SST -115

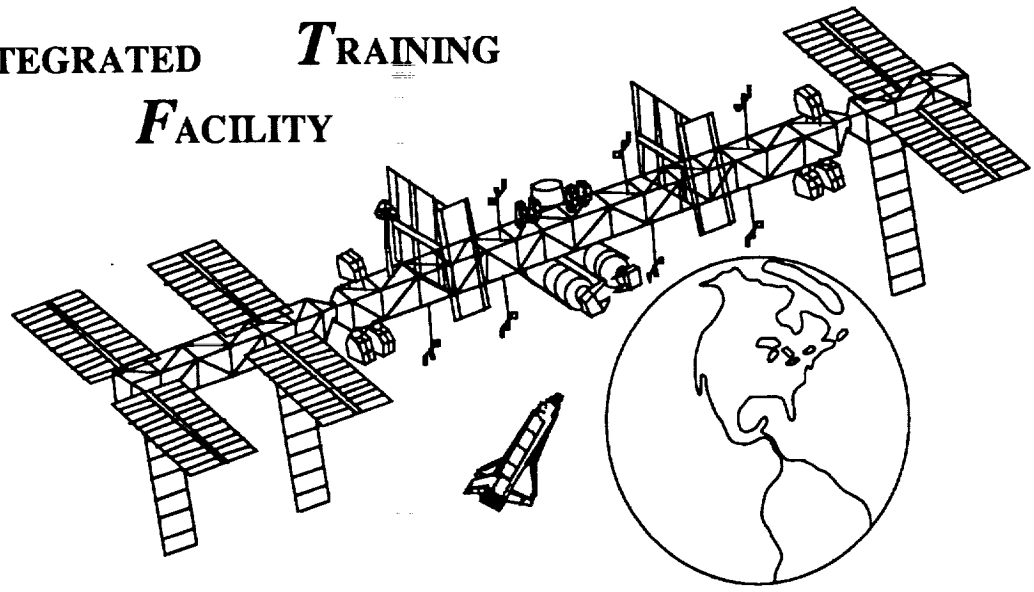
Software Architecture Standard for Simulation Virtual Machine

Version 2.0

NAS9-18181

20 April 1994

***I*NTEGRATED *T*RAINING *F*ACILITY**



Prepared for:

National Aeronautics and Space Administration
Lyndon B. Johnson Space Center
Houston, Texas 77058

Prepared by:

CAE-Link Corporation
Houston Operations
2224 Bay Area Boulevard
Houston, Texas 77058

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	20 April 1994	Final

4. TITLE AND SUBTITLE

Simulation Virtual Machine
Software Architecture Standard

5. FUNDING NUMBERS

C - NAS9-18181

6. AUTHOR(S)

Sturtevant, Robert and William Wessale

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

CAE-Link
2224 Bay Area Blvd.
Houston, TX 77058

8. PERFORMING ORGANIZATION REPORT NUMBERS

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Hill, Ken/DK
Mission Operation Directorate
Lyndon B. Johnson Space Center
National Aeronautics and Space Administration
1200 NASA Road 1
Houston, TX 77058-3696
(713) 244-7250

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

SST-115

11. SUPPLEMENTARY NOTES

SVM is architecture of the simulation executive developed for and used in the Space Station Verification and Training Facility (SSVTF).

12a. DISTRIBUTION/AVAILABILITY STATEMENT

See NASA Handbook NHB 2200.2

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Simulation Virtual Machine (SBM) is an Ada architecture which eases the effort involved in the real-time software maintenance and sustaining engineering. The Software Architecture Standard defines the infrastructure which all the simulation models are built from. SVM was developed for and used in the Space Station Verification and Training Facility.

14. SUBJECT TERMS

Real-time, rate monotonic, Software architecture

15. NUMBER OF PAGES

730

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

1. Abstract 3

2. Definition of Terms 4

3. Overall Architecture 7

4. Real–Time Services 13

4.1. Generic Model (Model Executive Interface) ... 13
4.2. Simulator Moding ... 15
4.2.1 Simulation Set–Up ... 16
4.2.1.1 Register I/O / Set–Up ... 16
4.2.1.2 Create Data ... 16
4.2.2 Initialization ... 16
4.2.2.1 Full IC ... 16
4.2.2.2 State Adjustment ... 17
4.2.3 Self Initialize ... 17
4.2.4 System Initialize ... 18
4.2.5 Freeze ... 18
4.2.5.1 Asset Add ... 18
4.2.5.2 Asset Drop ... 18
4.2.6 Run ... 19
4.2.6.1 Asset Add ... 19
4.2.6.2 Asset Drop ... 19
4.2.6.3 Safestore ... 19
4.2.7 Hold ... 19
4.2.7.1 Datastore ... 19
4.2.7.2 Abort ... 19
4.2.8 Terminate ... 20
4.2.9 Run To Freeze Transition ... 20
4.2.10 States of a Training Session ... 20
4.2.11 States of an Asset ... 21
4.3. The Messaging System ... 23
4.3.1 One–to–Many (Normal) Communication ... 23
4.3.2 Many–to–One Communication ... 25
4.3.3 Remote Communication ... 26
4.3.4 Mailbox communication ... 26
4.3.4.1 Mailbox Reads by Partitions ... 27
4.4. The DIS Concept ... 28
4.4.1 What is the DIS? ... 28
4.4.2 How is the DIS Organized? ... 30
4.4.3 Connecting Terms, Prefixes, and Malfunctions ... 32
4.4.4 Handling Enters, Malfunctions, and Initialization data ... 34
4.4.5 How Will Off–line Tools Use the DIS? ... 36
4.5. Mapping Logical Name to Physical Address: DIS & Symbol Map ... 37
4.6. Datastore/Initialization ... 42
4.6.1 Perform a Datastore ... 42
4.6.2 Initialize to a Datastore ... 42
4.6.3 Partition Requirements ... 42
4.6.4 Datastore Notes ... 42
4.7. Safestore ... 47

4.7.1 Perform a Safestore ... 48
4.7.2 Return to a Safestore ... 48
4.7.3 Partition Requirements ... 48
4.7.4 Safestore Notes ... 48
4.8. interface agentS ... 50
4.8.1 introduction ... 50
4.8.1.1 What is an Asset ?? ... 50
4.8.1.2 What is an Interface Agent ?? ... 51
4.8.2 interface Agent General Notes ... 55
4.8.3 Interface agent for asset with SVM ... 56
4.8.3.1 Simulating Interface ... 56
4.8.3.1.1 Communication ... 56
4.8.3.1.2 Moding ... 56
4.8.3.1.3 Malfunctions ... 56
4.8.3.1.4 User–Requested Data Entry ... 56
4.8.3.2 Effecting Pass–Thru Interface ... 57
4.8.3.2.1 Communication ... 57
4.8.3.2.2 Moding ... 57
4.8.3.2.3 Malfunctions ... 57
4.8.3.2.4 User–Requested Data Entry ... 57
4.8.3.3 Adding Asset ... 57
4.8.3.4 Dropping Asset ... 58
4.8.4 Interface agent for asset without SVM ... 59
4.8.4.1 Simulating Interface ... 59
4.8.4.1.1 Communication ... 59
4.8.4.1.2 Moding ... 59
4.8.4.1.3 Malfunctions ... 59
4.8.4.1.4 User–Requested Data Entry ... 59
4.8.4.2 Effecting Pass–Thru Interface ... 60
4.8.4.2.1 Communication ... 60
4.8.4.2.2 Moding ... 60
4.8.4.2.3 Malfunctions ... 60
4.8.4.2.4 User–Requested Data Entry ... 60
4.8.4.3 Adding Asset ... 61
4.8.4.4 Dropping Asset ... 61
4.9. Asynchronous I/O ... 62

5. Non–Real–Time Section ... 66
5.1. Overall Structure ... 66
5.2. Classes and Instances ... 66
5.3. Inheritance and Composition ... 66
5.4. Operational Components ... 66
5.4.1 Communicating with Other Operational Components/Partitions ... 66
5.4.1.1 File Exchanging ... 67
5.4.1.2 Utilizing the Real–Time Interface ... 67
5.4.1.3 POSIX Interprocess Communication ... 67
5.5. Templates and Guidelines ... 67

6. Bibliography ... 69

7 Appendix I – Ada Structural Templates ... I–1
7.1 Class Template .. I–1

7.2 Class Template With Computed Period .. I-3
7.3 Partition Template .. I-5
7.4 Generic Partition Template . I-12

8. Appendix II – Real Time Interface Packages II-1
8.1. Generic Model . II-1
8.2. Message . II-3
8.3. Mailbox II-14
8.3.1 Enter_Mailbox II-20
8.3.2 Malfunction_Mailbox II-22
8.3.3 Safestore_Mailbox II-23
8.3.4 Mega_Mailbox II-24
8.4. DIS II-26
8.5. SSTF_Defs II-44
8.6. Timer_Services_Class II-55

9. Appendix III – Questions and Answers: III-1
9.1. Ada Structural Components: III-1
9.2. Executive Sequencing and Moding: III-4
9.3. Messaging: III-5
9.4. Generic Partition: III-6
9.5. DIS III-6
9.6. Datastore: III-6
9.7. Interface Agent: III-7

10. Appendix IV – Example Code (non-real-time) IV-1
11 Appendix V – Hydraulic System Example . V-1
11.1 Real World Hydraulic System . V-1
11.1.1 Fluid Pressurization Assembly . V-1
11.1.1.1 Motor . V-1
11.1.1.2 Gear Box . V-1
11.1.1.3 Pump . V-1
11.1.2 Valve . V-1
11.1.3 Accumulator . V-2
11.1.4 Reservoir . V-2
11.1.5 Reservoir Quantity Sensor . V-2
11.1.6 Pressure Sensor . V-2
11.1.7 Distribution System . V-2
11.1.8 Return Lines . V-2
11.2 Specification of the Software System . V-4
11.2.1 External Components . V-4
11.2.1.1 Control Surfaces . V-4
11.2.1.2 Landing Gear . V-4
11.2.1.3 Electrical System . V-4
11.2.1.4 Hydraulic Control Panel . V-4
11.2.1.5 IOS . V-5
11.2.1.6 Malfunctions . V-5
11.2.1.7 Look and Enter Data . V-5
11.2.1.8 Aural Cue . V-6
11.2.2 Internal Components . V-7

11.3 Transition to Design . V-9
11.3.1 Sensor Class . V-9
11.3.2 Reservoir Class . V-9
11.3.3 Drive Unit Class . V-9
11.3.4 Hydraulic Pump Class . V-9
11.3.5 Other Classes V-10
11.4 Class Specification V-11
11.4.1 Attributes V-11
11.4.2 Type Declarations V-11
11.4.3 Modifier Specifications V-12
11.4.3.1 Default Modifiers V-12
11.4.3.2 Update V-12
11.4.3.3 Request_State_Change V-12
11.4.3.4 Create V-12
11.4.4 Selector Specifications V-12
11.4.5 Textual Description V-12
11.5 Class Examples V-13
11.5.1 The Accumulator Class V-13
11.5.2 The Pressure and Quantity Sensor Class V-13
11.6 The Hydraulic System Partition V-14
11.6.1 Hydraulic System Partition Interfaces V-14
11.6.2 Hydraulic_System_Partition Package Specification V-14
11.6.3 Hydraulic_System_Partition Package Body V-14
11.6.3.1 Generic Class Instantiations V-14
11.6.3.2 Local Type Definitions V-14
11.6.3.3 Message Pointers V-14
11.6.3.4 Class Instances V-15
11.6.3.5 Internal Data V-15
11.6.3.6 Creating Thread_Exec V-15

Ada Unit 1 Accumulator_Class Package Specification V-17
Ada Unit 2 Accumulator_Class Package Body V-18
Ada Unit 3 Accumulator_Class.Report_Symbols Separate Procedure V-19
Ada Unit 4 Generic_Sensor_Class Package Specification V-20
Ada Unit 5 Generic_Sensor_Class Package Body V-21
Ada Unit 6 Generic_Sensor_Class.Report_Symbols Separate Procedure V-22
Ada Unit 7 Elec_Motor_Class Package Specification V-23
Ada Unit 8 Elec_Motor_Class Package Body V-24
Ada Unit 9 Elec_Motor_Class.Report_Symbols Separate Procedure V-25
Ada Unit 10 Dc_Motor_Class Package Specification V-26
Ada Unit 11 Dc_Motor_Class Package Body V-27
Ada Unit 12 Dc_Motor_Class.Report_Symbols Separate Procedure V-28
Ada Unit 13 Gear_Box_Class Package Specification V-29
Ada Unit 14 Gear_Box_Class Package Body V-30
Ada Unit 15 Gear_Box_Class.Report_Symbols Separate Procedure V-30
Ada Unit 16 Drive_Unit_Class Package Specification V-32
Ada Unit 17 Drive_Unit_Class Package Body V-33
Ada Unit 18 Drive_Unit_Class.Report_Symbols Separate Procedure V-34
Ada Unit 19 Drive_Unit_Class.Update Separate Procedure V-34
Ada Unit 20 Positive_Displacement_Pump_Class Package Specification V-36
Ada Unit 21 Positive_Displacement_Pump_Class Package Body V-37
Ada Unit 22 Positive_Displacement_Pump_Class.Report_Symbols Separate Procedure V-38
Ada Unit 23 Axial_Piston_Pump_Class Package Specification V-39
Ada Unit 24 Axial_Piston_Pump_Class Package Body V-40
Ada Unit 25 Axial_Piston_Pump_Class.Report_Symbols Separate Procedure V-42
Ada Unit 26 Actuator_Class Package Specification V-43
Ada Unit 27 Actuator_Class Package Body V-44
Ada Unit 28 Actuator_Class.Report_Symbols Separate Procedure V-45
Ada Unit 29 Centrifugal_Pump_Class Package Specification V-47
Ada Unit 30 Centrifugal_Pump_Class Package Body V-48
Ada Unit 31 Centrifugal_Pump_Class.Report_Symbols Separate Procedure V-48
Ada Unit 32 Hydraulic_Pump_Class Package Specification V-50
Ada Unit 33 Hydraulic_Pump_Class Package Body V-52
Ada Unit 34 Hydraulic_Pump_Class.Report_Symbols Separate Procedure V-53
Ada Unit 35 Hydraulic_Pump_Class.Update Separate Procedure V-53
Ada Unit 36 Distribution_System_Class Package Specification V-56
Ada Unit 37 Distribution_System_Class Package Body V-57
Ada Unit 38 Distribution_System_Class.Report_Symbols Separate Procedure V-58
Ada Unit 39 Generic_Reservoir_Class Package Specification V-59
Ada Unit 40 Generic_Reservoir_Class Package Body V-60
Ada Unit 41 Generic_Reservoir_Class.Report_Symbols Separate Procedure V-61
Ada Unit 42 Valve_Class Package Specification V-62
Ada Unit 43 Valve_Class Package Body V-63
Ada Unit 44 Valve_Class.Report_Symbols Separate Procedure V-64
Ada Unit 45 Elec_Sys_Intfc_Defs Package Specification V-65
Ada Unit 46 Hyd_Control_Panel_Intfc_Defs Package Specification V-66
Ada Unit 47 Hyd_Sys_Intfc_Defs Package Specification V-67
Ada Unit 48 Hydraulic_System_Partition Package Specification V-70
Ada Unit 49 Hydraulic_System_Partition Package Body V-70
Ada Unit 50 Hydraulic_System_Partition.Create_Data Separate Procedure V-73

Ada Unit 51 Hydraulic_System_Partition.Hold Separate Procedure V-74
Ada Unit 52 Hydraulic_System_Partition.Initialize_Model Separate Procedure V-74
Ada Unit 53 Hydraulic_System_Partition.Initialize_Outputs Separate Procedure V-74
Ada Unit 54 Hydraulic_System_Partition.Process_Mailbox Separate Procedure V-75
Ada Unit 55 Hydraulic_System_Partition.Register_Io Separate Procedure V-78
Ada Unit 56 Hydraulic_System_Partition.Report_Symbols Separate Procedure V-81
Ada Unit 57 Hydraulic_System_Partition.Run Separate Procedure V-81
Ada Unit 58 Hydraulic_System_Partition.Self_Init Separate Procedure V-81
Ada Unit 59 Hydraulic_System_Partition.Set_Up Separate Procedure V-82
Ada Unit 60 Hydraulic_System_Partition.System_Init Separate Procedure V-84
Ada Unit 61 Hydraulic_System_Partition.Term Separate Procedure V-84
Ada Unit 62 Hydraulic_System_Partition.Update_Hydraulic_System Separate Procedure ... V-85
Ada Unit 63 Hydraulic_System_Partition.Update_Inputs Separate Procedure V-85
Ada Unit 64 Hydraulic_System_Partition.Update_Outputs Separate Procedure V-86
Ada Unit 65 Hydraulic_System_Partition.Update_Press_Components Separate Procedure .. V-88
Ada Unit 66 Hydraulic_System_Partition.Update_Supply_Components Separate Procedure . V-89
Ada Unit 67 Orvc_Common_Types Package Specification V-91
Ada Unit 68 Orvc_Defs Package Specification V-91
Ada Unit 69 Hydraulic_System_Defs Package Specification V-91

1. ABSTRACT

The Space Station Verification and Training Facility (SSVTF) is using an object–oriented design (OOD) methodology for software design, a rate monotonic scheduling (RMS) and message passing system called "Simulation Virtual Machine" (SVM) to support the highly distributed execution environment, and the Ada language to implement most of the software. This architecture document specifies how the Ada language will be used, in general, to support SVM and implement OOD. It will define the Ada structure of "classes", "class instances", "algorithm packages", "partitions", and many other architectural elements of the system. It will give guidance on ways to decompose requirements into the various Ada structural elements. It will show how communication is implemented between objects at different levels of the software design (class instances, partitions). It will also specify how the simulation will model the required real–world space station communication and simulation requirements for specific types of interfaces (i.e., 1553, discretes, interfaces to real and simulation hardware).

This document does not detail the specific design of various models in the simulation – it simply (importantly) defines the infrastructure which all the simulation models are built from. Adhering to the concepts and templates in this document will support a consistent architecture across the program assuring that Ada features are used logically and within reason. This architectural specification will support the development of a quality product through consistent design, early analysis and documentation of the "big picture" requirements. It will also be the common location to document general architectural issues and solutions.

This document can be viewed as a software developer's users guide. The following describes the basic steps in implementing the real–time Ada architecture. Several steps should be done concurrently (1..3,5..7). These steps represent the general flow to implement the architecture – not a cookbook. Iterative and vertical slice development are highly encouraged.

1. Identify solution–space objects and classes via OORA and iterative development.
2. Determine how the class instances will be grouped (composition, inheritance, ASM, partition). Define the rate that the partitions will execute.
3. Identify all external interfaces (input and output) to the partitions.
4. Start implementing classes by copying the templete provided in Appendix I and filling in model–specific details (attributes, names, routines). Class structures do not have to follow the template exactly, but the semantic structure defined by the templates should be maintained.
5. Start implementing partitions by copying the template provided in Appendix I. Supply mode routines for the generic model and message variables for the messaging system.
6. Create interface definition packages owned by the partition. Find / coordinate other partition's interface definitions.
7. Create a "nominal" partition that drives default messages and executes at the required rate. No model code executes in a nominal partition, only the partition shell. Time burners and memory allocators should be defined. This shell will be used by others for unit testing and load analysis. More information and an example of a nominal partition will be provided in future documentation.
8. Develop DIS packages as required. Identify terms for datastore, safestore, IOS look, IOS enter, and IOS malfunctions. Add partition code to register DIS terms.
9. Refine "Process_Mailbox" procedure to handle DIS input terms (in partition body). Note option to process or "stuff" variables.
10. Refine mode routines, interfaces, and other partition / class structures as the design proceeds.

What is provided to developers:

1. Real–time interface packages shown in Appendix II. Developers use these packages to communicate across partitions (Messaging), to execute partitions in real–time (Generic_Model), and to communicate with the IOS and perform datastores (DIS).
2. Class and Partition templates shown in Appendix I. Developers may make a copy of the templates to get a head–start in the implementation.

2. DEFINITION OF TERMS

Abstract Data Type(ADT): Normal implementation of a class. The class exports visible operations in a limited private data type representing the class state in the specification of the package. The body of the class contains the operations. Classes never define variables outside the private type structure (no global data).

Abstract State Machine(ASM): Non–standard implementation of an object, sometimes using generics. Also used to describe partitions and Operational Components. An ASM is an Ada package that exports operations in the specification and defines state in the body.

Ada Main: A standalone procedure that WITHs all partitions that make up a single executable for a single cpu. The Ada main will not perform any processing or sequencing – this is done by the thread executive portion of SVM. The Ada main is only used to bind together a set (any set) of models so that they may be executed.

Algorithm Package: An Ada package that exports functions/procedures that perform simple operations (like transcendental functions). No state data is allowed in this package – all data referenced are formal parameters in the exported routines.

Aperiodic: An execution method (form of a thread executive) defined by the SVM "Ger ic_Model" package that allows aperiodic updates within an RMS base rate. The updates can b ggered by interrupts or other events, and a predetermined max number of events can be handl within the RMS period. All processing must complete within the period time boundaries.

Asset: Any computer node or device on the RTSN such as IOS, SNS, and CSIOP.

Batch: A transaction model that runs in background without any urgency in completion time.

C–Spec: Class Specification. Used in final phases of Object–Oriented Requirements Analysis. This document represents a minimal specification of the requirements in an object–oriented fashion.

Class: "A set of objects that share a common structure and a common behavior. The terms *class* and *type* are usually (but not always) interchangeable." [Booch 91] Classes are modeled as Ada abstract data type packages.

Composition: The creation of a new class by constructing it from other classes.

Datastore: A set of independent data items that is collected on demand and can be returned to models as an initialization point.

DIS (Distributed Identifier Specification): A method and set of Ada packages d structures that associate logical names to physical variables for datastore, safestore. look and enter, and malfunction data. DIS is also used to uniquely identify partitions and partition messages.

Generic Model: An SVM Ada package that provides the real–time execution capability for a partition. Partitions instantiate either a "Periodic" or "Aperiodic" thread executive from the Generic_Model package to enable real–time execution.

GP LAN: A general–purpose local area network for file download and non–time critical network operations between the OSS, IOS, and session computers.

Inheritance: The ability to extend the structure of a class, and possibly it's operations, to create a more specialized component. It differs from composition because inheritance always results in a more specialized version of it's parent class, whereas composition provides a more generalized abstraction.

Instance: "Something you can do things to. An instance has state, behavior, and identity. The structure and behavior of similar instances are defined in their common class. The terms instance and object are interchangeable." [Booch 91] An Instance is the object created from an ADT class package.

Interface Agent: A partition that provides simulation or pass–thru for an asset. Asset add/drop and asset management are also supported.

Interface_Defn: A "type" package that contains the definitions of messages output by a partition. It contains no executable code. Use of this package enforces type checking and interface control between partitions.

Mailbox Message: A form of command and control, non–real–world interface message transmission on the software backplane. This form is used primarily by the IOS. Mailbox messages are free–form, non–typed binary messages (unpacker must understand algorithm of packer).

Many–to–One Message: A form of message transmission on the software backplane where a partition defines the message structure for a message that it will receive from other partitions (in its interface definition package). Many other partitions use the message definition to send message to the single partition. Messages are queued. This is a special case messaging method to support partitions who receive many identical messages from different senders.

Message Package: A SVM Ada package that is the interface to the software backplane for a modeler's partition. It provides services to register and attach input/output messages, and it provides put / get operations for partitions.

Model: A general term to describe a simulation software model such as propulsion, orbiter, and inertial sensor assembly. Models are codified into 1 or more Ada partitions.

Moding: Distinct modes which all real–time models operate in. Modes include freeze, run, hold, initialize, etc.

Nominal Partition: A partition shell for an actual model which executes a null procedure and reads / sends default messages at the desired rate. A time burner and memory allocator are implemented for timing/sizing analysis (activation is optional). No class structures or model–specific code is implemented. Basically, it is a shell that includes all the SVM hooks that are specific to the actual model. It is also used for by other modelers to provide active stubs of external partitions for unit testing.

Node: A single computer assembly containing several cpus connected through shared memory and a system bus.

Object: See instance.

Object–Oriented Design: The design process whereby the software architecture is organized around meaningful objects, rather than functions.

One–to–Many Message: A form of message transmission where a partition sends a message and any number of partitions may receive the message. This is the primary method of sending messages on SSVTF.

Operational Component: Largest unit of documentation in the O–Spec. The approximate real–time equivalent of the Operational Component is the Partition.

O–Spec: Abbreviation for "Object Specification". This is the document that describes a CI in terms of an object oriented perspective.

Partition: A self–contained code unit encompassing a single thread executive. It is an ASM that exports nothing in the package specification. It internally holds instances of classes and iterates them correctly. Internally, it uses Ada parameters to pass data between class instances. Externally, partitions use an SVM message scheme to communicate. Documented as an Operational Component. Note that 1 or more partitions may represent a single documented operational component and vise–versa. The general code size of an Partition will be from 5 to 20KSLOC.

Periodic: An execution method (form of a thread executive) defined by the SVM "Generic_Model" package that provides periodic updates at a specified hertz rate. All processing must complete within the period time boundaries.

Rate–Monotonic Scheduling (RMS): A non–frame–based scheduling approach where models execute at a periodic rate for a specified worst case time. Each model runs independently – RMS algorithms assure models will meet their iteration rates.

Real_Time Model: An application model that simulates a real–world structure, assembly, or function and iterates over a pre–defined time interval at a specified rate. The model's effects appear to be running "in normal human–perceivable time" – not faster than normal, not batch. Real–time also includes potentially fast processing to simulate missing hardware boxes – other real hardware would not know the difference.

RT LAN (RTSN): A real–time local area network (FDDI) for high–speed network communication between assets for a training session.

Safestore: A set of time–dependent data sent by models and captured during RUN at specified intervals. The data is used to recover to the safestore time if required.

Selector: A function in a class ADT that returns an attribute value (state variable) of the class.

Session: The main computer(s) and simulation program that run the simulation. The IOS, SNS, OSS, and CSIOP computers are not part of the session computers.

Simulation Virtual Machine (SVM): The SSVTF executive structure that provides an RMS–based executive and a messaging system for the distributed operating environment

Software Backplane: A term used to describe all the SVM software components that are involved in the transmission of messages between partitions, cpus, nodes, and assets. It is a passive structure that "wires together" the partitions and provides communication capabilities. The backplane provides several message transmission methods (1–to–many, many–to–1, mailbox) and time–consistent data transfer for the entire SSVTF.

Simulation State: The state of the simulation (not mode). There are 3 states pre–session, active, post–session. In these states, nodes are:

pre–session – loaded, connected, waiting for something to do (asset)
session active – part of session (asset)
post–session – disconnected from session (asset)

A nucleus is a training session with at least the RTSC with an IOS and optional Data Management System (DMS) string.

State: Any persistent data defined by a class or partition. State is defined by a class's private type and exists in the instance of the class. State may also exist in the partition's body. Messages between instances or partitions are reflections of the state, not the state itself (no global data). State is modified by normal iteration of the model or by "request state change" calls to modify state (such as insertion of malfunctions).

Thread Executive: A SVM component that gets created when a partition instantiates the "Generic_Model" packages. This component sequences the partition's mode routines at the appropriate time. It is the thread of control of a partition. There should be only one created per partition.

Transaction Model: Non–periodic, event driven processing that spans indeterminate time spans. The model may need to run quickly to emulate real–time data streams, but it is not periodic.

Vertical Slice: A code implementation where a developer implements a narrow design slice from top (partition / interfaces) to bottom (class structures / instances) to prove out the design concept (structure, timing, overhead, algorithm organization / implementation, etc.)

3. OVERALL ARCHITECTURE

The SSVTF Ada software architecture must support a general distributed hardware environment. Figure 3–1 shows the general SSVTF hardware architecture with two session computers and the various non–session assets connected via the RT LAN (the CSIOPs, IOSs, Visual, etc) and the nodes on the GPLAN (OSS, IOS). Also shown are the multiple cpus per node and multiple nodes per session computer system. Cpus communicate in local memory, nodes communicate via reflective memory, and other assets communicate on the RT LAN.

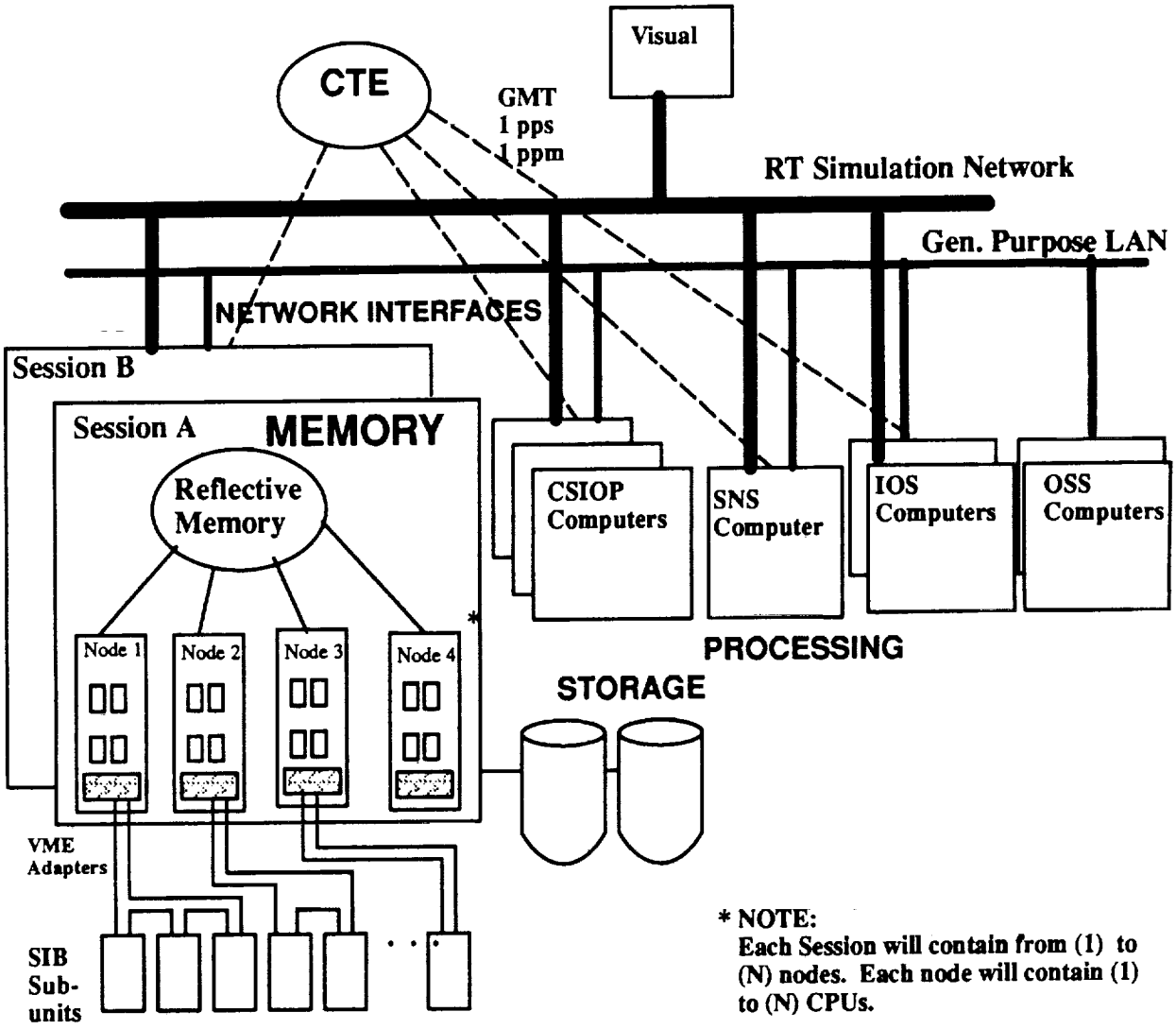


Figure 3–1

Each SSVTF software model is decomposed in an object–oriented fashion based on real–world structures and assemblies. Object–oriented means that data and the data's associated operations are grouped into "class" structures. A class structure encapsulates the hidden portion of the object's attributes and operations

and exports the data type that abstractly represents the object and the valid operations. The class structure on SSVTF is implemented as an Ada abstract data type (ADT) in the form shown in Appendix I, 7.1. An "object" is created when an "instance" of the class abstract type is declared. The class should represent real–world "objects" to the greatest extent possible. Classes/objects are initially defined during the Object–Oriented Requirements Analysis (OORA) phase.

