIS_LIST_MANAGEMENT.LIST_TYPE;
    subtype ACCESS_RIGHTS is STRING;

    function ALL_RIGHTS
        return DISCRETIONARY_ACCESS_LIST;
    procedure SET_GRANTED_RIGHTS (NODE:       in NODE_TYPE;
                                  GROUP_NODE: in NODE_TYPE;
                                  GRANT:      in GRANT_VALUE);
    procedure SET_GRANTED_RIGHTS (NAME:       in PATHNAME;
                                  GROUP_NAME: in PATHNAME;
                                  GRANT:      in GRANT_VALUE);
    procedure DELETE_GRANTED_RIGHTS (NODE:       in NODE_TYPE;
                                     GROUP_NODE: in NODE_TYPE);
    procedure DELETE_GRANTED_RIGHTS (NAME:       in PATHNAME;
                                     GROUP_NAME: in PATHNAME);
    procedure GET_GRANTED_RIGHTS (NODE:       in     NODE_TYPE;
                                  GROUP_NODE: in     NODE_TYPE;
                                  GRANT:      in out GRANT_VALUE);
    procedure GET_GRANTED_RIGHTS (NAME:       in     PATHNAME;
                                  GROUP_NAME: in     PATHNAME;
                                  GRANT:      in out GRANT_VALUE);
    function IS_APPROVED (OBJECT_NODE:  in NODE_TYPE;
                          ACCESS_RIGHT: in ACCESS_RIGHTS)
        return BOOLEAN;
    function IS_APPROVED (OBJECT_NAME:  in PATHNAME;
                          ACCESS_RIGHT: in ACCESS_RIGHTS)
        return BOOLEAN;
```

Downloaded from http://www.everyspec.com


```
     procedure ADOPT_ROLE (GROUP_NODE:   in NODE_TYPE;
                           KEY:          in RELATIONSHIP_KEY := LATEST_KEY;
                           INHERITABLE:  in BOOLEAN := TRUE);
     procedure ADOPT_ROLE (GROUP_NAME:   in PATHNAME;
                           KEY:          in RELATIONSHIP_KEY := LATEST_KEY;
                           INHERITABLE:  in BOOLEAN := TRUE);
     procedure UNADOPT_ROLE (KEY: in RELATIONSHIP_KEY);

end CAIS_ACCESS_CONTROL_MANAGEMENT;



with CAIS_DEFINITIONS;
with CAIS_ACCESS_CONTROL_MANAGEMENT;
with CAIS_LIST_MANAGEMENT;
package CAIS_STRUCTURAL_NODE_MANAGEMENT is

     use CAIS_DEFINITIONS;
     use CAIS_LIST_MANAGEMENT;

     procedure CREATE_NODE
       (NODE:                   in out NODE_TYPE;
        BASE:                   in     NODE_TYPE;
        KEY:                    in     RELATIONSHIP_KEY := LATEST_KEY;
        RELATION:               in     RELATION_NAME := DEFAULT_RELATION;
        INTENT:                 in     INTENT_ARRAY := (1=>WRITE);
        ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
        DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                   CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
        MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
     procedure CREATE_NODE
       (NODE:                   in out NODE_TYPE;
        NAME:                   in     PATHNAME;
        INTENT:                 in     INTENT_ARRAY := (1=>WRITE);
        ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
        DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                  CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
        MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
     procedure CREATE_NODE
       (BASE:                   in NODE_TYPE;
        KEY:                    in RELATIONSHIP_KEY := LATEST_KEY;
        RELATION:               in RELATION_NAME := DEFAULT_RELATION;
        INTENT:                 in INTENT_ARRAY := (1=>WRITE);
        ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
        DISCRETIONARY_ACCESS:   in DISCRETIONARY_ACCESS_LIST :=
                                   CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
        MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST);
     procedure CREATE_NODE
       (NAME:                   in PATHNAME;
        INTENT:                 in INTENT_ARRAY := (1=>WRITE);
        ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
        DISCRETIONARY_ACCESS:   in DISCRETIONARY_ACCESS_LIST :=
                                   CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
        MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST);

end CAIS_STRUCTURAL_NODE_MANAGEMENT;
```

Downloaded from http://www.everyspec.com


```
with CAIS_DEFINITIONS;
with CAIS_LIST_MANAGEMENT;
package CAIS_PROCESS_DEFINITIONS is

     use CAIS_DEFINITIONS;

     type PROCESS_STATUS_KIND is (READY, SUSPENDED, ABORTED, TERMINATED);

     subtype RESULTS_LIST   is CAIS_LIST_MANAGEMENT.LIST_TYPE;
     subtype RESULTS_STRING is STRING;
     subtype PARAMETER_LIST is CAIS_LIST_MANAGEMENT.LIST_TYPE;

     ROOT_PROCESS:     constant PATHNAME := "'CURRENT_JOB";
     STANDARD_INPUT:   constant PATHNAME := "'STANDARD_INPUT";
     STANDARD_OUTPUT:  constant PATHNAME := "'STANDARD_OUTPUT";
     STANDARD_ERROR:   constant PATHNAME := "'STANDARD_ERROR";

     EXECUTABLE_IMAGE_ERROR:  exception;

end CAIS_PROCESS_DEFINITIONS;


with CAIS_STANDARD;
with CAIS_CALENDAR;
with CAIS_DEFINITIONS;
with CAIS_LIST_MANAGEMENT;
with CAIS_PROCESS_DEFINITIONS;
with CAIS_ACCESS_CONTROL_MANAGEMENT;
package CAIS_PROCESS_MANAGEMENT is

     use CAIS_STANDARD;
     use CAIS_DEFINITIONS;
     use CAIS_LIST_MANAGEMENT;
     use CAIS_PROCESS_DEFINITIONS;

     procedure SPAWN_PROCESS
        (NODE:                   in out NODE_TYPE;
         FILE_NODE:              in     NODE_TYPE;
         INTENT:                 in     INTENT_ARRAY;
         INPUT_PARAMETERS:       in     PARAMETER_LIST := EMPTY_LIST;
         KEY:                    in     RELATIONSHIP_KEY := LATEST_KEY;
         RELATION:               in     RELATION_NAME := DEFAULT_RELATION;
         DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
         ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
         INPUT_FILE:             in     PATHNAME := STANDARD_INPUT;
         OUTPUT_FILE:            in     PATHNAME := STANDARD_OUTPUT;
         ERROR_FILE:             in     PATHNAME := STANDARD_ERROR;
         ENVIRONMENT_NODE:       in     PATHNAME := CURRENT_NODE);
     procedure SPAWN_PROCESS
        (NODE:                   in out NODE_TYPE;
         FILE_NODE:              in     NODE_TYPE;
         INTENT:                 in     INTENT_SPECIFICATION := READ_ATTRIBUTES;
         INPUT_PARAMETERS:       in     PARAMETER_LIST := EMPTY_LIST;
         KEY:                    in     RELATIONSHIP_KEY := LATEST_KEY;
```

Downloaded from http://www.everyspec.com


```
    RELATION:                in    RELATION_NAME := DEFAULT_RELATION;
    DISCRETIONARY_ACCESS:    in    DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:        in    MANDATORY_ACCESS_LIST := EMPTY_LIST;
    ATTRIBUTES:              in    ATTRIBUTE_LIST := EMPTY_LIST;
    INPUT_FILE:              in    PATHNAME := STANDARD_INPUT;
    OUTPUT_FILE:             in    PATHNAME := STANDARD_OUTPUT;
    ERROR_FILE:              in    PATHNAME := STANDARD_ERROR;
    ENVIRONMENT_NODE:        in    PATHNAME := CURRENT_NODE);
procedure AWAIT_PROCESS_COMPLETION
                (NODE:       in    NODE_TYPE;
                 TIME_LIMIT: in    CAIS_DURATION := LONG_DELAY);
procedure AWAIT_PROCESS_COMPLETION
                  (NODE:              in      NODE_TYPE;
                   RESULTS_RETURNED:  in out  RESULTS_LIST;
                   STATUS:               out  PROCESS_STATUS_KIND;
                   TIME_LIMIT:        in      CAIS_DURATION := LONG_DELAY);
procedure INVOKE_PROCESS
     (NODE:                   in out NODE_TYPE;
      FILE_NODE:              in     NODE_TYPE;
      INTENT:                 in     INTENT_ARRAY;
      RESULTS_RETURNED:       in out RESULTS_LIST;
      STATUS:                    out PROCESS_STATUS_KIND;
      INPUT_PARAMETERS:       in     PARAMETER_LIST;
      KEY:                    in     RELATIONSHIP_KEY := LATEST_KEY;
      RELATION:               in     RELATION_NAME := DEFAULT_RELATION;
      DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
      MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
      ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
      INPUT_FILE:             in     PATHNAME := STANDARD_INPUT;
      OUTPUT_FILE:            in     PATHNAME := STANDARD_OUTPUT;
      ERROR_FILE:             in     PATHNAME := STANDARD_ERROR;
      ENVIRONMENT_NODE:       in     PATHNAME := CURRENT_NODE;
      TIME_LIMIT:             in     CAIS_DURATION := LONG_DELAY);
procedure INVOKE_PROCESS
   (NODE:                   in out NODE_TYPE;
    FILE_NODE:              in     NODE_TYPE;
    INTENT:                 in     INTENT_SPECIFICATION := READ_ATTRIBUTES;
    RESULTS_RETURNED:       in out RESULTS_LIST;
    STATUS:                    out PROCESS_STATUS_KIND;
    INPUT_PARAMETERS:       in     PARAMETER_LIST;
    KEY:                    in     RELATIONSHIP_KEY := LATEST_KEY;
    RELATION:               in     RELATION_NAME := DEFAULT_RELATION;
    DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                               CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
    ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
    INPUT_FILE:             in     PATHNAME := STANDARD_INPUT;
    OUTPUT_FILE:            in     PATHNAME := STANDARD_OUTPUT;
    ERROR_FILE:             in     PATHNAME := STANDARD_ERROR;
    ENVIRONMENT_NODE:       in     PATHNAME := CURRENT_NODE;
    TIME_LIMIT:             in     CAIS_DURATION := LONG_DELAY);
procedure CREATE_JOB
   (FILE_NODE:              in NODE_TYPE;
    INPUT_PARAMETERS:       in PARAMETER_LIST := EMPTY_LIST;
    KEY:                    in RELATIONSHIP_KEY := LATEST_KEY;
```

Downloaded from http://www.everyspec.com


```
        DISCRETIONARY_ACCESS:    in DISCRETIONARY_ACCESS_LIST :=
                                   CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
        MANDATORY_ACCESS:        in MANDATORY_ACCESS_LIST := EMPTY_LIST;
        ATTRIBUTES:              in ATTRIBUTE_LIST := EMPTY_LIST;
        INPUT_FILE:              in PATHNAME := STANDARD_INPUT;
        OUTPUT_FILE:             in PATHNAME := STANDARD_OUTPUT;
        ERROR_FILE:              in PATHNAME := STANDARD_ERROR;
        ENVIRONMENT_NODE:        in PATHNAME := CURRENT_USER;
        DELETE_WHEN_TERMINATED: in BOOLEAN := TRUE);
procedure DELETE_JOB (NODE: in out NODE_TYPE);
procedure DELETE_JOB (NAME: in PATHNAME);
procedure APPEND_RESULTS (RESULTS: in RESULTS_STRING);
procedure WRITE_RESULTS (RESULTS: in RESULTS_STRING);
procedure GET_RESULTS (NODE:      in      NODE_TYPE;
                       RESULTS:   in out  RESULTS_LIST);
procedure GET_RESULTS (NODE:      in      NODE_TYPE;
                       RESULTS:   in out  RESULTS_LIST;
                       STATUS:       out  PROCESS_STATUS_KIND);
procedure GET_RESULTS (NAME:      in      PATHNAME;
                       RESULTS:   in out  RESULTS_LIST;
                       STATUS:       out  PROCESS_STATUS_KIND);
procedure GET_RESULTS (NAME:      in      PATHNAME;
                       RESULTS:   in out  RESULTS_LIST);
function CURRENT_STATUS (NODE: in NODE_TYPE)
    return PROCESS_STATUS_KIND;
function CURRENT_STATUS (NAME: in PATHNAME)
    return PROCESS_STATUS_KIND;
procedure GET_PARAMETERS (PARAMETERS: in out PARAMETER_LIST);
procedure ABORT_PROCESS (NODE:     in NODE_TYPE;
                         RESULTS:  in RESULTS_STRING);
procedure ABORT_PROCESS (NAME:     in PATHNAME;
                         RESULTS:  in RESULTS_STRING);
procedure ABORT_PROCESS (NODE: in NODE_TYPE);
procedure ABORT_PROCESS (NAME: in PATHNAME);
procedure SUSPEND_PROCESS (NODE: in NODE_TYPE);
procedure SUSPEND_PROCESS (NAME: in PATHNAME);
procedure RESUME_PROCESS (NODE: in NODE_TYPE);
procedure RESUME_PROCESS (NAME: in PATHNAME);
function OPEN_NODE_HANDLE_COUNT (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
function OPEN_NODE_HANDLE_COUNT (NAME: in PATHNAME)
    return CAIS_NATURAL;
function IO_UNIT_COUNT (NODE: in NODE_TYPE)
    return CAIS_NATURAL;
function IO_UNIT_COUNT (NAME: in PATHNAME)
    return CAIS_NATURAL;
function TIME_STARTED (NODE: in NODE_TYPE)
    return CAIS_CALENDAR.TIME;
function TIME_STARTED (NAME: in PATHNAME)
    return CAIS_CALENDAR.TIME;
function TIME_FINISHED (NODE: in NODE_TYPE)
    return CAIS_CALENDAR.TIME;
function TIME_FINISHED (NAME: in PATHNAME)
    return CAIS_CALENDAR.TIME;
function MACHINE_TIME (NODE: in NODE_TYPE)
    return CAIS_DURATION;
function MACHINE_TIME (NAME: in PATHNAME)
```

Downloaded from http://www.everyspec.com


```
        return CAIS_DURATION;

    function PROCESS_SIZE (NODE: in NODE_TYPE)
        return CAIS_NATURAL;
    function PROCESS_SIZE (NAME: in PATHNAME)
        return CAIS_NATURAL;

end CAIS_PROCESS_MANAGEMENT;


package CAIS_DEVICES is

    type ACCESS_METHOD_KIND is (DIRECT, SEQUENTIAL, TEXT,
                    implementation_defined);

    type DEVICE_KIND_TYPE is (SCROLL_TERMINAL, PAGE_TERMINAL,
                FORM_TERMINAL, MAGNETIC_TAPE_DRIVE, implementation_defined);

end CAIS_DEVICES;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_DEVICES;
with CAIS_LIST_MANAGEMENT;
package CAIS_IO_DEFINITIONS is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;

    type FILE_KIND is (SECONDARY_STORAGE, QUEUE, DEVICE);

    type QUEUE_KIND is
            (SYNCHRONOUS_SOLO, NONSYNCHRONOUS_SOLO,
             NONSYNCHRONOUS_COPY, NONSYNCHRONOUS_MIMIC);

    type DEVICE_KIND_ARRAY is array (CAIS_POSITIVE range <>)
                        of CAIS_DEVICES.DEVICE_KIND_TYPE;

    IN_INTENT:     constant INTENT_ARRAY := (1=>READ_CONTENTS);
    INOUT_INTENT:  constant INTENT_ARRAY := (READ_CONTENTS, WRITE_CONTENTS);
    OUT_INTENT:    constant INTENT_ARRAY := (1=>WRITE_CONTENTS);
    APPEND_INTENT: constant INTENT_ARRAY := (1=>APPEND_CONTENTS);

    UNBOUNDED_FILE_SIZE:  constant CAIS_NATURAL := 0;
    UNBOUNDED_QUEUE_SIZE: constant CAIS_NATURAL := 0;

    DATA_ERROR:                exception;
    END_ERROR:                 exception;
    FILE_KIND_ERROR:           exception;
    FORM_STATUS_ERROR:         exception;
    FUNCTION_KEY_STATUS_ERROR: exception;
    LAYOUT_ERROR:              exception;
    MODE_ERROR:                exception;
    TERMINAL_POSITION_ERROR:   exception;
```

Downloaded from http://www.everyspec.com


```
end CAIS_IO_DEFINITIONS;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_DEVICES;
with CAIS_IO_DEFINITIONS;
package CAIS_IO_ATTRIBUTES is

     use CAIS_STANDARD;
     use CAIS_DEFINITIONS;
     use CAIS_IO_DEFINITIONS;

     function ACCESS_METHOD (NODE: in NODE_TYPE)
          return CAIS_DEVICES.ACCESS_METHOD_KIND;
     function ACCESS_METHOD (NAME: in PATHNAME)
          return CAIS_DEVICES.ACCESS_METHOD_KIND;
     function KIND_OF_FILE (NODE: in NODE_TYPE)
          return FILE_KIND;
     function KIND_OF_FILE (NAME: in PATHNAME)
          return FILE_KIND;
     function KIND_OF_QUEUE (NODE: in NODE_TYPE)
          return QUEUE_KIND;
     function KIND_OF_QUEUE (NAME: in PATHNAME)
          return QUEUE_KIND;
     function KIND_OF_DEVICE (NODE: in NODE_TYPE)
          return DEVICE_KIND_ARRAY;
     function KIND_OF_DEVICE (NAME: in PATHNAME)
          return DEVICE_KIND_ARRAY;
     function CURRENT_FILE_SIZE (NODE: in NODE_TYPE)
          return CAIS_NATURAL;
     function CURRENT_FILE_SIZE (NAME: in PATHNAME)
          return CAIS_NATURAL;
     function MAXIMUM_FILE_SIZE (NODE: in NODE_TYPE)
          return CAIS_NATURAL;
     function MAXIMUM_FILE_SIZE (NAME: in PATHNAME)
          return CAIS_NATURAL;
     function CURRENT_QUEUE_SIZE (NODE: in NODE_TYPE)
          return CAIS_NATURAL;
     function CURRENT_QUEUE_SIZE (NAME: in PATHNAME)
          return CAIS_NATURAL;
     function MAXIMUM_QUEUE_SIZE (NODE: in NODE_TYPE)
          return CAIS_NATURAL;
     function MAXIMUM_QUEUE_SIZE (NAME: in PATHNAME)
          return CAIS_NATURAL;

end CAIS_IO_ATTRIBUTES;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_IO_DEFINITIONS;
with CAIS_LIST_MANAGEMENT;
with CAIS_ACCESS_CONTROL_MANAGEMENT;
generic
```

Downloaded from http://www.everyspec.com