Class structures may be made up of other classes by declaring instances of lower–level classes in the object–attribute record of the higher–level class. If the higher level class represents a less abstract form of the lower–level class, then this structure is defined as "inheritance". If the higher–level class represents an assembly where the lower–level classes are sub–parts of the higher level class, then the structure is called a "composition". In most cases, composition structures will be used on SSVTF. The depth of the hierarchy of classes is dependent on the particular model – one to three levels are common.

At some point in the hierarchy of classes, something must define instances of the highest–level classes. There are three possibilities in Ada – an Ada main program, a task, or an abstract state machine (ASM) package. On SSVTF, the top–level Ada architectural decomposition structure for a model will be an abstract state machine (ASM) package called a "partition" (template in Appendix I, 7.3). The partition performs two types of functionality – (1) defines the state of a model and sequences the model over time; (2) connects the model to the real–time distributed system interfaces.

The state of a model partition will be located in the package body of the partition. It will primarily consist of instances of classes. Since the partition is an object itself (and an ASM), it may also contain non–class related variables defined in the partition body. This data is either "temporary" data required for transformations of external data into data forms required by the classes, or it is real state data that persists cycle to cycle. In general however, class instances should contain the state of the model, not the partition. The partition defines the instances and connects and iterates the instances of the class structures. Instances of classes are connected via procedure calls and parameters.

The real–time system interfaces include a generic thread executive that provides a periodic RMS task to cycle the model (partition) at a given rate, a messaging system that allows partitions to communicate in a distributed environment, and the Distributed Identifier Specification (DIS) which provides the association of logical names to physical data variables for IOS display/manipulation and for datastore/safestore.

The real–time system services provide a virtual machine on which models (partitions) execute. These services support a distributed Ada environment. Fig. 3–2 shows the topology of the system with respect to models. Each model exists in a self–contained structure denoted as the partition. Externally, partitions interface through the software backplane via the package "Message". The backplane provides the messaging capability on the multi–cpu, multi–node distributed system. The backplane also allows the partitions to be very decoupled. The interfaces (messages) between partitions are defined by the "Interface_Defn" Ada type packages shown. These packages contain records defining the format of messages sent between partitions. This structure allows the Ada compiler to verify that interfaces have no inconsistencies. The messages are therefore defined using normal Ada constructs and then sent as messages via the software backplane to other partitions. At the bottom of the figure, the "DIS" (distributed identifier specification) is used to map logical names to physical variables for the purpose of IOS display and datastores.

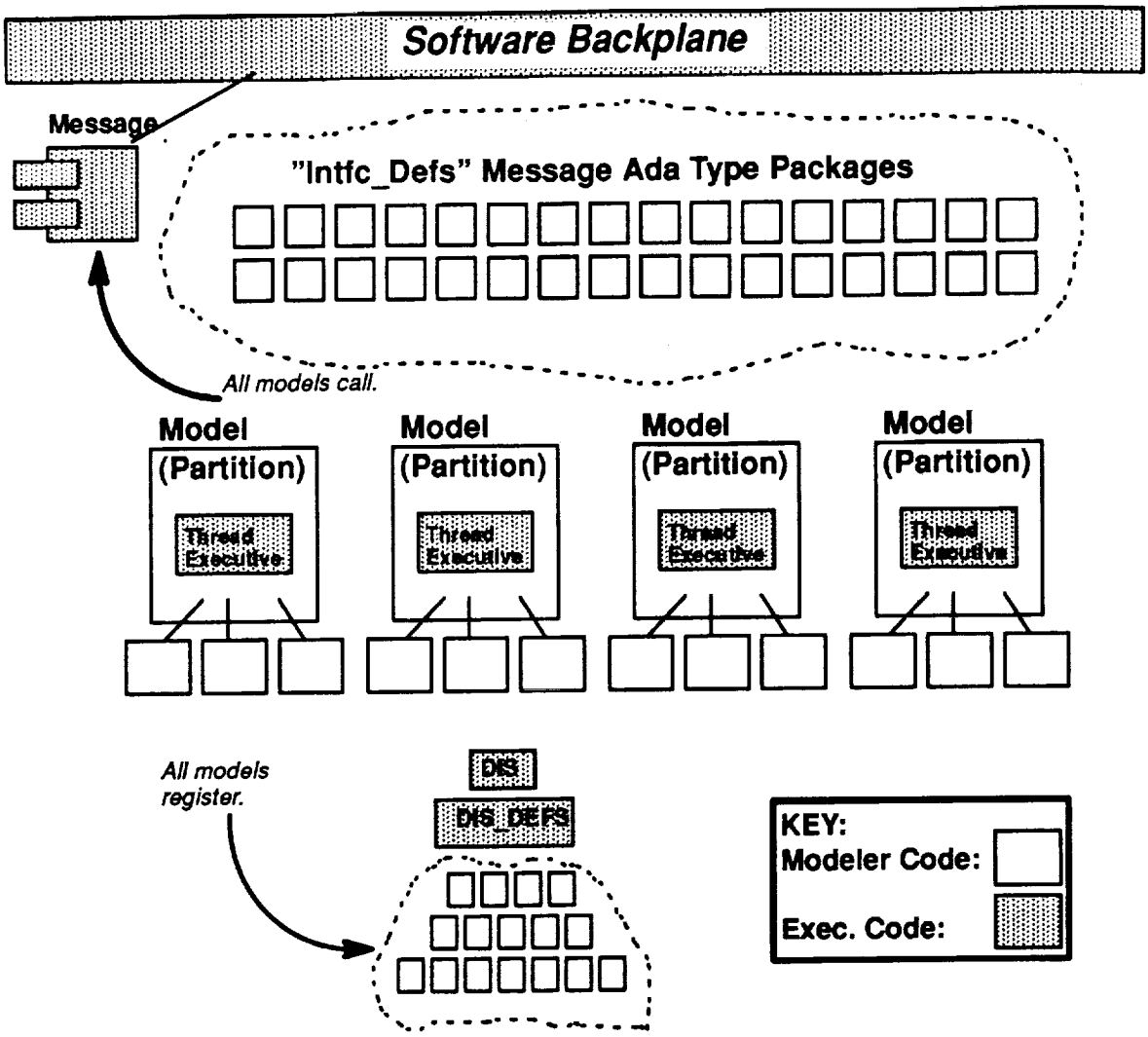


FIGURE 3–2

The following discussion explains the various parts of a partition. Reference the code templates in Appendix I for specific detail on code structures and actual implementation.

Partition Structure:

Figure 3–3 shows the various structures related to a partition. The large box labeled "Partition" represents an Ada package ASM. In the body of the package (hidden from external view) are the instances of classes (objects), local variables, and local subprograms. At the bottom of the page are the "class" packages that are used internally by the partition. The code template for class packages is shown in the "Class Template" section of this document. On the lower right side of the partition box, the "Thread_Exec" is shown. This is the SVM distributed executive that is an instantiation of the "periodic" package defined by the "Generic_Model" package (see appendix II). The partition supplies the mode routines during the instantiation. The thread exec executes the mode routines at the appropriate times. The two "Intfc_Defs" packages at the top are Ada type packages that define the messages that are produced by partitions. "External_Intfc_Defs" defines the messages of another external partition, and "Partition_Intfc_Defs" defines the messages owned by this parti-

tion. By WITHing in interface definition packages, a partition gains the type–checking features of Ada and the exact specification of the interface messages. They can be thought of as mini–interface control documents between partitions. The interface definition packages contain no executable code – only type structures. Inside the partition body, variables are declared using the interface definition package and the "Message" package. These variables are used to send and receive messages. The code template for this set of modules is located in the "Partition Template" section of this document.

Internal Partition Object Communication:

Within a partition, normal Ada language constructs are used to attach, iterate, and communicate data between class instances. Associations (message passing) between classes are done in a vertical fashion. Class structures themselves do not laterally invoke routines of other class structures primarily because the instances of the classes are not "known" by the classes themselves. A higher order module must create the instance and provide the associations between the instances. The class structure does not "know" or have access to instances of other class structures, so a class calling another class's exported routines is rendered impossible by the imposed structure (ref. class template). Note that this does not apply to class compositions or inheritance structures. In compositions and inheritance, the instance of a superclass is defined within the state definition type of the subclass. A call to the subclass can then update the superclass instance.

As shown in figure 3–3, instances of class structures are declared in the body of the partition. The partition provides the mode routines that iterate the instances of the classes.

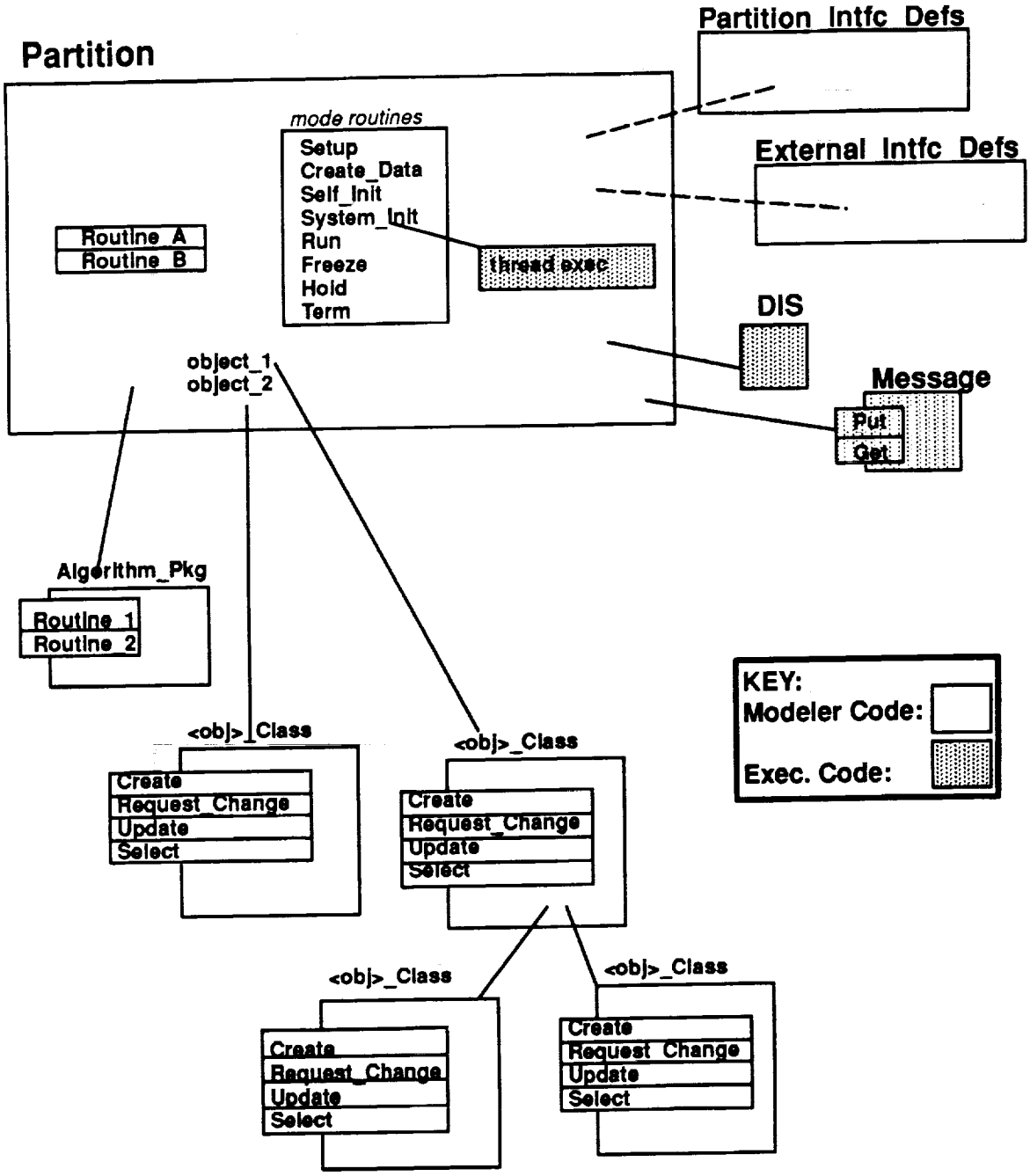


FIGURE 3–3

Figure 3–4 shows the local area network, interface agents representing the LAN nodes, and the SVM parts. Interface agents simulate LAN nodes when the node is not active or they pass data through from the LAN interface to the other models if the node is active. They will be discussed later in this document.

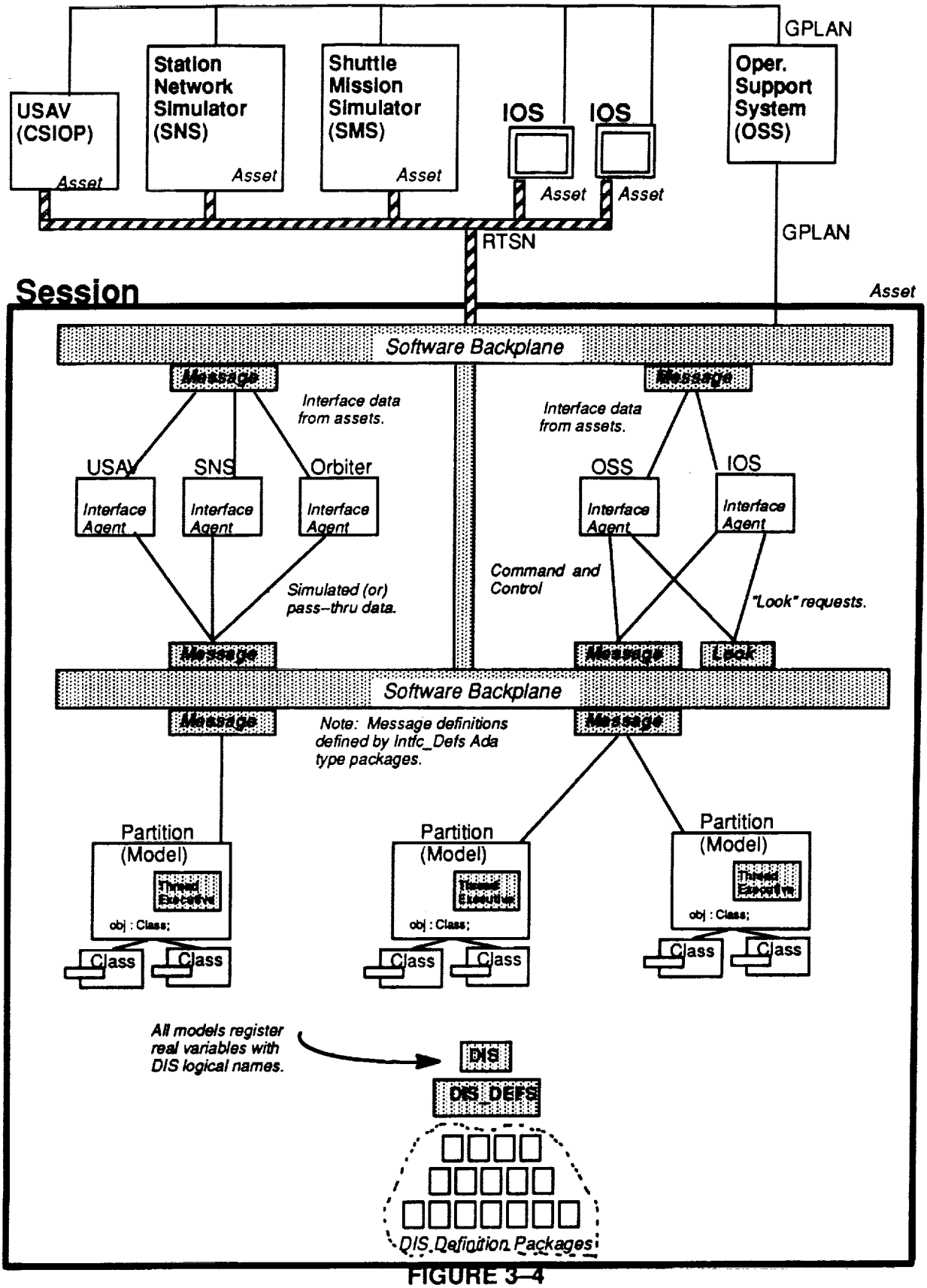

GPLAN
USAV (CSIOP)
Asset
Station Network Simulator (SNS)
Asset
Shuttle Mission Simulator (SMS)
Asset
IOS
IOS
Asset
Asset
Oper. Support System (OSS)
GPLAN
RTSN
Session
Asset
Software Backplane
Message
Interface data from assets.
Message
Interface data from assets.
USAV
SNS
Orbiter
OSS
IOS
Interface Agent
Interface Agent
Interface Agent
Interface Agent
Interface Agent
Command and Control
Simulated (or) pass-thru data.
"Look" requests.
Message
Message
Lock
Software Backplane
Message
Message
Note: Message definitions defined by Intfc_Defs Ada type packages.
Partition (Model)
Thread Executive
obj : Class;
Class
Class
Partition (Model)
Thread Executive
obj : Class;
Class
Class
Partition (Model)
Thread Executive
obj : Class;
Class
Class
All models register real variables with DIS logical names.
DIS
DIS_DEFS
DIS Definition Packages

FIGURE 3-4

4. REAL–TIME SERVICES

The real–time services include the following:

1. Moding and Control
2. RMS–based scheduling (thread executive)
3. Simulation Clock
4. Messaging System (1–to–Many, Many–to–1, Mailbox)
5. Distributed Identifier System (DIS) (for IOS and Datastore/Safestore Variables)
6. Datastore and Safestore Operations
7. Device Drivers
8. Architectural constraints (partition, messaging, DIS, interface agent)

Executive, moding, the messaging system, DIS, datastore/safestore, and interface agents are discussed below.

4.1. Generic Model (Model Executive Interface)

In order to execute a model in real–time, the model partition must use the SVM package "Generic_Model" to obtain the real–time scheduling services. This package specification is shown in Appendix II, and its use is shown in Appendix I under the section "Partition Template". Figure 4.1–1 illustrates the executive software. The Generic_Model contains two generic subpackages "Periodic" and "Aperiodic". Both are RMS scheduled which implies the allocated CPU time for the model is based on the period time and period rate. Partitions must run within their max period time otherwise period overruns will occur and simulation will be stopped.

The rates supported in the "Generic_Model" are described in the "Periodic_Type" and "Aperiodic_Type" enumeration values. SVM is not limited to these rates, but the rates being supported are shown here (if other rates are needed, they can be added). Note that whole (non–fractional) hertz rates are used since it is desired to have a repeating major cycle every 1 or 2 seconds. Fractional hertz rates would complicate mode transitions since the entire system must wait until the end of a major period when changing modes. Rates supported by SVM are to facilitate modeling the real world or to support interfaces with real world components in the simulator, and in following the guidelines of RMS, do not have to be harmonic. When data is shared between models executing at different rates that are not harmonic, the data consumed will appear to be

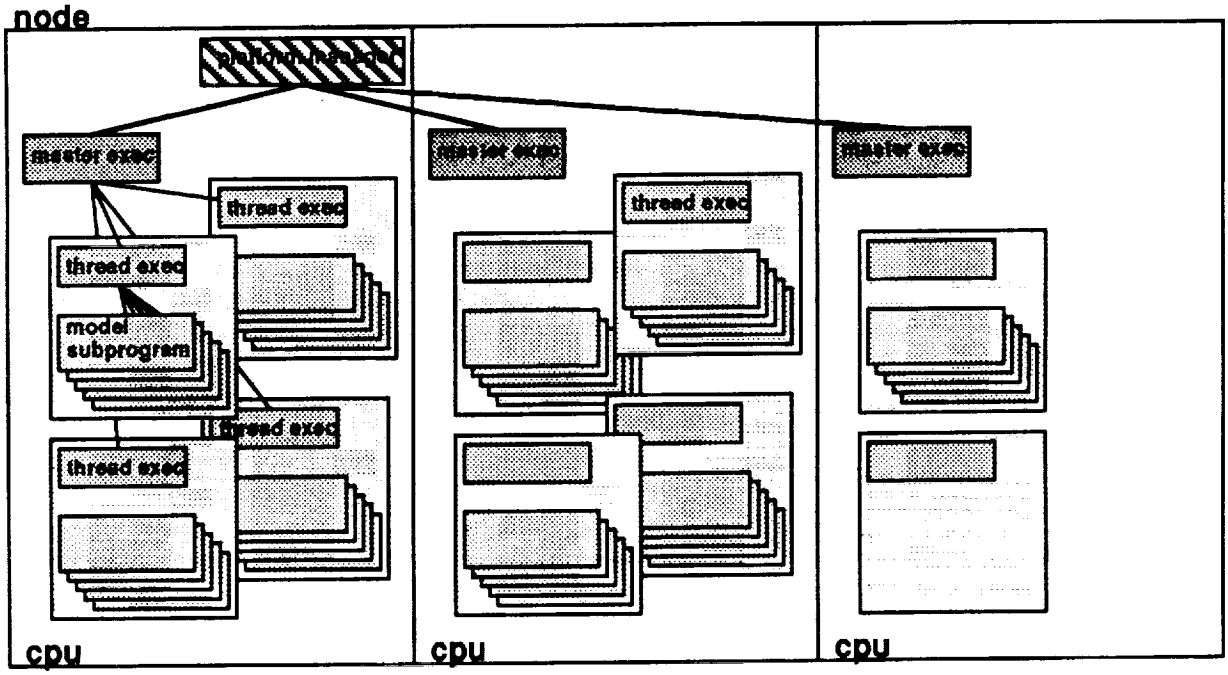


FIGURE 4.1–1 Executive Software

produced in irregular and disproportionate intervals. The modeler should be aware of the relationship between the producing and consuming partitions when choosing execution rates.

Instantiation of the "Periodic" package results in a thread executive for the partition that runs the partition's mode routines cyclically at the requested period (expressed in hertz). The modeler supplies mode routines in the partition body and uses them to instantiate the thread executive package. The mode routines include setup, create_data, self_init, system_init, run, freeze, hold, and terminate. The mode routines are explained in section 4.2. The modeler also provides the required rate and the name of the partition during the instantiation. The name is used to identify the partition if problems are detected. An optional parameter is available to specify the partition task's stack size; a larger stack is necessary to correct *Storage_Errors* for memory–intensive computations. The number of DIS terms that are anticipated to be retrieved from the partition is specified in the parameter, Max_DIS_Terms. This is used to distribute partition processing across execution frames. During execution, the thread executive will call the various mode routines, then process requests for retrieving the DIS term values.

The generic Thread Exec package contains subprograms which can be called to obtain characteristics of the instantiated executive software. Two functions, "Delta_Time" and "Rate_Of_Execution", provide the modeler with information concerning the characteristics of the thread exec. "Delta_Time" is exported by the instantiated Periodic package and provides time representing the interval time in seconds of the period (10 hz = 0.100 seconds in all modes except in freeze when 10hz = 0.0). This time should be used when updating the model and calculating integration 'constants.' "Rate_Of_Execution" returns the execution rate; the same as the generic parameter supplied at the time of instantiation. This function is to be used when supplying information to the software backplane. The function "A_Full_Ic_Is_Required" provides information concerning the type of system initialization conducted (refer to section 4.2.2). A call to this function is made from the self–init procedure. "Ready_To_Transition" is called by the partition when it completes certain mode transitions (refer to sections 4.2.1 through 4.2.4). This signals the master executive that the partition is ready to change mode if commanded. The procedure has an optional parameter that allows the Self_Init procedure to continue cycling when it is set to *true*. The two functions G_M_T and S_G_M_T return GMT time and SGMT time. The time returned is relative to the period of the partition (if the partition runs at 10 hz, GMT will tick in a 100ms interval). GMT or SGMT should only be used if required – models should use Delta_Time for propagating state.

Instantiation of the "Aperiodic" package results in the creation of a thread executive for the partition that runs the partition's mode routines in a periodic time reference but activated on an event. The generic formal parameters are similar to the "Periodic" package with the addition of "Iterations" and "Vector". "Iterations" defines the maximum number of times the aperiodic scheduler may run in a given period, and "vector" is the method to attach an interrupt or event to the aperiodic scheduler. This scheduling method is still RMS–based which means that a worst case time per period and period rate are required. Worst case time is computed as the period time per iteration times the number of iterations allowed. The partition must honor the RMS periodic scheduling time intervals (it cannot run as a transaction model).

Instantiation of the "Asynchronous" package creates a thread executive for the partitions that are non–rate based; these partitions execute in a CPU dedicated to asynchronous activity within an asset. These partitions run only when needed, to support the real-time simulation. The generic formal parameters are similar to the "Periodic" package with the substitution of "Delay_Time" for "Rate". "Delay_Time" is the amount of time to wait before the partition is allowed to execute again. This scheduling method is **not** RMS–based; all Asynchronous partitions will run at the same priority and execute when CPU time is available. These partitions will not execute in synch with the Periodic partitions. Typical partitions of this type include those buffering real-time data for collection/display, and those reading or writing to disk.

Package "Clock" is renamed and USEd in the thread executive so that the partition can have access to all the binary operations on "Time" in simulation clock without having to WITH Simulation_Clock in the partition. Time retrieved from the Periodic functions for GMT and SGMT will reflect time at the start of the partition's period. Aperiodic partition time reflects the start of the last period that has started. No accurate time can be guaranteed Asynchronous partitions, so the function is not available. Time is available from a message broadcast by SVM on the software backplane (20hz resolution). This time can be used for low–fidelity time requirements (since there will be inherent delays from the time the sender generates the time and the receiver reads it).

4.2. Simulator Moding

The various software modes that will be used in the SSVTF are described below. Following this discussion is a pictorial representation of mode transitions (figure 4.2–1). Note the shaded area of the diagram represents modes in which partitions execute in a one–pass manner and overruns are not detected. In these modes, partitions will not be called repeatedly by RTSSW software in order to complete their processing. Partitions have as much time as needed to complete processing and are therefore not considered executing in "realtime". (See section 7.3 for templates of the mode procedures).

Included at the end of this section is a discussion detailing the various States of the Training Session, and the States of an Asset. Figures 4.2.10–1 and 4.2.11–1 pictorially represent these transitions.

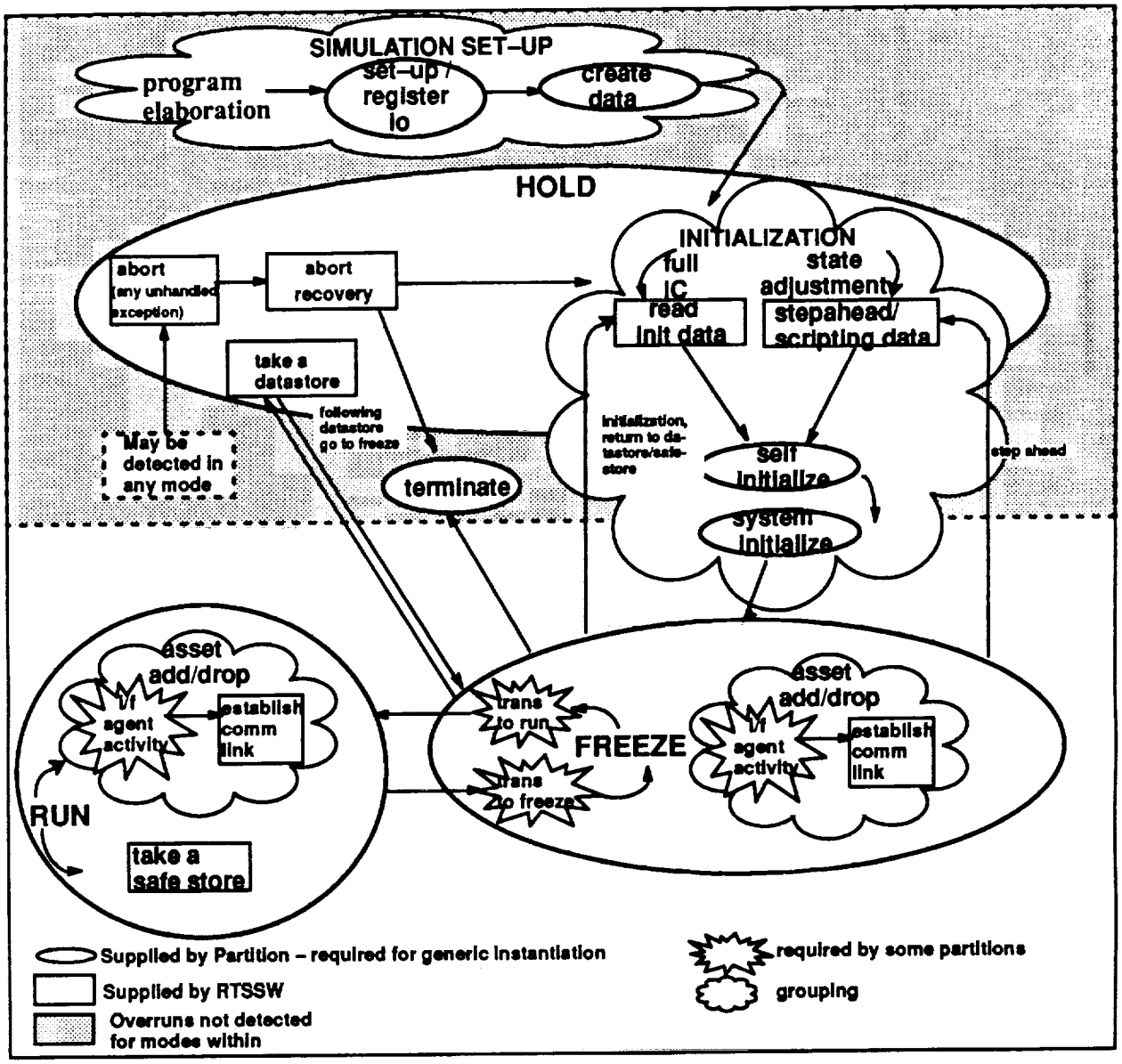


FIGURE 4.2–1 Mode Transition

4.2.1 Simulation Set–Up

4.2.1.1 Register I/O / Set–Up

- Set–Up
  - •• Partitions create objects (class instances) as required
  - •• Partitions connect addresses to DIS term identifiers and 'prefix' information to component identifiers (see section 4.4 for details).
  - •• Partitions connect Dis identifiers to symbol name(s) for each term to be displayed by IOS by calling Dis.Connect_Term.
- Register I/O
  - •• Required by partitions needing to communicate with other partitions and receive mailbox messages
  - •• Partitions register or identify their input and output messages with the RTSSW communication software. This allows the communication routing tables, which ar necessary for the communication to take place, to be set up.
  - •• Mailbox creation is also performed in this routine for Partitions requiring mailbox communication.

4.2.1.2 Create Data

Create_Data is the second phase of the partition to partition communication set up.

- Partitions provide the necessary information to the RTSSW communication software in order to create their communication (message) buffers.
- Partitions must then initialize these buffers (Only One_To_Many output messages) by setting all output messages to default values.
  - •• This activity sets up the message buffers used by the RTSSW communication software in order to pass messages between partitions.
- Platform Manager partitions populate and send a registration message to the Training Session Manager.
  - •• Included in the message is the platform's worst case transition time for the run to freeze and freeze to run transitions.