```
    type ELEMENT_TYPE is private;
package CAIS_DIRECT_IO is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;
    use CAIS_IO_DEFINITIONS;
    use CAIS_LIST_MANAGEMENT;

    type    COUNT          is range 0 .. implementation_defined;
    subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

    type FILE_TYPE is limited private;

    type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);

    procedure CREATE
      (NODE:                  in out NODE_TYPE;
       FILE:                  in out FILE_TYPE;
       BASE:                  in     NODE_TYPE;
       KEY:                   in     RELATIONSHIP_KEY := LATEST_KEY;
       RELATION:              in     RELATION_NAME := DEFAULT_RELATION;
       INTENT:                in     INTENT_ARRAY := INOUT_INTENT;
       MODE:                  in     FILE_MODE := INOUT_FILE;
       ATTRIBUTES:            in     ATTRIBUTE_LIST := EMPTY_LIST;
       MAXIMUM_FILE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
       DISCRETIONARY_ACCESS:  in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
       MANDATORY_ACCESS:      in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
    procedure CREATE
      (NODE:                  in out NODE_TYPE;
       FILE:                  in out FILE_TYPE;
       NAME:                  in     PATHNAME;
       INTENT:                in     INTENT_ARRAY := INOUT_INTENT;
       MODE:                  in     FILE_MODE := INOUT_FILE;
       ATTRIBUTES:            in     ATTRIBUTE_LIST := EMPTY_LIST;
       MAXIMUM_FILE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
       DISCRETIONARY_ACCESS:  in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
       MANDATORY_ACCESS:      in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
    procedure OPEN (FILE: in out FILE_TYPE;
                    NODE: in     NODE_TYPE;
                    MODE: in     FILE_MODE);
    procedure CLOSE (FILE: in out FILE_TYPE);
    procedure RESET (FILE: in out FILE_TYPE;
                     MODE: in     FILE_MODE);
    procedure SYNCHRONIZE (FILE: in FILE_TYPE);

    function MODE (FILE: in FILE_TYPE)
        return FILE_MODE;
    function IS_OPEN (FILE: in FILE_TYPE)
        return BOOLEAN;
    procedure READ (FILE: in     FILE_TYPE;
                    ITEM:    out ELEMENT_TYPE;
                    FROM: in     POSITIVE_COUNT);
    procedure READ (FILE: in     FILE_TYPE;
                    ITEM:    out ELEMENT_TYPE);
    procedure WRITE (FILE: in     FILE_TYPE;
```

Downloaded from http://www.everyspec.com


```
                  ITEM: in      ELEMENT_TYPE;
                  TO:   in      POSITIVE_COUNT);
    procedure WRITE (FILE: in   FILE_TYPE;
                  ITEM: in      ELEMENT_TYPE);
    procedure SET_INDEX (FILE: in FILE_TYPE;
                      TO:    in POSITIVE_COUNT);
    function INDEX (FILE: in FILE_TYPE)
        return POSITIVE_COUNT;
    function SIZE (FILE: in FILE_TYPE)
        return COUNT;
    function END_OF_FILE (FILE: in FILE_TYPE)
        return BOOLEAN;

private
    type FILE_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_DIRECT_IO;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_IO_DEFINITIONS;
with CAIS_LIST_MANAGEMENT;
with CAIS_ACCESS_CONTROL_MANAGEMENT;
generic
    type ELEMENT_TYPE is private;
package CAIS_SEQUENTIAL_IO is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;
    use CAIS_IO_DEFINITIONS;
    use CAIS_LIST_MANAGEMENT;

    type FILE_TYPE is limited private;

    type FILE_MODE is (IN_FILE, OUT_FILE, APPEND_FILE);

    procedure CREATE
        (NODE:                  in out NODE_TYPE;
         FILE:                  in out FILE_TYPE;
         BASE:                  in     NODE_TYPE;
         KEY:                   in     RELATIONSHIP_KEY := LATEST_KEY;
         RELATION:              in     RELATION_NAME := DEFAULT_RELATION;
         INTENT:                in     INTENT_ARRAY := OUT_INTENT;
         MODE:                  in     FILE_MODE := OUT_FILE;
         ATTRIBUTES:            in     ATTRIBUTE_LIST := EMPTY_LIST;
         MAXIMUM_FILE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
         DISCRETIONARY_ACCESS:  in     DISCRETIONARY_ACCESS_LIST :=
                                 CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:      in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
    procedure CREATE
        (NODE:                  in out NODE_TYPE;
         FILE:                  in out FILE_TYPE;
         NAME:                  in     PATHNAME;
         INTENT:                in     INTENT_ARRAY := OUT_INTENT;
         MODE:                  in     FILE_MODE := OUT_FILE;
```

Downloaded from http://www.everyspec.com


```
        ATTRIBUTES:            in     ATTRIBUTE_LIST := EMPTY_LIST;
        MAXIMUM_FILE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
        DISCRETIONARY_ACCESS:  in     DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
        MANDATORY_ACCESS:      in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
    procedure OPEN (FILE:     in out FILE_TYPE;
                    NODE:     in     NODE_TYPE;
                    MODE:     in     FILE_MODE);
    procedure CLOSE (FILE: in out FILE_TYPE);
    procedure RESET (FILE: in out FILE_TYPE;
                     MODE: in     FILE_MODE);
    procedure SYNCHRONIZE (FILE: in FILE_TYPE);

    function MODE (FILE: in FILE_TYPE)
        return FILE_MODE;
    function IS_OPEN (FILE: in FILE_TYPE)
        return BOOLEAN;
    function END_OF_FILE (FILE: in FILE_TYPE)
        return BOOLEAN;
    procedure READ (FILE: in      FILE_TYPE;
                    ITEM:   out   ELEMENT_TYPE);
    procedure WRITE (FILE: in FILE_TYPE;
                     ITEM: in ELEMENT_TYPE);

private
    type FILE_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_SEQUENTIAL_IO;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_IO_DEFINITIONS;
with CAIS_LIST_MANAGEMENT;
with CAIS_ACCESS_CONTROL_MANAGEMENT;
package CAIS_TEXT_IO is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;
    use CAIS_IO_DEFINITIONS;
    use CAIS_LIST_MANAGEMENT;

    type    COUNT          is range 0 .. implementation_defined;
    subtype POSITIVE_COUNT is COUNT range 1 .. COUNT'LAST;

    subtype FIELD       is CAIS_INTEGER range 0 .. implementation_defined;
    subtype NUMBER_BASE is CAIS_INTEGER range 2 .. 16;

    type TYPE_SET is (LOWER_CASE, UPPER_CASE);

    type FILE_TYPE is limited private;

    type FILE_MODE is (IN_FILE, OUT_FILE, APPEND_FILE);

    procedure CREATE
        (NODE:                        in out NODE_TYPE;
```

Downloaded from http://www.everyspec.com


```
    FILE:                    in out FILE_TYPE;
    BASE:                    in     NODE_TYPE;
    KEY:                     in     RELATIONSHIP_KEY := LATEST_KEY;
    RELATION:                in     RELATION_NAME := DEFAULT_RELATION;
    INTENT:                  in     INTENT_ARRAY := OUT_INTENT;
    MODE:                    in     FILE_MODE := OUT_FILE;
    ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
    MAXIMUM_FILE_SIZE:       in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
    DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
procedure CREATE
   (NODE:                    in out NODE_TYPE;
    FILE:                    in out FILE_TYPE;
    NAME:                    in     PATHNAME;
    INTENT:                  in     INTENT_ARRAY := OUT_INTENT;
    MODE:                    in     FILE_MODE := OUT_FILE;
    ATTRIBUTES:              in     ATTRIBUTE_LIST := EMPTY_LIST;
    MAXIMUM_FILE_SIZE:       in     CAIS_NATURAL := UNBOUNDED_FILE_SIZE;
    DISCRETIONARY_ACCESS:    in     DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
    MANDATORY_ACCESS:        in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
procedure OPEN (FILE: in out FILE_TYPE;
                NODE: in     NODE_TYPE;
                MODE: in     FILE_MODE);
procedure CLOSE (FILE: in out FILE_TYPE);
procedure RESET (FILE: in out FILE_TYPE;
                 MODE: in     FILE_MODE);
procedure SYNCHRONIZE (FILE: in FILE_TYPE);

function MODE (FILE: in FILE_TYPE)
    return FILE_MODE;
function IS_OPEN (FILE: in FILE_TYPE)
    return BOOLEAN;
procedure SET_INPUT (FILE: in FILE_TYPE);
procedure SET_OUTPUT (FILE: in FILE_TYPE);
function CURRENT_INPUT
    return FILE_TYPE;
function CURRENT_OUTPUT
    return FILE_TYPE;
procedure SET_LINE_LENGTH (FILE: in FILE_TYPE;
                           TO:   in COUNT);
procedure SET_LINE_LENGTH (TO: in COUNT);
procedure SET_PAGE_LENGTH (FILE: in FILE_TYPE;
                           TO:   in COUNT);
procedure SET_PAGE_LENGTH (TO: in COUNT);
function LINE_LENGTH (FILE: in FILE_TYPE)
    return COUNT;
function LINE_LENGTH
    return COUNT;
function PAGE_LENGTH (FILE: in FILE_TYPE)
    return COUNT;
function PAGE_LENGTH
    return COUNT;
procedure NEW_LINE (FILE:    in FILE_TYPE;
                    SPACING: in POSITIVE_COUNT := 1);
procedure NEW_LINE (SPACING: in POSITIVE_COUNT := 1);
```

Downloaded from http://www.everyspec.com


```
procedure SKIP_LINE (FILE:    in FILE_TYPE;
                     SPACING: in POSITIVE_COUNT := 1);
procedure SKIP_LINE (SPACING: in POSITIVE_COUNT := 1);
function END_OF_LINE (FILE: in FILE_TYPE)
    return BOOLEAN;
function END_OF_LINE
    return BOOLEAN;
procedure NEW_PAGE (FILE: in FILE_TYPE);
procedure NEW_PAGE;
procedure SKIP_PAGE (FILE: in FILE_TYPE);
procedure SKIP_PAGE;
function END_OF_PAGE (FILE: in FILE_TYPE)
    return BOOLEAN;
function END_OF_PAGE
    return BOOLEAN;
function END_OF_FILE (FILE: in FILE_TYPE)
    return BOOLEAN;
function END_OF_FILE
    return BOOLEAN;
procedure SET_COL (FILE: in FILE_TYPE;
                   TO:   in POSITIVE_COUNT);
procedure SET_COL (TO: in POSITIVE_COUNT);
procedure SET_LINE (FILE: in FILE_TYPE;
                    TO:   in POSITIVE_COUNT);
procedure SET_LINE (TO: in POSITIVE_COUNT);
function COL (FILE: in FILE_TYPE)
    return POSITIVE_COUNT;
function COL
    return POSITIVE_COUNT;
function LINE (FILE: in FILE_TYPE)
    return POSITIVE_COUNT;
function LINE
    return POSITIVE_COUNT;
function PAGE (FILE: in FILE_TYPE)
    return POSITIVE_COUNT;
function PAGE
    return POSITIVE_COUNT;
procedure GET (FILE: in     FILE_TYPE;
               ITEM:    out CHARACTER);
procedure GET (ITEM: out CHARACTER);
procedure PUT (FILE: in FILE_TYPE;
               ITEM: in CHARACTER);
procedure PUT (ITEM: in CHARACTER);
procedure GET (FILE: in     FILE_TYPE;
               ITEM:    out STRING);
procedure GET (ITEM: out STRING);
procedure PUT (FILE: in FILE_TYPE;
               ITEM: in STRING);
procedure PUT (ITEM: in STRING);
procedure GET_LINE (FILE: in     FILE_TYPE;
                    ITEM:    out STRING;
                    LAST:    out CAIS_NATURAL);
procedure GET_LINE (ITEM: out STRING;
                    LAST: out CAIS_NATURAL);
procedure PUT_LINE (FILE: in FILE_TYPE;
                    ITEM: in STRING);
procedure PUT_LINE (ITEM: in STRING);
```

Downloaded from http://www.everyspec.com


```
                    LAST:     out  CAIS_POSITIVE);

     procedure PUT  (TO:        out STRING;
                    ITEM: in       ENUM;
                    SET:  in    TYPE_SET := DEFAULT_SETTING);

   end ENUMERATION_IO;


private
   type FILE_TYPE is (IMPLEMENTATION_DEFINED);
   -- This type should be defined by the implementer.
end CAIS_TEXT_IO;



with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_IO_DEFINITIONS;
with CAIS_LIST_MANAGEMENT;
with CAIS_ACCESS_CONTROL_MANAGEMENT;
package CAIS_QUEUE_MANAGEMENT is

   use CAIS_STANDARD;
   use CAIS_DEFINITIONS;
   use CAIS_IO_DEFINITIONS;
   use CAIS_LIST_MANAGEMENT;
   use CAIS_ACCESS_CONTROL_MANAGEMENT;

   procedure CREATE_NONSYNCHRONOUS_COPY_QUEUE
        (QUEUE_NODE:             in out NODE_TYPE;
         FILE_NODE:              in     NODE_TYPE;
         QUEUE_BASE:             in     NODE_TYPE;
         QUEUE_KEY:              in     RELATIONSHIP_KEY := LATEST_KEY;
         QUEUE_RELATION:         in     RELATION_NAME := DEFAULT_RELATION;
         INTENT:                 in     INTENT_ARRAY := IN_INTENT;
         ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
         DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                     CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
         MAXIMUM_QUEUE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
   procedure CREATE_NONSYNCHRONOUS_COPY_QUEUE
        (QUEUE_NODE:             in out NODE_TYPE;
         FILE_NODE:              in     NODE_TYPE;
         QUEUE_NAME:             in     PATHNAME;
         INTENT:                 in     INTENT_ARRAY := IN_INTENT;
         ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
         DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                     CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
         MAXIMUM_QUEUE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
   procedure CREATE_NONSYNCHRONOUS_COPY_QUEUE
        (FILE_NODE:              in NODE_TYPE;
         QUEUE_BASE:             in NODE_TYPE;
         QUEUE_KEY:              in RELATIONSHIP_KEY := LATEST_KEY;
         QUEUE_RELATION:         in RELATION_NAME := DEFAULT_RELATION;
         INTENT:                 in INTENT_ARRAY := IN_INTENT;
```

Downloaded from http://www.everyspec.com


```
         ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
         DISCRETIONARY_ACCESS: in DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST;
         MAXIMUM_QUEUE_SIZE:     in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_NONSYNCHRONOUS_COPY_QUEUE
        (FILE_NODE:              in NODE_TYPE;
         QUEUE_NAME:             in PATHNAME;
         INTENT:                 in INTENT_ARRAY := IN_INTENT;
         ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
         DISCRETIONARY_ACCESS: in DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST;
         MAXIMUM_QUEUE_SIZE:     in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
        (QUEUE_NODE:             in out NODE_TYPE;
         FILE_NODE:              in     NODE_TYPE;
         QUEUE_BASE:             in     NODE_TYPE;
         QUEUE_KEY:              in     RELATIONSHIP_KEY := LATEST_KEY;
         QUEUE_RELATION:         in     RELATION_NAME := DEFAULT_RELATION;
         INTENT:                 in     INTENT_ARRAY := IN_INTENT;
         ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
         DISCRETIONARY_ACCESS: in     DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
         MAXIMUM_QUEUE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
        (QUEUE_NODE:             in out NODE_TYPE;
         FILE_NODE:              in     NODE_TYPE;
         QUEUE_NAME:             in     PATHNAME;
         INTENT:                 in     INTENT_ARRAY := IN_INTENT;
         ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
         DISCRETIONARY_ACCESS: in     DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
         MAXIMUM_QUEUE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
        (FILE_NODE:              in NODE_TYPE;
         QUEUE_BASE:             in NODE_TYPE;
         QUEUE_KEY:              in RELATIONSHIP_KEY := LATEST_KEY;
         QUEUE_RELATION:         in RELATION_NAME := DEFAULT_RELATION;
         INTENT:                 in INTENT_ARRAY := IN_INTENT;
         ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
         DISCRETIONARY_ACCESS: in DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST;
         MAXIMUM_QUEUE_SIZE:     in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_NONSYNCHRONOUS_MIMIC_QUEUE
        (FILE_NODE:              in NODE_TYPE;
         QUEUE_NAME:             in PATHNAME;
         INTENT:                 in INTENT_ARRAY := IN_INTENT;
         ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
         DISCRETIONARY_ACCESS: in DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
         MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST;
         MAXIMUM_QUEUE_SIZE:     in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
```

Downloaded from http://www.everyspec.com


```
generic
    type NUM is range <>;
package INTEGER_IO is

    DEFAULT_WIDTH: FIELD := NUM'WIDTH;
    DEFAULT_BASE: NUMBER_BASE := 10;

    procedure GET (FILE:   in      FILE_TYPE;
                   ITEM:      out  NUM;
                   WIDTH:  in      FIELD := 0);
    procedure GET (ITEM:       out NUM;
                   WIDTH:  in      FIELD := 0);
    procedure PUT (FILE:   in FILE_TYPE;
                   ITEM:   in NUM;
                   WIDTH:  in FIELD := 0;
                   BASE:   in NUMBER_BASE := DEFAULT_BASE);
    procedure PUT (ITEM:   in NUM;
                   WIDTH:  in FIELD := 0;
                   BASE:   in NUMBER_BASE := DEFAULT_BASE);
    procedure GET (FROM: in       STRING;
                   ITEM:    out   NUM;
                   LAST:    out   CAIS_POSITIVE);
    procedure PUT (TO:          out STRING;
                   ITEM: in       NUM;
                   BASE: in       NUMBER_BASE := DEFAULT_BASE);

end INTEGER_IO;


generic
    type NUM is digits <>;
package FLOAT_IO is

    DEFAULT_FORE: FIELD := 2;
    DEFAULT_AFT:  FIELD := NUM'DIGITS-1;
    DEFAULT_EXP:  FIELD := 3;

    procedure GET (FILE:   in      FILE_TYPE;
                   ITEM:      out  NUM;
                   WIDTH:  in      FIELD := 0);
    procedure GET (ITEM:       out NUM;
                   WIDTH:  in      FIELD := 0);
    procedure PUT (FILE: in FILE_TYPE;
                   ITEM: in NUM;
                   FORE: in FIELD := DEFAULT_FORE;
                   AFT:  in FIELD := DEFAULT_AFT;
                   EXP:  in FIELD := DEFAULT_EXP);
    procedure PUT (ITEM:  in NUM;
                   FORE:  in FIELD := DEFAULT_FORE;
                   AFT:   in FIELD := DEFAULT_AFT;
                   EXP:   in FIELD := DEFAULT_EXP);
    procedure GET (FROM: in       STRING;
                   ITEM:    out   NUM;
                   LAST:    out   CAIS_POSITIVE);
    procedure PUT (TO:          out STRING;
                   ITEM: in       NUM;
                   AFT:  in  FIELD := DEFAULT_AFT;
```

Downloaded from http://www.everyspec.com