4.2.2 Initialization

There are two forms of initialization, a "full Initial Condition (IC) reset" or a "state adjustment". The full IC form of initialization occurs when an initialization point, return to datastore or return to safestore is requested from the IOS. Also, the automatic initialization that occurs following the Start–up phase is considered a full IC. Initialization that occurs following a request to perform a step–ahead is considered the state adjustment form of initialization.

4.2.2.1 Full IC

The purpose of the Full IC initialization is to allow the simulation to be reset to a new starting point. For example, if a return to datastore or new initialization point is requested by the IOS, the simulation transitions into a HOLD mode in which the partition's execution is temporarily halted. The following steps are taken in order to start over.

- Once Initialization is entered, RTSSW reads the initialization values from disk and loads them into the various partition mailboxes. Read Init Data phase is now complete.
- The simulation automatically enters Self_Init.

- Partitions reset their internal state to default values, (predetermined safe starting values).
- Partitions will read their mailboxes and set their internal state to the values supplied and/or ramp their models to the desired state. Self–Init is now complete.

4.2.2.2 State Adjustment

The State Adjustment initialization is used to perform a system Step Ahead. The Environment partition will receive a point in time in which to step ahead, while other Paritions receive new values in which to set their internal states. The key difference between this initialization and Full IC is that the internal state is not reset to default values. The new internal state values, provided through mailbox messages are simply applied to the existing internal state. The following steps are taken to perform a State Adjustment.

- RTSSW is notified by the IOS to perform a Step Ahead. This causes the simulation to transition into the Initialization instructor mode.
- Because this is a Step Ahead, the Instructor is involved and is responsible for providing the Step Ahead time as well as any state change values that may be applied to various partitions.
- The IOS is requested to send all state change data to the various partitions. IOS notifies RTSSW when complete. The Step–Ahead/Scripting Data phase is now complete.
- The simulation now enters Self_Init.
- Partitions read their mailbox messages and will perform whatever tasks they are instructed. Mailbox messages may include information pertaining to the time to step–ahead, and/or new state data information.
  - Partitions needing to perform a step ahead will provide a routine to STEP to the new point in time inside their Self_Init procedure. Step ahead may not be completed in a single pass in which case the partition would be responsible for controlling it's internal execution until the desired point in time is reached. Partitions will execute until completed and are not considered running iteratively.
- Self–Init is now complete.

4.2.3 Self Initialize

- A function will be called in the self–init procedure that will identify if this will be an IC reset or state adjustment self–initialize. The logic of the Partition's self_init procedure must use this to determine how to process the mode request. See section 4.2.2 for more information regarding the different initialization types.
- Partitions may need to read their input data (messages) prior to execution in this mode. If so they are doing so at their own risk. This may be old or inconsistent data for what they are trying to do in this mode.
- Full IC reset – the Partition's internal state is cleared to some predetermined starting state.
  - Partition reads its mailbox for new internal state values and applies them to the internal state
- State Adjustment – a Step Ahead was requested by the IOS
  - Partitions will extract receive the target step ahead time via the generic model "S_G_M_T" function call
  - Partitions may also receive some state change information via the mailbox

- •• Env will advance to the target time and then apply any necessary state change information to the internal state
- •• Other partitions may receive only state change information – and will apply it to the internal state

- Self Init will remain a one pass procedure. Partitions that need to iterate will do so by executing until complete. Each partition will notify RTSSW when complete by calling the Ready_To_Transition procedure.

4.2.4 System Initialize

During System Initialize, partitions initialize with each other (both within an asset and between assets in a session) via their System_Init procedures.

- Partitions use the messaging system in order to pass data to other Partitions allowing values to be ramped and achieving a steady state for the simulation.
- Ready_To_Transition is called by the Partition when it has determined it's internal state is steady and at the appropriate values in order to begin the simulation.

When all Partitions have successfully initialized, the session will automatically transition to freeze mode. Note that System_Init is an iterative procedure running at the rate of the partition, i.e. delta time is equal to the partition's period time. The RTSSW executive software will detect overruns in this mode.

4.2.5 Freeze

Freeze mode is an iterative procedure in which RTSSW will detect overruns.

- RTSSW sets delta time to 0.
- Class structures should be able to run with a delta time equal to zero or greater.
- Partitions will execute a procedure that takes the delta time change into account. Two methods to accomplish this:
  - •• Partitions may use their existing RUN procedure if it is able to take into account the reset of delta time.
  - •• Partitions must supply a unique FREEZE procedure if special processing must be performed due to delta time being set to zero.

Messages will continue to be sent, received, and responded to by the partitions. Malfunctions will be held at the IOS until the Freeze mode is complete.

4.2.5.1 Asset Add

Prior to attempting to add an asset, the asset will have completed the PROGRAM ELABORATION, SETUP/REGISTER I/O, and CREATE DATA steps. An asset may be added while the session is in Freeze, or Run mode.

When adding an asset during run mode, one–way communication is established with the asset prior to passing data. Data is then passed to the asset so that it can initialize itself with the ongoing simulation. When everything is synchronized and it is time to join the asset to the simulation, the communication becomes two-way and the interface agent acts as a pass–through for the data transfer. The same basic steps apply when adding an asset during freeze mode; however, a system initialization may take place after communication is established. After all partitions have completed system initialization and checked in, the simulation will automatically transition to freeze mode. Refer to section ( ) for more information regarding Interface Agents.

4.2.5.2 Asset Drop

An asset can be dropped in Freeze, Run, or Terminate mode. The interface agents are the only Partitions with activity in this phase. They will receive the Drop command from the training session mode manager and

cease communication with the asset. The interface agent is now responsible for simulating the asset's outputs rather than acting as a pass through for the asset.

4.2.6 Run

During run, partitions iterate with their period time equal to delta time via their Run procedure. Messages are sent and received. Malfunctions and other commands will be entered and processed. Note that Run is an iterative procedure. RTSSW will detect overruns.

4.2.6.1 Asset Add

Refer to section 4.2.5.1.

4.2.6.2 Asset Drop

Refer to section 4.2.5.2.

4.2.6.3 Safestore

Refer to section 4.6.

4.2.7 Hold

In HOLD mode, partitions are in a suspended state and not executing, therefore overruns are not detected. This mode is used to process a Datastore or an Abort request. Hold will also be entered to initiate an initialization. Mailboxes will be populated with data for initialization if appropriate, but will not be read until Initialization is commanded by the IOS (that is when the Self_Init procedure is executed). Malfunction information and messages will not be passed between partitions during this mode.

4.2.7.1 Datastore

When a datastore is requested, the session transitions from FREEZE to HOLD mode and RTSSW collects all datastore terms that have been identified in the DIS. Taking a datastore in this manner ensures that a time–homogeneous data set is collected. During Datastore, partitions are in a suspended state (Hold mode) and do nothing. Refer to section 4.5 for more information about Datastores.

4.2.7.2 Abort

Abort conditions are detected by RTSSW. These are severe conditions that will not allow processing to continue. Due to the severity of this condition, the Abort detection must be processed immediately. Therefore, this transition is not an orderly one. During other mode transitions, the simulation does not begin executing in the new mode until all partitions have completed execution in the current mode. When the Abort transition occurs, partitions are commanded to transition to a Hold or suspended state as soon as the command is received (i.e., the next time they are released for execution). Partitions may not have completed their processing in the current mode when they receive the new mode. The following steps describe the Abort sequence:

- RTSSW detects an unrecoverable error condition and sends an Abort command to the Training Manager.
- Upon receipt, the Training Manager issues the Abort to all assets. The transition to Abort (actual transition to HOLD mode, the partition's are not executing) is processed immediately and is not dependent upon the OBCS's requirements for advanced notification of mode transitions. This disorderly shutdown may cause the OBCS to be placed in an unstable state.
- Partitions will each complete their current period's execution and then transition to Hold mode.
- RTSSW will then receive a command to initialize either through a return to data store point or initialization point. However, the error condition may be deemed

too severe to attempt a recovery. In this case, RTSSW will receive a command transition to terminate. In either case, RTSSW will wait for further instructions from the IOS.

- •• RTSSW will read the initialization data and populate the partition's message buffers.
- •• Partitions will self initialize.
- •• The system will then Initialize and an automatic transition to freeze will occur.

4.2.8 Terminate

During terminate, assets are dropped, partitions complete execution, and the RTSSW executive and communication software gracefully ceases execution. Each partition provides a Terminate procedure which shall allow for a graceful termination of that partition. Note that Terminate is a one–pass procedure. The RTSSW executive software will not detect overruns during terminate.

4.2.9 Run To Freeze Transition

- The request to Freeze is issued by the OS to RTSSW.
- Training Session Manager computes based on the registered worst case run to freeze transition times (See section 4.2.1), the earliest point in time the simulation can transition to freeze.
- Training Session Manager commands the Platform Managers to transition their Mode Controllers in each CPU to Freeze
  - •• The time to transition is included in the command to the Platform Managers.
- When the time to transition is reached, the simulation will transition to the new mode.

Ovals represent procedures that the partition developer will provide when the generic model is instantiated, while the explosions represent special case processing in which only a few partitions may need to supply procedures. (Note: these procedures will not be required for the generic instantiation.) Rectangles represent software that RTSSW is responsible for providing and clouds represent logical groupings of activity.

4.2.10 States of a Training Session

Figure 4.2.10–1 denotes the states of a training session: null, session nucleus, and target session active. A training session starts out as null; that is, no session exists. Before the OSS attempts to establish a new training session, several things are assumed to be established. 1) The Session Computer (SC) operating system will be configured for simulation and loaded into the correct CPU's. 2) A SaC process will be running on the SC and will send status information to the OSS computer. (This information will be used to determine the availability of the SC for configuration into a training session.) 3) The OSS is responsible for down–loading the training files into the correct machines. 4) The executable code is brought up on the SC in the correctly configured CPU's.

When a new training session is desired, the OSS first determines the availability of a session computer (SC) with its associated Data Management Set (DMS) string and at least one Instructor Station. Then the OSS directs the SC to establish a new training session with its DMS string and the available instructor station. If communication between the SC and its DMS string or between the SC and the Instructor Station cannot be established, the training session is not established (training session remains in the null state) and the three assets remain available for configuration into another training session. When the training session is established, it transitions from the null state to the session nucleus state. While in this state, it may commanded by the OSS to add or drop other assets as required to form the desired hardware configuration for the training session. When the OSS detects that the desired hardware configuration has been reached, it commands the training session to transition from the session nucleus state to the target session active state. At this time, command and control of the training session is passed from the OSS to the Instructor Station(s).

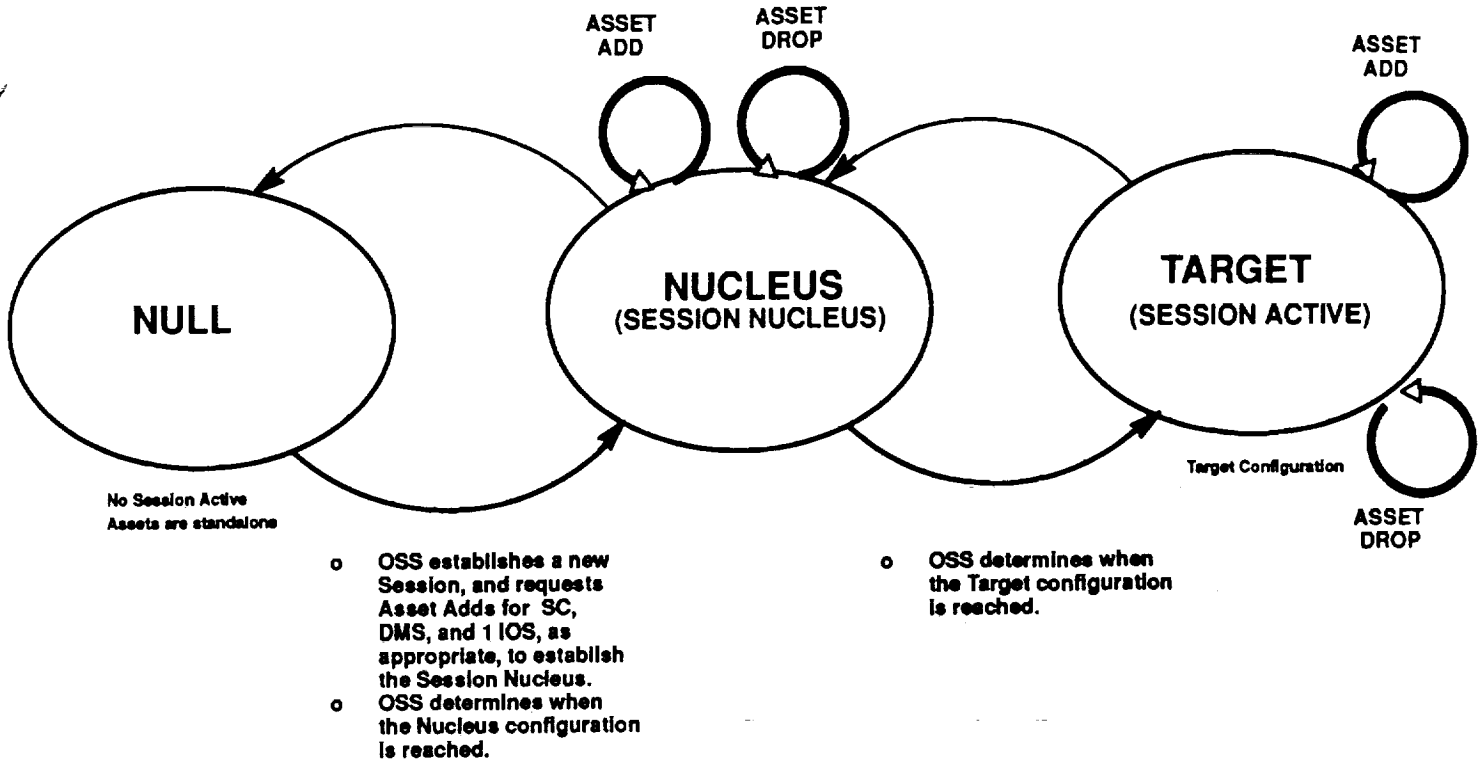


Figure 4.2.10–1 Training Session States

While in the target session active state, the training session may be commanded by the OSS to add or drop assets.

4.2.11 States of an Asset

Figure 4.2.11–1 provides the states of an asset: maintenance, pre–session, session active, and post–session. While in the maintenance state, tests and checkout procedures (and other activities involved with off–line testing) are performed on the asset's hardware. When the OSS detects that the asset's hardware is operational (on–line), the OSS transitions the asset from the maintenance state to the pre–session state (ready–to–load substate).

An asset will remain in the ready–to–load substate until it is required for configuration into a training session. When the OSS has successfully loaded and started the asset, it transitions from the ready–to–load substate into the loaded–and–ready substate. In this substate, the Asset sends out an Up_And_Ready message to the OSS, following completion of Simulation_Setup Processing. This notifies SaC that the Asset is ready for configuration instructions. Next, the asset awaits for a Create_Session command from the OSS. When the OSS commands the asset to create a training session, the asset transitions into the session connect substate. While in the session connect state, the Training Manager for the asset performs the necessary processing to create a training session then notifies Sac that it is Ready_To_Configure meaning the Asset is now ready to add assets to its established training session or to be added to another training session. In either case when communication is established between an asset and the training session, the asset transitions from the pre–session state (session connect substate) to the session active state.

While in the session active state, an asset provides services that make it worthy of being configured in the training session. An asset's responsibilities vary between assets. During session active state, an asset may transition between many different modes (such as Initialization, Run, Freeze, Hold, etc.). Also, the asset may receive an asset drop command from the training session (training session manager). Upon reception of an asset drop command, the asset performs processing that will remove it from the training session.

Upon completion of the asset drop, the asset transition into the post–session state. During the post–session state, the OSS saves asset–resident data which were created during the training session. Other activities

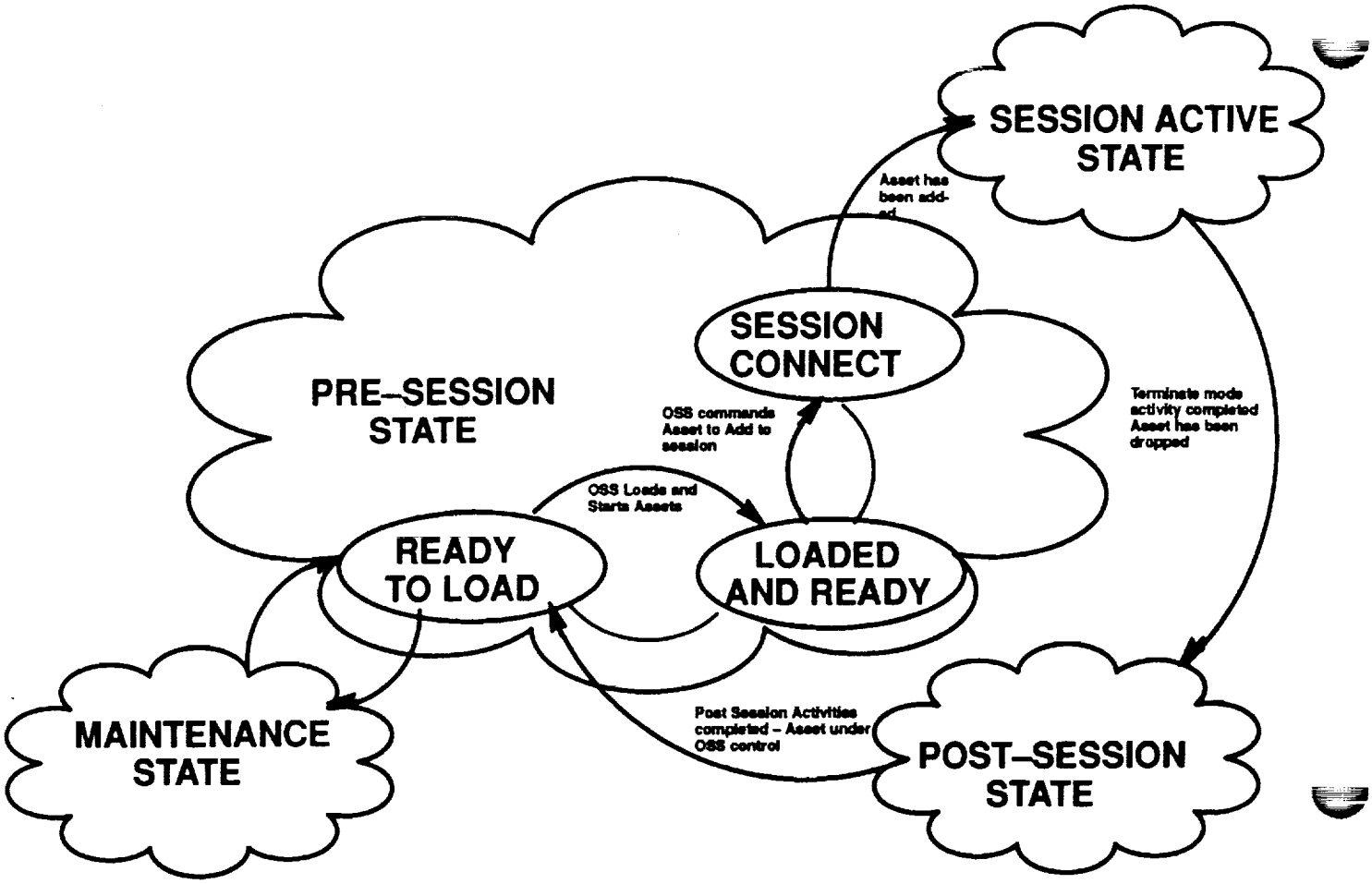


Figure 4.2.11–1 Asset States

may also occur during post–session state, depending on the requirements of the asset (such as running the OSS Productivity Monitoring Tool at an Instructor Station). When all post–session state activities are complete, the asset transitions from the post–session state to the pre–session state (into either the ready–to–load substate or the loaded–and–ready substate).

4.3. The Messaging System

There are four types of communication which are supported by the communication mechanism (software backplane), see figure 4.3–1. The first type is one producer sending to one or more receivers (one–to–many). This is the normal method for modeling real world interfaces (wiring, plumbing, talking, etc.). In this type of communication messages are queued in rate based queues to insure that the receiver will receive time consistent messages based upon the relative exxecution rates of the sender and receiver. For example, if the receiver runs four times as fast as the sender, the receiver will receive every message sent four times. If the receiver runs one fourth as fast as the sender, the receiver will receive every fourth message sent. Even if the sender and receiver are not running at harmonic rates, time consistent messages will be received.

The second type of communication is many producers sending to one receiver (many–to–one). This is a special case and should only be used by select systems (ENV, OBCS, EPS). Systems using the many–to–one interface will have to compute worst case message bursts for queue limit setups. In many–to–one communication, messages are queued in a FIFO queue to insure that all messages sent to the receiver will be received independent of the rate at which the sender and receiver execute.

The third type of communication is remote communication. It is used for messages that are sent to a remote node. Remote communication is not used by partitions, it is used by interface agents, RTSSW software, and a few special systems such as IOS which do not have Real-time Sessions Software running on them. Remote messages are also queued in FIFO queues. The one–to–many, many–to–one, and remote communication routines are located in package *Message* (see section 8.2).

Users of the messaging system declare taypes for their message in an interface definition package. Pointer types (access types) to each of the message types are also declared in this package. Partitions wishing to communicate WITH the appropriate interface definition packages and declare local objects of the access types for the messages they wish to send and receive. These local pointer objects will be registered with the software backplane. The software backplane will control them so that they point to the appropriate locations in the message buffers. Therefore, to send and receive messages, partitions just reference and de–reference their local pointers; the data may or may not actually get copied. Because messages are referenced by pointers, and because discriminants of a variant record referenced by a pointer cannot be changed, messages **cannot** be variant records and receive the benefits of variant records.

The fouth type of communication is mailbox communication. Mailbox communication is a special slow rate command and control messaging operation that is intended mainly for initialization, return to safestore, malfunction requests, and IOS enter operations. It is not encouraged as a general, partition to partition, communication mechanism. Mailbox messages are a stream of packed bytes that require packing and unpacking of data by the senders and receivers. It does not enforce Ada strong typing constructs. It is up to the sender and receiver to insure that they are using the same data types for mailbox messages. Unlike the messaging system where senders and receivers must register for specific messages, mailbox messages are dynamically routed. Each mailbox has a FIFO queue that holds incoming messages.

Partitions may have more than one mailbox. Each mailbox must have a unique prefix. Prefixes are associated with the owner of the mailbox through the DIS.Register_Component operation (see section 4.4.1). The mailbox system currently supports six types of messages, return to safestore messages, return to datastore messages, malfunction messages, enter messages, mega messages, and user defined messages. The first five types (safestore, datastore, malfunction, enter, and mega) are referred to as predefined messages because they have predefined data types. The predefined data types along with their operations are defined in four support packages: Safestore_Mailbox, Malfunction_Mailbox, Enter_Mailbox, and Mega_Mailbox. The mega message is used for return to datastore messages. In general, mega messages are used to send sets of data. These data sets may be logically related data items such as the x, y, and z values of a state vector which must be received by the model at the same time, or they could be a group of related data items such as the terms for a return to datastore. User defined messages are used for messages other than one of the predefined messages. The mailbox communication routines are located in the package *Mailbox* (see section 8.3).

4.3.1 One–to–Many (Normal) Communication

The producing partition registers its output messages with the software backplane using the "Register_To_Send_Msg" operation during the Setup submode. During the Create Data submode, the producing

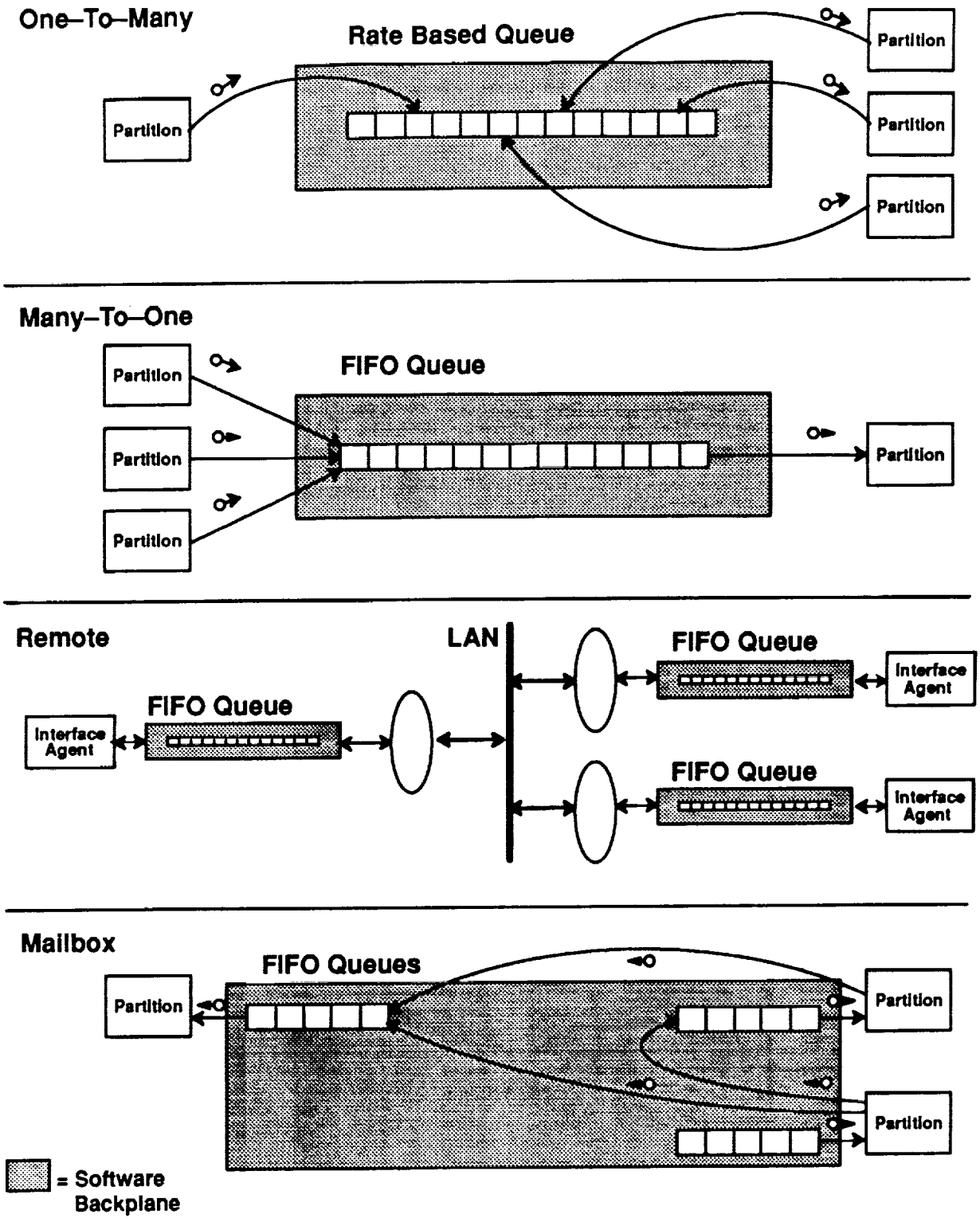


FIGURE 4.3–1

partition calls "Create_Msg". If buffers for this message have already been created in the software backplane the producers local pointers will be set to point to the message buffers. Otherwise the buffers will be created and then the pointers will be set. After ~Create_Msg" has been called the receivers local pointer will be point-

ing to the first write buffer for the particular messages. The partition should also initialize its output messages during the Create Data submode. This is done by updating the local pointers and then calling the "Put" operation. Note that local pointers **cannot** be updated until after the "Create_Msg" operation has been called. As the partition executes, it continues to update its local pointers and send out messages with the "Put" operation. The messaging system does not automatically refresh data. This means that after calling the "Put" operation, the partitions local pointer will be pointing to a new memory location and the partition should not make any assumptions about the values in this memory location. Therefore, partitions should output data in complete messages and should not read from their output pointers. If automatic data refresh turns out to be needed, an option to provide this capability may be added to the messaging system in the future.

The receiving partition registers to receive input messages using the "Register_To_Recv_Msg" operation during the Setup submode. During the Create Data submode, the receiving partition calls "Create_Msg". If buffers for this message have already been created in the software backplane the receivers local pointers will be set to point to the message buffers. Otherwise the buffers will be created and then the pointers will be set. To receive a message the partition calls the "Get" operation and de–references its local pointer. The local pointer must be de–referenced after the "Get" operation has been called and during the same period. The local pointer is only valid for one period. There are two variations of the "Get" operation: "Get" and "Get_Latest". The "Get" operation provides time consistent messages relative to the execution rate of the consumer. This guarantees that, for example, a receiver executing half as fast as the producer will always get every other message produced. Time consistent messages are guaranteed by giving the receiving partition the most recent message that was valid at the beginning of its current period. Therefore, the partition is receiving data that was produced during its previous period. The "Get_Latest" operation allows the requesting partition to receive the most recent message sent by the producer. Note that this operation does provide time homogenous data but not time consistent data. The time deltas between the messages received will vary depending upon the relative execution order of the producer and consumer. Both the "Get" and "Get_Latest" operations optionally return the time that the message was sent if the "Msg_Time" parameter is supplied.

4.3.2 Many–to–One Communication

The many–to–one communication works similar to the normal communication (one–to–many). The producing partitions register their output messages using the "Register_To_Send_Msg" operation during the Setup submode. During the Create Data submode, the producing partition calls "Create_Msg". If buffers for this message have already been created in the software backplane the producers local pointers will be set to point to the message buffers. Otherwise the buffers will be created and then the pointers will be set. As the partitions execute, they update their local pointers and call the "Put" operation to send the messages. All messages sent are placed in the receivers queue. The exception "Queue_Full" is raised if the receiver's queue is full when the "Put" operation is called.

The receiving partition registers to receive input messages using the "Register_To_Recv_Msg" operation during the Setup submode. During the Create Data submode, the receiving partition calls "Create_Msg". If buffers for this message have already been created in the software backplane the receivers local pointers will be set to point to the message buffers. Otherwise the buffers will be created and then the pointers will be set. To receive a message, the partition calls the "Get" operation and de–references its local pointer. The local pointer must be de–referenced after calling the "Get" operation and during the same period in which it was called. All messages sent to the receiver are queued in FIFO order, the "Get" operation retrieves the next message in the queue. The size of the queue is specified as a parameter to the "Register_To_Recv_Msg" operation. The queue size should be determined based upon two factors. First, the number of possible senders and second, the relative execution rates of the senders and the receiver. If the receiver is executing faster than the senders or at the same rate as the senders, the queue size must be at least as large as two times the number of senders. If the receiver is executing slower than the senders the following formula can be used to calculate the queue size: [(senders rate / receivers rate) x 2] x # senders. For example, if the receiver is executing at 10Hz with three senders executing at 40 Hz the queue size should be [(40 / 10) x 2] x 3 = 24.

The "Number_Of_Msgs_To_Get" operation returns the number of messages that have been sent and are available for the receiving partition to retrieve. The "Get" operation will raise the exception "No_Messages" if it is called when there are no messages to be retrieved. The "Get" operation also optionally returns the time that the message was sent if the "Msg_Time" parameter is supplied.

4.3.3 Remote Communication

Remote communication is not used by partitions, it is used by interface agents, RTSSW software, and a few special systems such as IOS which do not have Real-time Sessions Software running on them. Remote communication works similar to the many–to–one communication. The producers register their output messages using the "Register_To_Send_Msg" operation during the Setup submode. During the Create Data submode, the producing partition calls "Create_Msg". If buffers for this message have already been created in the software backplane the producers local pointers will be set to point to the message buffers. Otherwise the buffers will be created and then the pointers will be set. To send messages the producers update their local pointers and call the "Put" operation. As messages are sent they are picked up by the router and transmitted to the destination node. There can be more than one receiver of a remote message on a node.