```
                     EXP:  in  FIELD := DEFAULT_EXP);

end FLOAT_IO;


generic
     type NUM is delta <>;
package FIXED_IO is

     DEFAULT_FORE: FIELD := NUM'FORE;
     DEFAULT_AFT:  FIELD := NUM'AFT;
     DEFAULT_EXP:  FIELD := 0;

     procedure GET (FILE:  in     FILE_TYPE;
                    ITEM:     out NUM;
                    WIDTH: in     FIELD := 0);
     procedure GET (ITEM:     out NUM;
                    WIDTH: in     FIELD := 0);
     procedure PUT (FILE: in FILE_TYPE;
                    ITEM: in NUM;
                    FORE: in FIELD := DEFAULT_FORE;
                    AFT:  in FIELD := DEFAULT_AFT;
                    EXP:  in FIELD := DEFAULT_EXP);
     procedure PUT (ITEM:  in NUM;
                    FORE:  in FIELD := DEFAULT_FORE;
                    AFT:   in FIELD := DEFAULT_AFT;
                    EXP:   in FIELD := DEFAULT_EXP);
     procedure GET (FROM: in     STRING;
                    ITEM:    out NUM;
                    LAST:    out CAIS_POSITIVE);
     procedure PUT (TO:      out STRING;
                    ITEM: in     NUM;
                    AFT:  in  FIELD := DEFAULT_AFT;
                    EXP:  in  FIELD := DEFAULT_EXP);

end FIXED_IO;


generic
     type ENUM is (<>);
package ENUMERATION_IO is

     DEFAULT_WIDTH: FIELD := 0;
     DEFAULT_SETTING: TYPE_SET := UPPER_CASE;

     procedure GET (FILE: in     FILE_TYPE;
                    ITEM:    out ENUM);
     procedure GET (ITEM: out ENUM);
     procedure PUT (FILE:  in FILE_TYPE;
                    ITEM:  in ENUM;
                    WIDTH: in FIELD := DEFAULT_WIDTH;
                    SET:   in TYPE_SET := DEFAULT_SETTING);
     procedure PUT (ITEM:  in ENUM;
                    WIDTH: in FIELD := DEFAULT_WIDTH;
                    SET:   in TYPE_SET := DEFAULT_SETTING);
     procedure GET (FROM: in     STRING;
                    ITEM:    out ENUM;
```

Downloaded from http://www.everyspec.com


```
    (QUEUE_NODE:             in out NODE_TYPE;
     QUEUE_BASE:             in     NODE_TYPE;
     QUEUE_KEY:              in     RELATIONSHIP_KEY := LATEST_KEY;
     QUEUE_RELATION:         in     RELATION_NAME := DEFAULT_RELATION;
     INTENT:                 in     INTENT_ARRAY := IN_INTENT;
     ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
     DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
     MAXIMUM_QUEUE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
    (QUEUE_NODE:             in out NODE_TYPE;
     QUEUE_NAME:             in     PATHNAME;
     INTENT:                 in     INTENT_ARRAY := IN_INTENT;
     ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
     DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
     MAXIMUM_QUEUE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
    (QUEUE_BASE:             in NODE_TYPE;
     QUEUE_KEY:              in RELATIONSHIP_KEY := LATEST_KEY;
     QUEUE_RELATION:         in RELATION_NAME := DEFAULT_RELATION;
     INTENT:                 in INTENT_ARRAY := IN_INTENT;
     ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
     DISCRETIONARY_ACCESS:   in DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST;
     MAXIMUM_QUEUE_SIZE:     in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE
    (QUEUE_NAME:             in PATHNAME;
     INTENT:                 in INTENT_ARRAY := IN_INTENT;
     ATTRIBUTES:             in ATTRIBUTE_LIST := EMPTY_LIST;
     DISCRETIONARY_ACCESS:   in DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:       in MANDATORY_ACCESS_LIST := EMPTY_LIST;
     MAXIMUM_QUEUE_SIZE:     in CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
generic
    type ELEMENT_TYPE is private;
procedure CREATE_NONSYNCHRONOUS_SOLO_SEQUENTIAL_QUEUE
    (QUEUE_NODE:             in out NODE_TYPE;
     QUEUE_BASE:             in     NODE_TYPE;
     QUEUE_KEY:              in     RELATIONSHIP_KEY := LATEST_KEY;
     QUEUE_RELATION:         in     RELATION_NAME := DEFAULT_RELATION;
     INTENT:                 in     INTENT_ARRAY := IN_INTENT;
     ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
     DISCRETIONARY_ACCESS:   in     DISCRETIONARY_ACCESS_LIST :=
                                CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
     MANDATORY_ACCESS:       in     MANDATORY_ACCESS_LIST := EMPTY_LIST;
     MAXIMUM_QUEUE_SIZE:     in     CAIS_NATURAL := UNBOUNDED_QUEUE_SIZE);
procedure CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
    (QUEUE_NODE:             in out NODE_TYPE;
     QUEUE_BASE:             in     NODE_TYPE;
     QUEUE_KEY:              in     RELATIONSHIP_KEY := LATEST_KEY;
     QUEUE_RELATION:         in     RELATION_NAME := DEFAULT_RELATION;
     INTENT:                 in     INTENT_ARRAY := IN_INTENT;
     ATTRIBUTES:             in     ATTRIBUTE_LIST := EMPTY_LIST;
```

Downloaded from http://www.everyspec.com


```
          DISCRETIONARY_ACCESS: in     DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
          MANDATORY_ACCESS:     in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
    procedure CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
         (QUEUE_NODE:           in out NODE_TYPE;
          QUEUE_NAME:           in     PATHNAME;
          INTENT:               in     INTENT_ARRAY := IN_INTENT;
          ATTRIBUTES:           in     ATTRIBUTE_LIST := EMPTY_LIST;
          DISCRETIONARY_ACCESS: in     DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
          MANDATORY_ACCESS:     in     MANDATORY_ACCESS_LIST := EMPTY_LIST);
    procedure CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
         (QUEUE_BASE:           in NODE_TYPE;
          QUEUE_KEY:            in RELATIONSHIP_KEY := LATEST_KEY;
          QUEUE_RELATION:       in RELATION_NAME := DEFAULT_RELATION;
          INTENT:               in INTENT_ARRAY := IN_INTENT;
          ATTRIBUTES:           in ATTRIBUTE_LIST := EMPTY_LIST;
          DISCRETIONARY_ACCESS: in DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
          MANDATORY_ACCESS:     in MANDATORY_ACCESS_LIST := EMPTY_LIST);
    procedure CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE
         (QUEUE_NAME:           in PATHNAME;
          INTENT:               in INTENT_ARRAY := IN_INTENT;
          ATTRIBUTES:           in ATTRIBUTE_LIST := EMPTY_LIST;
          DISCRETIONARY_ACCESS: in DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
          MANDATORY_ACCESS:     in MANDATORY_ACCESS_LIST := EMPTY_LIST);
    generic
         type ELEMENT_TYPE is private;
    procedure CREATE_SYNCHRONOUS_SOLO_SEQUENTIAL_QUEUE
         (QUEUE_NODE:           in out NODE_TYPE;
          QUEUE_BASE:           in     NODE_TYPE;
          QUEUE_KEY:            in     RELATIONSHIP_KEY := LATEST_KEY;
          QUEUE_RELATION:       in     RELATION_NAME := DEFAULT_RELATION;
          INTENT:               in     INTENT_ARRAY := IN_INTENT;
          ATTRIBUTES:           in     ATTRIBUTE_LIST := EMPTY_LIST;
          DISCRETIONARY_ACCESS: in     DISCRETIONARY_ACCESS_LIST :=
                                    CAIS_ACCESS_CONTROL_MANAGEMENT.ALL_RIGHTS;
          MANDATORY_ACCESS:     in     MANDATORY_ACCESS_LIST := EMPTY_LIST);

end CAIS_QUEUE_MANAGEMENT;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_IO_DEFINITIONS;
package CAIS_SCROLL_TERMINAL_IO is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;
    use CAIS_IO_DEFINITIONS;

    type FILE_TYPE is limited private;

    type FILE_MODE is (IN_FILE, OUT_FILE, INOUT_FILE);
```

Downloaded from http://www.everyspec.com


```
type CHARACTER_ARRAY is array (CHARACTER) of BOOLEAN;

type FUNCTION_KEY_DESCRIPTOR is limited private;

subtype FUNCTION_KEY_NAME is STRING;

type TERMINAL_POSITION_TYPE is
    record
        ROW:    CAIS_POSITIVE;
        COLUMN: CAIS_POSITIVE;
    end record;

type TAB_STOP_KIND is (HORIZONTAL, VERTICAL);

procedure OPEN (TERMINAL: in out FILE_TYPE;
                NODE:     in     NODE_TYPE;
                MODE:     in     FILE_MODE);
procedure CLOSE (TERMINAL: in out FILE_TYPE);
function IS_OPEN (TERMINAL: in FILE_TYPE)
    return BOOLEAN;
function NUMBER_OF_FUNCTION_KEYS (TERMINAL: in FILE_TYPE)
    return CAIS_NATURAL;
function INTERCEPTED_INPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
function INTERCEPTED_OUTPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
procedure ENABLE_FUNCTION_KEYS (TERMINAL: in FILE_TYPE;
                                ENABLE:   in BOOLEAN);
function FUNCTION_KEYS_ARE_ENABLED (TERMINAL: in FILE_TYPE)
    return BOOLEAN;
procedure SET_ACTIVE_POSITION (TERMINAL: in FILE_TYPE;
                               POSITION: in TERMINAL_POSITION_TYPE);
function ACTIVE_POSITION (TERMINAL: in FILE_TYPE)
    return TERMINAL_POSITION_TYPE;
function PAGE_SIZE (TERMINAL: in FILE_TYPE)
    return TERMINAL_POSITION_TYPE;
procedure SET_TAB_STOP (TERMINAL: in FILE_TYPE;
                        KIND:     in TAB_STOP_KIND := HORIZONTAL);
procedure CLEAR_TAB_STOP (TERMINAL: in FILE_TYPE;
                          KIND:     in TAB_STOP_KIND := HORIZONTAL);
procedure CLEAR_ALL_TAB_STOPS (TERMINAL: in FILE_TYPE;
                               KIND:     in TAB_STOP_KIND := HORIZONTAL);
procedure TAB (TERMINAL: in FILE_TYPE;
               COUNT:    in CAIS_POSITIVE := 1;
               KIND:     in TAB_STOP_KIND := HORIZONTAL);
procedure SOUND_BELL (TERMINAL: in FILE_TYPE);
procedure PUT (TERMINAL: in FILE_TYPE;
               ITEM:     in CHARACTER);
procedure PUT (TERMINAL: in FILE_TYPE;
               ITEM:     in STRING);
procedure GET (TERMINAL: in     FILE_TYPE;
               ITEM:        out CHARACTER;
               KEYS:     in out FUNCTION_KEY_DESCRIPTOR);
procedure GET (TERMINAL: in     FILE_TYPE;
               ITEM:        out STRING;
               LAST:        out CAIS_NATURAL;
               KEYS:     in out FUNCTION_KEY_DESCRIPTOR);
```

Downloaded from http://www.everyspec.com


```
    procedure CREATE_FUNCTION_KEY_DESCRIPTOR
                  (KEYS:           in out FUNCTION_KEY_DESCRIPTOR;
                   MAXIMUM_COUNT: in    CAIS_POSITIVE);
    procedure DELETE_FUNCTION_KEY_DESCRIPTOR
                  (KEYS: in out FUNCTION_KEY_DESCRIPTOR);
    function FUNCTION_KEY_COUNT (KEYS: in FUNCTION_KEY_DESCRIPTOR)
        return CAIS_NATURAL;
    procedure GET_FUNCTION_KEY
                 (KEYS:            in      FUNCTION_KEY_DESCRIPTOR;
                  INDEX:           in      CAIS_POSITIVE;
                  KEY_IDENTIFIER:     out  CAIS_POSITIVE;
                  POSITION:           out  CAIS_NATURAL);
    function FUNCTION_KEY_IDENTIFICATION
                        (TERMINAL:        in FILE_TYPE;
                         KEY_IDENTIFIER:  in CAIS_POSITIVE)
        return FUNCTION_KEY_NAME;
    function MODE (TERMINAL: in FILE_TYPE)
        return FILE_MODE;
    procedure BACKSPACE (TERMINAL: in FILE_TYPE;
                         COUNT:    in CAIS_POSITIVE := 1);
    procedure NEW_LINE (TERMINAL: in FILE_TYPE;
                        COUNT:    in CAIS_POSITIVE := 1);
    procedure NEW_PAGE (TERMINAL: in FILE_TYPE);
    procedure RESET (TERMINAL: in out FILE_TYPE;
                     MODE:     in     FILE_MODE);
    procedure SYNCHRONIZE (TERMINAL: in FILE_TYPE);
    procedure ENABLE_SYNCHRONIZATION (TERMINAL: in FILE_TYPE;
                                      ENABLE:   in BOOLEAN);
    function SYNCHRONIZATION_IS_ENABLED (TERMINAL: in FILE_TYPE)
        return BOOLEAN;

private
    type FILE_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
    type FUNCTION_KEY_DESCRIPTOR is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_SCROLL_TERMINAL_IO;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_IO_DEFINITIONS;
package CAIS_PAGE_TERMINAL_IO is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;
    use CAIS_IO_DEFINITIONS;

    type FILE_TYPE is limited private;

    type FILE_MODE is (IN_FILE, OUT_FILE, INOUT_FILE);

    type CHARACTER_ARRAY is array (CHARACTER) of BOOLEAN;

    type FUNCTION_KEY_DESCRIPTOR is limited private;
```

Downloaded from http://www.everyspec.com


```
subtype FUNCTION_KEY_NAME is STRING;

type TERMINAL_POSITION_TYPE is
     record
          ROW:      CAIS_POSITIVE;
          COLUMN:   CAIS_POSITIVE;
     end record;

type TAB_STOP_KIND is (HORIZONTAL, VERTICAL);

type SELECT_RANGE_KIND is
     (FROM_ACTIVE_POSITION_TO_END,
      FROM_START_TO_ACTIVE_POSITION,
      ALL_POSITIONS);

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

type GRAPHIC_RENDITION_ARRAY is array (GRAPHIC_RENDITION_KIND)
     of BOOLEAN;

DEFAULT_GRAPHIC_RENDITION: constant GRAPHIC_RENDITION_ARRAY :=
          (PRIMARY_RENDITION => TRUE, BOLD..REVERSE_IMAGE => FALSE);

procedure OPEN (TERMINAL: in out FILE_TYPE;
                NODE:     in     NODE_TYPE;
                MODE:     in     FILE_MODE);
procedure CLOSE (TERMINAL: in out FILE_TYPE);
function IS_OPEN (TERMINAL: in FILE_TYPE)
     return BOOLEAN;
function NUMBER_OF_FUNCTION_KEYS (TERMINAL: in FILE_TYPE)
     return CAIS_NATURAL;
function INTERCEPTED_INPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
     return CHARACTER_ARRAY;
function INTERCEPTED_OUTPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
     return CHARACTER_ARRAY;
procedure ENABLE_FUNCTION_KEYS (TERMINAL: in FILE_TYPE;
                                ENABLE:   in BOOLEAN);
function FUNCTION_KEYS_ARE_ENABLED (TERMINAL: in FILE_TYPE)
     return BOOLEAN;
procedure SET_ACTIVE_POSITION (TERMINAL: in FILE_TYPE;
                               POSITION: in TERMINAL_POSITION_TYPE);
function ACTIVE_POSITION (TERMINAL: in FILE_TYPE)
     return TERMINAL_POSITION_TYPE;
function PAGE_SIZE (TERMINAL: in FILE_TYPE)
     return TERMINAL_POSITION_TYPE;
procedure SET_TAB_STOP (TERMINAL: in FILE_TYPE;
                        KIND:     in TAB_STOP_KIND := HORIZONTAL);
procedure CLEAR_TAB_STOP (TERMINAL: in FILE_TYPE;
                          KIND:     in TAB_STOP_KIND := HORIZONTAL);
procedure CLEAR_ALL_TAB_STOPS (TERMINAL: in FILE_TYPE;
```

Downloaded from http://www.everyspec.com


```
                                      KIND:       in TAB_STOP_KIND := HORIZONTAL);
procedure TAB (TERMINAL: in FILE_TYPE;
               COUNT:    in CAIS_POSITIVE  := 1;
               KIND:     in TAB_STOP_KIND  := HORIZONTAL);
procedure SOUND_BELL (TERMINAL: in FILE_TYPE);
procedure PUT (TERMINAL: in FILE_TYPE;
               ITEM:     in CHARACTER);
procedure PUT (TERMINAL: in FILE_TYPE;
               ITEM:     in STRING);
procedure GET (TERMINAL: in     FILE_TYPE;
               ITEM:        out CHARACTER;
               KEYS:     in out FUNCTION_KEY_DESCRIPTOR);
procedure GET (TERMINAL: in     FILE_TYPE;
               ITEM:        out STRING;
               LAST:        out CAIS_NATURAL;
               KEYS:     in out FUNCTION_KEY_DESCRIPTOR);
procedure CREATE_FUNCTION_KEY_DESCRIPTOR
                (KEYS:          in out FUNCTION_KEY_DESCRIPTOR;
                 MAXIMUM_COUNT: in     CAIS_POSITIVE);
procedure DELETE_FUNCTION_KEY_DESCRIPTOR
                (KEYS: in out FUNCTION_KEY_DESCRIPTOR);
function FUNCTION_KEY_COUNT (KEYS: in FUNCTION_KEY_DESCRIPTOR)
    return CAIS_NATURAL;
procedure GET_FUNCTION_KEY
            (KEYS:            in     FUNCTION_KEY_DESCRIPTOR;
             INDEX:           in     CAIS_POSITIVE;
             KEY_IDENTIFIER:     out CAIS_POSITIVE;
             POSITION:           out CAIS_NATURAL);
function FUNCTION_KEY_IDENTIFICATION
                        (TERMINAL:       in FILE_TYPE;
                         KEY_IDENTIFIER: in CAIS_POSITIVE)
    return FUNCTION_KEY_NAME;
function MODE (TERMINAL: in FILE_TYPE)
    return FILE_MODE;
procedure DELETE_CHARACTER (TERMINAL: in FILE_TYPE;
                            COUNT:    in CAIS_POSITIVE  := 1);
procedure DELETE_LINE (TERMINAL: in FILE_TYPE;
                       COUNT:    in CAIS_POSITIVE  := 1);
procedure ERASE_CHARACTER (TERMINAL: in FILE_TYPE;
                           COUNT:    in CAIS_POSITIVE  := 1);
procedure ERASE_IN_DISPLAY (TERMINAL:  in FILE_TYPE;
                            SELECTION: in SELECT_RANGE_KIND);
procedure ERASE_IN_LINE (TERMINAL:  in FILE_TYPE;
                         SELECTION: in SELECT_RANGE_KIND);
procedure INSERT_SPACE (TERMINAL: in FILE_TYPE;
                        COUNT:    in CAIS_POSITIVE  := 1);
procedure INSERT_LINE (TERMINAL: in FILE_TYPE;
                       COUNT:    in CAIS_POSITIVE  := 1);
function GRAPHIC_RENDITION_IS_SUPPORTED
                         (TERMINAL:   in FILE_TYPE;
                          RENDITION:  in GRAPHIC_RENDITION_ARRAY)
    return BOOLEAN;
procedure SELECT_GRAPHIC_RENDITION
                   (TERMINAL:  in FILE_TYPE;
                    RENDITION: in GRAPHIC_RENDITION_ARRAY :=
                                    DEFAULT_GRAPHIC_RENDITION);
```

Downloaded from http://www.everyspec.com