The receivers register to receive input messages using the "Register_To_Recv_Msg" operation during the Setup submode. During the Create Data submode, the receivers call "Create_Msg". If buffers for this message have already been created in the software backplane the receivers local pointers will be set to point to the message buffers. Otherwise the buffers will be created and then the pointers will be set. To receive a message, the receiver calls the "Get" operation and de–references its local pointer. The local pointer must be de–referenced only after calling the "Get" operation and during the same period in which it was called. All messages sent to the receiver are queued in FIFO order, the "Get" operation retrieves the next message in the queue. The size of the queue is specified as a parameter to both the "Register_To_Recv_Msg" and "Register_To_Send_Msg" operations. The queue size should be determined based upon two factors. First, the number of possible senders, and second, the relative execution rates of the senders and the receiver. If the receiver is executing faster than the senders or at the same rate as the senders, the queue size must be at least twice as large as the number of senders. If the receiver is executing slower than the senders the following formula can be used to calculate the queue size: [(senders rate / receivers rate) x 2] x # senders. For example, if the receiver is executing at 10Hz with three senders executing at 40 Hz the queue size should be [(40 / 10) x 2] x 3 = 24. All senders and receivers should use the same queue size.

The "Number_Of_Msgs_To_Get" operation returns the number of messages that have been sent and are available for the receiving partition to retrieve. The "Get" operation will raise the exception No_Messages if the "Get" operation is called when there are no messages to be retrieved. The "Get" operation also optionally returns the time that the message was sent if the Msg_Time parameter is supplied.

4.3.4 Mailbox communication

Mailbox are registered using the "Register_Mailbox" operation during the Setup submode. Anyone wishing to receive mail messages must register a mailbox.

The sender of a predefined mail message creates the local message using the "Create" operation in the appropriate support package (Safestore_Mailbox.Create, Malfunction_Mailbox.Create, etc.). The message may then be sent using the appropriate put operation in package Mailbox (Put_Safestore_Msg, Put_Malfunction_Msg, etc.).

The receiver of a mail message must check its mailbox to determine the number of messages present. This is done using the "Num_Mail_Msgs" operation. Then, for each message in the mailbox, the receiver calls the "Get_Next_Msg_Type" operation to determine the type of the next message. The "Get_Next_Msg_Type" operation will return one of the six supported types, Return_To_Safestore, Return_To_Datastore, Malfunction, Enter, Mega, or User_Defined. If the type is one of the predefined types, the receiver calls the appropriate get operation (Get_Safestore, Get_Malfunction, etc.) to receive the message. The support packages can be used to interpret messages of predefined types.

Some mailbox users will need to use mail messages for purposes other than Safestore, Datastore, Malfunction or Enter. For this reason the mailbox system provides support for user defined mail messages. Instead of using the types and operations in Safestore_Mailbox, Malfunction_Mailbox, Enter_Mailbox or Mega_Mailbox, the user defines their own type for a mail message and is responsible for packing it themselves (as opposed to calling a "Create" operation in a support package). Support for user defined messages is provided through generic operations: "Get_User_Defined_Msg_Type", "Get_User_Defined_Msg", and "Put_User_Defined_Message". The sender and receiver must declare a type that will allow them to uniquely

identify the user defined message. This is referred to as the User_Defined_Msg_Types. It is recommended that an enumeration type be used for this. There are two restrictions placed on this type. First, it must have a size of 32 bits, and second, the values that objects of the type take must be positive. The sender instantiates the "Put_User_Defined_Msg" operation with the type for User_Defined_Message_Types and with the type for the mail message itself. After the sender builds the message by assigning to its local copy of the mail message, the instantiation of "Put_User_Defined_Msg" is called to send the message. The receiver instantiates the "Get_User_Defined_Msg_Type" operation with the type for User_Defined_Message_Types and the "Get_User_Defined_Msg" operation with the type for the mail message. The receiver checks its mailbox for messages using the "Num_Mail_Msgs" operation. For each message in the mailbox the receiver calls the "Get_Next_Msg_Type" operation to determined the type of the next message. If it is a user defined message the instantiation of "Get_User_Defined_Msg_Types" is called to determine which user defined message it is. Once the receiver knows which user defined message it is receiving, it can call the appropriate instantiation of "Get_User_Defined_Msg" to receive the message. It is up to the sender and receiver to ensure that they are both using the same data type for user defined mail messages.

4.3.4.1 Mailbox Reads by Partitions

On an initialization to a Datastore/Safestore, partitions will receive messages containing the datastore/safestore data via their partition mailbox. The partitions are responsible for interpreting the data stream. The data stream will contain the identifier and value of each data item. See section 4.4.4 for an example of a mailbox processing procedure.

4.4. The DIS Concept

4.4.1 What Is the DIS?

The DIS is used to build a symbol table which identifies data items in the running session. The identifiers are created using Ada code, and can be used by on–line code as well as by off–line tools. It will be used to support the following SSVTF capabilities:

- IOS Look & Enter capability
- Datastore/Initialize
- Identification of messages passed between SSVTF partitions
- Data Logging

DIS stands for Distributed Identifier Specification; the identifiers created using it can be distributed anywhere in the network for data requests. The DIS is composed of a top–level package of general definitions and a set of packages that declare identifiers for different SSVTF systems. The top–level package, called DIS, is written by the RTSSW group, and the general definitions it contains are for identifier types and subprograms which operate on these types (see an outline of the specification in section 8.4). The body of the DIS package holds the symbol table; identifiers are added to it by calling the DIS's 'Register' subprograms. The set of packages which declare identifiers and call these registration functions are referred to as DIS–related packages. These packages are to be written by the model developers, and must follow certain rules (presented later in this discussion) for their format and names.

There are five major abstract data types in the top–level DIS package: Component_ID, Type_ID, Term_ID, Message_ID, and Malfunction_ID.

A *Component_ID* gives a name to a configuration component of the SSVTF. A Component_ID may refer to a high–level, or large, component like Robotics or Environment, or it may refer to a much lower–level component like the left arm of the SPDM mechanism within the Robotics system. Each Component_ID is registered 'below' some other component. For example, if Robotics is composed of SPDM, SSRMS, and MT components, these are registered with Robotics as the parent component. In this way, a hierarchy of components can be established. Component_IDs, Type_IDs, Term_IDs, and Malfunction_IDs can be registered at any level in the hierarchy of components. The levels are pictorially represented in figure 4.4–1. A Component_ID can be registered as an *array* by setting the 'Length' parameter to the number of elements desired. This is a way of registering a single name that represents a set of components which contain identical elements but are distinct instances; for example a component array can be used to register four heat transfer units if they are all alike. Each element of a Component_ID array is itself a unique Component_ID. The component array can be indexed using ordinal numbers (from 1 to *Length*) or string labels (*e.g.*, Left, Middle, Right). A *prefix* indicator can be supplied with the registration call. In most instances, this means that the Component_ID being registered is also the identifier for a partition and it's mailbox. (If there is more than one mailbox for a partition, each should be identified by a different Component_ID prefix.)

A *Type_ID* is a descriptor for data items and can be used by Term_IDs to provide mappings for complex types. The DIS supports integer types (of 8, 16, or 32 bits), floating point types (single or double precision), the type String (fixed–length), type Character, and enumeration types. The DIS declares a Type_Tag which is used to distiguish between these options. An integer, floating point, or character type identifier may be supplied with upper and/or lower bounds. Enumeration type identifiers must be supplied with a list of labels that represent the enumeration literals. Enumeration representation values may optionally be supplied at registration time. *Subtypes* are Type_IDs which are based on previously registered Type_IDs, but with (possibly) different upper and lower bounds. They can be registered using a *Register_Subtype* routine — there is one each for integer, floating point and character types. The name of the subtype is the same as its base type, unless an optional name parameter is supplied to Register_Subtype. It is important to name types and subtypes so that they indicate clearly the engineering units that the user will see at the IOS.

A *Term_ID* gives a symbol name to a data item within a software model. The registration of the name includes the necessary type information with a Type_ID. A Length parameter greater than one registers a *Term_ID*

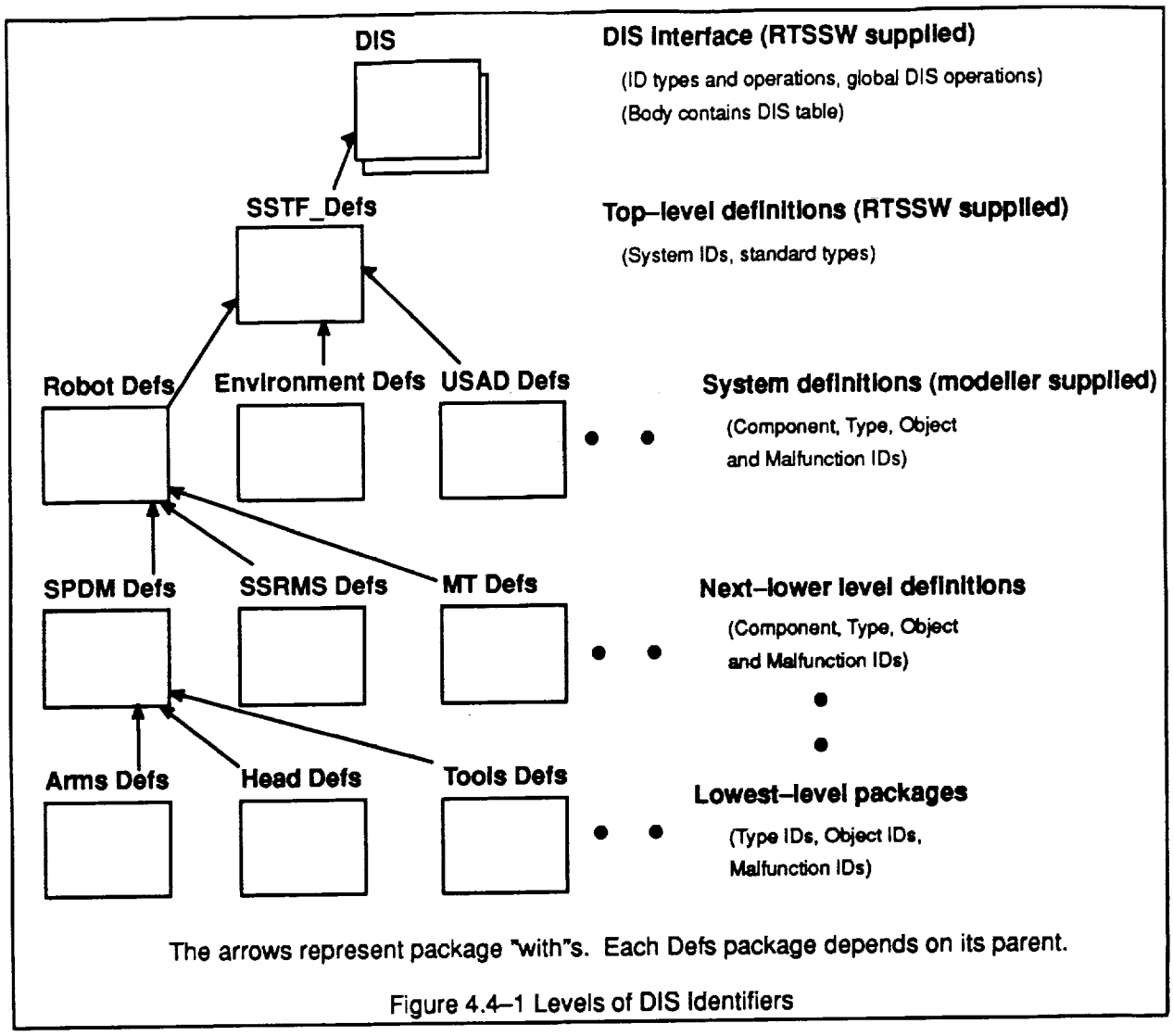


Figure 4.4–1 Levels of DIS Identifiers

array, multiple terms of the same type. Each element of a Term_ID array is a separate DIS Term_ID; like a Component_ID array, this can be indexed by ordinal numbers or string labels.

A *Message_ID* is a symbol for identifying messages transmitted in the software backplane. Message_IDs are registered in a partition's interface definition package. The only required information about a Message_ID is the size, in bits, of the data to be sent. In addition, one can specify that it identifies a safestore message.

A *Malfunction_ID* is a name which can be used by the IOS page developers to invoke a malfunction in the running session. The malfunction identifier includes descriptor information that determines the kind of parameters which can be sent with the identifier to the host. A *Malfunction_ID array* can be used to register a malfunction that applies to many entities; as with Term_ID arrays, this is accomplished by setting the Length parameter the the number of malfunctions desired. As with Term_ID arrays and Component_ID arrays, each element of a Malfunction_ID array is a unique identifier, and the array can be indexed by ordinal numbers or a set of string labels.

There are four categories of malfunctions, distinguished by the kind of parameters which can be passed to them. First is the "*simple*" or "parameterless" malfunction; this is registered with no parameter information; when an instructor activates this kind of malfunction, no input parameters are passed to the model along with it. Second is the "*options*" malfunction; a single enumeration–type parameter is passed to the malfunction to indicate the behavior desired. For example, if a valve can be failed in one of four positions, an enumeration

type identifier would be reigistered which lists the four positions; this list will be displayed to the instructor when the malfunction is activated. The instructor will select the correct position at which to fail the valve; this selection will be passed to the model as the parameter for the malfunction. Third is the "*P1*" malfunction; a single floating point value is supplied with the malfunction. The fourth kind of malfunction, "P1_P2", is supplied with two floating point values; these are typically used for scale and bias. When registering a malfunction which has floating point type parameters, the Px_Name and Px_Type parameters are always required. Px_Low and Px_High limits can be supplied; if they are not supplied, the high and low limits of the malfunction parameter are taken from the type registration for Px_Type.

Each "*..._ID*" type is declared "private" in the DIS package spec. For each, there is a "Register_..." function that returns a value of the respective type (e.g., there is a "Register_Term" function which returns a Term_ID). These are the functions that are called by the DIS–related packages which are written by the SSVTF model developers. Each of these functions has the side–effect of adding the identifier to the symbol table in the DIS package body. Because the identifiers being added to the DIS are not to be changed at runtime, the identifiers are to be declared constants in the DIS–related packages.

Along with the identifier types and registration functions, a number of other supporting types and functions are declared in the top–level DIS package. "User" is an enumeration type which provides tags that indicate the uses of a Term_ID; *e.g.*, a Term_ID with User "Look_Enter" means that the term being identified can be examined or changed by IOS or data logging. When registering a Term_ID, a list of users is supplied as an array—the User_List parameter. The default combination of "Look_Enter" and "Initialize" in this list means that the term can be examined and changed by IOS, and is to be inculded in the Datastore/Initialize data set.

Supporting functions include selectors for information associated with identifiers, such as the string name of an identifier, the number of levels associated with Component_ID, the type tag associated with a Term_ID, and the different descriptor information associated with a Type_ID.

Entities in the real software are "connected" to DIS identifiers by the operations Connect_Term, Connect_Malfunction, and Connect_Prefix. The "Register_..." functions are called from the DIS–related packages written by the model developers, and provide the *static* DIS information needed by off–line tools; the "Connect_..." operations supply the additional *run–time* information needed to locate data and partitions. The Connect_Term procedure makes an association between a Term_ID and the address of the data which is referred to by the identifier. This allows the data to be retrieved "through the back door". Connect_Malfunction is called for malfunctions which need to be datastored or which will be "poked' using the Malfunction_Mailbox package. An address is required for each parameter to be associated with the malfunction, as well as an address of a Boolean value, the "active" flag, which indicates whether or not the malf is currently activated. The Connect_Prefix procedure's Component_ID parameter must have been registered with Prefix => True. If a partition will receive messages in more than one mailbox, each should be identified with a different prefix Component_ID, and each of these needs to be connected with Connect_Prefix. The prefix name should uniquely identify a particular mailbox.

4.4.2 How Is the DIS Organized?

The DIS–related packages are organized in a hierarchy. The top–level DIS–related package is called SSTF_Defs (all DIS–related packages have the suffix "_Defs"), and is written and maintained by the RTS group. This package registers the Component_IDs for the top–level systems in the SSVTF, which includes Robotics, Environment, USAD, USAV, Visual, SNS, and others. It also registers Type_IDs and supplies Type_Tags which correspond to the ones in the SSVTF Standard_Engineering_Types package. See Appendix II (section 8.5) for the complete specification of the SSTF_Defs package.

For each Component_ID in the above package, another package must be created which defines the identifiers that exist at the next lower level in the DIS hierarchy (this is a general rule for DIS–related packages). Thus, there must be a package for Robotics (which would be written by the Robotics group), and it will look something like this:

```
with DIS, SSTF_Defs;
package Robotics_Defs is
```

```
        package DD renames SSTF_Defs;

        -- Identifiers for Robotics components.

        MT      : constant DIS.Component_ID := DIS.Register_Component
                           (DD.Robotics, "MT");
        SSRMS   : constant DIS.Component_ID := DIS.Register_Component
                           (DD.Robotics, "SSRMS");
        SPDM    : constant DIS.Component_ID := DIS.Register_Component
                           (DD.Robotics, "SPDM", Prefix => True);
        AVU     : constant DIS.Component_ID := DIS.Register_Component
                           (DD.Robotics, "AVU", Length => 4);
        MBS     : constant DIS.Component_ID := DIS.Register_Component
                           (DD.Robotics, "MBS");
        MMD     : constant DIS.Component_ID := DIS.Register_Component
                           (DD.Robotics, "MMD");

end Robotics_Defs;
-------------------------------------------------------------------------------
```

This package would be written by the Robotics group. Similar packages would be developed for USAD, Environment, etc. Notice that the hierarchy concept is recursive: for the Robotics systems there must be an SSRMS_Defs package, an MT_Defs package, an SPDM_Defs package, etc.; and for each Component_ID registered in these packages, another package must be written. For a component array like AVU, only one AVU_Defs package needs to be written; the identifiers defined in this package will be duplicated the appropriate number of times; in this case four. The current design of the DIS permits up to seven (7) levels of components to be registered. Term_IDs, Type_IDs, and Malfunction_IDs can be registered at any level in the hierarchy. See Figure 4.4–1. Two rules must be followed to ensure that the Dis is created properly: (1) *all IDs in the same "_Defs" package must be registered with the same Parent Component_ID*; and (2) *the same Component_ID must not be used as a Parent in more than one "_Defs" packages.*

The hierarchy of DIS–related packages should reflect the *hardware, not the software, structure* of the modeled system. There are two reasons for this: (1) the DIS exists mainly to provide a window into the system for IOS and system initialization; the people performing these duties are not likely to know (nor should they have to know) the software organization of the system (i.e., the partitions, object classes, etc.); and (2) the software structure of the simulator will probably change more frequently than the hardware configuration, and changes to the DIS hierarchy should be minimized, since this has adverse effects on re–compilation. Therefore, the way a system (like Robotics or USAD) is organized *from the user's point of view* is how the DIS–related hierarchy should be organized. The relationship between this organization and the "partition" organization is discussed below: see "Connecting Terms and Prefixes".

(Note that Message_IDs do not appear in "_Defs" packages, but in the appropriate partition's interface definition package, with the suffix "_Intfc_Defs". They do not form any part of the "_Defs" hierarchy.)

In addition to the hierarchy rule, other guidelines need to be followed in order for the DIS to work properly. All of these DIS–related packages should have no "state"; i.e., all of the identifiers are constants, and no variable data should be declared in these specs. Furthermore, no subprograms may be exported from these packages. The "_Defs" packages should not have package bodies. Also, "_Defs" packages should not "with" other packages which have state or subprograms. This is because the entire "_Defs" hierarchy is to be "withed" offline for use by off–line tools.

Each Term_ID must include type information in order to permit interpretation of the data being examined. The code below shows some examples of Type_ID, Term_ID, and Malfunction_ID registrations.

```
with DIS;
with SSTF_Defs;
```

```
with Robotics_Defs;
package SPDM_Defs is

        Circle_Degrees   : constant DIS.Type_ID := DIS.Register_Subtype
                           (Robotics_Defs.SPDN, Base => SSTF_Defs.Degrees,
                            Low_Bound => -180.0, High_Bound => 180.0);

        Left_Arm_Yaw     : constant DIS.Term_ID := DIS.Register_Term
                           (Robotics_Defs.SPDM, "Left_Arm_Yaw",
                            The_Type => Circle_Degrees);

        Left_Arm_Pitch   : constant DIS.Term_ID := DIS.Regsiter_Term
                           (Robotics_Defs.SPDM, "Left_Arm_Pitch",
                            The_Type => Circle_Degrees);

        Left_Arm_Roll    : constant DIS.Term_ID := DIS.Register_Term
                           (Robotics_Defs.SPDM, "Left_Arm_Roll", DIS.Float_Tag,
                           Users => (DIS.Look, DIS.Initialize));

        Direction_Labels : constant String := "Yaw, Pitch, Roll";

        Right_Arm        : constant DIS.Term_ID := DIS.Register_Term
                           (Robotics_Defs.SPDM, "Right_Arm",
                            The_Type => Circle_Degrees, Length => 3,
                            Labels => Direction_Labels);

        Fail_Left_Arm    : constant DIS.Malfunction_ID := DIS.Register_Malfunction
                           (Robotics_Defs.SPDM, "Fail_Left_Arm",
                            Options => SSTF_Defs.On_Off);

        Fail_Right_Arm   : constant DIS.Malfunction_ID := DIS.Register_Malfunction
                           (Robotics_Defs.SPDM, "Fail_Right_Arm",
                            P1_Name => "Degrees", P1_Type => Circle_Degrees);

end SPDM_Defs
```

A Component_ID which is registered as an array (by supplying a length parameter to the Register_Component routine) requires only one "Defs" package which uses the ID as its parent. The DIS will automatically duplicate the contents of the "Defs" package to each component represented by the multiple.

4.4.3 Connecting Terms, Prefixes, and Malfunctions

In the partition code, a modeler needs to supply a procedure to associate (or *connect*) addresses to term, malfunction, and prefix identifiers to the partition. This needs to be called in the Setup procedure . Here is an example:

```
with Mailbox, Generic_Model, SET, Robots_Types;
package body SPDM_Partition is

        Left_Arm_Yaw     : Float;
        Left_Arm_Pitch   : Float;
        Left_Arm_Roll    : Float;
        Left_Arm_On_Off  : SET.On_Off;
```

```
        Right_Arm        : array (1..3) of Float;
        Fail_Right_Arm_Active : Boolean := False;
        Fail_Left_Arm_Active : Boolean := False;
        Right_Arm_Degrees : Robots_Types.Degrees;

        Partition_Name : constant String := "SPDM_Partition";
        Mb       : Mailbox.Mailboxes; -- my mailbox

        procedure Update is separate;
        procedure Freeze is separate;
        -- ...etc. for mode procedures.

        procedure Setup is separate;

        procedure Process_Mailbox_Requests is separate;

        package My_Thread_Exec is new Generic_Model.Periodic
                                    (Name   => Partition_Name,   --- etc...

end SPDM_Partition;
```

Each Term_ID and Malfunction_ID must belong to a partition in the system. By "belonging" we simply mean that the data identified by a Term_ID is located in an "owning" partition, and that the handling of a malfunction identified by a Malfunction_ID is done within a "owning" partition. No data item or malfunction handling is shared between partitions; there is only one "owner" per id. A *prefix* is a Component_ID that directly identifies a single partition. The prefix of a Term_ID or Malfunction_ID is that portion of the identifier which indicates the partition to which that identifier belongs. More than one prefix may identify a partition; also, all identifiers registered under a particular prefix must be located within one partition.

As an example of this, consider a partition that combines the SPDM arms and the power supply for the arms. Suppose that the following four components have been registered: **Robotics.SPDM.Arms**, **Robotics.SPDM.Arms.Left_Arm**, **Robotics.SPDM.Arms.Right_Arm**, and **USAD.EPS.Arm_Power**. If the data and malfunctions for all the identifiers in the packages Arms_Defs, Left_Arm_Defs, and Right_Arm_Defs are to be located in this partition, then **Robotics.SPDM.Arms** might be registered with Prefix => True, and the partition calls "Connect_Prefix" with the Component_ID **SPDM_Defs.Arms**. This partition also connects the prefix **USAD.EPS.Arm_Power** (which should also have been registered with Prefix => True), if the data and malfunctions for all the identifiers in that package are located in this partition.

In the following example of a Setup procedure, the data items being connected to identifiers are not complex types. This is not a realistic example, and is provided only to illustrate the way to call the Connect routines. The way to connect addresses to identifiers representing selected pieces of complex objects is to use the Symbol_Map package; this is discussed in Section 4.5. The "DIS.Connect_" procedures should be called after creating the object instances in the Setup procedure.

```
with DIS, SPDM_Defs;
with Local;  -- a package to get local system information
procedure Setup is
begin
        ....  -- Create objects (see Sec 4.2.1)

        DIS.Connect_Term
                (SPDM_Defs.Left_Arm_Yaw,  -- first parameter is the Term_ID;
                 Left_Arm_Yaw'Address);   -- second is the actual data address.

        DIS.Connect_Term (SPDM_Defs.Left_Arm_Pitch, Left_Arm_Pitch'Address);
        DIS.Connect_Term (SPDM_Defs.Left_Arm_Roll, Left_Arm_Roll'Address);
```

```
        DIS.Connect_Term_Array (SPDM_Defs.Right_Arm, Right_Arm(1)'Address);
        DIS.Connect_Prefix (Robotics_Defs.SPDM, Local.Get_Node_Name,
                            Local.Get_Process_ID, Partition_Name);
                -- if more than one prefix relates to this partition,
                -- do another "Connect_Prefix" call for that one.

        DIS.Connect_Malfunction (SPDM.Fail_Left_Arm,
                 (Active_Addr  => Fail_Left_Arm_Active'Address,
                  Options_Addr => Left_Arm_On_Off'Address,
                  others => Dis.Null_Address);
        DIS.Connect_Malfunction (SPDM.Fail_Right_Arm,
                 (Active_Addr => Fail_Right_Arm_Active'Address,
                  P1_Addr => Right_Arm_Degrees'Address,
                  others => DIS.Null_Address));

end Setup;
-------------------------------------------------------------------------------
```

4.4.4 Handling Enters, Malfunctions, and Initialization data

Connecting addressed to identifiers is enough to permit IOS to look at the data items. In order to allow IOS to "enter" values, it is necessary to receive messages from IOS through the partition mailbox. The reason for this is that a change of state like this cannot be done the way a read is done (backdoor via address); it is necessary to incorporate the new value in a controlled way that cannot corrupt the system in the middle of computation. A procedure should be written to handle this; in the example below, it is called Process_Mailbox_Requests. The mailbox is used to accept requests for IOS enters, system initialization values, and malfunctions. This procedure should check for mailbox inputs and process whatever has arrived, applying the malfunction or new data. This procedure should be called at the beginning of the Run and Freeze mode subprograms. Thus, the update is incorporated in a controlled way.

Notice that the different mailbox packages have the ability to "poke" the data coming in. This takes the data that has come into the mailbox and puts it into the address that was connected for that data, whether it is malfunction data or term data being entered from IOS or returned from a datastore Initialize operation. The example shows how to treat some entered data with special processing (data which cannot just be directly placed into the target address), and how to poke the rest.

```
-------------------------------------------------------------------------------
with DIS, SPDM_Defs;
with Mailbox, Enter_Mailbox, Malfunction_Mailbox, Safestore_Mailbox;
separate (SPDM_Partition)
procedure Process_Mailbox_Requests is
    Num_Msgs : Natural := Mailbox.Num_Mail_Msgs (Mb);
    More     : Boolean;
    Msg_Type : Mailbox.Msg_Types;
    Size     : Natural;
    E_Msg    : Enter_Mailbox.Enter_Msg;
    M_Msg    : Malfunction_Mailbox.Malfunction_Msg;
    S_Msg    : Safestore_Mailbox.Safestore_Msg;
    G_Msg    : Mega_Mailbox.Mega_Msg;
begin
    for I in 1..Num_Msgs loop
        Msg_Type := Mailbox.Get_Next_Msg_Type(Mb);
        case Msg_Type is
            when Mailbox.Enter =>
```

```
                Mailbox.Get_Enter_Msg (E_Msg, Mb);
                if Enter_Mailbox.ID(E_Msg) = SPDM_Defs.Left_Arm_Yaw then
                    -- You only need to check the ID if you need to do
                    -- special processing on the incoming data...
                    Left_Arm_Yaw := Enter_Msgs.Convert_Float (E_Msg);
                    -- ... etc.
                elsif Enter_Mailbox.ID(E_Msg) = SPDM_Defs.Right_Arm_Yaw then
                    Right_Arm_Yaw := Enter_Msgs.Convert_Float (E_Msg);
                    -- etc ...
                elsif
                    -- etc...
                else
                    -- for all other enters, just call poke, which directly
                    -- places the data into its address...
                    Enter_Mailbox.Poke (E_Msg);
                end if;
            when Mailbox.Return_to_Datastore | Mailbox.Mega =>  -- datastore
                Mailbox.Get_Mega_Msg (G_msg, Mb);
                Mega_Mailbox.Go_To (G_Msg, SPDM_Defs.Right_Arm_Yaw, Found);
                if Found then
                    Mega_Mailbox.Value (G_Msg, a-floating-point-variable);
                    -- process the floating point variable before assigning.
                    -- getting the value of a mega entry ensures that its
                    -- value will not be poked by a poke_all call.
                end if;
                Mega_Mailbox.Poke_All (G_Msg); -- simply poke all other entries
            when Mailbox.Malfunction =>
                Mailbox.Get_Malfunction_Msg (M_Msg, Mb);
                if Malfunction_Mailbox.ID(M_Msg) = SPDM_Defs.Fail_Left_Arm then
                    --  do whatever it takes ...
                else
                    Malfunction_Mailbox.Poke (M_Msg)
                end if;
            when Mailbox.Return_To_Safestore =>  -- safestore
                Mailbox.Get_Safestore_Msg (S_Msg, Mb);
                if Safestore_Mailbox.ID(S_Msg) = A_Safestore_Message_Id then
                    The_Safestore_Data_Object.all :=
                        Move_Data (Safestore_Mailbox.Value(S_Msg));
                elsif ... -- a different id then
                    ... -- move the return value to the data item
                end if;
        end case;
    end loop;
end Process_Mailbox_Requests;
```

The DIS term registration can be used to tag datastore items for eventual retrieval. Each item tagged for datastore will be retrieved "in the background", like an IOS look. Each item tagged for safestore will be retrieved through the software backplane; the partition must create software backplane "output messages" for these items. Both datastore and safestore items will be sent back (for return to datastore and return to safestore) to the partition through the partition's mailbox — the partition must have a special procedure to read this mailbox during initialization.

4.4.5 How Will Off–line Tools Use the DIS?

The identifiers registered with the DIS through the '_Defs' packages are entered into the DIS body's data structure at elaboration time, before the start of the main program. Then a program may access any of the DIS data by calling functions and procedures in the DIS spec. While this works out well for real time models, it is not good for off–line tools to be dependent (in the Ada sense) on these packages. If any change is made to the registered identifiers, the off–line tool using the DIS would have to be re–linked in order to get the new identifier information. So, for off–line tools, the DIS tree will not be populated by the elaboration of '_Defs' packages, but by the reading of a file. The DIS.Report routine saves the entire set of registered static information in a file which can be loaded into the DIS tree using the DIS.Load routine. The off–line tool is dependent on the file instaead of the packages; in order to get new versions of the DIS, a tool will have to load new versions of this file, but it will not have to be re–linked.

An example of such an off–line tool is the DIS Browser, which displays the registered identifiers in a workstation window so that IOS page creators can select an identifier and associate it with a screen gauge, button, or meter. (On–line tools that require the presence of the entire set of DIS identifiers, like the Datastore partition and the Central look–at engine facility, will also use the report files rather than the '_Defs' packages.)

The DIS.Load routine can work in two ways: when the File_List parameter is False (default), the From_File string parameter is the name of a file containing the output of a DIS.Report call. When the File_List parameter is True, From_File indicates a file that contains a list of files, each of which was created by a DIS.Report call. Each group (USAD, Robotics, Environment, etc.) will create a different file using DIS.Report. Then these files will be listed in the file passed to DIS.Load; in this way the entire set if DIS identifiers will be loaded for tools which need to see the whole of it.

The Dis.Report creates a *non–expanded* report file by default. The report contains all information necessary for the Dis.Load call, and the output is summarized such that only a few lines are used for an id array, even if the array has hundreds of elements. By setting the "Expand" parameter to True, and *expanded* report file is produced. Each line in the file is exactly one identifier—all of the arrays are expanded out. (Unlike the non–expanded report, the expanded report does not contain enough information to re–create the entire Dis via a Load call.) The identifier on the line can be converted to its internal representation using the appropriate Dis.Convert call. The expanded report is useful for visual inspection of the Dis contents and for tools that need to search through or manipulate the entire Dis.

Another way of using the Dis is through the Navigate package, a sub–package available in the Dis spec. This permits a tool familiar with the Dis structure to traverse through the Dis tree using the different types of "handles" defined by the Dis.

4.5. Mapping Logical Name to Physical Address: DIS & Symbol Map

In the SSVTF simulation data needs to be displayed at Instructor/Operator Stations (IOS). The DIS was created to assist in this problem. The DIS provides the IOS a logical view of the simulation by defining a method of relating simulation terms to IOS page terms. However, the DIS in itself does not solve completely the problem of mapping physical Ada simulation terms to the IOS logical term.

The SSVTF architecture encourages the use of partitions, classes, and class compositions. A class represents a specific object and should not be aware of where it is used in the simulation—i.e., which other structures (partitions, classes) may inherit it. However, there are Ada terms in the individual classes that may need to be visible at an IOS. How can these terms be registered in the DIS?

A register symbol structure has been defined that can solve the problem. A class provides a procedure that registers Ada terms in the class with a symbol list. A structure which inherits the class (a parent) provides the class its name (the parent name) in the Create procedure. Thus, when class terms are registered in the symbol list the parentage is contained in the term name. In this manner, Ada terms within a class are registered with the symbol list. This provides a physical mapping of terms to simulation physical addresses.

The specification for the Symbols package which manages the symbol list follows.

```
with System;
package Symbols is

   type Base_Types is (Integer, Real, Enum, Boolean);

   --*************************************************************************
   --| Register associates an actual variable name with its type, address, and size.
   --| ** NOTE ** Register is only valid during Set_Up mode.
   --|
   --| Parameters:
   --|    Name: the full name of the variable
   --|    Base_Type      : the base type of the variable
   --|    Tick_Address   : the address of the variable
   --|                     ** Must use Variable'Address **
   --|    Tick_Size      : the size in bits of the variable
   --|                     ** Must use Variable'Size **
   --|
   --| Exceptions Raised:
   --|    Duplicate_Name : raised if the same name is in the database
   --|    Register_Mode  : raised if system is not in Set_Up mode
   --|
   --| Example of how to use:
   --|    Register (Name          => Parent & ".item",
   --|              Base_Type     => Symbols.Integer,
   --|              Tick_Address  => Instance.Item'Address,
   --|              Tick_Size     => Instance.Item'Size);
   --*************************************************************************
   procedure Register (Name          : in String;
                       Base_Type     : in Base_Types;
                       Tick_Address  : in System.Address;
                       Tick_Size     : in Natural);

   -- *************************************************************************
   --| Is_Address is a function that returns the address of a registered symbol
   --| ** NOTE ** Is_Address also removes the symbol from the symbols database
   --|
   --| Parameter:
   --|    Name      : the full name of the registered variable
   --|
   --| Returns:
   --|    Address   : the address of the registered variable
   --|
```

```
            --| Exceptions Raised:
            --|    Name_Not_Found : raised if the name is not in the database
            --| ***************************************************************************
            function Is_Address (Name : in String) return System.Address;

            -- ***************************************************************************
            --| Report is a procedure that prints the contents of the Symbols_Table to a
            --| data file.
            --| ** NOTE ** This routine is supplied for testing only, and should not be
            --|            called in real-time.
            --|
            --| Parameter:
            --|    Filename  : the name of the output file
            --|
            --| Exceptions Raised:
            --|    Those propagated by Text_Io.
            -- ***************************************************************************
            procedure Report (Filename : in String);

            -- ***************************************************************************
            --| Clear will remove all the remaining symbols from the Symbols_Table.
            --| ** NOTE ** This routine should be called by the partition in Set_Up, after all
            --|            of the Dis.COnnects have completed.
            --|
            --| Parameter:
            --|    none
            --|
            --| Exceptions Raised:
            --|    Those propagated
            -- ***************************************************************************
            procedure Clear;

            ---------------------------------------------------------------------------
            --| Exceptions
            ---------------------------------------------------------------------------
            Name_Not_Found : exception;
            -- raised by Is_Address if the name is not currently in the database

            Duplicate_Name : exception;
            -- raised by Resister if the same name is currently in the database

            Register_Mode : exception;
            -- raised by Register if the system is not in Set_Up mode

end Symbols;
-------------------------------------------------------------------------------------
--|  Abstract:        Symbols is a service package that is used to associate
--|                   variable names with their Type, Address, and Size
--|                   attributes.
--|
--|                   How to use:
--|
--|                     for a Class (in the Create operation)
--|                       call Symbols.Register for required variables
--|
--|                     for a Partition (in Set_Up)
--|                       process all Class.Create operations
--|                       call Symbols.Register for required partition symbols (optional)
--|                       call Symbols.Report to show all registered symbols (optional)
--|                       process all Dis.Connect operations
--|                       call Symbols.Clear to remove any unused symbols
--|
```

```
--|  Warnings:          The Register parameters Tick_Address and Tick_Size must
--|                     be values that are the direct result of using the Ada
--|                     Predefined Language Attributes P'Address and P'Size.
--|
--|                     A call to the Is_Address operation returns the address
--|                     of the symbol, but also removes the symbol from the database.
-------------------------------------------------------------------------------
```

Now we need to map the physical address to the logical name registered in the DIS. This is achieved with the DIS.Connect_Term procedure.

```
package DIS is
    • • •
       procedure Connect_Term (          Term       : in Term_Id;  ---DIS.Term_Id (Logical)
                                         Symbol     : in String);  --- Symbol.Register name
                                                                   --- (Physical)
    • • •
end DIS;
```

In summary, the IOS logical view of the simulation is defined via the DIS and DIS_Defs packages. The physical address of simulation terms is captured via the symbol list (package Symbols). The two are joined via the DIS.Connect_Term procedure.

The following figures depict how the logical to physical mapping works. The figure 4.5–1 provides code excerpts of a partition, its associated DIS_Defs, and Class packages. The figure 4.5–2 provides a conceptual view of how the Set_Up procedure ties everything together.

```
package Partition;
-------------------------------
with Class_A, DIS, My_Defs;
package body Partition is
   Zebra : Class_A.Objects;
   ...
procedure Set_Up is
begin
  Class_A.Create
        (Instance => Zebra,
         Parent => "Partition.Zebra");
  ...

  DIS.Connect_Term
     ( Term   => My_Defs.Value_X,
       Symbol => "Partition.Zebra.X"
  DIS.Connect_Term
     ( Term   => My_Defs.Value_Y,
       Symbol => "Partition.Zebra.Y"
  DIS.Connect_Term
     ( Term   => My_Defs.Command,
       Symbol => "Partition.Zebra.Cmd"
     ...
end Set_Up;
     ...
end Partition;
```

```
with Other_Defs;
package My_Defs is
  Value_X : constant DIS.Term_Id :=
    DIS.Register_Term (Parent   => Other_Defs.Sys,
                       Name     => "Vaue_X",
                       The_Tag  => DIS.Float_Tag,
                       Users    => (1=> DIS.Look));
  Value_Y : constant DIS.Term_Id :=
    DIS.Register_Term (Parent   => Other_Defs.Sys,
                       Name     => "Vaue_Y",
                       The_Tag  => DIS.Integer_Tag,
                       Users    => (1=> DIS.Look));
  Command : constant DIS.Term_Id :=
    DIS.Register_Term (Parent   => Other_Defs.Sys,
                       Name     => "Command",
                       The_Tag  => DIS.Enum_Tag
                       Users    => (1=> DIS.Look));
    ...
end My_Defs;
```

Symbol List

Name	Type	Size (Bits)	Address
Partition.Zebra.X	Real	32	FAC0
Partiion.Zebra.Y	Integer	32	FAC4
Partition.Zebra.Cmd	Enum	8	FAC8
Partition.Zebra.Dog.A	Boolean	8	FAC9
Partition.Zebra.Dog.B	Integer	32	FACC

```
package Class_A is
    type Object is limited private;
    type Commands is (Set_Qty, Leak_Oil);

    procedure Create (Instance : in out Object;
                      Parent   : in String);

    procedure Request_State_Change
                     (Instance  : in out Object;
                      Command : in Commands;
                      Val       : in Integer);
    ...
private
   type Object is
     record
       X    : Real;
       Y    : Integer;
       Cmd : Commands;
       Dog  : Class_B.Object;
   end record;
end Class_A;
-------------------------------
with Class_B, Symbols;
package body Class_A is

    procedure Report_Symbols
                     (Instance : in out Objects;
                      Parent   : in String) is separate;

    procedure Create (Instance : in out Objects;
                      Parent   : in String)  is
    begin
      Report_Symbols (Instance => Instance,
                      Parent => Parent);
        -- Make a class composition
      Class_B.Create (Instance => Instance.Dog,
                      Parent => Parent & ".Dog");
      ...
    end Create;
-------------------------------
separate (Class_A)

procedure Report_Symbols is
                    (Instance : in out Objects;
                     Parent   : in String) is
begin
     Symbols.Register (Name       => Parent & ".X",
                       Base_Type  => Symbols.Real,
                       Tick_Address => Instance.X'Address,
                       Tick_Size    => Instance.X'Size);

     Symbols.Register (Name       => Parent & ".Y",
                       Base_Type  => Symbols.Integer,
                       Tick_Address => Instance.Y'Address,
                       Tick_Size    => Instance.Y'Size);

     Symbols.Register (Name       => Parent & ".Cmd",
                       Base_Type  => Symbols.Enum,
                       Tick_Address => Instance.Com'Address,
                       Tick_Size    => Instance.Com'Size);
end Report_Symbols;
```

Figure 4.5–1

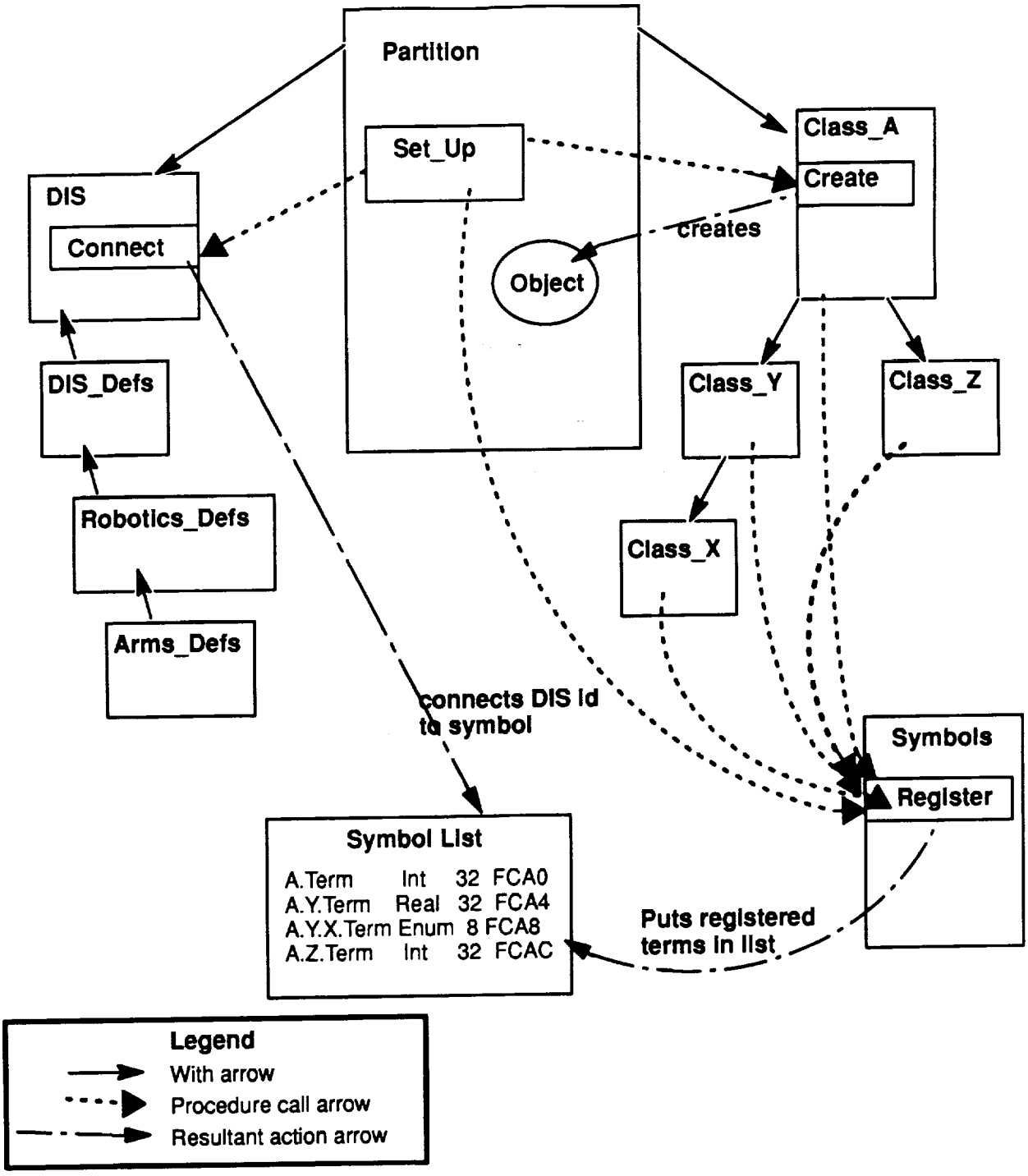


Figure 4.5–2 Mapping Logical Terms to Physical Address at Set_Up

4.6. Datastore/Initialization

The following sections provide a textual description of datastore activities and requirements. The figures, at the end of this section, depict how a datastore is performed, where datastores are performed and how a return to datastore occurs, respectively.

A datastore is an instructor initiated activity. The state of the simulation session is captured and saved to a disk file. The datastore may be saved and used in other simulation sessions. The datastore retains enough information to initialize the simulation to the same state at which the datastore was taken.

4.6.1 Perform a Datastore

The following steps are performed when a datastore is requested by the instructor. An instructor enters a datastore command along with some type of datastore ID. The IOS sends the datastore command and ID to RTSSW. RTSSW transmits the datastore command to all Ada mains and platforms (including APM and JEM). The session transitions to the Datastore mode synchronously. In the datastore mode, no data transfers occur except for those partitions responsible for communicating with hardware devices (to keep them from dropping off-line). The Datastore object, using the IOS Look–At technique and DIS, reads data from the simulation partitions (note that OBCS may be an exception to this method). The Datastore object buffers the data and writes it out to an ASCII disk file in records containing the fully qualified Ada DIS name, type, and value. RTSSW provides datastore status to SaC as required. Lastly, the session transitions to the Freeze mode, and RTSSW sends IOS the Freeze mode notification.

4.6.2 Initialize to a Datastore

The following steps are performed when an initialize to a datastore is requested. An instructor enters the initialize to datastore command and corresponding datastore id. The IOS confirms the data entry and sends both the command and datastore id to RTSSW. The session transitions to the Initialization mode synchronously. The Datastore object opens the datastore file and reads the datastore data from disk. The Datastore object parses the datastore data and sends the datastore data to the appropriate partitions via mailbox messages. Each partition reads its mailbox messages and self–initializes to its internal datastore values. The session then initializes to the datastore point during the system init mode.

4.6.3 Partition Requirements

For a partition to successfully be involved in a datastore event, the following rules must be adhered to:

- Each datastore item must be registered in the DIS.
- Each partition has a mailbox.
- Each partition provides the software to process (input) the data from the mailbox.
- Each partition provides a self–initialize routine to internally initialize to the datastore state.

4.6.4 Datastore Notes

RECON will not be dependent on the DIS to process datastore data.

The datastore file will be ASCII to the extent practical.

OBCS datastore (flight software terms) may be a special case. Due to the size and nature of the OBCS, the OBCS binary data may be handled in a different manner. OBCS will be responsible for the format of the binary data. OBCS data will probably be maintained in its own datastore file; the file name will correlate to the regular datastore file name/id.

The datastore file(s) will have an id associating it to the datastore id, SGMT, and load id. RTSSW will create the datastore file name. A title and short description of the datastore will be entered by the instructor and placed in the datastore file.

CSIOP does not do a datastore to the CSIOP platform (the CSIOP interface agent in the Session Host provides the CSIOP datastore data).

SNS will do a datastore to the disk local to the SNS platform. The SNS datastore filename will correspond to the Session Host name for correlation by OSS/Recon in the datastore repository.

Procedures are supplied (by RTSSW) to build and parse the mailbox headers for the datastore message data. (Refer to section 7.3 in Appendix I).

The datastore data will be buffered by bytes, not by specific Ada data type. The Datastore object will supply the procedures necessary for converting the datastore information to the byte form. The return to datastore partition software will need to convert the 'bytes' to the appropriate Ada data type.

On return to datastore, all the data for a partition will be buffered together in a single mailbox message.

FREEZE mode, submode of Hold

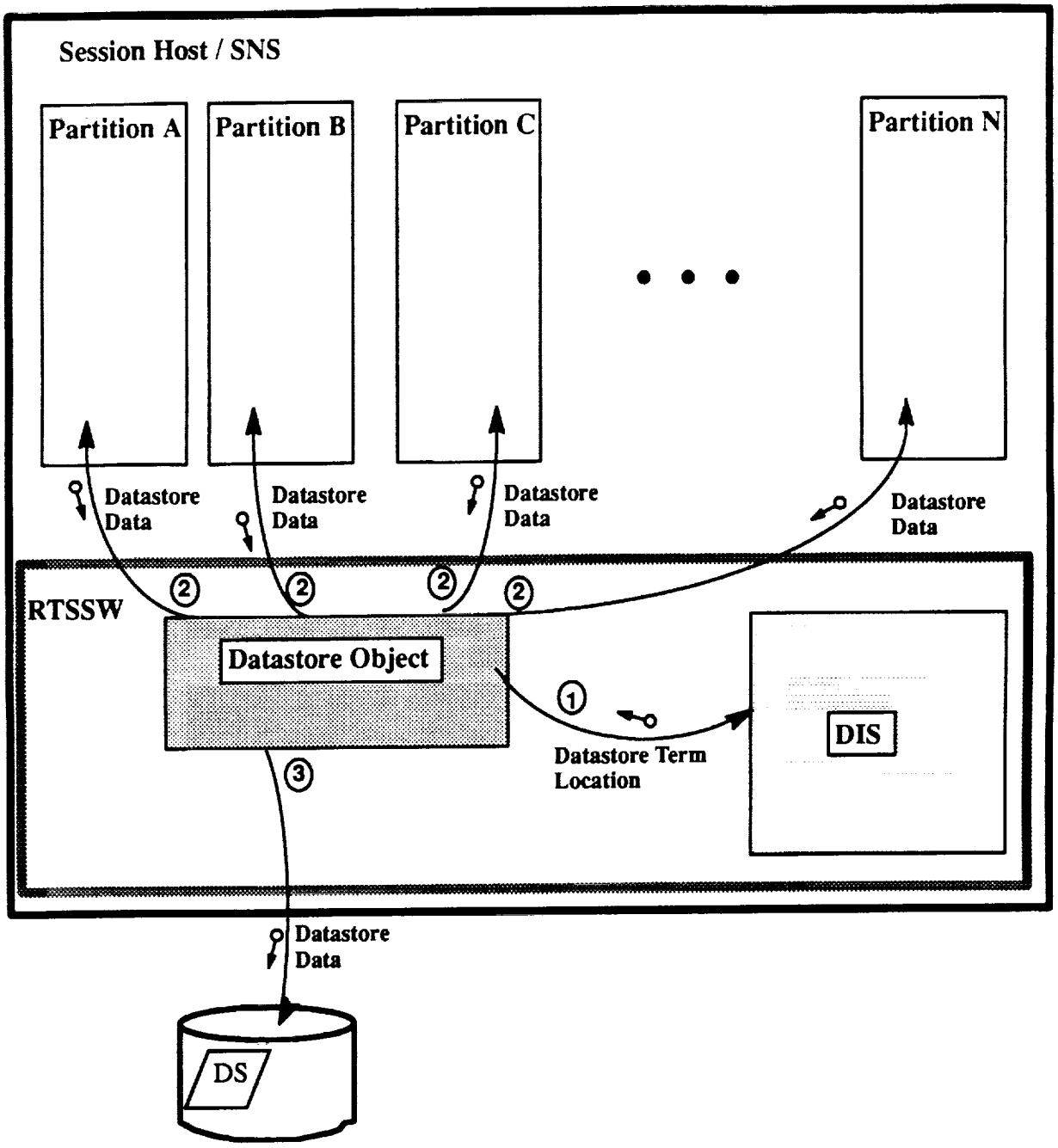


Figure 4.6–1. Taking a Datastore

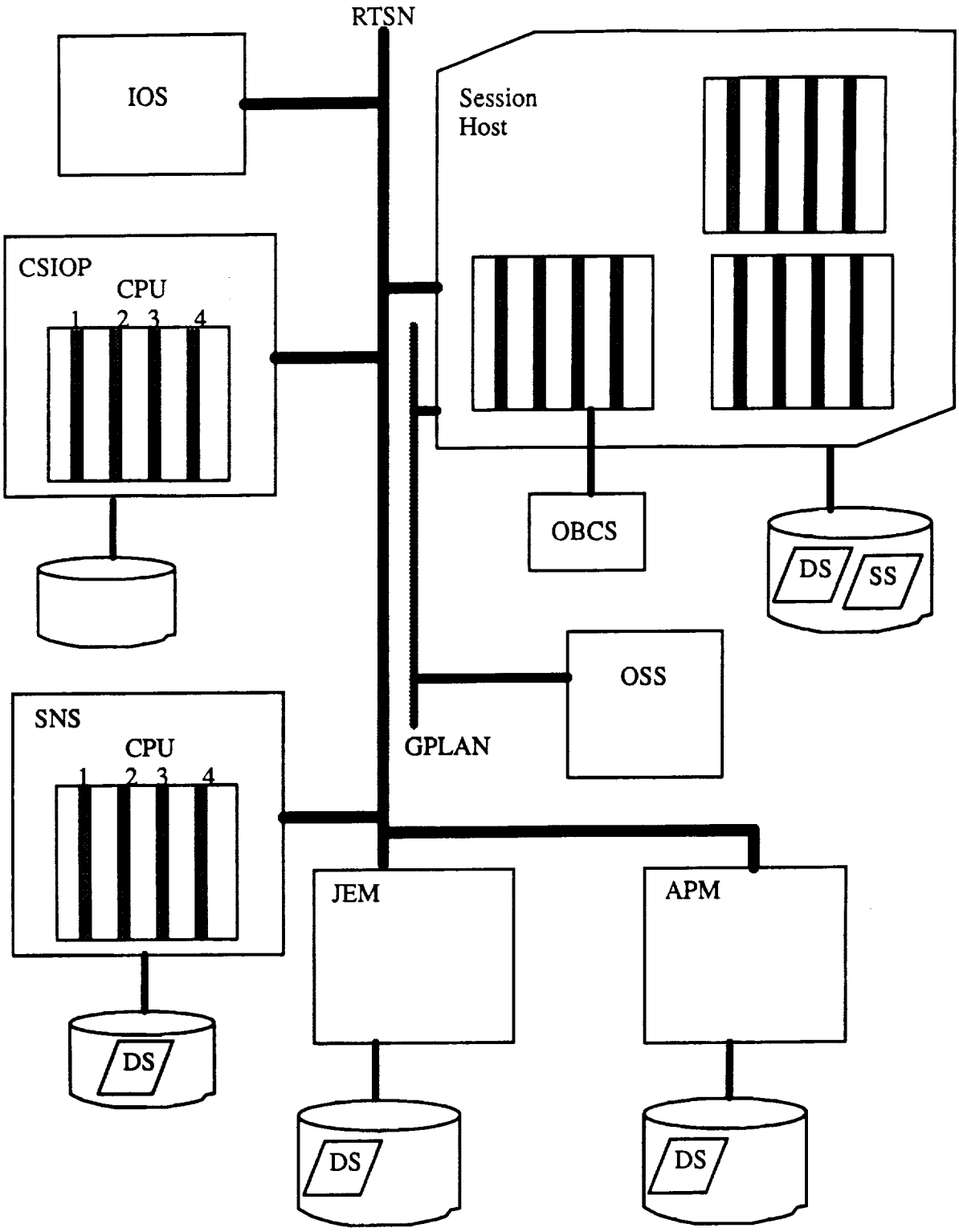
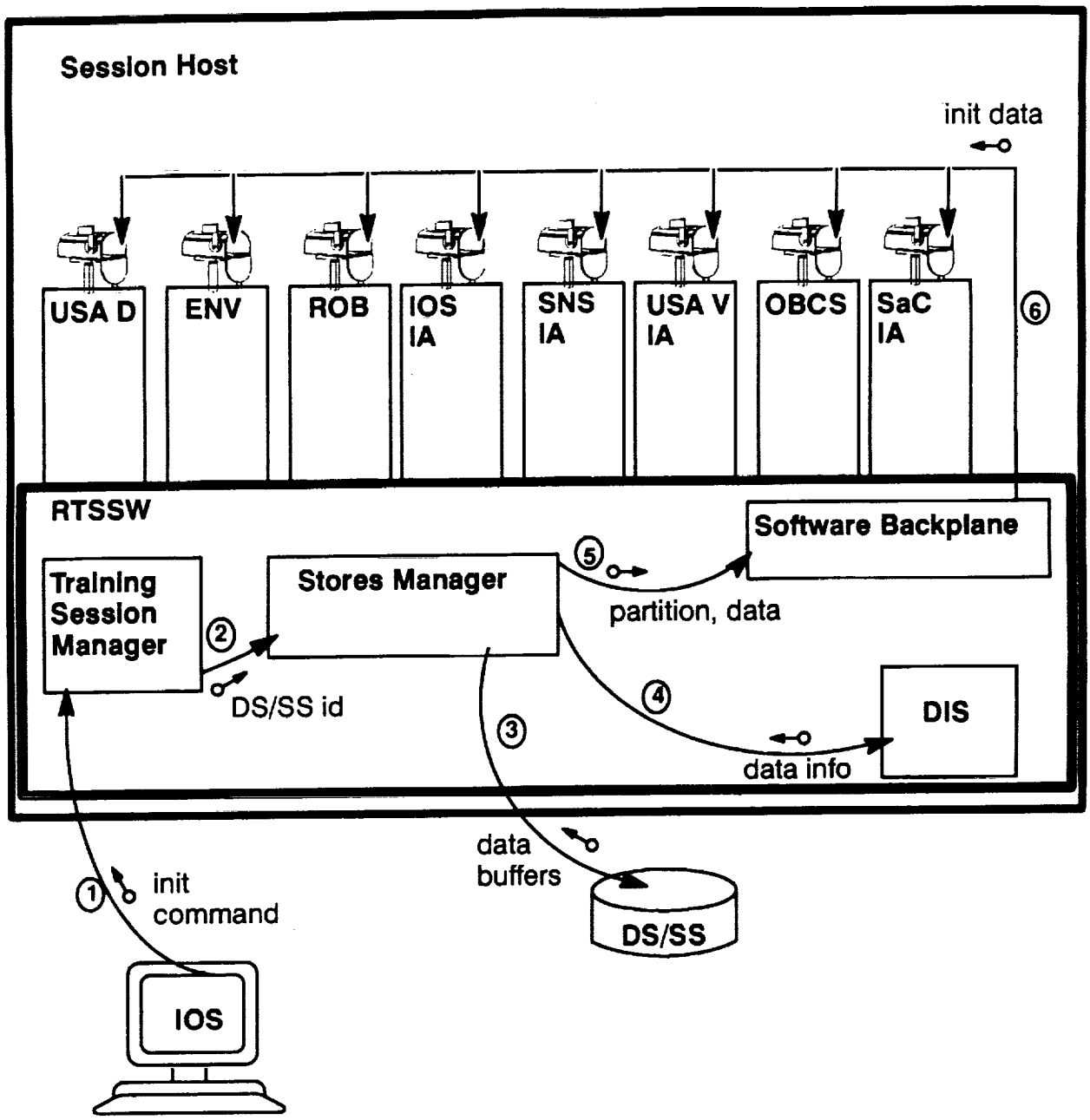


Figure 4.6–2. Physical Layout of Training Session



Note: Initialization to a Datastore/Safestore will occur in an analogous manner on the SNS platform.

Figure 4.6–3. Initialization to Datastore/Safestore

4.7. Safestore

The following sections provide a textual description of safestore activities and requirements. Figure 4.7–1 depicts how a safestore occurs. Figure 4.6–3 in the preceding section depicts a return to a safestore.