```
    function END_POSITION_SUPPORT (TERMINAL: in FILE_TYPE)
        return BOOLEAN;
    procedure RESET (TERMINAL: in out FILE_TYPE;
                     MODE:     in     FILE_MODE);
    procedure SYNCHRONIZE (TERMINAL: in FILE_TYPE);
    procedure ENABLE_SYNCHRONIZATION (TERMINAL: in FILE_TYPE;
                                      ENABLE:   in BOOLEAN);
    function SYNCHRONIZATION_IS_ENABLED (TERMINAL: in FILE_TYPE)
        return BOOLEAN;

private
    type FILE_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
    type FUNCTION_KEY_DESCRIPTOR is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_PAGE_TERMINAL_IO;



with CAIS_STANDARD;
with CAIS_DEFINITIONS;
with CAIS_IO_DEFINITIONS;
package CAIS_FORM_TERMINAL_IO is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;
    use CAIS_IO_DEFINITIONS;

    type FILE_TYPE is limited private;

    type FORM_TYPE is limited private;

    type CHARACTER_ARRAY is array (CHARACTER) of BOOLEAN;

    subtype FUNCTION_KEY_NAME is STRING;

    type TERMINAL_POSITION_TYPE is
        record
            ROW:    CAIS_POSITIVE;
            COLUMN: CAIS_POSITIVE;
        end record;

    type AREA_INTENSITY_KIND is
        (NONE,
         NORMAL,
         HIGH);

    type AREA_PROTECTION_KIND is
        (UNPROTECTED,
         PROTECTED);

    type AREA_INPUT_KIND is
        (GRAPHIC_CHARACTERS,
         NUMERICS,
         ALPHABETICS);

    type AREA_VALUE_KIND is
```

Downloaded from http://www.everyspec.com


```
         (NO_FILL,
          FILL_WITH_ZEROES,
          FILL_WITH_SPACES);

subtype PRINTABLE_CHARACTER is CHARACTER range ' ' .. ASCII.TILDE;

procedure OPEN (TERMINAL: in out FILE_TYPE;
                NODE:     in     NODE_TYPE);
procedure CLOSE (TERMINAL: in out FILE_TYPE);
function IS_OPEN (TERMINAL: in FILE_TYPE)
    return BOOLEAN;
function NUMBER_OF_FUNCTION_KEYS (TERMINAL: in FILE_TYPE)
    return CAIS_NATURAL;
function INTERCEPTED_INPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
function INTERCEPTED_OUTPUT_CHARACTERS (TERMINAL: in FILE_TYPE)
    return CHARACTER_ARRAY;
procedure CREATE_FORM
                (FORM:                             in out FORM_TYPE;
                 ROWS:                             in     CAIS_POSITIVE;
                 COLUMNS:                          in     CAIS_POSITIVE;
                 AREA_QUALIFIER_REQUIRES_SPACE:    in     BOOLEAN);
procedure DELETE_FORM (FORM: in out FORM_TYPE);
procedure COPY_FORM (FROM: in     FORM_TYPE;
                     TO:   in out FORM_TYPE);
procedure DEFINE_QUALIFIED_AREA
               (FORM:       in out FORM_TYPE;
                INTENSITY:  in     AREA_INTENSITY_KIND := NORMAL;
                PROTECTION: in     AREA_PROTECTION_KIND := PROTECTED;
                INPUT:      in     AREA_INPUT_KIND := GRAPHIC_CHARACTERS;
                VALUE:      in     AREA_VALUE_KIND := NO_FILL);
procedure REMOVE_AREA_QUALIFIER (FORM: in out FORM_TYPE);
procedure SET_ACTIVE_POSITION (FORM:     in out FORM_TYPE;
                               POSITION: in     TERMINAL_POSITION_TYPE);
function ACTIVE_POSITION (FORM: in FORM_TYPE)
    return TERMINAL_POSITION_TYPE;
procedure ADVANCE_TO_QUALIFIED_AREA (FORM:  in out FORM_TYPE;
                                     COUNT: in     CAIS_POSITIVE := 1);
procedure PUT (FORM: in out FORM_TYPE;
               ITEM: in     PRINTABLE_CHARACTER);
procedure PUT (FORM: in out FORM_TYPE;
               ITEM: in     STRING);
procedure ERASE_AREA (FORM: in out FORM_TYPE);
procedure ERASE_FORM (FORM: in out FORM_TYPE);
procedure ACTIVATE (TERMINAL: in     FILE_TYPE;
                    FORM:     in out FORM_TYPE);
procedure GET (FORM: in out FORM_TYPE;
               ITEM:    out PRINTABLE_CHARACTER);
procedure GET (FORM: in out FORM_TYPE;
               ITEM:    out STRING);
function IS_FORM_UPDATED (FORM: in FORM_TYPE)
    return BOOLEAN;
function FUNCTION_KEY_IDENTIFICATION
                      (TERMINAL:       in FILE_TYPE;
                       KEY_IDENTIFIER: in CAIS_POSITIVE)
    return FUNCTION_KEY_NAME;
function TERMINATION_KEY (FORM: in FORM_TYPE)
```

Downloaded from http://www.everyspec.com


```
        return CAIS_NATURAL;

    function FORM_SIZE (FORM: in FORM_TYPE)
        return TERMINAL_POSITION_TYPE;
    function TERMINAL_SIZE (TERMINAL: in FILE_TYPE)
        return TERMINAL_POSITION_TYPE;
    function AREA_QUALIFIER_REQUIRES_SPACE (FORM: in FORM_TYPE)
        return BOOLEAN;
    function AREA_QUALIFIER_REQUIRES_SPACE (TERMINAL: in FILE_TYPE)
        return BOOLEAN;

private
    type FILE_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
    type FORM_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_FORM_TERMINAL_IO;


with CAIS_STANDARD;
with CAIS_DEFINITIONS;
package CAIS_MAGNETIC_TAPE_IO is

    use CAIS_STANDARD;
    use CAIS_DEFINITIONS;

    type FILE_TYPE is limited private;

    type FILE_MODE is (IN_FILE, OUT_FILE);

    subtype TAPE_NAME is STRING;

    subtype TAPE_BLOCK is STRING;

    type TAPE_DRIVE_STATUS_KIND is
          (OPENED,
           MOUNT_REQUESTED,
           MOUNTED,
           LOADED,
           CLOSED);

    type TAPE_POSITION_KIND is
          (BEGINNING_OF_VOLUME,
           END_OF_VOLUME,
           END_OF_TAPE,
           AFTER_TAPE_MARK,
           OTHER);

    type TAPE_RECORDING_METHOD_KIND is
          (NON_RETURN_TO_ZERO_INVERTED,
           PHASE_ENCODED,
           GROUP_CODED_RECORDING);

    TAPE_STATUS_ERROR: exception;

    procedure OPEN (TAPE_DRIVE: in out FILE_TYPE;
```

Downloaded from http://www.everyspec.com


```
                    NODE:          in      NODE_TYPE;
                    MODE:          in      FILE_MODE);
    procedure CLOSE (TAPE_DRIVE: in out FILE_TYPE);
    function IS_OPEN (TAPE_DRIVE: in FILE_TYPE)
        return BOOLEAN;
    function MODE (TAPE_DRIVE: in FILE_TYPE)
        return FILE_MODE;
    procedure REQUEST_MOUNT (TAPE_DRIVE:            in FILE_TYPE;
                             NAME:                  in TAPE_NAME;
                             RECORDING_METHOD:      in TAPE_RECORDING_METHOD_KIND;
                             INSTALL_WRITE_RING:    in BOOLEAN := FALSE);
    procedure LOAD (TAPE_DRIVE: in FILE_TYPE;
                    BLOCK_SIZE: in CAIS_POSITIVE);
    procedure UNLOAD (TAPE_DRIVE: in FILE_TYPE);
    procedure REQUEST_DISMOUNT (TAPE_DRIVE: in FILE_TYPE);
    function POSITION (TAPE_DRIVE: in FILE_TYPE)
        return TAPE_POSITION_KIND;
    procedure REWIND_TAPE (TAPE_DRIVE: in FILE_TYPE);
    procedure SKIP_TAPE_MARK (TAPE_DRIVE: in FILE_TYPE;
                              COUNT:      in CAIS_POSITIVE := 1);
    procedure WRITE_TAPE_MARK (TAPE_DRIVE: in FILE_TYPE);
    function STATUS (TAPE_DRIVE: in FILE_TYPE)
        return TAPE_DRIVE_STATUS_KIND;
    function RECORDING_METHOD (TAPE_DRIVE: in FILE_TYPE)
        return TAPE_RECORDING_METHOD_KIND;
    function IS_WRITE_RING_INSTALLED (TAPE_DRIVE: in FILE_TYPE)
        return BOOLEAN;
    procedure SKIP_BLOCK (TAPE_DRIVE: in FILE_TYPE;
                          COUNT:      in CAIS_POSITIVE := 1);
    procedure READ_BLOCK (TAPE_DRIVE:      in     FILE_TYPE;
                          BLOCK:              out TAPE_BLOCK;
                          LAST:               out CAIS_NATURAL;
                          BLOCK_OVERFLOW:     out CAIS_NATURAL);
    procedure WRITE_BLOCK (TAPE_DRIVE: in FILE_TYPE;
                           BLOCK:      in TAPE_BLOCK);
    procedure RESET (TAPE_DRIVE: in out FILE_TYPE;
                     MODE:       in     FILE_MODE);

private
    type FILE_TYPE is (IMPLEMENTATION_DEFINED);
    -- This type should be defined by the implementer.
end CAIS_MAGNETIC_TAPE_IO;


with CAIS_DEFINITIONS;
with CAIS_LIST_MANAGEMENT;
package CAIS_IMPORT_EXPORT is

    use CAIS_DEFINITIONS;
    use CAIS_LIST_MANAGEMENT;

    procedure IMPORT_CONTENTS (FROM:             in STRING;
                               TO:               in NODE_TYPE;
                               CHARACTERISTICS:  in LIST_TYPE := EMPTY_LIST);
    procedure IMPORT_CONTENTS (FROM:             in STRING;
                               TO:               in PATHNAME;
```

Downloaded from http://www.everyspec.com


```
                                      CHARACTERISTICS: in LIST_TYPE := EMPTY_LIST);
    procedure EXPORT_CONTENTS (FROM:                 in NODE_TYPE;
                               TO:                   in STRING;
                               CHARACTERISTICS: in LIST_TYPE := EMPTY_LIST);
    procedure EXPORT_CONTENTS (FROM:                 in PATHNAME;
                               TO:                   in STRING;
                               CHARACTERISTICS: in LIST_TYPE := EMPTY_LIST);