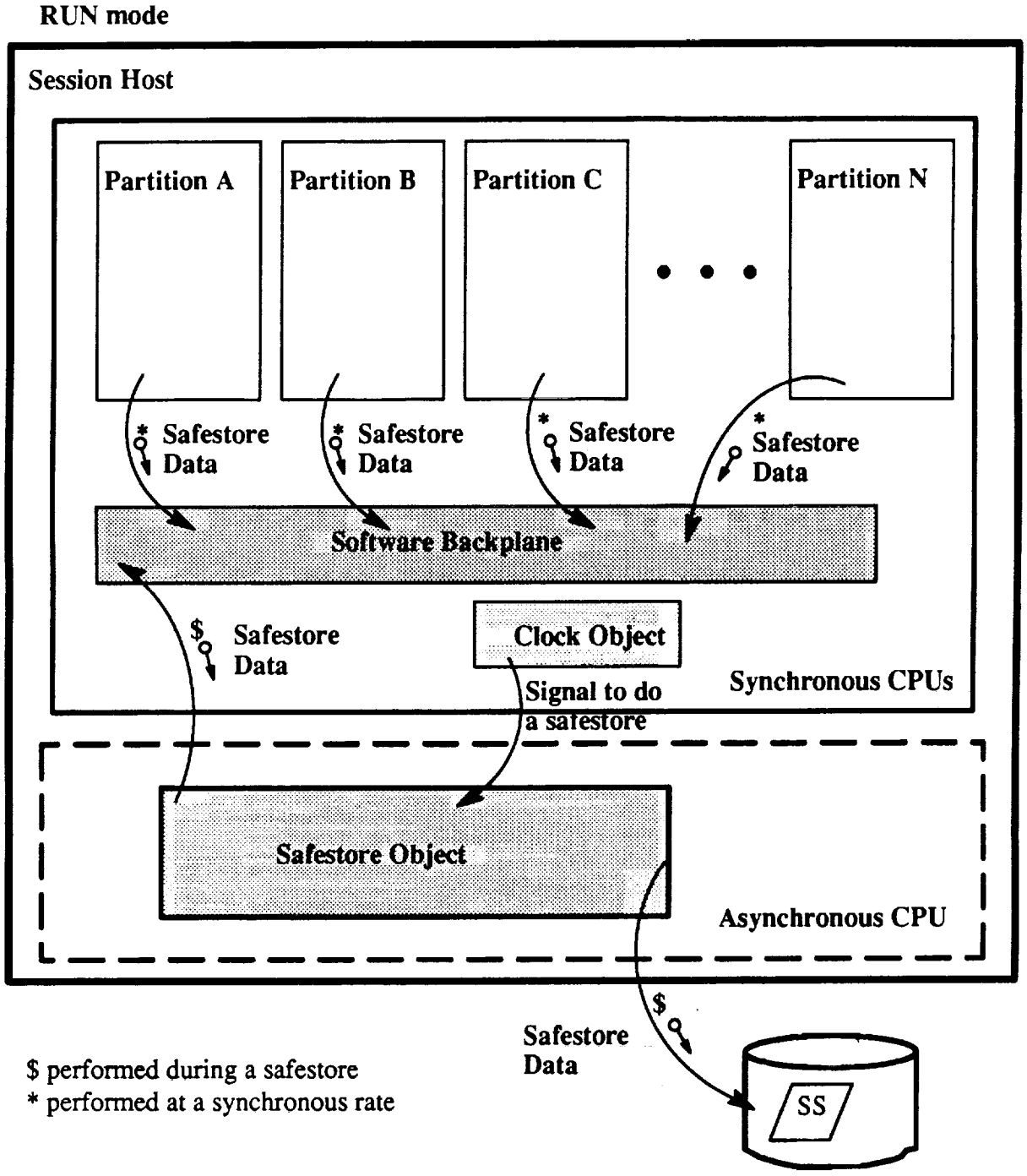


Figure 4.7–1. Taking a Safestore

Safestore is taking a snapshot of the simulation environment at consistent intervals during the simulation run mode. The purpose of the safestore is for recovery following an expected termination of the simulation. A recovery to a stable point before the termination occurred can be accomplished by initializing first to the last datastore or initialization point and then applying one of the last four safestores. Note that a safestore set is much smaller than a datastore set.

4.7.1 Perform a Safestore

The Safestore interval defaults to 15 minutes. An instructor may set the safestore interval to a different value via the IOS. The IOS checks the validity of the specified interval and then transmits the valid safestore interval to RTSSW. Valid intervals range from a minimum of five minutes to a maximum of fifteen minutes in increments of a minute. RTSSW sets the safestore interval as required/requested. Each partition produces safestore messages at a consistent rate (minimum of 1 hertz). The safestore object determines the expiration of the safestore interval. The Safestore object collects all safestore messages from the partitions. The software backplane mechanism ensures that the safestore messages are time–homogeneous at the 1 hertz rate. The Safestore object buffers the data and outputs it to disk.

4.7.2 Return to a Safestore

The IOS receives a return to safestore command and safestore id from the instructor. The IOS sends RTSSW the return to safestore command and id. The safestore object opens the safestore file and reads the safestore data from disk. The safestore object parses the data and places it in the mailbox for all appropriate partitions. Each partition processes (inputs) its mailbox message and self–initializes to the safestore state.

4.7.3 Partition Requirements

For a partition to successfully be involved in a safestore event, the following rules must be adhered to:

- Register safestore messages with the software backplane.
- Output safestore messages consistently at a minimum of 1 hertz.
- Have a mailbox.
- Process safestore data from the mailbox during a 'return to safestore'.
- Self–initialize to the safestore state.

4.7.4 Safestore Notes

Four (4) safestores are maintained per training session.

CSIOP does not perform a safestore.

SNS does not perform a safestore.

Propulsion, Environment, & GNC (on the Session Host) produce safestore data.

Safestore files are not kept after a session is normally terminated.

With safestore object on an asynchronous CPU, the safestore does not disturb the RMS algorithms. However, the safestore interval software interrupt may not be received immediately if the asynchronous CPU is 'busy'.

An instructor may 'protect' one of the four safestores during a session. The protected safestore will not be overwritten.

The safestore interval object will need to be part of the asynchronous simulation in order to be aware of simulation modes (to reset after certain modes and not issue interrupt during a freeze).

To reset to a safestore, first an initialization to the original initialization point (or datastore point) is performed followed by the application of the latest safestore.

On initialization to a new datastore point (or initialization point), previous safestores are essentially flushed. New safestores will relate to the current initialization point.

4.8. INTERFACE AGENTS

4.8.1 INTRODUCTION

This discussion of interface agents is limited to those innerface agents in the Full Task Trainer (FTT) of the Space Station Verification and Training Facility (SSVTF). In particular, this discussion is limited to interface agents that are hosted on assets with SVM.

To aid in understanding what an interface agent is and what is does (and maybe get some inkling how an interface agent should do its work), the following background material about assets and interface agents is provided.

4.8.1.1 What is an Asset ??

An asset is an SSVTF FTT hardware entity attached to the real–time simulation network (RTSN) which can be used as an element of a training session. Table 4.8–1 provides a list of the FTT assets and how many assets may be configured into a training session. An asset cannot be configured in more than one training session at a time. An asset may be configured into and out of an active training session.

FTT Asset Class Name	Asset Owner	Total # of Asset Instances	Min / Max # Configured in a Training Session	Asset with SVM	Asset without SVM
RT Session Computer (RTSC)	RTS	2	1 / 1	X	
SNS	SNS	1	0 / 1	X	
OTW Visual	VIS	1	0 / 1		X
CCTV Visual	VIS	1	0 / 1		X
IOS Work Station	IOS	14	1 / 14	X	
DMS String (with SIB)	OBCS	2	1 / 1		X
C&T String	USA D	1	0 / 1		X
Crew Station	USA V	3	0 / 3	X	
APM	APM	1	0 / 1	?	? (X)
JEM	JEM	1	0 / 1	?	? (X)
SMS	SMTF	1	0 / 1		X

Table 4.8–1 FTT Assets within a Training Session

Some assets contain the real–time system software (RTSSW) executive and communication environment known as the Simulation Virtual Machine (SVM). These assets include the RTSC, Crew Station host (CSIOP), and SNS. Since these assets contain the RTSSW environment, these assets are referred to as "assets with SVM" throughout this document. Some assets, such as IOS Work Stations, contain only the SVM communication environment. In this document, there is no differentiation made between assets with SVM and asset with only the SVM communication environment; these assets will be treated alike.

Most assets are black boxes which need to be stimulated in order to work properly in the FTT. Examples of these black box assets include the OTW and CCTV IGs and the DMS String. Since these assets do not contain the RTSSW environment, these assets are referred to as "assets without SVM" throughout this document.

All assets have the ability to operate with other assets when configured into a training session. During this "integrated" or "configured" mode of operation, an asset may communicate with one or more other assets.

Some assets have an additional ability to operate by themselves (in a "standalone" mode of operation). These assets include the SNS, OTW and CCTV IGs, IOS Work Stations, DMS String, and SMTF. (It is expected that the APM and JEM simulators will also have the capability of standalone operation.)

4.8.1.2 What is an Interface Agent ??

An interface agent is the software that provides model data from one asset to another through a controlled interface. In essence, an interface agent provides an abstraction of its parent asset. The asset providing the interface agent is called the "parent asset". The asset where the interface agent resides is called the "host asset". Note that the location of an interface agent (its host asset) depends on the parent asset's requirements for integrated and standalone modes of operation and whether the parent asset is an asset with or without SVM; the general rule is that interface agents will only reside in assets with SVM.

- Figure 4.8–1 provides a simple example of an interface agent between two assets. In this figure, Asset B has some need (requirement) to use some data produced by Asset A. (Let's postpone discussions about implementing an interface agent until later.) In order to support Asset B's need for data, Asset A employs an interface agent to provide Asset B with the required interface to Asset A. When Asset B needs some data produced by Asset A, Asset B uses the interface agent to get that data. Note that in this simple example, Asset B is an asset with SVM.
- Figure 4.8–2 provides a general association diagram of an interface agent (a non–IOS interface agent). In this figure, the interface agent is effecting a pass–thru interface between its host asset (Asset A) and its parent asset (Asset B). The interface agent exchanges data with some of Asset A's models (called Partition A, Partition Y, and Partition Z). The interface agent receives malfunction and enter value requests from an IOS interface agent. The interface agent receives add/drop asset commands from its Platform Manager, and returns the asset add/drop status to both the Platform Manager and a Status and Control (SaC) agent. Note that in this example, Asset A is an asset with SVM.

  An interface agent may play one of two roles while controlling virtually all information transmitted between its parent asset and host asset. These two roles are:

  A. Simulating an asset that is not configured in the training session

  B. Effecting a pass–thru interface with an asset that is configured in the training session

- In a training session where a given asset is not present, as shown in Figure 4.8–3, the interface agent will simulate the interaction between the "missing" parent asset and the host asset at some fidelity (the minimum fidelity required for meeting the resource and consumable demands of the host asset). The fidelity of asset simulation will depend on the requirements imposed on and capabilities of the interface agent. (Note that the interface agent should use static values wherever possible when simulating its parent asset's interface.)
- In a training session where a given asset is present, as shown in Figures 4.8–4 and 4.8–5, the interface agent will effect a high fidelity interchange between the "present" parent asset and the host asset. Note that Figure 4.8–4 shows the communication path when both assets have SVM, and Figure 4.8–5 shows the communication path when only one asset has SVM.

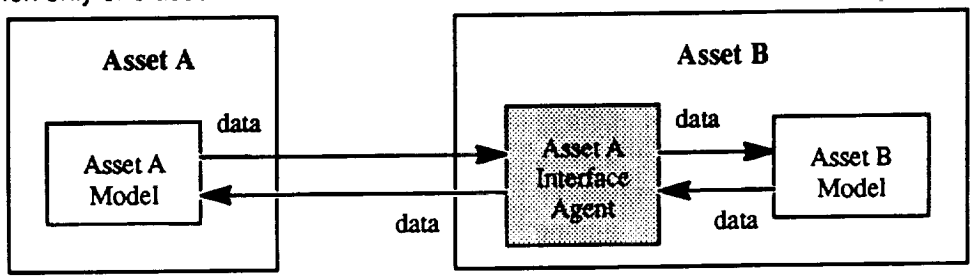


Figure 4.8–1 Simple Example of an Interface Agent

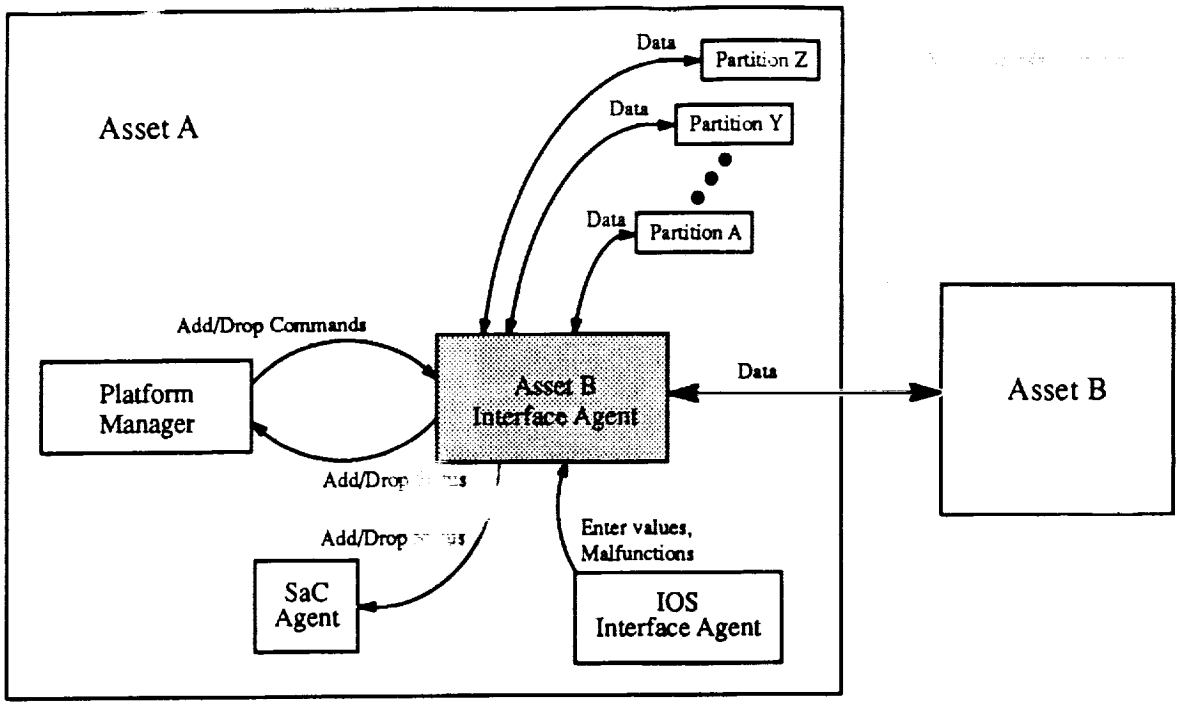
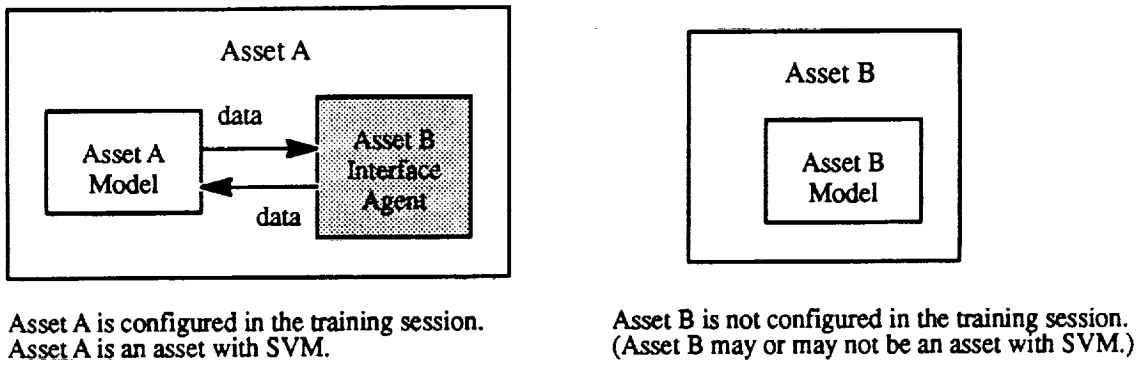


Figure 4.8–2 General Interface Agent Association Diagram



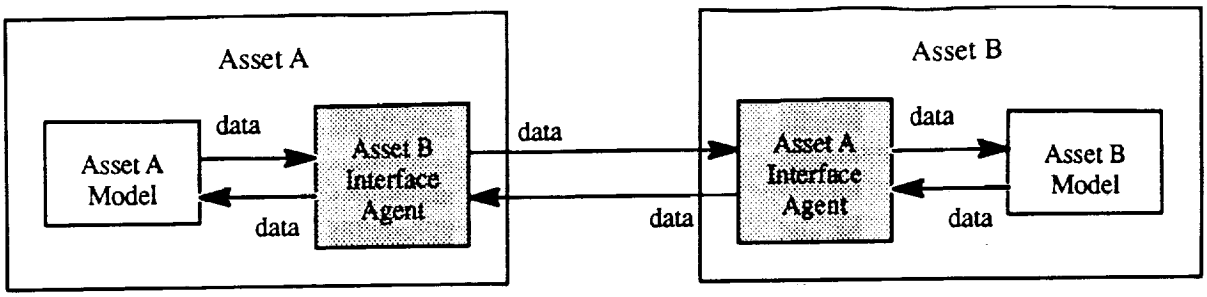
Asset A is configured in the training session.
Asset A is an asset with SVM.

Asset B is not configured in the training session.
(Asset B may or may not be an asset with SVM.)

Figure 4.8–3 Interface Agent Playing Roll as Asset Simulation

Figure 4.8–6 provides a general state diagram for an interface agent. On startup, the interface agent defaults to simulating the parent/host asset interface. An interface agent effects the pass–thru interface when the parent asset has been successfully integrated into the host asset's training session. Allowing an interface agent to (easily) switch between the roles of an interface simulator and an interface stimulator provides for a well–controlled, constant interface relationship between the assets.

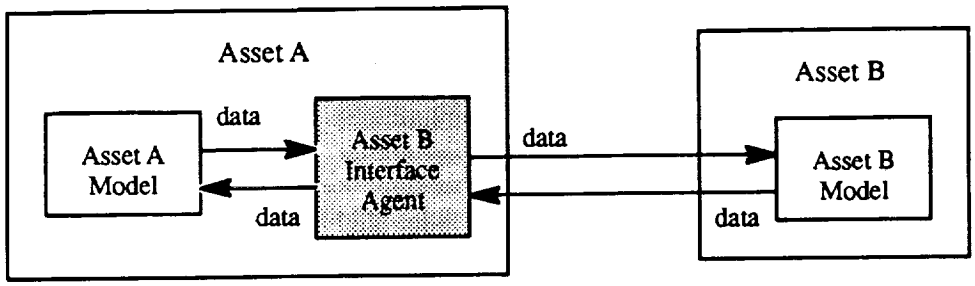
Figure 4.8–7 shows a diagram of a better–detailed communication path between two assets with SVM when both assets are integrated in the same training session. Note that two interface agents are employed.



Asset A is configured in the training session.
Asset A is an asset with SVM.

Asset B is configured in the training session.
Asset B is an asset with SVM.

Figure 4.8–4 Interface Agent Playing Role as Asset (with SVM) Stimulator



Asset A is configured in the training session.
Asset A is an asset with SVM.

Asset B is configured in the training session.
Asset B is an asset without SVM.

Figure 4.8–5 Interface Agent Playing Role as Asset (without SVM) Stimulator

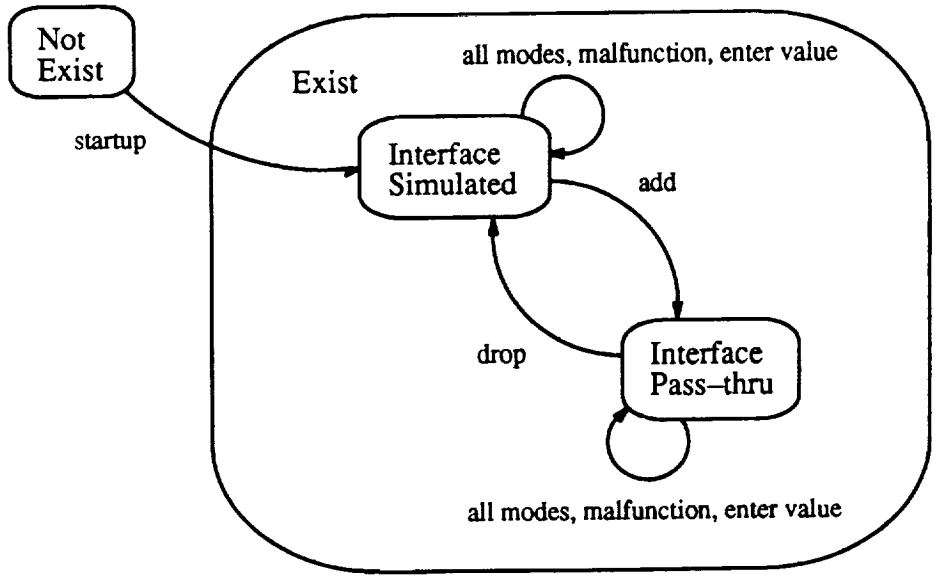


Figure 4.8–6 General Interface Agent State Diagram

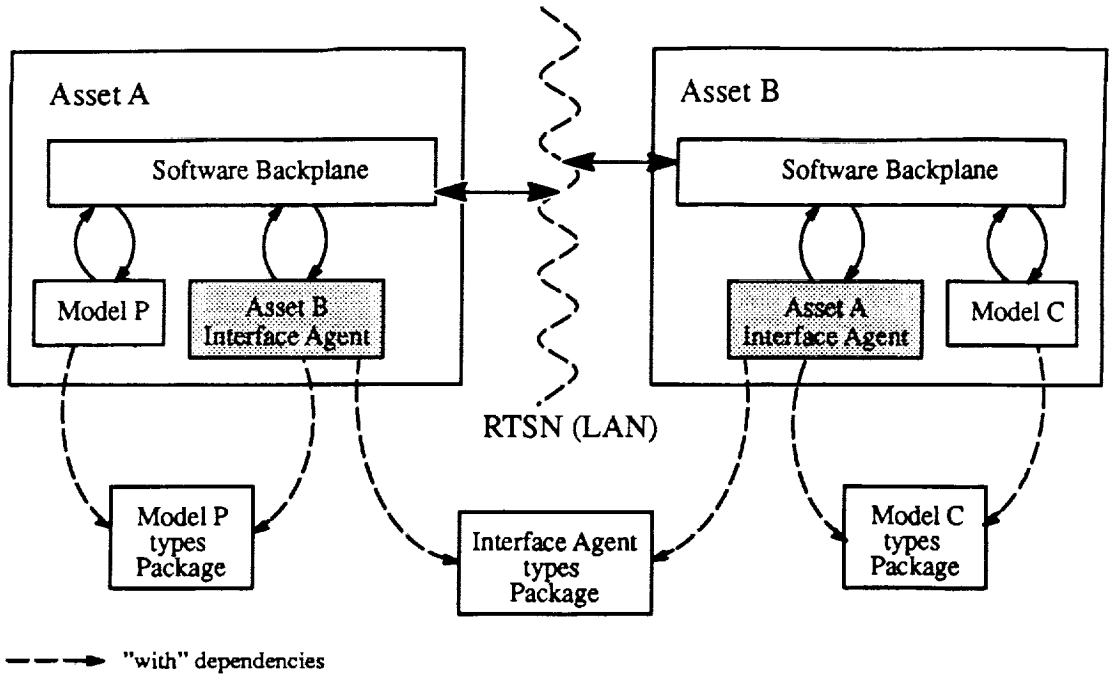


Figure 4.8–7 General Interface Agents within Two Assets with SVM

4.8.2 INTERFACE AGENT GENERAL NOTES

Rules

1. All output message types in a partition's interface are defined via Interface Definition Types Packages. The Distributed Identifier Specification (DIS) creates identifiers (Message_IDs) for the messages listed in a partition's interface definition packages. The SVM communication software uses these message identifiers to determine the location of the output messages. A partition creates the actual output message object (the data itself) using the SVM communication software. Interfaces between partitions are effected by registering input and output identifiers with the SVM communication software.
2. It is greatly desired that a training session should not stop when an asset goes down or goes improperly off-line.
3. SGMT will be provided on demand from a generic clock model.

Assumptions

1. Generally, do not mix data at different rates in the same message on the LAN. The idea here is to ensure that high–rate data is not starved by waiting for low–rate data to be ready for transfer in the same data block. One simple work around is to issue the data block at the higher rate, and only update the low–rate data as necessary in the block. Of course, the receiver must be ready to deal with getting multi–rate data in a data block.
2. An interface agent (or at least some part of an interface agent) will be packaged as (and treated like) a partition. An interface agent will register with SVM communication and executive software in the same manner as a partition. An interface agent can do sub–scheduling within itself where needed. (This capability for sub–scheduling within a partition may be provided by the SVM executive software.)
3. Generally, there will be only one interface agent per asset class. The interface agent shall control virtually all communication between the host asset and the parent assets.
4. An interface agent is responsible for resolving word size or word definition differences between the host and parent asset via bit–fiddling, byte–swapping, or whatever other means are available to the interface agent. If the byte–swapping or bit–fiddling is a general problem of the asset interface (i.e., it's a problem for every basic data type), then the Network Services part of Connectivity might be able to perform these actions on all data transferred across the interface. (Network Services will not be able to handle type–specific conversions – it's all or nothing.)
5. An interface agent helps to optimize FDDI packets (helps reduce amount of little packets sent across LAN), thus helping the RTSN and GP LAN to provide better response to every user.
6. Mode transitions commands (from the master Platform Manager) should be "disabled" during asset add and asset drop activities. This will allow the asset to be added or dropped in a "stable" mode. Also, the master Platform Manager should not issue asset add and drop commands while a Datastore or Initialization is occurring.

Other Notes

All interface agents provide the following four main capabilities:

1. Simulating an asset interface
2. Effecting pass–thru interface for an asset (with or without SVM)
3. Adding an asset
4. Dropping an asset

Each capability is discussed at length in the following sections. Whether an interface agent is simulating an asset interface or effecting a pass–thru interface, it must still deal with the issues of communication, modes, malfunctions, and user–requested data entry.

4.8.3 INTERFACE AGENT FOR ASSET WITH SVM

4.8.3.1 Simulating Interface

4.8.3.1.1 Communication

The interface agent shall use the SVM communication software to communicate with other partitions. No moding of the parent asset is performed during the asset interface simulation.

During the startup phase, the interface agent must set up all communication paths (for simulation of the interface as well as effecting a pass–thru interface) before beginning its simulated interface behavior. This allows for a quick (and controlled) behavioral change into the pass–thru interface behavior when the parent asset is added.

4.8.3.1.2 Moding

The portion of the interface agent that is periodic (the partition portion) shall respond to modes (just like a normal partition). The interface agent's particular interface modelling responsibilities depend on the required fidelity of the interface simulation. Note that interface agents must read their mailbox, perform an asset add upon request, and return the success/fail status of the add to OSS (SaC) and the master Platform Manager. In the unlikely event that an asset drop is requested, the interface agent should return an error status (i.e., requested asset is not currently configured in training session) to both the OSS (SaC) and master Platform Manager.

It is anticipated that interface agents (for assets with SVM) will not have to do anything for Step Ahead while simulating the interface.

When the "simulated interface" contains data that should be Datastored, the interface agent must register each Datastore item with the DIS (via DIS–related packages). The interface agent must provide the software to process the return–to–datastore data from its mailbox.

When the "simulated interface" contains data that should be Safestored, the interface agent must register each Safestore item with the DIS (via DIS–related packages) as well as with the SVM communication software. Interface agents must update these safestore terms at a minimum of once a second (at 1 hertz). The interface agent must provide the software to process the return–to–safestore data from its mailbox.

Upon entering TERMINATE mode, an interface agent should stop simulating the asset interface. (Essentially, the interface agent should quit.) There is no requirement for an interface agent to shutdown its parent asset when the asset is not configured in the training session.

4.8.3.1.3 Malfunctions

When the "simulated interface" contains data that is affected by malfunctions, the interface agent must register each malfunction with the DIS (via DIS–related packages). An interface agent must provide the software to effect the malfunction in the simulated interface. An interface agent should inform the IOS when a malfunction request (for the simulated interface) cannot be serviced.

In the event that some malfunction is processed by both the interface agent (during interface simulation) and its parent asset (when configured in the training session), the interface agent shall issue that malfunction request to its parent asset.

Maybe malfunctions shouldn't be handled (anything really done) by interface agent when it is simulating the asset interface: Why should an interface agent care about malfunctioning something in a low–fidelity interface ??

Maybe the interface agent (for a parent asset with SVM) won't have to deal with malfunction logic: the IOS may not allow selection of malfunctions which are hosted (belong to) an asset with SVM which isn't configured into the training session.

4.8.3.1.4 User–Requested Data Entry

When the "simulated interface" contains data that can be over–written by an instructor or operator (via user–requested data entry), the interface agent must register each item (targeted for a controlled value override) with the DIS (via DIS–related packages). An interface agent must provide the software to enter the user–supplied data (from its mailbox) to the simulated interface. An interface agent should inform the IOS when an enter value request cannot be serviced.

4.8.3.2 Effecting Pass–Thru Interface

4.8.3.2.1 Communication

The interface agent shall use the SVM communication software to communicate with other partitions. The communication paths do not need to be set up at this point, since they were set up during the startup phase.

4.8.3.2.2 Moding

All mode transition commands (including requested mode and mode–specific parameters) shall be sent directly from the master Platform Manager to the asset's Platform Manager; the interface agent is not involved with this message transfer. (Note that each asset with SVM shall have a Platform Manager.)

The portion of the interface agent that is periodic (the partition portion) shall respond to modes (just like a normal partition). The interface agent's particular interface modelling responsibilities depend on the required fidelity of effecting the pass–thru interface. Note that interface agents must read their mailbox, perform an asset drop upon request, and return the success/fail status of the drop to OSS (SaC) and the master Platform Manager. In the unlikely event that another asset add is requested, the interface agent should return an error status (i.e., requested asset is not currently configured in training session) to both the OSS (SaC) and master Platform Manager.

It is anticipated that interface agents (for assets with SVM) will not have to do anything for Step Ahead while effecting a pass–thru interface.

Since models within the asset shall register Datastore and Safestore terms (local to that asset), the interface agent will not have to do anything for Datastore or Safestore. While effecting a pass–thru interface, the interface agent will not have to update its "simulated interface" data terms.

4.8.3.2.3 Malfunctions

For an asset with SVM, the interface agent is not required to register malfunctions (as long as the interface agent does not have to malfunction the "simulated interface").

In the event that some malfunction is processed by both the interface agent (during interface simulation) and its parent asset (when configured in the training session), the interface agent shall issue that malfunction request to its parent asset, upon request by the IOS.

4.8.3.2.4 User–Requested Data Entry

For an asset with SVM, the interface agent is not required to register data items targeted for a controlled value override.

4.8.3.3 Adding Asset

The master Platform Manager (Asset Manager), upon command from the OSS, will issue an "Add Asset" request to the interface agent. This request will identify which asset instance (i.e., which unique asset) should be added into the training session. An asset may be added only during FREEZE and RUN modes. Note that prior to issuing the "Add Asset" request, the master Platform Manager will have ensured (with OSS SMaCT or SaC help ?) that the asset has successfully completed its Startup Activity (including the PROGRAM ELAB-

ORATION, SETUP/REGISTER I/O, and CREATE DATA steps) and that it's communicating on the RTSN. The master Platform Manager shall also be responsible for ensuring that an asset add will not occur during a Datastore operation. The interface agent should be simulating the asset interface at this time.

Responses of interface status (Asset Add successful, Asset Add failed, etc.) during an Asset Add shall be sent from the interface agent to both SaC and the master Platform Manager (Asset Manager).

When adding an asset during run, one–way communication is first established with the asset. Data is passed to the asset so that it can initialize itself with the ongoing simulation. When everything is synchronized and it is time to join the asset to the simulation, the communication becomes two–way and the interface agent acts as a pass–thru for the data transfer.

When a new asset is being added, there may be a need for a "controls not in agreement" capability. this would involve the IOS, the interface agent, and the actual asset. This capability would allow the asset to be "in configuration" prior to being added so that a large jump would not be detected when they were actually added (if the asset was not near the current simulated configuration). *[Aside: according to SET team, we will ignore the controls not in agreement capability.]*

4.8.3.4 Dropping Asset

The master Platform Manager (Asset Manager), upon command from the OSS, will issue a "Drop Asset" request to the interface agent. This request will identify which asset instance (i.e., which unique asset) should be dropped from the training session. An asset may be dropped only during FREEZE, RUN and TERMINATE modes. The master Platform Manager shall be responsible for ensuring that an asset drop will not occur during a Datastore operation. The interface agent should be effecting a pass–thru interface at this time.

Responses of interface status (Asset Drop in progress, Asset Drop successful, Asset Drop failed, etc.) during an Asset Drop shall be sent from the interface agent to both SaC and the master Platform Manager (Asset Manager).

4.8.4 INTERFACE AGENT FOR ASSET WITHOUT SVM

4.8.4.1 Simulating Interface

4.8.4.1.1 Communication

The interface agent shall use the SVM communication software to communicate with other partitions. No moding of the parent asset is performed during the asset interface simulation.