end CAIS_IMPORT_EXPORT;
```

Downloaded from http://www.everyspec.com

Appendix C
Cross Reference of CAIS Procedures and Functions

The material contained in this appendix is not a mandatory part of the standard.

This appendix lists the CAIS procedures and functions in order to allow the reader ready access to a description of a particular capability in the CAIS.

List Of Tables

Table Name	Page
Access Control	580
Attribute Iterators	580
File Handles	583
Files	575
Float Items	593
Form Terminal	588
Identifier Items	592
Import Export	591
Input Output	582
Integer Items	593
List Items	592
List Management	591
Magnetic Tape	589
Node Attributes	578
Node Handles	573
Node Iterators	579
Nodes	573
Page Terminal	586
Pathnames	576
Process Control	581
Process Information	581
Queues	590
Relationship Attributes	577
Relationships	576
Scroll Terminal	585

Downloaded from http://www.everyspec.com

List Of Tables -- Continued.

Table Name	Page
String Items	593
Time and Calendar	594

Downloaded from http://www.everyspec.com

Node Handles		
Description of Operations	Applicable Interfaces	Page
Opening a handle to a named node	OPEN	63
Closing a handle	CLOSE	66
Changing intent	CHANGE_INTENT	67
Query for open-ness	IS_OPEN	69
Query for intent	INTENT	70
Obtaining an index for a node handle	INDEX	84
Opening a handle to parent	OPEN_PARENT	85
Getting next open node handle on an iterator	GET_NEXT	111
Getting a handle to current node	GET_CURRENT_NODE	117
Query for number of open node handles of a process	OPEN_NODE_HANDLE_COUNT	202
Query for number of open file handles associated with a file node handle	OPEN_FILE_HANDLE_COUNT	72

Nodes		
Description of Operations	Applicable Interfaces	Page
Query for node kind	KIND_OF_NODE	71
Query for obtainability or accessiblity	IS_OBTAINABLE	81
Query for sameness	IS_SAME	82
Duplicating a node	COPY_NODE	87
Duplicating a tree of nodes	COPY_TREE	89
Renaming a node	RENAME	92
Deleting a node	DELETE_NODE	94
Deleting a tree of nodes	DELETE_TREE	96
Deleting a job	DELETE_JOB	188
Get next open node handle on an iterator	GET_NEXT	111
Get path element to next node on an iterator	NEXT_NAME	114

Downloaded from http://www.everyspec.com

Nodes -- Continued.		
Description of Operations	Applicable Interfaces	Page
Skip next node on an iterator	SKIP_NEXT	113
Getting creation time for a node	TIME_CREATED	119
Getting the last time that any relationship on a node was modified	TIME_RELATIONSHIP_WRITTEN	120
Getting the last time that any attribute on a node was modified	TIME_ATTRIBUTE_WRITTEN	122
Getting the last time that node contents were written	TIME_CONTENTS_WRITTEN	121
Setting access control information for a node	SET_GRANTED_RIGHTS	154
Deleting access control information for a node	DELETE_GRANTED_RIGHTS	156
Getting the access method for a node	ACCESS_METHOD	218
Query for kind of file node	KIND_OF_NODE	71
Query for kind of queue node	KIND_OF_QUEUE	220
Query for kind of device file node	KIND_OF_DEVICE	221
Setting CURRENT_NODE	SET_CURRENT_NODE	116
Creating a structural node	CREATE_NODE	164
Creating a process node	SPAWN_PROCESS	174
Creating a process node	INVOKE_PROCESS	180
Creating a root process node	CREATE_JOB	185
Creating a direct file and its node	CREATE	228
Creating a sequential file and its node	CREATE	237
Creating a text file and its node	CREATE	246
Creating a nonsynchronous copy queue node	CREATE_NONSYNCHRONOUS_ COPY_QUEUE	257
Creating a nonsynchronous mimic queue node	CREATE_NONSYNCHRONOUS_ MIMIC_QUEUE	262
Creating a nonsynchronous solo text queue node	CREATE_NONSYNCHRONOUS_ SOLO_TEXT_QUEUE	267

Downloaded from http://www.everyspec.com

Nodes -- Continued.		
Description of Operations	Applicable Interfaces	Page
Creating a nonsynchronous solo sequential queue node	CREATE_NONSYNCHRONOUS_ SOLO_SEQUENTIAL_QUEUE	271
Creating a synchronous solo text queue node	CREATE_SYNCHRONOUS_ SOLO_TEXT_QUEUE	274
Creating a synchronous solo sequential queue node	CREATE_SYNCHRONOUS_ SOLO_SEQUENTIAL_QUEUE	278

Files		
Description of Operations	Applicable Interfaces	Page
Read and write operations	Same as specified in [1815A]	
Input and output operations	See subsections of 5.3	
Querying the kind of file	KIND_OF_FILE	219
Determining the current size of a file	CURRENT_FILE_SIZE	222
Determining the maximum size of a file	MAXIMUM_FILE_SIZE	223
Determining the last time that the contents of a file were updated	TIME_CONTENTS_WRITTEN	121
Getting the access method for a file	ACCESS_METHOD	218
Creating a direct file	CREATE	228
Opening a direct file	OPEN	231
Closing a direct file	CLOSE	232
Resetting a direct file	RESET	233
Synchronizing direct files	SYNCHRONIZE	234
Creating a sequential file	CREATE	237
Opening a sequential file	OPEN	240
Closing a sequential file	CLOSE	241
Resetting a sequential file	RESET	242
Synchronizing sequential files	SYNCHRONIZE	243
Creating a text file	CREATE	246
Opening a text file	OPEN	249
Closing a text file	CLOSE	250

Downloaded from http://www.everyspec.com

Files -- Continued.		
Description of Operations	Applicable Interfaces	Page
Resetting a text file	RESET	251
Synchronizing text files	SYNCHRONIZE	252
Importing the contents of a file	IMPORT_CONTENTS	415
Exporting the contents of a file	EXPORT_CONTENTS	417

Pathnames		
Description of Operations	Applicable Interfaces	Page
Unique primary pathname	PRIMARY_NAME	73
Last key of unique primary pathname	PRIMARY_KEY	74
Last relation name of unique primary pathname	PRIMARY_RELATION	75
Relationship key of last pathname element for handle	PATH_KEY	76
Relation name of last pathname element for handle	PATH_RELATION	77
Base pathname	BASE_PATH	78
Relation name of last pathname element	LAST_RELATION	79
Relationship key of last pathname element	LAST_KEY	80
Get path element to next node on an iterator	NEXT_NAME	114

Relationships		
Description of Operations	Applicable Interfaces	Page
Renaming a primary relationship	RENAME	92
Deleting a primary relationship	DELETE_NODE	94
Deleting a tree of primary relationships	DELETE_TREE	96
Deleting a job	DELETE_JOB	188
Creating a secondary relationship	CREATE_SECONDARY_RELATIONSHIP	98

Downloaded from http://www.everyspec.com

Relationships -- Continued.		
Description of Operations	Applicable Interfaces	Page
Deleting a secondary relationship	DELETE_SECONDARY_RELATIONSHIP	100
Setting inheritance property of a relationship	SET_INHERITANCE	102
Query for inheritance property of a relationship	IS_INHERITABLE	104
Setting the CURRENT_NODE	SET_CURRENT_NODE	116
Query last time a relationship was modified	TIME_RELATIONSHIP_WRITTEN	120
Deleting access relationship	DELETE_GRANTED_RIGHTS	156

Predefined Relations		
Description of Operations	Applicable Interfaces	Page
Getting the CURRENT_NODE	GET_CURRENT_NODE	117
Creating the JOB relationship	CREATE_JOB	185
Deleting the JOB relationship	DELETE_JOB	188

Relationship Attributes		
Description of Operations	Applicable Interfaces	Page
Creating a path attribute	CREATE_PATH_ATTRIBUTE	127
Deleting a path attribute	DELETE_PATH_ATTRIBUTE	131
Setting a path attribute	SET_PATH_ATTRIBUTE	135
Getting a path attribute	GET_PATH_ATTRIBUTE	139
Setting inheritance property of a relationship	SET_INHERITANCE	102
Query for inheritance property of a relationship	IS_INHERITABLE	104
Query last time a relationship was modified	TIME_RELATIONSHIP_WRITTEN	120
Getting value of GRANT attribute	GET_GRANTED_RIGHTS	158

Downloaded from http://www.everyspec.com

Node Attributes		
Description of Operations	Applicable Interfaces	Page
Creating a node attribute	CREATE_NODE_ATTRIBUTE	125
Deleting a node attribute	DELETE_NODE_ATTRIBUTE	129
Setting a node attribute	SET_NODE_ATTRIBUTE	133
Getting a node attribute	GET_NODE_ATTRIBUTE	137
Query last time that node attribute was modified	TIME_ATTRIBUTE_WRITTEN	122
Appending to RESULTS attribute	APPEND_RESULTS	190
Replacing the RESULTS attribute	WRITE_RESULTS	191
Reading the RESULTS attribute	GET_RESULTS	192
Reading the CURRENT_STATUS attribute	CURRENT_STATUS	194
Reading the PARAMETERS attribute	GET_PARAMETERS	195
Reading the OPEN_NODE_HANDLE_COUNT attribute	OPEN_NODE_HANDLE_COUNT	202
Reading the IO_UNIT_COUNT attribute	IO_UNIT_COUNT	203
Reading the TIME_ATTRIBUTE_WRITTEN attribute	TIME_ATTRIBUTE_WRITTEN	122
Reading the TIME_CONTENTS_WRITTEN attribute	TIME_CONTENTS_WRITTEN	121
Reading the TIME_CREATED attribute	TIME_CREATED	119
Reading the TIME_RELATIONSHIP_WRITTEN attribute	TIME_RELATIONSHIP_WRITTEN	120
Reading the TIME_STARTED attribute	TIME_STARTED	204
Reading the TIME_FINISHED attribute	TIME_FINISHED	205

Downloaded from http://www.everyspec.com

Node Attributes -- Continued.		
Description of Operations	Applicable Interfaces	Page
Reading the MACHINE_TIME attribute	MACHINE_TIME	206
Reading the PROCESS_SIZE attribute	PROCESS_SIZE	207
Reading the ACCESS_ METHOD attribute	ACCESS_METHOD	218
Reading the FILE_KIND attribute	KIND_OF_FILE	219
Reading the QUEUE_KIND attribute	KIND_OF_QUEUE	220
Reading the DEVICE_KIND attribute	KIND_OF_DEVICE	221
Reading the CURRENT_FILE_SIZE attribute	CURRENT_FILE_SIZE	222
Reading the MAXIMUM_FILE_SIZE attribute	MAXIMUM_FILE_SIZE	223
Reading the CURRENT_QUEUE_SIZE attribute	CURRENT_QUEUE_SIZE	224
Reading the MAXIMUM_QUEUE_SIZE attribute	MAXIMUM_QUEUE_SIZE	225

Node Iterators		
Description of Operations	Applicable Interfaces	Page
Creating a node iterator	CREATE_ITERATOR	107
Query for more nodes on iterator	MORE	109
Approximate size of iterator	APPROXIMATE_SIZE	110
Get next open node handle on an iterator	GET_NEXT	111
Skip next node on an iterator	SKIP_NEXT	113
Get path element to next node on an iterator	NEXT_NAME	114
Delete a node iterator	DELETE_ITERATOR	115

Downloaded from http://www.everyspec.com

Attribute Iterators		
Description of Operations	Applicable Interfaces	Page
Creating an iterator over node attributes	CREATE_NODE_ATTRIBUTE_ITERATOR	142
Creating an iterator over relationship attributes	CREATE_PATH_ATTRIBUTE_ITERATOR	144
Query for more attributes on iterator	MORE	146
Approximate size of iterator	APPROXIMATE_SIZE	147
Get next attribute name on an iterator	NEXT_NAME	148
Get next attribute value on an iterator	GET_NEXT_VALUE	149
Skip next attribute on an iterator	SKIP_NEXT	150
Delete an attribute iterator	DELETE_ITERATOR	151

Access Control		
Description of Operations	Applicable Interfaces	Page
Value of all predefined access rights	ALL_RIGHTS	153
Setting access control information for a node	SET_GRANTED_RIGHTS	154
Deleting access control information for a node	DELETE_GRANTED_RIGHTS	156
Get value of granted rights	GET_GRANTED_RIGHTS	158
Query approved access rights for a process	IS_APPROVED	159
Adopting a role	ADOPT_ROLE	160
Unadopting a role	UNADOPT_ROLE	162

Downloaded from http://www.everyspec.com

Process Control		
Description of Operations	Applicable Interfaces	Page
Creating a spawned process and its process node	SPAWN_PROCESS	174
Creating a process node and calling the process	INVOKE_PROCESS	180
Creating a root process node	CREATE_JOB	185
Deleting a job	DELETE_JOB	188
Waiting for a process to complete	AWAIT_PROCESS_COMPLETION	178
Aborting a process	ABORT_PROCESS	196
Suspending a process	SUSPEND_PROCESS	198
Resuming a suspended process	RESUME_PROCESS	200

Process Information		
Description of Operations	Applicable Interfaces	Page
Appending to RESULTS attribute	APPEND_RESULTS	190
Replacing the RESULTS attribute	WRITE_RESULTS	191
Reading the RESULTS attribute	GET_RESULTS	192
Determing the current status of a process	CURRENT_STATUS	194
Getting the parameters list of a process	GET_PARAMETERS	195
Query for number of open handles of a process	OPEN_NODE_HANDLE_COUNT	202
Query for number of input and output units used by a process	IO_UNIT_COUNT	203
Query for time that a process started	TIME_STARTED	204
Query for time that a process finished	TIME_FINISHED	205
Query for time that a process has been active	MACHINE_TIME	206
Query for the size of a process	PROCESS_SIZE	207

Downloaded from http://www.everyspec.com

Process Information -- Continued.		
Description of Operations	Applicable Interfaces	Page
Query approved access rights for a process	IS_APPROVED	159

Input Output		
Description of Operations	Applicable Interfaces	Page
Read and write operations	Same as specified in [1815A]	
Input and output operations	See subsections of 5.3	
Creating a direct file and its node	CREATE	228
Opening a direct file	OPEN	231
Closing a direct file	CLOSE	232
Resetting a direct file	RESET	233
Synchronizing direct files	SYNCHRONIZE	234
Creating a sequential file and its node	CREATE	237
Opening a sequential file	OPEN	240
Closing a sequential file	CLOSE	241
Resetting a sequential file	RESET	242
Synchronizing sequential files	SYNCHRONIZE	243
Creating a text file and its node	CREATE	246
Opening a text file	OPEN	249
Closing a text file	CLOSE	250
Resetting a text file	RESET	251
Synchronizing text files	SYNCHRONIZE	252
Query the last time that node contents were written	TIME_CONTENTS_WRITTEN	121
Query number of input and output units used by a process	IO_UNIT_COUNT	203
Query access method of a file node	ACCESS_METHOD	218
Query the kind of file node	KIND_OF_FILE	219
Query the kind of queue file node	KIND_OF_QUEUE	220

Downloaded from http://www.everyspec.com

Input Output -- Continued.		
Description of Operations	Applicable Interfaces	Page
Query the kind of device file node	KIND_OF_DEVICE	221
Query the current size of a file	CURRENT_FILE_SIZE	222
Query the maximum size of a file	MAXIMUM_FILE_SIZE	223
Query the current size of a queue	CURRENT_QUEUE_SIZE	224
Query the maximum size of a queue	MAXIMUM_QUEUE_SIZE	225

File Handles		
Description of Operations	Applicable Interfaces	Page
Query for number of open file handles associated with a file node handle	OPEN_FILE_HANDLE_COUNT	72
Opening a direct file handle	OPEN	231
Closing a direct file handle	CLOSE	232
Resetting a direct file handle	RESET	233
Synchronizing direct files	SYNCHRONIZE	234
Opening a sequential file handle	OPEN	240
Closing a sequential file handle	CLOSE	241
Resetting a sequential file handle	RESET	242
Synchronizing sequential files	SYNCHRONIZE	243
Opening a text file handle	OPEN	249
Closing a text file handle	CLOSE	250
Resetting a text file handle	RESET	251
Synchronizing text files	SYNCHRONIZE	252
Opening a scroll terminal file handle	OPEN	287
Query open-ness of a scroll terminal file handle	IS_OPEN	289
Closing a scroll terminal file handle	CLOSE	288

Downloaded from http://www.everyspec.com

File Handles -- Continued.		
Description of Operations	Applicable Interfaces	Page
Resetting a scroll terminal file handle	RESET	315
Synchronizing scroll terminal files	SYNCHRONIZE	316
Setting scroll terminal file handle synchronization	ENABLE_SYNCHRONIZATION	317
Query scroll terminal file handle synchronization	SYNCHRONIZATION_IS_ENABLED	318
Opening a page terminal file handle	OPEN	323
Query open-ness of a page terminal file handle	IS_OPEN	325
Closing a page terminal file	CLOSE	324
Resetting a page terminal file handle	RESET	358
Synchronizing page terminal files	SYNCHRONIZE	359
Setting page terminal file handle synchronization	ENABLE_SYNCHRONIZATION	360
Query page terminal file handle synchronization	SYNCHRONIZATION_IS_ENABLED	361
Opening a form terminal file handle	OPEN	364
Query open-ness of a form terminal file handle	IS_OPEN	366
Closing a form terminal file handle	CLOSE	365
Opening a tape drive file handle	OPEN	395
Query open-ness of a tape drive file handle	IS_OPEN	397
Closing a tape drive file handle	CLOSE	396
Resetting a tape drive file handle	RESET	413

Downloaded from http://www.everyspec.com

Scroll Terminal		
Description of Operations	Applicable Interfaces	Page
Opening a scroll terminal file	OPEN	287
Query open-ness of a scroll terminal file handle	IS_OPEN	289
Closing a scroll terminal file	CLOSE	288
Resetting a scroll terminal file	RESET	315
Setting scroll terminal file handle synchronization	ENABLE_SYNCHRONIZATION	317
Query scroll terminal file handle synchronization	SYNCHRONIZATION_IS_ENABLED	318
Synchronizing scroll terminal files	SYNCHRONIZE	316
Query number of function keys	NUMBER_OF_FUNCTION_KEYS	290
Intercepted input characters	INTERCEPTED_INPUT_ CHARACTERS	291
Intercepted output characters	INTERCEPTED_OUTPUT_ CHARACTERS	292
Enabling function keys for terminal	ENABLE_FUNCTION_KEYS	293
Query of function keys enabled	FUNCTION_KEYS_ARE_ENABLED	294
Setting the active position	SET_ACTIVE_POSITION	295
Query of the active position	ACTIVE_POSITION	296
Query of the size of the terminal	PAGE_SIZE	297
Setting tab stops on the terminal	SET_TAB_STOP	298
Clearing tab stops on the terminal	CLEAR_TAB_STOP CLEAR_ALL_TAB_STOPS	299 300
Advancing to next tab position on the terminal	TAB	301
Sounding the terminal bell	SOUND_BELL	302
Writing to the terminal	PUT	303
Reading a character from the terminal	GET	304
Reading all available characters from the terminal	GET	305

Downloaded from http://www.everyspec.com

Scroll Terminal -- Continued.		
Description of Operations	Applicable Interfaces	Page
Creating a function key descriptor	CREATE_FUNCTION_KEY_DESCRIPTOR	306
Deleting a function key descriptor	DELETE_FUNCTION_KEY_DESCRIPTOR	307
Query number of function keys read from terminal	FUNCTION_KEY_COUNT	308
Query function key usage for the terminal	GET_FUNCTION_KEY	309
Get Function key identification	FUNCTION_KEY_IDENTIFICATION	310
Query for mode of the terminal	MODE	311
Backspacing	BACKSPACE	312
Advancing to next line	NEW_LINE	313
Advancing to next page	NEW_PAGE	314

Page Terminal		
Description of Operations	Applicable Interfaces	Page
Opening a page terminal file handle	OPEN	323
Query open-ness of a page terminal file handle	IS_OPEN	325
Closing a page terminal file	CLOSE	324
Resetting a page terminal file handle	RESET	358
Synchronizing page terminal files	SYNCHRONIZE	359
Setting page terminal file handle synchronization	ENABLE_SYNCHRONIZATION	360
Query page terminal file handle synchronization	SYNCHRONIZATION_IS_ENABLED	361
Number of function keys	NUMBER_OF_FUNCTION_KEYS	326
Intercepted input characters	INTERCEPTED_INPUT_ CHARACTERS	327
Intercepted output characters	INTERCEPTED_OUTPUT_ CHARACTERS	328

Downloaded from http://www.everyspec.com

Page Terminal -- Continued.		
Description of Operations	Applicable Interfaces	Page
Enabling function keys for terminal	ENABLE_FUNCTION_KEYS	329
Query of function keys enabled	FUNCTION_KEYS_ARE_ENABLED	330
Setting the active position	SET_ACTIVE_POSITION	331
Query of the active position	ACTIVE_POSITION	332
Query of the size of the terminal	PAGE_SIZE	333
Setting tab stops on the terminal	SET_TAB_STOP	334
Clearing tab stops on the terminal	CLEAR_TAB_STOP CLEAR_ALL_TAB_STOPS	335 336
Advancing to next tab position on the terminal	TAB	337
Sounding the terminal bell	SOUND_BELL	338
Writing to the terminal	PUT	339
Reading a character from the terminal	GET	340
Reading all available characters from the terminal	GET	341
Creating a function key descriptor	CREATE_FUNCTION_KEY_DESCRIPTOR	342
Deleting a function key descriptor	DELETE_FUNCTION_KEY_DESCRIPTOR	343
Query number of function keys read from terminal	FUNCTION_KEY_COUNT	344
Query function key usage for the terminal	GET_FUNCTION_KEY	345
Get function key identification	FUNCTION_KEY_IDENTIFICATION	346
Query for mode of the terminal	MODE	347
Deleting characters	DELETE_CHARACTER	348
Deleting lines	DELETE_LINE	349
Erasing characters	ERASE_CHARACTER	350
Erasing characters in a display	ERASE_IN_DISPLAY	351
Erasing characters in a line	ERASE_IN_LINE	352
Inserting spaces in a line	INSERT_SPACE	353
Inserting lines in the output terminal	INSERT_LINE	354

Downloaded from http://www.everyspec.com

Page Terminal -- Continued.		
Description of Operations	Applicable Interfaces	Page
Query for graphic rendition support	GRAPHIC_RENDITION_IS_SUPPORTED	355
Selecting the graphic rendition	SELECT_GRAPHIC_RENDITION	356
Determining the effect of writing to the end position of the terminal	END_POSITION_SUPPORT	357

Form Terminal		
Description of Operations	Applicable Interfaces	Page
Opening a form terminal file	OPEN	364
Query open-ness of a file handle	IS_OPEN	366
Closing a form terminal file	CLOSE	365
Number of function keys	NUMBER_OF_FUNCTION_KEYS	367
Intercepting input characters	INTERCEPTED_INPUT_ CHARACTERS	368
Intercepting output characters	INTERCEPTED_OUTPUT_ CHARACTERS	369
Creating a form	CREATE_FORM	370
Deleting a form	DELETE_FORM	371
Copying a form	COPY_FORM	372
Defining a qualified area	DEFINE_QUALIFIED_AREA	373
Removing an area qualifier	REMOVE_AREA_QUALIFIER	374
Setting the active position	SET_ACTIVE_POSITION	375
Query of the active position	ACTIVE_POSITION	376
Moving to the next qualified area	ADVANCE_TO_QUALIFIED_AREA	377
Writing to the form	PUT	378
Erasing a qualified area	ERASE_AREA	379
Erasing a form	ERASE_FORM	380
Activating a form on the terminal	ACTIVATE	381
Reading from the form	GET	382
Query for changes to a form	IS_FORM_UPDATED	383

Downloaded from http://www.everyspec.com

Form Terminal -- Continued.		
Description of Operations	Applicable Interfaces	Page
Get function key identification	FUNCTION_KEY_IDENTIFICATION	384
Query for the termination key	TERMINATION_KEY	385
Query for size of the form	FORM_SIZE	386
Query for the terminal size	TERMINAL_SIZE	387
Query for the area qualifier requiring space in the form	AREA_QUALIFIER_REQUIRES_SPACE	388
Query for the area qualifier requiring space on the terminal	AREA_QUALIFIER_REQUIRES_SPACE	389

Magnetic Tape		
Description of Operations	Applicable Interfaces	Page
Opening a tape drive file	OPEN	395
Query open-ness of a tape drive file handle	IS_OPEN	397
Closing a tape drive file	CLOSE	396
Resetting a tape drive file handle	RESET	413
Query for mode of the tape drive	MODE	398
Requesting a tape mount	REQUEST_MOUNT	399
Requesting a tape dismount	REQUEST_DISMOUNT	402
Loading a tape	LOAD	400
Unloading a tape	UNLOAD	401
Query for the positon of a tape	POSITION	403
Query for the recording mode of a tape	RECORDING_METHOD	408
Query status of a tape drive	STATUS	407
Query whether or not a write ring is installed	IS_WRITE_RING_INSTALLED	409
Rewind the tape	REWIND_TAPE	404
Skipping tape marks on a tape	SKIP_TAPE_MARK	405
Writing tape marks on a tape	WRITE_TAPE_MARK	406
Reading blocks from a tape	READ_BLOCK	411
Writing blocks to a tape	WRITE_BLOCK	412
Skipping blocks on a tape	SKIP_BLOCK	410

Downloaded from http://www.everyspec.com

Queues		
Description of Operations	Applicable Interfaces	Page
Opening a queue file node handle	OPEN	63
Closing a queue file node handle	CLOSE	66
Opening a sequential queue file handle	OPEN	240
Reading elements from a sequential queue	CAIS_SEQUENTIAL_IO.READ	
Writing elements to a sequential queue	CAIS_SEQUENTIAL_IO.WRITE	
Closing a sequential queue file handle	CLOSE	241
Resetting a sequential queue file handle	RESET	242
Opening a text queue file handle	OPEN	249
Reading elements from a text queue	CAIS_TEXT_IO subprograms	
Writing elements to a text queue	CAIS_TEXT_IO subprograms	
Closing a text queue file handle	CLOSE	250
Resetting a text queue file handle	RESET	251
Querying the kind of queue	KIND_OF_QUEUE	220
Determining the current size of a queue	CURRENT_QUEUE_SIZE	224
Determining the maximum size of a queue	MAXIMUM_QUEUE_SIZE	225
Creating a nonsynchronous copy queue node	CREATE_NONSYNCHRONOUS_ COPY_QUEUE	257
Creating a nonsynchronous mimic queue node	CREATE_NONSYNCHRONOUS_ MIMIC_QUEUE	262
Creating a nonsynchronous solo text queue node	CREATE_NONSYNCHRONOUS_ SOLO_TEXT_QUEUE	267
Creating a nonsynchronous solo sequential queue node	CREATE_NONSYNCHRONOUS_ SOLO_SEQUENTIAL_QUEUE	271
Creating a synchronous solo text queue node	CREATE_SYNCHRONOUS_ SOLO_TEXT_QUEUE	274

Downloaded from http://www.everyspec.com

Queues -- Continued.		
Description of Operations	Applicable Interfaces	Page
Creating a synchronous solo sequential queue node	CREATE_SYNCHRONOUS_ SOLO_SEQUENTIAL_QUEUE	278

Import Export		
Description of Operations	Applicable Interfaces	Page
Importing the contents of a file	IMPORT_CONTENTS	415
Exporting the contents of a file	EXPORT_CONTENTS	417

List Management		
Description of Operations	Applicable Interfaces	Page
Copying a list	COPY_LIST	427
Making a list empty	SET_TO_EMPTY_LIST	428
Converting from text to list form	CONVERT_TEXT_TO_LIST	429
Converting from list form to text	TEXT_FORM	430
Query equality of two lists	IS_EQUAL	431
Deleting an item from linear list	DELETE	432
Query kind of list	KIND_OF_LIST	433
Query kind of list item	KIND_OF_ITEM	434
Inserting sequence of items into list	SPLICE	435
Concatenating two lists	CONCATENATE_LISTS	436
Extracting a list from a list	EXTRACT_LIST	437
Query number of items in a list	NUMBER_OF_ITEMS	438
Query position of current list	POSITION_OF_CURRENT_LIST	439
Query whether the current list is outermost list	CURRENT_LIST_IS_OUTERMOST	440
Making the next outer linear list the current list	MAKE_CONTAINING_LIST_CURRENT	441
Making this item's list the current list	MAKE_THIS_ITEM_CURRENT	442
Query the length of the text form of a list or of a list item	TEXT_LENGTH	444

Downloaded from http://www.everyspec.com

List Management -- Continued.		
Description of Operations	Applicable Interfaces	Page
Getting the name of a named item	GET_ITEM_NAME	446
Query the position of a named item	POSITION_BY_NAME	447

List Items		
Description of Operations	Applicable Interfaces	Page
Extracting a list value from a list	EXTRACT_VALUE	449
Replacing a list value in a list	REPLACE	451
Inserting a list value in a list	INSERT	453
Locating a list item by value in a list	POSITION_BY_VALUE	455

Identifier Items		
Description of Operations	Applicable Interfaces	Page
Copying a token	COPY_TOKEN	458
Converting an identifier from text to token form	CONVERT_TEXT_TO_TOKEN	459
Converting an identifier from token to text form	TEXT_FORM	460
Query equality of two identifier tokens	IS_EQUAL	461
Extracting an identifier item from a list	EXTRACT_VALUE	462
Replacing an identifier item in a list	REPLACE	464
Inserting an identifier item into a list	INSERT	466
Locating an identifier-valued item by value within a list	POSITION_BY_VALUE	468

Downloaded from http://www.everyspec.com

Integer Items		
Description of Operations	Applicable Interfaces	Page
Converting an integer to its text form	TEXT_FORM	470
Extracting an integer item from a list	EXTRACTED_VALUE	471
Replacing an integer item in a list	REPLACE	473
Inserting an integer item into a list	INSERT	475
Locating an integer-valued item by value within a list	POSITION_BY_VALUE	477

Float Items		
Description of Operations	Applicable Interfaces	Page
Converting a floating point value to its text form	TEXT_FORM	479
Extracting a floating point value item from a list	EXTRACTED_VALUE	480
Replacing a floating point value item in a list	REPLACE	482
Inserting a floating point value item into a list	INSERT	484
Locating a floating point-valued item by value within a list	POSITION_BY_VALUE	486

String Items		
Description of Operations	Applicable Interfaces	Page
Extracting a string item from a list	EXTRACTED_VALUE	489
Replacing a string item in a list	REPLACE	491
Inserting a string item into a list	INSERT	493
Locating a string-valued item by value within a list	POSITION_BY_VALUE	495

Downloaded from http://www.everyspec.com

Time and Calendar		
Description of Operations	Applicable Interfaces	Page
Getting the current time	CLOCK	499
Extracting year from time	YEAR	500
Extracting month from time	MONTH	501
Extracting day from time	DAY	502
Extracting seconds from time	SECONDS	503
Converting time into year, month, day and seconds	SPLIT	504
Combining year, month, day and seconds into time	TIME_OF	505
Addition of time and duration	“+”	506
Subtraction of time and duration	“-”	507
Comparing values of time	“<”, “<=”, “>”, “>=”	508

Downloaded from http://www.everyspec.com

Appendix D
Syntax Summary

The material contained in this appendix is not a mandatory part of the standard.

This appendix summarizes the syntax descriptions given throughout this document. Lexical categories not defined here are found in [1815A] Section 2. The notation used is a form of Backus-Naur Form (BNF):