During the startup phase, the interface agent must set up all communication paths (for simulation of the interface as well as effecting a pass–thru interface) before beginning its simulated interface behavior. This allows for a quick (and controlled) behavioral change into the pass–thru interface behavior when the parent asset is added.

4.8.4.1.2 Moding

The portion of the interface agent that is periodic (the partition portion) shall respond to modes (just like a normal partition). The interface agent's particular interface modelling responsibilities depend on the required fidelity of the interface simulation. Note that interface agents must read their mailbox, perform an asset add upon request, and return the success/fail status of the add to OSS (SaC) and the master Platform Manager. In the unlikely event that an asset drop is requested, the interface agent should return an error status (i.e., requested asset is not currently configured in training session) to both the OSS (SaC) and master Platform Manager.

It is anticipated that interface agents (for assets without SVM) will not have to do anything for Step Ahead while simulating the interface.

When the "simulated interface" contains data that should be Datastored, the interface agent must register each Datastore item with the DIS (via DIS–related packages). The interface agent must provide the software to process the return–to–datastore data from its mailbox.

When the "simulated interface" contains data that should be Safestored, the interface agent must register each Safestore item with the DIS (via DIS–related packages) as well as with the SVM communication software. Interface agents must update these safestore terms at a minimum of once a second (at 1 hertz). The interface agent must provide the software to process the return–to–safestore data from its mailbox.

Upon entering TERMINATE mode, an interface agent should stop simulating the asset interface. (Essentially, the interface agent should quit.) There is no requirement for an interface agent to shutdown its parent asset when the asset is not configured in the training session.

4.8.4.1.3 Malfunctions

When the "simulated interface" contains data that is affected by malfunctions, the interface agent must register each malfunction with the DIS (via DIS–related packages). An interface agent must provide the software to effect the malfunction in the simulated interface. An interface agent should inform the IOS when a malfunction request (for the simulated interface) cannot be serviced.

The IOS shall send malfunction requests to the interface agent (in accordance with the malfunction's DIS registration). During RUN and FREEZE modes, the interface agent shall read its mailbox and effect malfunction (for the simulated interface) as required.

In the event that some malfunction is processed by both the interface agent (during interface simulation) and its parent asset (when configured in the training session), the interface agent shall issue that malfunction request to its parent asset.

4.8.4.1.4 User–Requested Data Entry

When the "simulated interface" contains data that can be over–written by an instructor or operator (via user–requested data entry), the interface agent must register each item (targeted for a controlled value override)

with the DIS (via DIS–related packages). An interface agent must provide the software to enter the user–supplied data (from its mailbox) to the simulated interface.

The IOS shall send enter value requests to the interface agent (in accordance with the data item's DIS registration). During RUN and FREEZE modes, the interface agent shall read its mailbox and process the enter value requests (as allowed by the peculiar capabilities provided by the asset without SVM). The interface agent shall inform the IOS when an enter value request (for the simulated interface) cannot be serviced.

4.8.4.2 Effecting Pass–Thru Interface

4.8.4.2.1 Communication

The interface agent shall use the SVM communication software to communicate with other partitions.

During the startup phase, the interface agent must set up all communication paths (for simulation of the interface as well as effecting a pass–thru interface) before beginning its simulated interface behavior. This allows for a quick (and controlled) behavioral change into the simulated interface behavior when the parent asset is dropped.

4.8.4.2.2 Moding

When an asset without SVM is integrated, its interface agent must deal with mode transition logic: it should know how to "mode" its parent asset. In a sense, the interface agent acts as a pseudo–Platform Manager for the asset.

During INITIALIZATION (including Self Init and System Init) the interface agent shall receive initialization data from its mailbox and apply it to the host–to–parent transfer buffers (as appropriate). Thus, the interface agent shall stimulate asset so that it receives and processes the initialization data until the asset reaches a steady state.

During FREEZE (and HOLD and STEP AHEAD), the interface agent must keep the asset running (either continue data transfers, or command the asset to freeze). In some cases, additional messages may be sent to trick asset into its "freeze" logic. Despite the method used, the interface agent should take care of all special processing to ensure that the asset is frozen when the training session enters FREEZE.

The interface agent must register all necessary DATASTORE terms with the DIS. These terms should be terms within the asset's pass–thru interface or derived from data within the asset's pass–thru interface.

During RUN, the interface agent should communicate with its asset as required to effect the pass–thru interface. On a FREEZE to RUN transition, the interface agent should take care of all special processing to ensure that the asset begins interface processing when the training session enters RUN mode.

If required, the interface agent must register all necessary SAFESTORE terms with the DIS. Note that it is not generally expected that assets without SVM will have safestore data.

4.8.4.2.3 Malfunctions

The interface agents must register all malfunctions for its parent asset. This is necessary since the parent asset is unable to register malfunctions by itself. Each malfunction must be registered with the DIS.

The IOS shall send malfunction requests to the interface agent (in accordance with the malfunction's DIS registration). During RUN and FREEZE modes, the interface agent shall read its mailbox and malfunction the asset as required. The interface agent must know how to effect malfunctions on its parent asset. The interface agent shall inform the IOS when a malfunction request cannot be serviced.

4.8.4.2.4 User–Requested Data Entry

The interface agent must register each data item (targeted for a controlled value override) with the DIS. This is necessary because the parent asset is unable to register these terms by itself.

The IOS shall send enter value requests to the interface agent (in accordance with the data item's DIS registration). During RUN and FREEZE modes, the interface agent shall read its mailbox and process the enter value requests (as allowed by the peculiar capabilities provided by the asset without SVM). The interface agent shall inform the IOS when an enter value request cannot be serviced.

4.8.4.3 Adding Asset

The master Platform Manager (Asset Manager), upon command from the OSS, will issue an "Add Asset" request to the interface agent. This request will identify which asset instance (i.e., which unique asset) should be added into the training session. An asset may be added only during FREEZE and RUN modes. Note that prior to issuing the "Add Asset" request, the master Platform Manager will have ensured (with OSS SMaCT or SaC help ?) that the asset has successfully completed its Startup Activity (whatever this means for the asset without SVM) and that it's communicating on the RTSN. The master Platform Manager shall also be responsible for ensuring that an asset add will not occur during a Datastore operation. The interface agent should be simulating the asset interface at this time.

Responses of interface status (Asset Add successful, Asset Add failed, etc.) during an Asset Add shall be sent from the interface agent to both SaC and the master Platform Manager (Asset Manager).

When adding an asset during run, one–way communication is first established with the asset. The interface agent shall deal with foreign connections, as required. Data is passed to the asset so that it can initialize itself with the ongoing simulation. When everything is synchronized and it is time to join the asset to the simulation, the communication becomes two–way and the interface agent acts as a pass–thru for the data transfer.

When a new asset is being added, there may be a need for a "controls not in agreement" capability. this would involve the IOS, the interface agent, and the actual asset. This capability would allow the asset to be "in configuration" prior to being added so that a large jump would not be detected when they were actually added (if the asset was not near the current simulated configuration). *[Aside: according to SET team, we will ignore the controls not in agreement capability.]*

4.8.4.4 Dropping Asset

The master Platform Manager (Asset Manager), upon command from the OSS, will issue a "Drop Asset" request to the interface agent. This request will identify which asset instance (i.e., which unique asset) should be dropped from the training session. An asset may be dropped only during FREEZE, RUN and TERMINATE modes. The master Platform Manager shall be responsible for ensuring that an asset drop will not occur during a Datastore operation. The interface agent should be effecting a pass–thru interface at this time.

Responses of interface status (Asset Drop in progress, Asset Drop successful, Asset Drop failed, etc.) during an Asset Drop shall be sent from the interface agent to both SaC and the master Platform Manager (Asset Manager).

If a hardware device is attached, the interface agent may need to shutdown that device as part of the asset's drop procedure.

4.9. Asynchronous I/O

Asynchronous I/O provides real–time disk i/o operations for real–time models. Disk i/o operations are usually time consuming operations. The calling model must wait for the i/o operation to complete before it can continue processing. In a real–time simulation, such i/o delays cause overruns to occur and can not be tolerated. However, asynchronous i/o permits real–time models to 'post' i/o operations to a shared memory area. A partition in the asynchronous processor which is not bound by real–time time constraints processes the posted i/o operations. The real–time model making the request picks up the result of the i/o request on a subsequent execution. For the model, the time consumed for i/o operations is that for memory to memory data transfers which is much more efficient than memory to/from disk data transfers.

Asynchronous I/O permits a real–time model to open a file for read access or write access, not both simultaneously. The data is accessed as in stream i/o. Sequential reads/writes of data in terms of bits/bytes is performed. The model is responsible for file format and data type information.

Real–time application models requiring disk access use the services of the Realtime_Io package. Realtime_Io operations post i/o requests in a shared memory buffer area. Several models, within the same or different CPUs, may make real-time i/o requests. Built into CPU 0 is Async_Io_Partition. Async_Io_Partition scans the shared memory buffer area for i/o requests and processes them appropriately (see figure 4.9–1).

Models requiring disk i/o should with package Realtime_IO. For each file that is to be operated on simultaneously, the model should Register a buffer area specific to the file in Set_Up. Update should be called in each simulation mode until a Status of Registered is received. Once the buffer for a file has been registered, the file may be opened (Open) or created (Create). Update should be called until a Status of Opened or Created is returned. Before data is read, a check should be made to ensure There_Is_Data_For the read and that the End_Of_File has not been reached. The Read may then be performed. Again Update should be called before the next call to any Realtime_Io service. Similarly, for a write, a check should be made to ensure There_Is_Room_For the data to be written in the buffer area for this file. The Write may then be performed and a call to Update made before another Realtime_Io operation is made. The file may be closed (Close) or

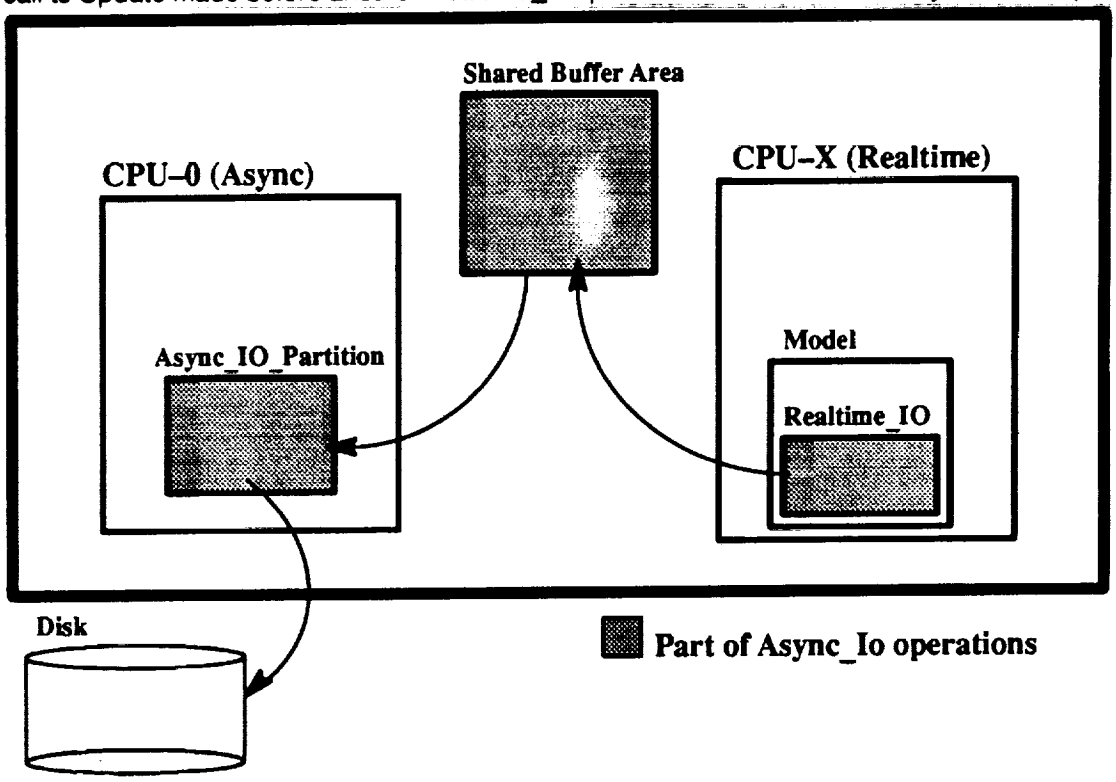


Figure 4.9–1 Asynchronous I/O Overview

deleted (Delete) following the completion of all requested read or write operations. Once a file has been closed or deleted, the Realtime_Io object may be used to operate on another file.

The Async_Io_Partition processes i/o operations posted by the Realtime_IO package. Async_IO_Partition executes at 1 hz. Thus, several i/o requests may have been posted by real-time models between Async_IO_Partition executions. Async_IO_Partition scans the shared memory buffer areas and processes any posted i/o operations. The response to a Realtime_Io call is not completed until Async_IO_Partition has executed. A requesting model may have to call Realtime_Io.Update several times before it receives a completed status. However, Reads may be done by the real-time model until its buffer area is depleted without an execution by Async_IO_Partition; similarly, writes may be performed until the buffer for the file is full without an execution by Async_IO_Partition.

The Realtime_Io package spec follows:

```
with System;
with Io_Exceptions;

package Realtime_Io is

    --|
    type File_Type is limited private;

    --|
    type File_Mode is (In_File, Out_File);

    --|
    type File_Size is (Small, Large); -- estimated buffer size needed

    --|
    type File_Status is (None, Error, Registering, Registered, Creating,
                         Created, Opening, Opened, Writing, Written,
                         Reading, Read, Closing, Closed, Deleting, Deleted);

    ------------------------------------------------------------------------
    --| Register: allows use of other Realtime_Io routines.
    --|
    --| ***  Register must be the first Realtime_IO routine  ***
    --| ***  called. It must be called during Set_Up.        ***
    ------------------------------------------------------------------------
    procedure Register (File : in out File_Type;
                        Mode : in     File_Mode := Out_File;
                        Size : in     File_Size := Small);

    ------------------------------------------------------------------------
    --| Update: allows Realtime_IO to update the File object.
    --|
    --| ***  Update must be the first Realtime_IO routine  ***
    --| ***  called in a given period for all modes other  ***
    --| ***  than Set_Up.                                  ***
    ------------------------------------------------------------------------
    procedure Update (File : in out File_Type);

    ------------------------------------------------------------------------
    --| Destroy: Kills an instance of File_Type.
    --|
    --| ***  Must call Register to use this instance again. ***
    ------------------------------------------------------------------------
    procedure Destroy (File : in out File_Type);

    ------------------------------------------------------------------------
    --| Create: creates a disk file of the supplied Name.
    ------------------------------------------------------------------------
```

```
    procedure Create (File : in out File_Type; Name : in String);

    ------------------------------------------------------------------------
    --| Open: opens a disk file of the supplied Name.
    ------------------------------------------------------------------------
    procedure Open (File : in out File_Type; Name : in String);

    ------------------------------------------------------------------------
    --| Close: closes the currently opened file.
    ------------------------------------------------------------------------
    procedure Close (File : in out File_Type);

    ------------------------------------------------------------------------
    --| Delete: deletes the currently opened file;
    ------------------------------------------------------------------------
    procedure Delete (File : in out File_Type);

    ------------------------------------------------------------------------
    --| Read: reads data from the currently opened file.
    ------------------------------------------------------------------------
    procedure Read (File         : in out File_Type;
                    Address      : in     System.Address;
                    Size_In_Bits : in     Natural);

    ------------------------------------------------------------------------
    --| Write: writes data to the currently opened file.
    ------------------------------------------------------------------------
    procedure Write (File         : in out File_Type;
                     Address      : in     System.Address;
                     Size_In_Bits : in     Natural);

    ------------------------------------------------------------------------
    --| Selector Functions
    ------------------------------------------------------------------------
    function Mode (File : in File_Type) return File_Mode;
    function Name (File : in File_Type) return String;
    function Status (File : in File_Type) return File_Status;
    function End_Of_File (File : in File_Type) return Boolean;
    function There_Is_Room_For (File : in File_Type; Number_Of_Bits : in Natural) return Boolean;
    function There_Is_Data_For (File : in File_Type; Number_Of_Bits : in Natural) return Boolean;

    ------------------------------------------------------------------------
    --| Exceptions this package can raise
    ------------------------------------------------------------------------
    Name_Error   : exception renames Io_Exceptions.Name_Error;
    Use_Error    : exception renames Io_Exceptions.Use_Error;
    Status_Error : exception renames Io_Exceptions.Status_Error;
    Mode_Error   : exception renames Io_Exceptions.Mode_Error;
    Device_Error : exception renames Io_Exceptions.Device_Error;
    End_Error    : exception renames Io_Exceptions.End_Error;
    Data_Error   : exception renames Io_Exceptions.Data_Error;
    --
    Mismatch_Error : exception;

------------------------------------------------------------------------
private

    type State;

    type File_Type is access State;

end Realtime_Io; -- package spec
```

```
-------------------------------------------------------------------------------
--|  Abstract:  This package provides a real-time interface for models wanting
--|             real-time write capabilities.
--|
--|  Warnings:  Async_Io_Partition and Realtime_Io are co-programs.
--|             Realtime_Io loads data into a shared buffer_area and
--|             Async_Io_Partition processes that data.
-------------------------------------------------------------------------------
```

5. NON–REAL–TIME SECTION

5.1. Overall Structure

<To be finalized in later revision>

Non–real–time (NRT) components are constructed in a manner similar to real–time (RT) components. Since the NRT system doesn't require the RT Thread Executive, messaging mechanisms are a little more relaxed (see section 5.4.1). Missing from NRT components are requirements to have "Update" or "Request_State_Change" operations or Interface Definition Packages. Updating of components is accomplished entirely by the controlling component (Operational Component) calling operations of subordinate instances (See Fig 5–1 — Structural View of an Operational Component).

5.2. Classes and Instances

<To be finalized in later revision>

Like the previous section, all instances exist via creation from ADTs or Generic ADTs.

5.3. Inheritance and Composition

<To be finalized in later revision>

Both inheritance and composition of objects have played a large role in the analysis of our systems. To convert this effort to Ada, we must address the needs of efficiency and maintainability, as well as the need to satisfy object–oriented approaches.

Composition is fairly straightforward, needed classes are "WITHed", then objects are declared within the structure of the newer class.

Inheritance is another matter completely. There are several documented forms of accomplishing inheritance using Ada — each has their advantages and disadvantages.

The approach used by Grady Booch [Booch 91] is what we will be using for the SSVTF. Although more wordy that other approaches, it lends itself to the easiest maintenance (and easiest migration should we go to Ada 9X, the next version of Ada). There are several other methods to accomplish inheritance, and it is worth investigation by the curious. For further reading on alternate approaches to inheritance, see [Atkinson 91], [Hirasuna 92], [Perez 88], or [Seidewitz 92].

Grady Booch states, "In practice, we find it common to design as if inheritance were possible, then use a variety of implementations to fake it if the language does not directly support inheritance." [Booch 91] This is exactly the case for SSVTF using the current version of Ada. In order to support inheritance in it's simplest form, we will use packages built from the class structures (or possibly generic class structures) defined in section 5.5 "Templates and Guidelines" along with "pass–through" calls. See Appendix III, Create operation, Lesson_Class package for an example of "pass–through" calling.

5.4. Operational Components

<To be finalized in later revision>

Operational Components represent the "main" program in Ada. This is typically an ASM (called from a procedure) that controls all instances of classes – much like the real–time Partition.

5.4.1 Communicating with Other Operational Components/Partitions

<To be finalized in later revision>

There are three mechanisms by which Operational Components may communicate: file exchanging, utilizing the real–time interface, or POSIX Interprocess Communication. Each mechanism has it's benefits and drawbacks, which will be explained in detail in the next sections.

5.4.1.1 File Exchanging

<To be published in next revision>

5.4.1.2 Utilizing the Real–Time Interface

<To be published in next revision>

5.4.1.3 POSIX Interprocess Communication

<To be published in next revision>

5.5. Templates and Guidelines

The following example is intended to be used as a prototype template for building ADTs in non–real–time systems:

```
with Std_Eng_Types;
package Non_Real_Time_Template_Class is
   package SET renames Std_Eng_Types; -- Simplifies Parameter names
   type Valve_State is (Open, Closed);
   type Object is limited private;  -- limited private is preferred.
                                    -- private may be used,
                                    -- unprotected types require SRB approval
   -- ************************* Modifiers ****************************** --
   procedure Create (Instance : in out Object);
     -- AVOID using generics if this form can be used to parameterize
        the object.
   procedure Destroy (Instance : in out Object);
   procedure Set_Valve (Instance : in out Object;
                        To       : in     Valve_State);
   procedure Set_Pressure (Instance : in out Object;
                           To       : in     SET.Psi);
   -- ************************* Selectors ****************************** --
   -- NOTE: These are only examples.  Note that all operations here
   --       return primitives.  A primitive is either a non-numeric
   --       type defined in package Standard, a previously declared
   --       enumerated type within this specification, or a type
   --       defined inside the package Std_Eng_Types.
   function Valve_Is_Open (Instance : in     Object) return Boolean;
   function Pressure_Of (Instance : in     Object) return SET.Psi;
private
   type State;
   type Object is access State; -- Note that the "attributes" of the object
                                -- are invisible in the specification!
end Non_Real_Time_Template_Class;
```

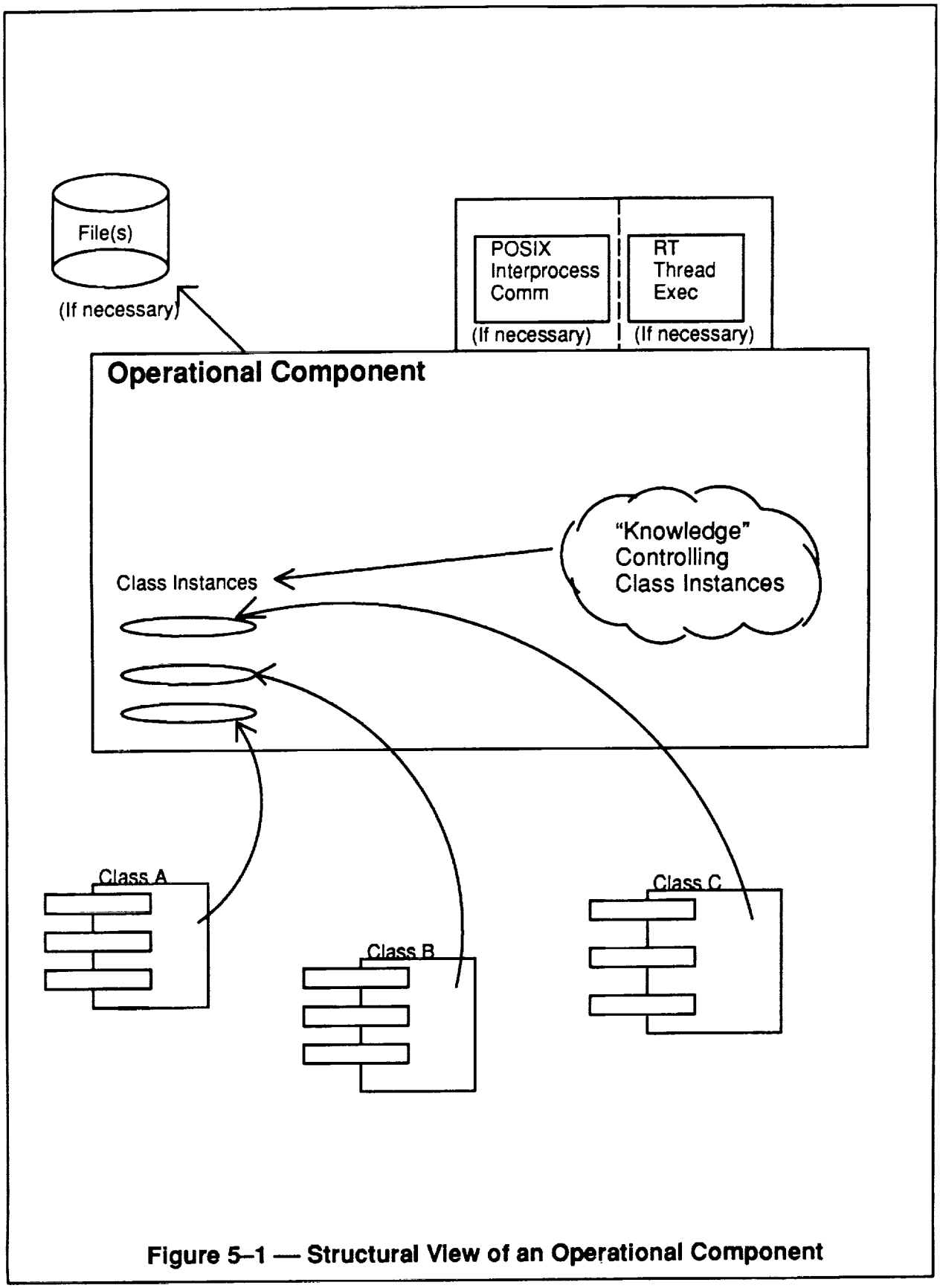


Figure 5–1 — Structural View of an Operational Component

6. BIBLIOGRAPHY

[Atkinson 91] Atkinson, Colin, Object–Oriented Reuse, Concurrency and Distribution, pages 183–228. ACM Press, NY, NY; 1991.

[Booch 91] Booch, Grady, Object Oriented Design with Applications, pages 443–470. Benjamin/Cummings Publishing Co., Menlo Park, CA; 1991.

[Gross and Stuckey, 1990] Gross, David C. and Stuckey, Lynn D., Jr., Ada Types: The Cornerstone of Simulation Modeling. (Source?).

[Hirasuna 92] Hirasuna, Michael, "Using Inheritance and Polymorphism with Ada in Government Sponsored Contracts", Ada Letters. Volume 12, Number 2, pages 43–56.

[Perez 88] Perez, Eduardo Perez, "Simulating Inheritance with Ada", Ada Letters, Volume 8, Number 5, pages 37–46, 1988.

[Seidewitz 92] Seidewitz, Ed, "Object–Oriented Programming with Mixins in Ada", Ada Letters. Volume 12, Number 2, pages 76–90.

7 APPENDIX I – ADA STRUCTURAL TEMPLATES

The following templates show the general Ada structural form for the class structures and partitions.

7.1 Class Template

The following example is intended to be used as a template for building ADTs in real–time systems. The ADT package exports an object of type "Object" and operations on that object. The operations are divided into 4 major categories – create, request state change, update, and selectors. The create is used to elaborate/initialize an instance. Request state change procedure(s) provide the capability of aperiodically modifying an instances state (such as the insertion of malfunctions or providing reset values). The Update procedure(s) iterate the instance of time. The selectors provide access to values held within the internal state of the instance.

```
with Std_Eng_Units;
package <name>_Class is

   package Seu renames Std_Eng_Units;  --Simplifies parameter names.

   type Object is limited private;     --Limited private is preferred.

   type Commands is (Reset, Malf1, Malf2, etc);

-- ******************************** Modifiers *********************************

   procedure Create (Instance        : in out Object;
                     Opt_Config_Var1 : in     Seu.Feet;
                     Opt_Config_Var2 : in     Seu.Psi;
                     Parent          : in     String);

   procedure Request_State_Change
                    (Instance        : in out Object;
                     Command         : in     Commands;
                     Input_1         : in     Seu.Feet := 0.0;
                     Input_2         : in     Seu.Psi  := 0.0);
        -- Used to modify the state of the instance.  Operation is performed
        -- aperiodically (i.e. applying a malfunction).

   procedure Update (Instance        : in out Object;
                     Delta_Time      : in     Seu.Seconds;
                     Input_One       : in     Seu.Feet;
                     Input_Two       : in     Seu.Psi);

-- ******************************** Selectors *********************************

   function Is_Selector1 (Instance : in  Object) return Boolean;

   function Is_Selector2 (Instance : in  Object) return Seu.Feet;

private

   type Object is record
      State_Var_1  : Seu.Feet  := 0.0;     --Note:  Always supply
      State_Var_2  : Seu.Psi   := 1.0;     --       default values.
      State_Var_3  : Boolean   := False;
   end record;

end <name>_Class;
------------------------------------------------------------------------------
--|
--| Abstract    : This is a general template form for a class structure.  Class
--|               structures should have this form in general when implemented.
--|               The actual class may have different routines, but each class
--|               should have Create, Request_State_Change, Update, and selector
--|               routines that basically follow this pattern.  This pattern will
--|               provide consistency for the software implementation of class
```

```
--                structures.
--
--  Warnings     : None.
--
-------------------------------------------------------------------------------



-------------------------------------------------------------------------------
-------------------------------------------------------------------------------

package body <name>_Class is

-- ***************************************************************************
   procedure Create (Instance         : in out Object;
                     Opt_Config_Var1  : in     Seu.Feet;
                     Opt_Config_Var2  : in     Seu.Psi;
                     Parent           : in     String) is
   begin
      null;  -- setup/init code goes here.
   end Create;

-- ***************************************************************************
   procedure Request_State_Change
                    (Instance         : in out Object;
                     Command          : in     Commands;
                     Input_1          : in     Seu.Feet := 0.0;
                     Input_2          : in     Seu.Psi  := 0.0) is
   begin
      case Command is
         when Reset => --reset code goes here.
         when Malf1 => --malf 1 insertion goes here.
         when Malf2 => --malf 2 insertion goes here.
         when ...;
      end case;
   end Request_State_Change;

-- ***************************************************************************
   procedure Update (Instance         : in out Object;
                     Delta_Time       : in     Seu.Seconds;
                     Input_One        : in     Seu.Feet;
                     Input_Two        : in     Seu.Psi) is
   begin
      null;  --update code goes here.
   end Update;

-- ***************************************************************************
   function Is_Selector1 (Instance    : in     Object) return Boolean is
   begin
      return Instance.State_Var_3;
   end Is_Selector1;

-- ***************************************************************************
   function Is_Selector2 (Instance    : in     Object) return Seu.Feet is
   begin
      return Instance.State_Var_1;
   end Is_Selector2;

end <name>_Class;
```

7.2 Class Template With Computed Period

The following class template is similar to the class template shown above except for the addition of a computed period capability. This capability allows an instance to be configured to run at a slower harmonic rate than the parent and at a relative period offset from the parent. This structure may be required if varying rate objects are placed under one rate–monotonically scheduled partition. This should be used in exceptional cases only. Note that this form will cause the partition modeler to perform manual period–leveling within the scope of the partition. The worst case period must then be used for RMS time allocations.

Two data types are provided in "Timer_Services_Class" (8.5) that support this option – "Rates" and "Period_Offsets". In the "Create" operation of the class structure, three parameters are shown – "Subrate", "Period_Base_Time", and "Period_Offset". "Subrate" specifies the rate relative to the parent base rate that the instance should update. The default value is "full" so that if the user does not want to use subrate scheduling, nothing has to be encoded and the instance will work normally. Any value passed in other than "full" will enable the subrate scheduling feature. "Period_Base_Time" is the base period rate of the parent (i.e. the RMS scheduled period of the partition in seconds). "Period_Offset" is the Nth period relative to the parent base period that the instance should update. This number is valid from 1 to (1/rate) of the subrate (i.e. 2, 4, 8, 16, 32, or 64).

For example: Assume a partition's base RMS period is 25 Hz (40 ms). If an object instance within the partition needs to run at a period of about 6 Hz (160 ms) or "quarter" rate, then the instance would be created using the following code segment:

```
Class.Create (Instance         => Instance,
              Subrate          => Quarter,
              Period_Base_Time => 0.04,
              Period_Offset    => 3);
```

The period offset of 3 would cause this instance to update on the third period of every consecutive 4 period cycles from the parent. ** Note that internally the class does not update using a counter (count 1..4, on 3 execute) – the update is performed based on delta time. This addresses the concern that if the parent "jumps ahead in time", the object will update based on that jump time, not the base period and count. The concept of passing delta time still applies completely.