```
Words   identify syntactic categories;
[ ]     identify optional items;
{ }     identify items which may be repeated zero or more times;
|       separates alternatives;
::=     separates the left-hand and right-hand sides of productions.
```

The following definitions are global.

```
identifier          ::= letter { [ underline ] letter_or_digit }
graphic_character   ::= letter
                        | digit
                        | special_character
                        | space_character
letter_or_digit     ::= letter | digit
letter              ::= upper_case_letter | lower_case_letter
underline           ::= _
digit               ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
upper_case_letter   ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
                        N | O | P | Q | R | S | T | U | V | W | X | Y | Z
lower_case_letter   ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
                        n | o | p | q | r | s | t | u | v | w | x | y | z
special_character   ::= " | # | & | ' | ( | ) | * | + | , | - | . | / | :
                        | ; | < | = | > | _ | |
```

10. Pathname Syntax

The syntax for pathnames is specified as follows:

```
pathname                     ::= relationship_key_designator{path_element}
                                 | path_element{path_element}
                                 | :
path_element                 ::= 'relation_name [ ( [relationship_key_designator] ) ]
                                 | .relationship_key_designator
relation_name                ::= identifier
relationship_key_designator  ::= relationship_key
                                 | [identifier_prefix] #
relationship_key             ::= identifier
identifier_prefix            ::= letter { [underline] letter_or_digit } [underline]
```

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

Downloaded from http://www.everyspec.com

20. GRANT Attribute Value Syntax

The syntax for GRANT attribute values is as follows:

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

30. Classification Attribute Value Syntax

The syntax for classification attribute values is as follows:

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

40. List External Representation Syntax

The syntax for the external representation of lists is as follows:

```
list           ::= named_list
                   | unnamed_list
                   | empty_list
named_list     ::= ( named_item { , named_item } )
unnamed_list   ::= ( item_value { , item_value } )
empty_list     ::= ()
named_item     ::= item_name => item_value
item_name      ::= identifier
item_value     ::= list
                   | quoted_string
                   | integer_number
                   | float_number
                   | identifier
integer_number ::= [-] integer
float_number   ::= [-] decimal_literal
quoted_string  ::= string_literal
```

Downloaded from http://www.everyspec.com

Appendix E
CAIS Access Control Management

The material contained in this appendix is a mandatory part of the standard.

The reference manual of each CAIS implementation must include an appendix (called Appendix E) that describes implementation-dependent aspects of access control management for that implementation.

10. Package Replacement

This section describes an implementation-dependent replacement for Package CAIS_ACCESS_CONTROL_MANAGEMENT as described in Section 5.1.4. The implementation behavior of such a replacement package must be documented herein. As a minimum, Appendix E must include:

a. A description of the security model, if a model different from the model described in Section 4.4 and Section 5.1.4 is chosen.

b. A description (using the method of description that is described in Section 4.2) of the interfaces replacing those in Section 5.1.4.

20. General Access Control

This section describes implementation-dependent items related to access control.

a. A description of the interfaces for the creation, modification or deletion of group nodes as well as the effects of the deletion of group nodes.

b. A description of the interfaces for the creation, modification and deletion of the relationships of the predefined relation DEFAULT_ROLE, in particular for those emanating from nodes representing the executable image of a program.

c. A description of the effects of alterations of group memberships or of relationships of the predefined relation DEFAULT_ROLE on concurrently executing processes.

d. A description of the keys of relationships of the predefined relation ADOPTED_ROLE when the relationships are created implicitly.

e. A description of the hierarchical classification level set and the non-hierarchical category set (when mandatory security is implemented).

f. A description of the criteria that allow an executing process (subject) to establish or alter an access relationship to a group node. For some interfaces, the exception ACCESS_VIOLATION is raised if the executing process (subject) is not allowed to establish or alter an access relationship to the given group node according to these criteria.

g. A description of the possible values of a node's classification as an object or as a subject.

Downloaded from http://www.everyspec.com

Appendix F
Implementation Dependencies

The material contained in this appendix is a mandatory part of the standard.

Reliance on any information provided in this appendix endangers transportability of tools.

This appendix describes those aspects of a CAIS implementation which are implementation-dependent. Some of these aspects are explicitly noted in the CAIS specification and the implementation behavior should be documented herein. Other aspects may be the result of implementation choices and ambiguity in the CAIS specification; it is recommended that such observed ambiguities be reported to the Ada Joint Program Office as design feedback. (See Section 1.2, page 2, on application guidance.)

The reference manual of each CAIS implementation must include an appendix (called Appendix F) that describes all implementation-dependent characteristics other than those covered in Appendix E. The Appendix F for a given implementation must list in particular:

a. Implementation-defined pragmatic limits.

b. Implementation-defined exceptions.

c. Whether and when nodes whose primary relationships have been deleted are actually removed.

d. The effect on existing node iterators of creation or deletion of relationships.

e. The effect on existing attribute iterators of creation or deletion of attributes or relationships.

f. The meaning of the values returned by the TIME_STARTED (see Section 5.2.2.16, page 204) and TIME_FINISHED (see Section 5.2.2.17, page 205) interfaces.

g. The package CAIS_DEVICES.

h. The revised Table XI and additional packages to support any allowed extensions made by the implementation to the types defined in the package CAIS_DEVICES.

i. Other aspects of any implementation explicitly noted as implementation-dependent in the CAIS specification; these must be identified by CAIS Section number and must describe the implementation choices made.

Downloaded from http://www.everyspec.com

Index

-- latest key 29

+, CAIS_CALENDAR 506

-, CAIS_CALENDAR 507

: -- current process node 31, 78

<, CAIS_CALENDAR 508
<=, CAIS_CALENDAR 508

>, CAIS_CALENDAR 508
>=, CAIS_CALENDAR 508

Abort 7, 168, 178
Abort process 94, 196
ABORT_PROCESS 196, 197
Access 7, 14, 36, 40, 154, 156, 172, 173, 511, 517, 519
  to a node 7, 35, 36
    See also Approved access rights
Access checking 7, 35
Access control 7, 19, 35, 597
  discretionary 7, 9, 11, 35, 36, 152
  information 54, 152, 154, 156, 164, 174, 180, 185, 246
  mandatory 7, 11, 12, 35, 48, 52, 56, 152
  mechanisms 152
  rules 48
Access relationship 7, 14, 40, 153, 158, 519
Access right 7, 35, 36, 40, 42, 48, 517, 519
  constraints 7, 35, 55
  discretionary 42
  EXISTENCE 7, 42
Access synchronization 57, 63
ACCESS_METHOD 218, 519
ACCESS_METHOD_KIND 214
ACCESS_VIOLATION 40, 55, 56
Accessible 7, 42
ACTIVATE 381, 385
Activation 174, 180, 185, 521
Active position 8, 284, 319, 362, 376
  advance 8, 284, 295, 301, 303, 313, 314, 319, 331, 337, 339, 362
  precede 13, 284, 295, 319, 362
ACTIVE_POSITION 296, 332, 376
Ada external file 22, 23
Ada Programming Support Environment 1, 8
Additional interfaces
  definition 20
Adopt a role 8, 37, 160
ADOPT_ROLE 37, 160
Adopted roles of a process 37, 42, 160, 511
ADOPTED_ROLE 8, 16, 37, 160, 162, 172, 173, 517, 597
ADVANCE_TO_QUALIFIED_AREA 377
ALL_RIGHTS 44, 153
APPEND 44, 58
APPEND_ATTRIBUTES 43, 59
APPEND_CONTENTS 44, 59
APPEND_INTENT 215
APPEND_RELATIONSHIPS 43, 60
APPEND_RESULTS 190, 192
Application Guidance 2
Approved access rights 8, 42, 48, 159
APPROXIMATE_SIZE 110, 147
APSE 8, 12
  See also Ada Programming Support Environment
Area qualifier 8, 362, 373, 374, 377, 382, 388, 389

AREA_INPUT_KIND 363
AREA_INTENSITY_KIND 363
AREA_PROTECTION_KIND 363
AREA_QUALIFIER_REQUIRES_SPACE 381, 388, 389
AREA_VALUE_KIND 363
Attribute 8, 22, 32, 87, 510
  ACCESS_METHOD 209, 218, 519
  CURRENT_FILE_SIZE 222, 519
  CURRENT_QUEUE_SIZE 224, 519
  CURRENT_STATUS 169, 194, 519
  DEVICE_KIND 209, 221, 519
  FILE_KIND 209, 219, 519, 520
  GRANT 40, 42, 152, 154, 158, 172, 173, 511, 517, 519, 596
  HIGHEST_CLASSIFICATION 52, 519
  INHERITABLE 519
  IO_UNIT_COUNT 169, 203, 519
  iteration types and subtypes 141
  iterator 8, 11, 141
  LOWEST_CLASSIFICATION 52, 519
  MACHINE_TIME 169, 206, 520
  MAXIMUM_FILE_SIZE 223, 520
  MAXIMUM_QUEUE_SIZE 225, 520
  name 32, 123
  NODE_KIND 519
  OBJECT_CLASSIFICATION 51, 520
  OPEN_NODE_HANDLE_COUNT 169, 202, 520
  PARAMETERS 169, 195, 520
  PROCESS_SIZE 169, 207, 520
  QUEUE_KIND 209, 253, 520
  RESULTS 169, 190, 191, 192, 520
  SUBJECT_CLASSIFICATION 51, 520
  TIME_ATTRIBUTE_WRITTEN 122, 521
  TIME_CONTENTS_WRITTEN 121, 521
  TIME_CREATED 119, 521
  TIME_FINISHED 169, 205, 521
  TIME_RELATIONSHIP_WRITTEN 120, 521
  TIME_STARTED 169, 204, 521
  value FILE 71
  value PROCESS 71
  value STRUCTURAL 71
ATTRIBUTE_ERROR 55
ATTRIBUTE_ITERATOR 141
ATTRIBUTE_LIST 54
ATTRIBUTE_NAME 54, 141
ATTRIBUTE_NAME_PATTERN 141
AWAIT_PROCESS_COMPLETION 178

BACKSPACE 312
Base node 8, 29, 31, 53
BASE_PATH 78

CAIS_ACCESS_CONTROL_MANAGEMENT 19, 40, 53, 152, 597
CAIS_ATTRIBUTE_MANAGEMENT 53
CAIS_CALENDAR 497
CAIS_DEFINITIONS 53
CAIS_DEVICES 214
CAIS_DIRECT_IO 226, 512
CAIS_DURATION 496
CAIS_FLOAT_ITEM 478
CAIS_FORM_TERMINAL_IO 362
CAIS_IDENTIFIER_ITEM 457
CAIS_IMPORT_EXPORT 414
CAIS_INTEGER 496
CAIS_INTEGER_ITEM 469
CAIS_IO_ATTRIBUTES 217

Downloaded from http://www.everyspec.com

CAIS_IO_DEFINITIONS 215
CAIS_LIST_ITEM 448
CAIS_LIST_MANAGEMENT 123, 425
CAIS_MAGNETIC_TAPE_IO 390
CAIS_NATURAL 496
CAIS_NODE_MANAGEMENT 53
CAIS_PAGE_TERMINAL_IO 319
CAIS_POSITIVE 496
CAIS_PRAGMATICS 509
CAIS_PROCESS_MANAGEMENT 57, 172
CAIS_PROCESS_DEFINITIONS 171
CAIS_QUEUE_MANAGEMENT 253
CAIS_SCROLL_TERMINAL_IO 284
CAIS_SEQUENTIAL_IO 235, 512
CAIS_STRING_ITEM 488
CAIS_STRUCTURAL_NODE_MANAGEMENT 53, 57, 163
CAIS_TEXT_IO 244
Canonical list text representation 8, 422, 430
Canonical text form representation 470, 479
CAPACITY_ERROR 509
Case distinction 123, 421, 423
CHANGE_INTENT 57, 58, 59, 60, 61, 67, 68
CHARACTER_ARRAY 286, 321, 363
Classification 48, 51, 52, 596
  hierarchical 48, 51
  label 52
  level 52
CLEAR_ALL_TAB_STOPS 300, 336
CLEAR_TAB_STOP 299, 335
CLOCK 499
CLOSE 57, 66, 68, 232, 241, 250, 281, 288, 324, 365, 396
Closed file handle 232, 241, 250, 288, 324, 396
Closed node handle 8, 53, 57
CONCATENATE_LISTS 436
Constant
  ACCESS_RELATIONSHIPS_OF_OBJECT 511
  ADOPTED_ROLES_OF_PROCESS 511
  ATTRIBUTES_PER_RELATIONSHIP 511
  ATTRIBUTES_PER_NODE 510
  CAIS_ACCESS_RELATIONSHIPS_OF_OBJECT 511
  CAIS_ADOPTED_ROLES_OF_PROCESS 511
  CAIS_ATTRIBUTES_PER_RELATIONSHIP 511
  CAIS_ATTRIBUTES_PER_NODE 510
  CAIS_DIRECT_IO_INDEX_RANGE_UPPER_BOUND 512
  CAIS_DIRECT_IO_RECORD_SIZE 512
  CAIS_ELEMENTS_OF_ATTRIBUTE_ITERATOR 510
  CAIS_ELEMENTS_OF_NODE_ITERATOR 510
  CAIS_EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE 510
  CAIS_EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE 510
  CAIS_FILE_HANDLES_PER_PROCESS 513
  CAIS_GRANT_ITEMS_ON_GRANT_ATTRIBUTE 511
  CAIS_GROUP_NODES 511
  CAIS_IDENTIFIER_LENGTH 509
  CAIS_IDENTIFIER_ITEM_LENGTH 513
  CAIS_LENGTH_OF_PRIMARY_PATH 511
  CAIS_LIST_LENGTH 513
  CAIS_LIST_MAXIMUM_DIGITS 514
  CAIS_LIST_TEXT_LENGTH 514
  CAIS_MAXIMUM_INTEGER 514
  CAIS_MAXIMUM_TAPE_BLOCK_LENGTH 513
  CAIS_MINIMUM_INTEGER 514
  CAIS_MINIMUM_TAPE_BLOCK_LENGTH 513
  CAIS_NODE_HANDLES_PER_PROCESS 509
  CAIS_NODES_IN_COPY_TREE 509
  CAIS_NODES_IN_DELETE_TREE 510
  CAIS_NUMBER_OF_NODES 511
  CAIS_PATHNAME_LENGTH 509
  CAIS_SEQUENTIAL_IO_FILE_SIZE 512
  CAIS_SEQUENTIAL_IO_RECORD_SIZE 512
  CAIS_SMALL_FOR_CAIS_DURATION 514
  CAIS_STRING_ITEM_LENGTH 514
  CAIS_TEXT_IO_COLUMNS_PER_LINE 512
  CAIS_TEXT_IO_LINES_PER_FILE 512
  CAIS_TEXT_IO_LINES_PER_PAGE 512
  CURRENT_NODE 55
  CURRENT_PROCESS 55
  CURRENT_USER 55, 172
  DEFAULT_GRAPHIC_RENDITION 321
  DEFAULT_RELATION 55
  DIRECT_IO_INDEX_RANGE_UPPER_BOUND 512
  DIRECT_IO_INDEX_RANGE_UPPER_BOUND 512
  DIRECT_IO_RECORD_SIZE 512
  DIRECT_IO_RECORD_SIZE 512
  ELEMENTS_OF_ATTRIBUTE_ITERATOR 510
  ELEMENTS_OF_NODE_ITERATOR 510
  EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE 510
  EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE 510
  FILE_HANDLES_PER_PROCESS 513
  FILE_STORAGE_UNIT_SIZE 513
  GRANT_ITEMS_ON_GRANT_ATTRIBUTE 511
  GROUP_NODES 511
  IDENTIFIER_ITEM_LENGTH 513
  IDENTIFIER_LENGTH 509
  LATEST_KEY 27, 55
  LIST_LENGTH 513
  LIST_MAXIMUM_DIGITS 514
  LIST_TEXT_LENGTH 514
  LONG_DELAY 55
  MAXIMUM_INTEGER 514
  MAXIMUM_TAPE_BLOCK_LENGTH 513
  MEMORY_STORAGE_UNIT_SIZE 513
  MINIMUM_INTEGER 514
  MINIMUM_TAPE_BLOCK_LENGTH 513
  NODE_HANDLES_PER_PROCESS 509
  NODES_IN_COPY_TREE 509
  NODES_IN_DELETE_TREE 510
  PATHNAME_LENGTH 509, 514
  QUEUE_STORAGE_UNIT_SIZE 513
  ROOT_PROCESS 171
  SEQUENTIAL_IO_FILE_SIZE 512
  SEQUENTIAL_IO_RECORD_SIZE 512
  SMALL_FOR_CAIS_DURATION 514
  STANDARD_ERROR 171
  STANDARD_INPUT 171
  STANDARD_OUTPUT 171
  STRING_ITEM_LENGTH 514
  TEXT_IO_COLUMNS_PER_LINE 512
  TEXT_IO_LINES_PER_FILE 512
  TEXT_IO_LINES_PER_PAGE 512
  UNRESTRICTED 509
Contents 8, 22
  of a file node 22
  of a process node 22
  of a structural node 22
CONTROL 44, 61
CONVERT_TEXT_TO_LIST 429
CONVERT_TEXT_TO_TOKEN 459
Copy queue 9
  See also Queue, copy
COPY_FORM 372
COPY_LIST 427
COPY_NODE 87, 88, 89
COPY_TOKEN 458
COPY_TREE 89, 90, 509
Copying nodes 57

Downloaded from http://www.everyspec.com

INDEX

Coupled file 9
Coupled file node 253
CREATE 57, 228, 237, 246
CREATE_FORM 370
CREATE_FUNCTION_KEY_DESCRIPTOR 306, 342
CREATE_ITERATOR 107, 110
CREATE_JOB 14, 27, 57, 152, 169, 185, 195
CREATE_NODE 57, 164, 165, 166
CREATE_NODE_ATTRIBUTE 125
CREATE_NODE_ATTRIBUTE_ITERATOR 142
CREATE_NONSYNCHRONOUS_COPY_QUEUE 257
CREATE_NONSYNCHRONOUS_MIMIC_QUEUE 262
CREATE_NONSYNCHRONOUS_SOLO_SEQUENTIAL_QUEUE 271
CREATE_NONSYNCHRONOUS_SOLO_TEXT_QUEUE 267
CREATE_PATH_ATTRIBUTE 127, 128
CREATE_PATH_ATTRIBUTE_ITERATOR 144
CREATE_SECONDARY_RELATIONSHIP 24, 98, 99
CREATE_SYNCHRONOUS_SOLO_SEQUENTIAL_QUEUE 278
CREATE_SYNCHRONOUS_SOLO_TEXT_QUEUE 274
Creating nodes
  file 87, 89, 228, 237, 246
  process 172, 174, 180, 185
  structural 87, 89, 163
Current job 9, 27, 171
Current linear list 421, 435, 436, 441, 442
Current node 9, 27, 55, 117
Current process 9, 29, 55, 116, 117, 159, 160
Current process node 9, 29, 31, 162
Current status 168
Current user 9, 27
CURRENT_FILE_SIZE 222, 519
CURRENT_JOB 9, 25, 27, 172, 173, 517
CURRENT_LIST_IS_OUTERMOST 440
CURRENT_NODE 9, 25, 27, 29, 31, 55, 116, 172, 173, 517
CURRENT_PROCESS 55
CURRENT_QUEUE_SIZE 224, 519
CURRENT_STATUS 169, 194, 519
CURRENT_USER 9, 25, 27, 31, 55, 172, 173, 185, 517

DATA_ERROR 215
DAY 502
DAY_DURATION 498
DAY_NUMBER 498
Default group node 9, 36, 37, 172, 173
Default relation name 55
DEFAULT_GRAPHIC_RENDITION 321
DEFAULT_RELATION 55
DEFAULT_ROLE 36, 37, 517, 597
DEFINE_QUALIFIED_AREA 373
DELETE 432
DELETE_CHARACTER 348
DELETE_FORM 371
DELETE_FUNCTION_KEY_DESCRIPTOR 307, 343
DELETE_GRANTED_RIGHTS 156
DELETE_ITERATOR 109, 110, 115, 151
DELETE_JOB 24, 188
DELETE_LINE 349
DELETE_NODE 24, 94, 95, 96, 188
DELETE_NODE_ATTRIBUTE 129
DELETE_PATH_ATTRIBUTE 131, 132
DELETE_SECONDARY_RELATIONSHIP 24, 95, 100, 101
DELETE_TREE 24, 96, 97, 510
Deleting nodes 24, 57, 94, 96, 186, 188, 510, 597
Dependent process 9, 24, 27
Dependent process node 52
Determining access rights 40
Device 9, 16, 25, 52, 517
Device file 9, 211
Device name 9, 25
DEVICE_ERROR 55
DEVICE_KIND 519
DEVICE_KIND_ARRAY 215
DEVICE_KIND_TYPE 214
Direct 227, 519
Direct file 512, 513
DIRECT_IO_INDEX_RANGE_UPPER_BOUND 512
DIRECT_IO_RECORD_SIZE 512
Discretionary access checking 48
Discretionary access control 7, 9, 35, 36, 152
Discretionary access control rights 11
Discretionary access rights 42, 153, 174, 180, 185
DISCRETIONARY_ACCESS 40, 164, 172, 173, 174, 180, 185
DISCRETIONARY_ACCESS_LIST 54, 153
DOT 10, 31, 36, 37, 517, 595

Element (of a file) 9, 228, 235, 244
EMANATING_PRIMARY_RELATIONSHIPS_PER_NODE 510
EMANATING_SECONDARY_RELATIONSHIPS_PER_NODE 510
Empty list 10, 419, 425, 428, 429, 432, 445
EMPTY_LIST 425
ENABLE_FUNCTION_KEYS 293, 329
ENABLE_SYNCHRONIZATION 317, 360
End position 10, 362, 378, 382
END_ERROR 215
END_OF_FILE 283
END_POSITION_SUPPORT 339, 357
ERASE_AREA 379
ERASE_CHARACTER 350
ERASE_FORM 380
ERASE_IN_DISPLAY 321, 351
ERASE_IN_LINE 321, 352
Exception
  ACCESS_VIOLATION 40, 55, 56
  ATTRIBUTE_ERROR 55
  CAPACITY_ERROR 216, 509
  DATA_ERROR 215
  DEVICE_ERROR 55
  END_ERROR 215
  EXECUTABLE_IMAGE_ERROR 171
  EXISTING_NODE_ERROR 55
  FILE_KIND_ERROR 215
  FORM_STATUS_ERROR 215
  FUNCTION_KEY_STATUS_ERROR 215, 216
  INTENT_VIOLATION 55
  ITEM_KIND_ERROR 425
  ITERATOR_ERROR 55
  LAYOUT_ERROR 215, 216
  LIST_KIND_ERROR 425
  LIST_POSITION_ERROR 425
  LOCK_ERROR 55, 94, 197
  MODE_ERROR 215, 216
  NAME_ERROR 52, 55, 56
  NAMED_LIST_ERROR 425
  NODE_KIND_ERROR 55, 56
  PATHNAME_SYNTAX_ERROR 55, 56
  PREDEFINED_ATTRIBUTE_ERROR 55, 56
  PREDEFINED_RELATION_ERROR 55, 56
  RELATIONSHIP_ERROR 55, 56
  RESOURCE_ERROR 509
  SEARCH_ERROR 425
  SECURITY_VIOLATION 52, 55, 56, 177
  STATUS_ERROR 55, 56
  SYNTAX_ERROR 55, 56, 425
  TAPE_STATUS_ERROR 394
  TERMINAL_POSITION_ERROR 215, 216
  TOKEN_ERROR 420, 425

Downloaded from http://www.everyspec.com

USE_ERROR 55, 56
Exceptions 56, 599
definition 20
implementation-defined 509
EXCLUSIVE_APPEND 58
EXCLUSIVE_APPEND_RELATIONSHIPS 60
EXCLUSIVE_APPEND_ATTRIBUTES 59
EXCLUSIVE_APPEND_CONTENTS 59
EXCLUSIVE_CONTROL 60
EXCLUSIVE_READ 58
EXCLUSIVE_READ_ATTRIBUTES 59
EXCLUSIVE_READ_CONTENTS 58
EXCLUSIVE_READ_RELATIONSHIPS 60
EXCLUSIVE_WRITE 58, 96, 188
EXCLUSIVE_WRITE_ATTRIBUTES 59
EXCLUSIVE_WRITE_CONTENTS 58
EXCLUSIVE_WRITE_RELATIONSHIPS 60
Executable image 37, 171, 172, 173, 517, 597
EXECUTABLE_IMAGE 172, 173, 517
EXECUTABLE_IMAGE_ERROR 171
EXECUTE 44, 61
Execute under the authority 40
Execution 11, 14, 15, 22, 168, 174, 180, 185
EXISTENCE 7, 43
Existence of a node 55
EXISTING_NODE_ERROR 55
EXPORT_CONTENTS 417
External file 10, 22, 23
External representation 514, 596
EXTRACT_LIST 437
EXTRACT_VALUE 449, 462
EXTRACTED_VALUE 471, 480, 489

File 10, 12, 22, 71, 519
File handle 10, 11, 208, 231
File node 10, 16, 22, 23, 25, 27, 52, 87, 89, 92
File transfer 414
FILE_KIND 215, 519, 520
FILE_KIND_ERROR 215
FILE_MODE 227, 236, 245, 286, 321, 392
FILE_TYPE 10, 208, 227, 236, 245, 286, 321, 363, 392
Floating point equality 420
Form 10, 14, 15, 362, 373
Form terminal 15, 211, 362, 363
FORM_SIZE 381, 386
FORM_STATUS_ERROR 215
FORM_TERMINAL 519
FORM_TYPE 363
FUNCTION_KEY_COUNT 304, 305, 308, 309, 340, 341, 344, 345
FUNCTION_KEY_DESCRIPTOR 286, 321
FUNCTION_KEY_IDENTIFICATION 310, 346, 384
FUNCTION_KEY_NAME 286, 321, 363
FUNCTION_KEY_STATUS_ERROR 215, 216
FUNCTION_KEYS_ARE_ENABLED 294, 330

GET 169, 282, 304, 305, 340, 341, 382, 519
GET_CURRENT_NODE 57, 117, 118
GET_FUNCTION_KEY 309, 345
GET_GRANTED_RIGHTS 158
GET_ITEM_NAME 446
GET_NEXT 57, 106, 107, 109, 110, 111, 112, 114
GET_NEXT_VALUE 141, 142, 144, 146, 147, 149
GET_NODE_ATTRIBUTE 137
GET_PARAMETERS 195
GET_PATH_ATTRIBUTE 139
GET_RESULTS 169, 178, 192
GRANT 40, 42, 152, 154, 158, 172, 173, 511, 517, 519, 596
GRANT_VALUE 152
GRAPHIC_CHARACTERS 363
GRAPHIC_RENDITION_ARRAY 321
GRAPHIC_RENDITION_IS_SUPPORTED 355
GRAPHIC_RENDITION_KIND 321
Group 10, 13, 16, 36, 172, 173, 518
Group name 10, 36
Group node 10, 13, 14, 36, 37, 40, 172, 173, 511, 517, 518

Hierarchical classification 15, 48
keyword 51
level 48, 51
HIGHEST_CLASSIFICATION 52, 519

Identification 10, 31
by pathname 32
of a node 10, 31
of a relationship 10, 32
Identifier equality 421, 461
Identifier item 513
Identifier text 10, 420, 421
IDENTIFIER_TEXT 420, 425, 426, 457
Identify a node 31
IMPORT_CONTENTS 415
IN_INTENT 215
Inaccessible 11, 36, 65, 71, 81, 82, 99, 108
INDEX 84
Inherit 27
Inheritable 11, 519
Initiate 11
Initiated process 11, 22
Initiating process 11, 22, 168
INOUT_INTENT 215
Input and Output 23
Scope 2
INSERT 453, 466, 475, 484, 493
INSERT_COUNT 425
INSERT_LINE 354
INSERT_SPACE 353
Integer equality 420
INTENT 48, 55, 57, 70, 177
APPEND 58
APPEND_ATTRIBUTES 59
APPEND_CONTENTS 59
APPEND_RELATIONSHIPS 60
CONTROL 60
EXCLUSIVE_APPEND 58
EXCLUSIVE_APPEND_RELATIONSHIPS 60
EXCLUSIVE_APPEND_ATTRIBUTES 59
EXCLUSIVE_APPEND_CONTENTS 59
EXCLUSIVE_CONTROL 60
EXCLUSIVE_READ 58
EXCLUSIVE_READ_ATTRIBUTES 59
EXCLUSIVE_READ_CONTENTS 58
EXCLUSIVE_READ_RELATIONSHIPS 60
EXCLUSIVE_WRITE 58, 96, 188
EXCLUSIVE_WRITE_ATTRIBUTES 59
EXCLUSIVE_WRITE_CONTENTS 58
EXCLUSIVE_WRITE_RELATIONSHIPS 60
EXECUTE 61
NO_ACCESS 36, 58, 82
READ 58
READ_ATTRIBUTES 59
READ_CONTENTS 58
READ_RELATIONSHIPS 60
WRITE 58
WRITE_ATTRIBUTES 59
WRITE_CONTENTS 58
WRITE_RELATIONSHIPS 60
INTENT_ARRAY 54
INTENT_SPECIFICATION 54
INTENT_VIOLATION 55

Downloaded from http://www.everyspec.com

INDEX

INTERCEPTED_INPUT_CHARACTERS 291, 327, 368
INTERCEPTED_OUTPUT_CHARACTERS 292, 328, 369
Interface 11
Internal file 11, 208
Interoperability 1, 11
INVOKE_PROCESS 57, 152, 169, 180, 183, 195
Invoked process 184
IO_UNIT_COUNT 169, 203, 519
IS_APPROVED 159
IS_EQUAL 425, 461
IS_FORM_UPDATED 383
IS_INHERITABLE 104
IS_OBTAINABLE 81
IS_OPEN 69, 289, 325, 366, 397
IS_SAME 82
IS_WRITE_RING_INSTALLED 409
Item name 11, 419, 421, 423
Item value 11, 419, 421
ITEM_KIND 425
ITEM_KIND_ERROR 425
Iterator 8, 11, 12
    See also Attribute Iterator and Node Iterator
ITERATOR_ERROR 55

Job 11, 25, 27, 185, 518

Key 11, 23, 27, 55, 98
Keyword 51
Keyword member 51
Kind 22
KIND_OF_DEVICE 221
KIND_OF_FILE 219
KIND_OF_ITEM 434
KIND_OF_LIST 433
KIND_OF_NODE 71
KIND_OF_QUEUE 220

Labeling 51
Last path element 74, 75, 76, 77, 78, 79, 80, 123
LAST_KEY 80, 114
LAST_RELATION 79, 114
Latest key 11, 29, 55
LATEST_KEY 27, 55
LAYOUT_ERROR 215, 216
Linear list 9, 10, 12, 17, 419, 421, 423
Linear list manipulations 9, 421
List 12, 14, 513, 514, 596
    classifications 419
    equality 423
    item 10, 12, 16, 419, 420, 421, 422, 423, 425, 429
    structure 9
    text representation 423
LIST_KIND 425
LIST_KIND_ERROR 425
LIST_POSITION_ERROR 425, 426
LIST_SIZE 425
LIST_TEXT 422, 425, 426
LIST_TYPE 9, 12, 32, 425
LOAD 400
Lock 58, 66, 87, 94, 96, 111, 188
LOCK_ERROR 55, 94, 197
LONG_DELAY 55
LOWEST_CLASSIFICATION 52, 519

MACHINE_TIME 169, 206, 520
Magnetic tape drive 23, 399
Magnetic tape drive file 12, 212, 390
MAGNETIC_TAPE_DRIVE 519
MAKE_CONTAINING_LIST_CURRENT 441
MAKE_THIS_ITEM_CURRENT 442
Mandatory access control 7, 11, 12, 35, 48, 52, 56, 152
MANDATORY_ACCESS 51, 164, 174, 180, 185, 228, 237, 246, 257, 262, 267, 271, 274, 278
MANDATORY_ACCESS_LIST 54
MAXIMUM_FILE_SIZE 223, 520
MAXIMUM_QUEUE_SIZE 225, 520
Mimic queue 12
    See also Queue, mimic
MIMIC_FILE 253, 262, 282, 518
MODE 311, 347, 398
MODE_ERROR 215, 216
MONTH 501
MONTH_NUMBER 498
MORE 106, 109, 141, 142, 144, 146

Name 12
NAME_ERROR 52, 55, 56
Named item 12, 419
Named list 12, 419, 429, 446
NAMED_LIST_ERROR 425, 426
Nested list structure 12, 421
Nested sublist 12, 421
NEW_LINE 313
NEW_PAGE 314
NEXT_NAME 106, 107, 114, 141, 142, 144, 147, 148
NO_ACCESS 36, 58, 82
Node 10, 12, 22
    creation 87
    deletion 57
    handle 8, 12, 13, 16, 48, 53, 63, 520
    identification 10, 31
    iterator 11, 12, 106, 107
    Management 53
    non-existing 12, 24, 32
    process 29
    system-level 25
    traversal 32
    unobtainable 24
Node kind 12
Node model 1, 19
    Scope 1
NODE_ITERATOR 106
NODE_KIND 54, 519
NODE_KIND_ARRAY 106
NODE_KIND_ERROR 55, 56
NODE_TYPE 12, 53, 54, 66
Non-hierarchical categories 15, 48, 51, 597
Nonsynchronous queue 12
    See also Queue, nonsynchronous
NONSYNCHRONOUS_COPY 520
NONSYNCHRONOUS_MIMIC 520
NONSYNCHRONOUS_SOLO 520
Notes
    definition 21
Null key 12, 29, 31
NUMBER_OF_FUNCTION_KEYS 290, 310, 326, 346, 367, 384
NUMBER_OF_ITEMS 438

Object 13, 35, 36, 40, 42, 48, 51, 159, 174, 180, 185, 517
Object classification 519, 520
Object classification label 52
OBJECT_CLASSIFICATION 51, 520
Obtainable 13, 24
OPEN 57, 63, 65, 66, 68, 85, 118, 167, 231, 240, 249, 281, 287, 323, 364, 390, 395
Open file handle 13, 66, 208, 287, 323, 513
Open node handle 13, 16, 48, 52, 53, 55, 57, 63, 65, 82, 111, 117, 240, 249
Open operation 57

Downloaded from http://www.everyspec.com

OPEN_FILE_HANDLE_COUNT 72
OPEN_NODE_HANDLE_COUNT 169, 202, 520
OPEN_PARENT 57, 85, 86
Operations 48
OUT_INTENT 215
Outermost linear list 437, 440

Page terminal 15, 211, 319
PAGE_SIZE 297, 333
PAGE_TERMINAL 519
PARAMETER_LIST 171
Parameters 520
  definition 20
Parent 13, 16, 24, 172, 173, 518
  relationship 92, 164
Path 13, 29
  element 8, 11, 13, 29, 31, 53, 114, 511
  primary 29
  unique primary 16
PATH_KEY 76, 108
PATH_RELATION 77, 108
Pathname 8, 9, 10, 11, 13, 29, 31, 54, 78, 79, 123, 509, 512, 517, 595
  : -- current process node 31
PATHNAME_LENGTH 514
PATHNAME_SYNTAX_ERROR 55, 56
Portability 1, 2
Position 10, 13, 284, 319, 362, 403
POSITION_BY_NAME 447
POSITION_BY_VALUE 455, 468, 477, 486, 495
POSITION_COUNT 425
POSITION_OF_CURRENT_LIST 439
Potential member 13, 37, 160, 518
POTENTIAL_MEMBER 10, 13, 36, 37, 40, 518
Pragmatics 2, 13, 509, 599
  Scope 2
Precede the active position 284, 295, 319, 362
Predefined access rights
  ALL_RIGHTS 44
  APPEND 44
  APPEND_ATTRIBUTES 43
  APPEND_CONTENTS 44
  APPEND_RELATIONSHIPS 43
  CONTROL 44
  EXECUTE 44
  EXISTENCE 43
  READ 44
  READ_ATTRIBUTES 43
  READ_CONTENTS 43
  READ_RELATIONSHIPS 43
  WRITE 44
  WRITE_ATTRIBUTES 43
  WRITE_CONTENTS 43
  WRITE_RELATIONSHIPS 43
Predefined relations 25, 169
  ACCESS 7, 14, 36, 40, 154, 156, 172, 173, 511, 517, 519
  ADOPTED_ROLE 8, 16, 37, 160, 162, 172, 173, 517, 597
  CURRENT_JOB 9, 25, 27, 172, 173, 517
  CURRENT_NODE 9, 25, 27, 29, 31, 116, 172, 173, 517
  CURRENT_USER 9, 25, 27, 31, 172, 173, 185, 517
  DEFAULT_ROLE 36, 37, 517, 597
  DEVICE 9, 25, 27, 172, 173, 517
  DOT 10, 31, 36, 37, 517, 595
  EXECUTABLE_IMAGE 172, 173, 517
  GROUP 36, 172, 173, 518
  JOB 25, 27, 185, 518
  MIMIC_FILE 518
  PARENT 13, 14, 24, 36, 164, 172, 173, 518
  POTENTIAL_MEMBER 10, 13, 36, 37, 40, 518
  STANDARD_ERROR 169, 172, 173, 518
  STANDARD_INPUT 169, 172, 173, 518
  STANDARD_OUTPUT 169, 172, 173, 518
  USER 17, 25, 27, 172, 173, 518
PREDEFINED_ATTRIBUTE_ERROR 55, 56
PREDEFINED_RELATION_ERROR 55, 56
Primary relationship 13, 14, 17, 24, 89, 92, 94, 95, 96, 164, 189, 510, 518
PRIMARY_KEY 74
PRIMARY_NAME 73
PRIMARY_RELATION 75
PRINTABLE_CHARACTER 363
Process 2, 12, 14, 15, 22, 71, 519
  abortion 94, 168, 169, 178, 180, 184, 196
  Ada tasks 22
  creation 172, 174, 180, 185
  current 29
  dependent 9, 24, 27
  initiated 11, 22
  initiating 11, 22
  initiation 168
  node 14, 22, 29, 94, 171, 172
  root 9, 11, 14, 27, 185, 188
  Scope 2
  termination 15, 168, 169, 172, 178, 180, 184
  tree 9, 11, 14, 27, 168
Process status 181, 184, 194
  ABORTED 181, 519
  READY 174, 180, 181, 185, 519
  SUSPENDED 181, 519
  TERMINATED 94, 181, 519
PROCESS_SIZE 169, 207, 520
PROCESS_STATUS_KIND 171
Program 14, 168
Purpose
  definition 20
PUT 169, 303, 339, 363, 378, 519

Qualified area 8, 14, 362, 373
Queue 14, 22, 23, 519, 520
  copy 9, 253, 257
  file 14, 211, 253, 513, 520
  mimic 9, 12, 253, 262, 518
  nonsynchronous 12, 254, 257, 262, 267, 271
  solo 15, 253
  solo sequential 271, 278
  solo text 267, 274
  synchronous 15, 253, 274, 278
  used for process communication 22, 253
QUEUE_KIND 215, 520

READ 44, 58, 282
READ_ATTRIBUTES 43, 59
READ_BLOCK 411
READ_CONTENTS 43, 58
READ_RELATIONSHIPS 43, 60, 65
READY 174, 180, 185, 519
RECORDING_METHOD 408
Relation 14, 23
  ACCESS 36, 40
  ADOPTED_ROLE 37, 597
  CURRENT_JOB 25, 27
  CURRENT_NODE 25, 27, 29, 31
  CURRENT_USER 25, 27, 31
  DEFAULT_ROLE 36, 37, 597
  DEVICE 9, 25, 27, 172, 173, 517
  DOT 31, 36, 37
  GROUP 36
  JOB 25, 27
  name 10, 13, 14, 23, 29, 31, 32, 75, 77, 98, 106, 114, 164
  PARENT 36, 164

Downloaded from http://www.everyspec.com

INDEX

POTENTIAL_MEMBER 36, 37, 40
predefined 25
USER 25, 27
RELATION_NAME 54
RELATION_NAME_PATTERN 106
Relationship 14, 22, 23
identification 10, 32
key 10, 11, 13, 14, 23, 29, 74, 76, 80, 106, 114, 595
key designator 14, 29, 31, 32, 164
latest key 29
primary 13, 14, 24
reading 35
secondary 14, 24
source node 23
target node 23
RELATIONSHIP_ERROR 55, 56
RELATIONSHIP_KEY 54
RELATIONSHIP_KEY_PATTERN 106
RELATIONSHIP_KIND 106
REMOVE_AREA_QUALIFIER 374
RENAME 24, 92, 93
Renaming a file 92
Renaming nodes 57
REPLACE 451, 464, 473, 482, 491
REQUEST_DISMOUNT 390, 402
REQUEST_MOUNT 399
RESET 233, 242, 251, 315, 358, 413
RESOURCE_ERROR 509
RESULTS 169, 190, 191, 192, 520
RESULTS_LIST 171
RESULTS_STRING 171
Resume 15, 168
RESUME_PROCESS 200
REWIND_TAPE 404
Role 14, 36, 37, 511, 517
Root process node 9, 11, 14, 27, 29, 37, 171, 172, 185, 188, 518
ROOT_PROCESS 171

Scroll terminal 15, 211, 284
SCROLL_TERMINAL 519
SEARCH_ERROR 425, 426
Secondary relationship 14, 24, 89, 95, 98, 100, 101, 174, 180, 185, 510
Secondary storage file 15, 22, 23, 211
SECONDS 503
Security level 15, 52
SECURITY_VIOLATION 52, 55, 56, 177
SELECT_GRAPHIC_RENDITION 356
SELECT_RANGE_KIND 321
Semantic Descriptions 20
additional interfaces 20
exceptions 20
notes 21
parameters 20
purpose 20
SEQUENTIAL 519
Sequential file 512, 513
SET_ACTIVE_POSITION 295, 331, 375
SET_CURRENT_NODE 116
SET_GRANTED_RIGHTS 40, 154
SET_INHERITANCE 102
SET_NODE_ATTRIBUTE 133
SET_PATH_ATTRIBUTE 135, 136
SET_TAB_STOP 298, 334
SET_TO_EMPTY_LIST 428
SKIP_BLOCK 410
SKIP_NEXT 106, 107, 109, 110, 113, 114, 141, 142, 144, 146, 150
SKIP_TAPE_MARK 405
Solo queue 15
See also Queue, solo
SOUND_BELL 302, 338
Source node 13, 14, 15, 16, 23, 24, 518
SPAWN_PROCESS 57, 152, 169, 174, 176, 180, 195
SPLICE 435
SPLIT 504
STANDARD_ERROR 169, 171, 172, 173, 518
STANDARD_INPUT 169, 171, 172, 173, 518
STANDARD_OUTPUT 169, 171, 172, 173, 518
Start position 15, 362, 378, 380, 382
STATUS 392, 407
STATUS_ERROR 55, 56
String equality 420
String item 514
STRUCTURAL 12, 71, 519
Structural node 15, 16, 22, 25, 87, 89, 92, 163, 164
contents 22
Subject 15, 35, 36, 40, 42, 48, 51, 159, 174, 180, 185, 517
Subject classification 520
Subject classification label 52
SUBJECT_CLASSIFICATION 51, 520
Suspend 15, 168, 178, 199
See also Process, suspension
SUSPEND_PROCESS 198, 199
Synchronization 15
SYNCHRONIZATION_IS_ENABLED 318, 361
SYNCHRONIZE 234, 243, 252, 316, 359
Synchronous queue 15
See also Queue, synchronous
SYNCHRONOUS_SOLO 520
SYNTAX_ERROR 55, 56, 425, 426
System-level node 15, 16, 25, 36, 517, 518, 523

TAB 301, 337
TAB_STOP_KIND 286, 321
Tape drive 22
Tape mark 405, 406
TAPE_BLOCK 392
TAPE_DRIVE_STATUS_KIND 392
TAPE_NAME 392
TAPE_POSITION_KIND 392
TAPE_RECORDING_METHOD_KIND 392
TAPE_STATUS_ERROR 394
Target node 12, 13, 14, 15, 16, 17, 23, 24, 36, 40, 71, 87, 89, 92, 107
Task 15, 22, 23
Terminal 22, 23
Terminal file 15, 211
TERMINAL_POSITION_ERROR 215, 216
TERMINAL_POSITION_TYPE 286, 321, 363
TERMINAL_SIZE 381, 387
TERMINATED 94
Termination of a process 15, 168
TERMINATION_KEY 385
TEXT 519
Text file 512
TEXT_FORM 430, 460, 470, 479
TEXT_LENGTH 444
TIME 497, 498
TIME_ATTRIBUTE_WRITTEN 122, 521
TIME_CONTENTS_WRITTEN 121, 521
TIME_CREATED 119, 521
TIME_ERROR 498
TIME_FINISHED 169, 205, 521
TIME_LIMIT 57
TIME_OF 505
TIME_RELATIONSHIP_WRITTEN 120, 521
TIME_STARTED 169, 204, 521
To obtain access 35
Token 16, 420, 421, 458, 459, 460, 461, 462, 513

Downloaded from http://www.everyspec.com

TOKEN_ERROR 420, 425, 426
TOKEN_TYPE 420, 425, 457
Tool 1, 16, 599
  set 1, 8, 16
Top-level node 16, 25, 36
  device node 27, 172
  file node 27, 517
  group node 36, 172, 518
  user node 9, 17, 25, 27, 29, 36, 55, 172, 517, 518
Track 16, 24, 53, 65, 92, 93
Transportability 1, 16, 599
Traversal of a node 16, 32, 35
Traversal of a relationship 16, 23

Unadopt a role 16, 37
UNADOPT_ROLE 37, 162
UNBOUNDED_FILE_SIZE 215
UNBOUNDED_QUEUE_SIZE 215
Undefined token 16, 420
Unique primary path 16, 29, 96
Unique primary pathname 16, 24, 31, 73, 74, 75, 92, 188
UNLOAD 390, 401
Unnamed item 16, 17, 419
Unnamed list 17, 419, 429
Unobtainable 17, 24, 36, 65, 71, 81, 82, 94, 99, 101, 108
UNRESTRICTED 509
USE_ERROR 55, 56
User 16, 17, 25, 27, 172, 173, 518
  name 17, 25
  node 25
Utilities 2
  Scope 2

WRITE 44, 58, 282, 512
WRITE_ATTRIBUTES 43, 59
WRITE_BLOCK 412
WRITE_CONTENTS 43, 58
WRITE_RELATIONSHIPS 43, 60
WRITE_RESULTS 191, 192
WRITE_TAPE_MARK 406

YEAR 500
YEAR_NUMBER 498

Downloaded from http://www.everyspec.com

Preparing Activity:
Air Force - 02

(Project MCCR/IPSC 0208)

Custodians:

Army - CR
Navy - EC
Air Force - 10

Review Activities:

Army - CR, AV
Navy - SH, TD
Air Force - 17, 02
Other - DC

User Activities:

Army -
Navy - OM, Naval Telecommunications Command
Air Force -
Other - DC, DH, Defense Mapping Agency

Agent: Ada Joint Program Office

Downloaded from http://www.everyspec.com

Postscript: Submission of Comments

For submission of comments on DOD-STD-1838, we would appreciate them being sent by ARPANET/MILNET to the address:

CAIS-COMMENT at ADA20.ISI.EDU

If you do not have ARPANET access, please send the comments by mail to:

Ada Joint Program Office
Room 3E114, Pentagon
Washington, DC 20301-3081

For mail comments, it will assist us if you are able to send them on 5 1/4-inch double-sided double-density 360 Kbyte floppy diskette formatted for MS-DOS. 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 four-line header.

! Section ...
! Version DOD-STD-1838
! Topic ...
! Rationale ...

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

! Section 03.02.01(12) A. Gargaro 82-04-26

The version line, for comments on the military standard, should only contain "DOD-STD-1838". 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 information 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 ..."

As a final example here is a complete comment:

! Section 03.02.01(12) A. Gargaro 85-01-15
! Version MIL-STD-CAIS
! Topic FILE NODE MANAGEMENT
Change "component" to "subcomponent" in 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.

Downloaded from http://www.everyspec.com

STANDARDIZATION DOCUMENT IMPROVEMENT PROPOSAL

(See Instructions – Reverse Side)

1. DOCUMENT NUMBER

2. DOCUMENT TITLE

3a. NAME OF SUBMITTING ORGANIZATION

4. TYPE OF ORGANIZATION *(Mark one)*

- ☐ VENDOR
- ☐ USER
- ☐ MANUFACTURER
- ☐ OTHER *(Specify)*: ______

b. ADDRESS *(Street, City, State, ZIP Code)*

5. PROBLEM AREAS

a. Paragraph Number and Wording:

b. Recommended Wording:

c. Reason/Rationale for Recommendation:

6. REMARKS

7a. NAME OF SUBMITTER *(Last, First, MI)* – Optional

b. WORK TELEPHONE NUMBER *(Include Area Code)* – Optional

c. MAILING ADDRESS *(Street, City, State, ZIP Code)* – Optional

8. DATE OF SUBMISSION *(YYMMDD)*

DD FORM 1426, 82 MAR

PREVIOUS EDITION IS OBSOLETE.