For this template, the partition must run at the fastest rate required by the entire system. A service package, Timer_Services_Class (8.5), is used to provide the mechanism to run class instances defined by the partition at a slower, harmonic relative rate. Using this mechanism, there are no issues at the first level of class composition below the partition level. However, for composition elements past the first level, several scheduling and timing issues arise. The recommendations are that only the first level below the partition be subscheduled, and that if odd scheduling rates are required, the model should be flattened to address the real–time execution concerns.

```
with Std_Eng_Units;
with Timer_Services_Class;
package <name>_Class is

   package Seu renames Std_Eng_Units;  --Simplifies parameter names.

   package Services renames Timer_Services_Class;  --Simplifies names

   type Object is limited private;     --Limited private is preferred.

   type Commands is (Reset, Malf1, Malf2, etc);

   procedure Create (Instance         : in out Object;
                     Opt_Config_var1  : in     Seu.Feet;
                     Opt_Config_var2  : in     Seu.Psi;
                     Parent           : in     String;
```

```
                    --Subrate scheduling option parameters: --
                    Subrate          : in     Services.Rates          := Services.Full;
                    Period_Base_Time : in     Seu.Seconds             := 0.0;
                    Period_Offset    : in     Services.Period_Offsets := 1);

   ...

private
   type Object is record
      Timer      : Services.Object;          --Subrate schedule instance
      .
      .
   end record;
end <name>_Class;
-------------------------------------------------------------------------------
--;
--| Abstract     : This form of the class structure allows instances to run at
--|                slower harmonic rates from the calling model. This form will
--|                allow an instance to run slower than the parent and at a
--|                specified period offset from the parent.  Note that the
--|                instance must be able to complete within the period of the
--|                parent!  Note also that at Create, the timing parameters are
--|                defaulted to update at the same (Full) rate of the parent.
--|
--| Warnings     : None.
--'
-------------------------------------------------------------------------------

--=============================================================================
--=============================================================================

package body <name>_Class is

-- ***************************************************************************
   procedure Create (Instance         : in out Object;
                     Opt_Config_Var1  : in     Seu.Feet;
                     Opt_Config_Var2  : in     Seu.Psi;
                     Parent           : in     String;
                     Subrate          : in     Services.Rates          := Services.Full;
                     Period_Base_Time : in     Seu.Seconds             := 0.0;
                     Period_Offset    : in     Services.Period_Offsets := 1);
   begin

       -- Create the timing part.
       Services.Create (Timer            => Instance.Timer,
                        Subrate          => Subrate,
                        Period_Base_Time => Period_Base_Time,
                        Period_Offset    => Period_Offset);

       -- setup/init code goes here.

   end Create;

   ...

-- ***************************************************************************
   procedure Update (Instance   : in out Object;
                     Delta_Time : in     Seu.Seconds;
                     Input_One  : in     Seu.Feet;
                     Input_Two  : in     Seu.Psi) is
   begin

       -- Update the timing data.
       Services.Update (Timer      => Instance.Timer,
                        Delta_Time => Delta_Time);

       -- Update the rest of the data if it is time.
       if Services.Time_To_Update (Timer => Instance.Timer) then
           -- Use Services.Actual_Delta_Time (Timer => Instance.Timer)
           --     to get the change in time.
```

```
            null;  --update code goes here.
         end if;

   end Update;

   ...

end <name>_Class;
```

7.3 Partition Template

The following template shows the basic form of a partition. The first package shows the partitions interface definition (message) output package.

```
with Std_Eng_Units;
package <name>_Interface_Defn is
   package SEU renames Std_Eng_Units;
   type Message_1 is
      record
         Value1 : SEU.Volts;
         Value2 : Integer;
         Value3 : SEU.Amps;
      end record;
   type M1_Ptrs is access Message_1;
   type Message_2 is
      record
         Value4 : SEU.Psi;
         Value5 : SEU.Feet;
         Value6 : Natural;
      end record;
   type M2_Ptrs is access Message_2;
end <name>_Interface_Defn;
-------------------------------------------------------------------------------
--|
--| Abstract     : This is a general template form a partition's interface definition
--|                package.  Note that there can be 1 message per package, multiple
--|                interface definition packages per partition.
--|
--| Warnings     : None.
--|
--|-----------------------------------------------------------------------------


with <name>_Interface_Defn;
package <name>_Partition is
end <name>_Partition;
-------------------------------------------------------------------------------
--|
--| Abstract     : This is a general template form for a partition package.
--|
--| Warnings     : None.
--|
--|-----------------------------------------------------------------------------


with Std_Eng_Units;
with Mailbox,         -- SVM Mailbox System
     Message,         -- SVM Message System
     Generic_Model;   -- SVM Thread Exec
with <name>_Class;    -- some class

package BODY <name>_Partition is

-- ---------------
```

```
-- Package Renames
-- ---------------
   package SEU renames Std_Eng_Units;

-- ----------------
-- Message Pointers
-- ----------------
   Mesg_1_Id : Message.One_To_Many.Out_Msgs;
   Mesg_1    : <name>_Interface_Defn.M1_Ptrs;
   Mesg_2_Id : Message.Many_To_One.Out_Msgs;
   Mesg_2    : <name>_Interface_Defn.M2_Ptrs;

-- ---------------------------------
-- Internal Partition Class Instances
-- ---------------------------------
   My_Instance : <name>_Class.Object;

-- -----------------------
-- Internal Partition Data
-- -----------------------
   Delta_Time       : Set.Seconds;
   Elapsed_Time     : Set.Seconds    := 0.0
   Partition_Name   : String(1..16)  := "<name>_Partition";
   Mailbox_Id       : Mailbox.Mailboxes;
   Stabilized       : Boolean        := False;

   My_Var           : Integer;

--vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

   procedure Set_Up;
   procedure Create_Data;
   procedure Self_Init;
   procedure System_Init;
   procedure Run;
   procedure Freeze;
   procedure Hold;
   procedure Term;

   package Thread_Exec is new Generic_Model.Periodic
                              (Name                       => Partition_Name,
                               Rate                       => Generic_Model.P40hz,
                               Execute_Set_Up_Model       => Set_Up,
                               Execute_Create_Data_Model  => Create_Data,
                               Execute_Self_Init_Model    => Self_Init,
                               Execute_System_Init_Model  => System_Init,
                               Execute_Run_Model          => Run,
                               Execute_Freeze_Model       => Freeze,   --note, RUN may be used.
                               Execute_Hold_Model         => Hold,
                               Execute_Terminate_Model    => Term);

   procedure Process_Mailbox         is separate;
   procedure Update_Inputs           is separate;
   procedure Update_Outputs          is separate;
   procedure Update_Some_Object      is separate;
   procedure Update_Somemore_Objects is separate;
   procedure Reset_Partition         is separate;
   procedure Register_IO             is separate;

   procedure Set_Up                  is separate;
   procedure Create_Data             is separate;
   procedure Self_Init               is separate;
   procedure System_Init             is separate;
   procedure Run                     is separate;
   procedure Freeze                  is separate;
   procedure Hold                    is separate;
   procedure Term                    is separate;

end <name>_Partition;
```

```
------------------------------------------------------------------------------
--|
--  Abstract    : This is a general template form for a partition body.
--|
--  Warnings    : None.
--|
--|----------------------------------------------------------------------------

------------------------------------------------------------------------------
------------------------------------------------------------------------------
with Safestore_Mailbox;
with Safestore_Msg_Ids;
with Mail_Msg_Types;
separate (<name>_Partition)
procedure Process_Mailbox is
   Num_Msgs : Natural := 0;
   Safestore_Value : <safestore_data_type>;
   function Get_Safestore_Value is new
            Safestore_Mailbox.Value(Data_Type => <safestore_data_type>);
begin

-- ---
-- Get
-- ---

   -- get all of the mail from mailbox

   Num_Msgs := Mailbox.Num_Mail_Msgs (Mailbox_Id => Mailbox_Id) loop
   for i in 1 .. Num_Msgs loop

      Mailbox.Get (Mailbox_Id => Mailbox_Id,
                   Mail_Msg_Type => Msg_Type);

      -- do case on the type of mail message
      case Msg_Type is
         when Mail_Msg_Types.Ret_To_Safestore =>
            -- after type of message is determined, the message must be split to place the
            -- data area into the local pointer.
            Mailbox.Split_Safestore_Msg (Safestore_Msg => <safestore_message>,
                                         Mailbox_ID    => Mailbox_ID);
            -- get the data
            Safestore_Value := Get_Safestore_Value(Msg => <safestore_message>);
         when others =>
            null;
      end case;

   end loop;

-- ----
-- Send    ** NOT NEEDED IN EVERY PARTITION **
-- ----

   Msg_Type := <type_of_mail_message_to_send>;

   -- do case on the type of mail message
   case Msg_Type is
      when Mail_Msg_Types.Ret_To_Safestore =>
         -- build the message
         Mailbox.Build_Safestore_Msg (Safestore_Msg => <safestore_message_variable>,
                                      Mailbox_ID    => Mailbox_ID);
         -- now send the mail message
         Mailbox.Put (Partition_Prefix => <identifies_receiving_partition>,
                      Mailbox_Id       => Mailbox_Id);
      when others =>
         null;
   end case;

end Process_Mailbox;
------------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure Update_Inputs is
   Num_Of_Msgs : Natural := 0;
begin

-- normal (one-to-many)

   -- get time consistent message
   Message.One_To_Many.Get (In_Msg_Id => <message_identifier>);

   -- get time consistent message along with the time that the message was sent
   Message.One_To_Many.Get (In_Msg_Id => <message_identifier>,
                            Msg_Time  => <simulation_clock_time>);

   -- get the latest message that was sent by the producer
   -- Note:
      --       This operation does not provide time consistent message
      --       retrieval.  The time deltas between the messages received
      --       will vary depending upon the relative execution order of
      --       the producer and consumer
   Message.One_To_Many.Get_Latest
          (In_Msg_Id  => <message_identifier>,
           Msg_Time => <requester's_current_period_start_time>);

-- special (many-to-one)

   Num_Of_Msgs :=
       Message.Many_To_One.Number_Of_Msgs_To_Get (In_Msg_Id => (message_identifier>);
   For i in 1..Num_Of_Msgs loop
      Message.Many_To_One.Get
              (In_Msg_Id => <message_identifier>);
      <process_message>;
   end loop;

end Update_Inputs;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure Update_Outputs is
begin

-- To send messages for other partitions to use:

   -- for one-to-many messages
   Message.One_To_Many.Put (Out_Msg_Id => <message_identifier>);

   -- for many-to-one
   Message.Many_To_One.Put (Out_Msg_Id => <message_identifier>);

end Update_Outputs;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure Update_Some_Object is
begin

   null;  -- Whatever

end Update_Some_Object;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure Update_Somemore_Objects is
begin

   null;  -- More whatever

end Update_Somemore_Objects;
```

```
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure Initialize_Model is
begin

   <name>_Class.Request_State_Change(Instance => <name>_Instance_1,
                                     Command  => Reset);
   <name>_Class.Request_State_Change(Instance => <name>_Instance_2,
                                     Command  => Reset);
   <name>_Class.Request_State_Change(Instance => <name>_Instance_N,
                                     Command  => Reset);

end Initialize_Model;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure Register_IO is
begin

-- --------------------------
-- Register Partition Mailbox
-- --------------------------
   Mailbox.Register_Mailbox (Partition_Prefix     => <identifier_of_registering_partition>
                             Mailbox_Id           => <identifies_the_mailbox>);

-- ---------------------------------------------------------------
-- Identify the messages to be sent to other partitions
-- (output messages from this partitions perspective)
-- Each output message will require a REGISTER_TO_SEND_MSG routine
-- ---------------------------------------------------------------

   -- normal (one-to-many)

   Message.One_To_Many.Register_To_Send_Msg
           (Out_Msg_Id           => <identifies_the_message>,
            Partition_Prefix     => <identifier_of_registering_partition>,
            Msg_Dis_Id           => <DIS_id_of_message_to_be_sent>,
            Msg_Bit _Size        => <message_size_in_BITS>,
            Execution_Rate       => <worst_case_delivery_rate>,
            Msg_Ptr_Addr         => <local_pointer_to_the_message>);

   -- special (many-to-one)

   Message.Many_To_One.Register_To_Send_Msg
           (Out_Msg_Id           => <identifies_the_message>,
            Partition_Prefix     => <identifier_of_the_requesting_partition>,
            Msg_Dis_Id           => <DIS_id_of_message_to_be_sent>,
            Msg_Ptr_Addr         => <local_pointer_to_the_message>)
-- ---------------------------------------------------------------
-- Identify the messages to be received from other partitions
-- (input messages from this partitions perspective)
-- Each input message will require a REQUEST_TO_RECV_MSG
-- ---------------------------------------------------------------

-- normal (one-to-many)

   Message.One_To_Many.Register_To_Recv_Msg
           (In_Msg_Id            => <identifies_the_message>,
            Partition_Prefix     => <identifier_of_the_requesting_partition>,
            Msg_Dis_Id           => <DIS_id_of_message_to_be_received>,
            Execution_Rate       => <rate at which receiving partition executes>,
            Msg_Ptr_Addr         => <local_pointer_to_the_message>);

-- special (many-to-one)

   Message.Many_To_One.Regester_To_Recv_Msg
           (In_Msg_Id            => <identifies_the_message>,
            Partition_Prefix     => <identifier_of_the_requesting_partition>,
```

```
            Msg_Dis_Id          => <DIS_id_of_message_to_be_sent>,
            Msg_Bit _Size       => <message_size_in_BITS>,
            Queue_Size          => <worst_case_queue_size>,
            Msg_Ptr_Addr        => <local_pointer_to_the_message> );

end Register_IO;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
with DIS, <name>_Defs;
separate (<name>_Partition)
procedure Set_Up is
begin

-- ----------------------------
-- Create instances of classes.
-- ----------------------------
   <name>_Class.Create(...);

-- ------------------------------------------------
-- Link actual variables names to the logical DIS terms.
-- ------------------------------------------------
   DIS.Connect_Term (Term    => <name>_Defs.<variable_name>, -- Term ID
                     Address => <variable_name>'address);    -- Variable's Address

   DIS.Connect_Term (Term   => <name>_Defs.<variable_name>, -- Term ID
                     Symbol => "<variable_name>");          -- Variable's Name

-- ----------------------------------------
-- Initialize the model(s) in this Partition.
-- ----------------------------------------
   Initialize_Model;

-- ----------------------------
-- Register inputs and outputs.
-- ----------------------------
   Register_IO;

end Set_Up;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure Create_Data is
begin

-- normal (one-to-many)

   -- for each one-to-many input message CREATE_MSG is required
   Message.One_To_Many.Create_Msg (In_Msg_Id => <identifies_the_message>);

   -- for each one-to-many output message CREATE_MSG is required
   Message.One_To_Many.Create_Msg (Out_Msg_Id => <identifies_the_message>);

   --  For each one-to-many output message, init the buffer with "good" data.
   --  This is just in case another partition were to try and "read" from this
   --  buffer -- it ensures no constraint errors because of no initialization.
   Message.One_To_Many.Put (Out_Msg_Id => <identifies_the_message>);

-- special (many-to-one)

   -- for each many-to-one input message CREATE_MSG is required
   Message.Many_To_One.Create_Msg (In_Msg_Id  =>  <identifies_the_message>);

   -- for each many-to-one output message CREATE_MSG is required
   Message.Many_To_One.Create_Msg (Out_Msg_Id  =>  <identifies_the_message>);

end Create_Data;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure Self_Init is
```

```
begin

-- This routine will be called after some type of initialization data has been
-- read from an initialization file and placed into the appropriate mailbox.
-- The parameter, Initialization_Type is used to identify the type of self-init
-- being requested i.e., a full IC or a state adjustment.  See section 4.2.2 for
-- more information.

   If Thread_Exec.A_Full_Ic_Is_Required then -- means we are doing a full_ic initialization
                                             -- see section 4.4.2
      Initialize_Model;
   end if;

-- Each partition will read the mailbox data and populate local variables to
-- their new values.  It will also perform any other necessary internal
-- initialization.
-- NOTE:  This is a one-pass initialization -- no iterating!

   Process_Mailbox;

   -- Setup flags used during System_Init
   Stabilized := False;
   Elapsed_Time := 0.0;

-- ------------------------
-- Update to the next mode.
-- ------------------------

--  This is to be called when the partition has completed self-init processing

   Thread_Exec.Ready_To_Transition;

end Self_Init;
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
separate (<name>_Partition)
procedure System_Init is
begin

-- This routine will be called after Self_Init is complete.
-- Partitions will be able to iterate in this mode until stable conditions
-- have been reached.

   Delta_Time := Thread_Exec.Delta_Time;

   Process_Mailbox;
   Update_Inputs;
   Update_Some_Object;
   Update_Somemore_Objects;
   Update_Outputs;

   -- Update the timer used to stabilize this model
   if not Stabilized then
      Elapsed_Time := Elapsed_Time + Delta_Time;
      if Elapsed_Time >= 5.0 then
         Stabilized := True;
         Thread_Exec.Ready_To_Transition;
      end if;
   end if;

end System_Init;
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
separate (<name>_Partition)
procedure RUN is
begin

-- ANY PROCESSING REQUIRED BY THIS PARTITION WILL BE PLACED IN HERE SOMEWHERE

   Delta_Time := Thread_Exec.Delta_Time;
```

```
   Process_Mailbox;
   Update_Inputs;
   Update_Some_Object;
   Update_Somemore_Objects;
   Update_Outputs;

end RUN;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure FREEZE is
begin

-- ANY PROCESSING REQUIRED BY THIS PARTITION WILL BE PLACED IN HERE SOMEWHERE
-- Partitions will still iterate, however integration constants will be set
-- to zero.  Overruns will be detected in this mode.  Messages can    passed
-- and malfunctions entered by IOS.

   Delta_Time := Thread_Exec.Delta_Time;

   Process_Mailbox;
   Update_Inputs;
   Update_Some_Object;
   Update_Somemore_Objects;
   Update_Outputs;

end FREEZE;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure HOLD is
begin

-- ANY PROCESSING REQUIRED BY THIS PARTITION WILL BE PLACED IN HERE.
-- GENERALLY, ONLY PARTITIONS NEEDING TO PERFORM:
--    1. I/O TO KEEP DEVICES FROM DROPPING OFF-LINE
--    2. SPECIAL PROCESSING FOR ASSET ADD/DROP (INTERFACE AGENTS)
-- NEED TO PROVIDE SPECIAL ROUTINES FOR THIS MODE.
--
-- NO MESSAGES WILL BE PASSED IN THIS MODE
-- NO MALFUNCTIONS ENTERED FROM IOS.

   null;

end HOLD;
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
separate (<name>_Partition)
procedure TERM is
begin

-- ANY SHUTDOWN PROCESSING REQUIRED BY THIS PARTITION WILL BE PLACED IN HERE

   null;

 end TERM;
```

7.4 Generic Partition Template

The following template shows the differences between a normal partition and a generic partition. This structure basically makes a class–like structure out of an RMS–scheduled partition

```
with DIS;
generic
   Partition_DIS_Id : in     DIS.variable_id;
package <name>_Partition;
```

8. APPENDIX II –REAL TIME INTERFACE PACKAGES

8.1. Generic Model

```
------------------------------------------------------------------------------
-- genmodl_s.a
------------------------------------------------------------------------------
--  this package provides generic packages instantiated by applications
--  for scheduling.

with Std_Eng_Units;
with Rts_Types;
with Simulation_Clock;

package Generic_Model is

        ------------------------------------------------------------------------
        -- periodic rates supported are 40, 30, 25, 20, 10, 5, 2, and 1 Hz
        type Periodic_Rate is new Rt.Execution_Rate range Rt.P40hz .. Rt.P1hz;

        generic
                Name : in String;
                Rate : in Periodic_Type;
                with procedure Execute_Set_Up_Model;
                with procedure Execute_Create_Data_Model;
                with procedure Execute_Self_Init_Model;
                with procedure Execute_System_Init_Model;
                with procedure Execute_Run_Model;
                with procedure Execute_Freeze_Model;
                with procedure Execute_Hold_Model;
                with procedure Execute_Terminate_Model;
                Storage_Bits : in Integer := 10240 * 8;
                Max_Dis_Terms : in Integer := 400;
        package Periodic is
                -- subprograms for Thread Exec characteristics
                function Delta_Time return Seu.Seconds;
                function Rate_Of_Execution return Rt.Execution_Rate;
                -- subprograms for partition moding
                function A_Full_Ic_Is_Required return Boolean;
                procedure Ready_To_Transition (Continue_Exec : in Boolean := False);
                package Clock renames Simulation_Clock;
                use Clock;
                function G_M_T   return Clock.Time;
                function S_G_M_T return Clock.Time;
        end Periodic;

        ------------------------------------------------------------------------
        -- aperiodic budgeted rates are 40, 30, 25, 20, 10, 5, 2, and 1 Hz
        type Aperiodic_Rate is new Rt.Execution_Rate range Rt.A40hz .. Rt.A1hz;

        generic
                Name : in String;
                Rate : in Aperiodic_Type;
                Iterations : in Integer;
                Vector : in Integer;
                with procedure Execute_Set_Up_Model;
                with procedure Execute_Create_Data_Model;
                with procedure Execute_Self_Init_Model;
                with procedure Execute_System_Init_Model;
                with procedure Execute_Run_Model;
                with procedure Execute_Freeze_Model;
```

```
            with procedure Execute_Hold_Model;
            with procedure Execute_Terminate_Model;
            Storage_Bits : in Integer := 10240 * 8;
        package Aperiodic is
            -- subprogram for Thread Exec characteristics
            function Rate_Of_Execution return Rt.Execution_Rate;
            -- time functions return latest time from SimClock
            package Clock renames Simulation_Clock;
            use Clock;
            function G_M_T   return Clock.Time;
            function S_G_M_T return Clock.Time;
        end Aperiodic;

        ------------------------------------------------------------------------
        -- asynchronous partitions execute in background
        generic
            Name : in String;
            Delay_Time : Seu.Seconds;
            with procedure Execute_Set_Up_Model;
            with procedure Execute_Create_Data_Model;
            with procedure Execute_Self_Init_Model;
            with procedure Execute_System_Init_Model;
            with procedure Execute_Run_Model;
            with procedure Execute_Freeze_Model;
            with procedure Execute_Hold_Model;
            with procedure Execute_Terminate_Model;
            Storage_Bits : in Integer := 10240 * 8;
        package Asynchronous is
            -- subprograms for Thread Exec characteristics
            function Rate_Of_Execution return Rt.Execution_Rate;
            procedure Ready_To_Transition (Continue_Exec : in Boolean := False);
            procedure Change_Delay_Time (Time : in Seu.Seconds);
        end Asynchronous;

end Generic_Model;
```

8.2. Message

```
with Dis, Rts_Types, Std_Eng_Units, Simulation_Clock,
     Message_Internal_Types, System;

package Message is

    ---------------------------  Exceptions  ----------------------------
    -- The following exception is raise if an error occurs while
    -- setting up the messaging system.  If this exception is raised
    -- the messaging system may not function properly.
    Message_System_Setup_Error : exception;

    -- The following exception is raised if there is an unrecoverable
    -- error in the messaging system.  If this error is raised the
    -- message system may not function properly.
    Message_Internal_Error : exception;

    -- The following exceptions are raised if the message can not be
    -- successfull registerd or created.
    Register_Message_Error : exception;
    Create_Message_Error : exception;

    -- Package One_To_Many should be used for all general communication needs.
    -- It supports one sender sending to one or more receivers.  It provides
    -- time homogeneous and time consistent data based on the relative rates
    -- of the sender and receiver(s) if the Get operation is used to retrieve
    -- messages.  If the Get_Latest operation is used it provides the latest
    -- (most recent) message that was sent by the producer.

    package One_To_Many is   -- Normal Communication

        type Out_Msg is limited private;
        type In_Msg is limited private;

        ---------------------------  Exceptions  ----------------------------
        -- The following exception is raised by the Get operation if the
        -- time consistant message that is to be received by the caller
        -- (based upon its execution rate) is nolonger in the message
        -- buffer.  This condition will arise if the caller is executing
        -- slower than the lowest supported rate, or if it has has an
        -- overrun which causes it to be executing slower than the lowest
        -- supported rate.
        Message_Not_Found : exception;

        ----------------  called by the producer  -----------------

        -- This operation must be called by the producer during the
        -- Register_I/O submode for each message that is to be sent.
        procedure Register_To_Send_Msg
                        (Out_Msg_Id : in out Out_Msg;
                         Partition_Prefix : in Dis.Component_Id;
                         Msg_Dis_Id : in Dis.Message_Id;
                         Msg_Bit_Size : in Natural;
                         Execution_Rate : in Rts_Types.Execution_Rate;
                         Msg_Ptr_Addr : in System.Address);

        -- This operation must be called by the producer during the
        -- Create_Data submode for each message that is to be sent.
        procedure Create_Msg (Out_Msg_Id : in out Out_Msg);

        -- The Put operation is called by the producer to send a message
        procedure Put (Out_Msg_Id : in out Out_Msg);
```

```
      ----------------   called by the receiver(s)   ------------------

      -- This operation must be called by the receiver during the
      -- Register_I/O submode for each message that is to be received.
      procedure Register_To_Recv_Msg
                     (In_Msg_Id : in out In_Msg;
                      Partition_Prefix : in Dis.Component_Id;
                      Msg_Dis_Id : in Dis.Message_Id;
                      Execution_Rate : in Rts_Types.Execution_Rate;
                      Msg_Ptr_Addr : in System.Address);

      -- This operation must be called by the receiver during the
      -- Create_Data submode for each message that is to be received.
      procedure Create_Msg (In_Msg_Id : in out In_Msg);

      -- The Get operations retrieve time consistent messages relative to
      -- the rate of the consumer.  The message retrieved will be the
      -- most recent message produced during the consumers previous
      -- period.
      procedure Get (In_Msg_Id : in out In_Msg);

      procedure Get (In_Msg_Id : in out In_Msg;
                     Msg_Time : out Simulation_Clock.Time);

      -- The Get_Latest operations retrieve the most recent message
      -- produced (relative to the rate of the producer).
      --   NOTE:
      --        These operations do not provide time consistent message
      --        retrieval.  The time deltas between the messages received
      --        will vary depending upon the relative execution order of
      --        the producer and consumer.
      procedure Get_Latest (In_Msg_Id : in out In_Msg);

      procedure Get_Latest (In_Msg_Id : in out In_Msg;
                                Msg_Time : out Simulation_Clock.Time);

      -- ---------------------------
      -- -- private
      -- --

private
      package Mit renames Message_Internal_Types;

      pragma Inline (Put);
      pragma Inline (Get);
      pragma Inline (Get_Latest);

      -- Record for SGI
      type Msg is
          record
                Buffer_Ptr : Mit.Msg_Buffer_Ptr;
                Desc_Ptr : Mit.Otm_Msg_Desc_Ptr;
                Tag_Ptr : Mit.Otm_Msg_Tag_Ptr;
                Partition_Ptr_Addr : System.Address;
                Routing_Table_Index : Integer := 0;
                Period : Rts_Types.Execution_Rate :=
                   Rts_Types.Execution_Rate'First;
          end record;

      type In_Msg is new Msg;
      type Out_Msg is new Msg;

end One_To_Many;
```

```
-- Package Many_To_One should be used only for special case communication.
-- All messages sent to the receiver are queued in FIFO order.  The message
-- received is not based on the rates of the sender and receiver, instead
-- the receiver receives all message sent.  This communication method
-- should be used when a single receiver receives the same message from
-- more than one producer or when a single receiver needs to receive all
-- messages sent in FIFO order.  When the receiver registers to reeceive
-- a message a queue size must be specified.  The queue size should be
-- determined based upon two factors.  First, the number of possible senders
-- and second, the relative execution rates of the senders and the receiver.
-- If the receiver is executing faster than or at the same rate as the
-- senders, the queue size must be at least as large as two times the number
-- of senders.  If the receiver is executing slower than the senders the
-- following formula can be used to calculate the queue size:
--                     [(senders rate / receivers rate) x 2] x #senders.
-- For example, if the receiver is executing at 10Hz with three senders
-- executing at 40 Hz the queue size should be [(40/10) x 2] x 3 = 24.

package Many_To_One is    -- Special Case Communication

        package Mit renames Message_Internal_Types;

        type Out_Msg is limited private;
        type In_Msg is limited private;

        -- 512 has been chosen as the max queue size, no particular reason, just
        -- a guess for now.
        subtype Queue_Sizes is Mit.Internal_Queue_Sizes range 0 .. 512;

        ----------------------------  Exceptions  ----------------------------
        -- The following exception is raised by Put when the queue is full
        Queue_Full : exception;

        -- The following exception is raised by Get when there are no messages
        No_Messages : exception;

        -----------------   called by producers of message   ------------------

        -- This operation must be called by the producers during the
        -- Register_I/O submode for each message that is to be sent.
        procedure Register_To_Send_Msg (Out_Msg_Id : in out Out_Msg;
                                                     Partition_Prefix : in Dis.Component_Id;
                                                     Msg_Dis_Id : in Dis.Message_Id;
                                                     Msg_Ptr_Addr : in System.Address);


        -- This operation must be called by the producers during the
        -- Create_Data submode for each message that is to be sent.
        procedure Create_Msg (Out_Msg_Id : in out Out_Msg);

        -- This operation will return true if the queue for Out_Msg_Id
        -- is full
        function Queue_Is_Full (Out_Msg_Id : in Out_Msg) return Boolean;

        -- The Put operation is called by the producers to send a message
        procedure Put (Out_Msg_Id : in out Out_Msg);

        -----------------   called by receiver of message   ------------------

        -- This operation must be called by the receiver during the
        -- Register_I/O submode for each message that is to be received.
        -- See the description at the begining of the Many_To_One package
        -- to determine the queue size.
        procedure Register_To_Recv_Msg (In_Msg_Id : in out In_Msg;
```

```
                                                Partition_Prefix : in Dis.Component_Id;
                                                Msg_Dis_Id : in Dis.Message_Id;
                                                Msg_Bit_Size : in Natural;
                                                Queue_Size : in Queue_Sizes;
                                                Msg_Ptr_Addr : in System.Address);

        -- This operation must be called by the receiver during the
        -- Create_Data st   ie for each message that is to be received.
        procedure Create_   (In_Msg_Id : in out In_Msg);

        -- This operation may be called by the receiver to determine the
        -- number of partitions registered to send a particular message.
        function Number_Of_Senders (In_Msg_Id : In_Msg) return Natural;

        -- This operation may be called by the receiver to determine the
        -- the number of messages available to get.
        function Number_Of_Msgs_To_Get (In_Msg_Id : In_Msg) return Natural;

        -- The Get operations will retrieve the next message in the FIFO queue.
        procedure Get (In_Msg_Id : in out In_Msg);

        procedure Get (In_Msg_Id : in out In_Msg;
                       Msg_Time : out Simulation_Clock.Time);

        -- ---------------------------
        -- -- private
        -- --

private

        pragma Inline (Get);
        pragma Inline (Put);
        pragma Inline (Number_Of_Msgs_To_Get);

        -- Record for SGI
        type Msg is
            record
                    Buffer_Ptr : Mit.Msg_Buffer_Ptr;
                    Desc_Ptr : Mit.Mto_Msg_Desc_Ptr;
                    Tag_Ptr : Mit.Mto_Msg_Tag_Ptr;
                    Partition_Ptr_Addr : System.Address;
                    Routing_Table_Index : Integer := 0;
                    Queue_Size : Mit.Internal_Queue_Sizes := 0;
            end record;

        type In_Msg is new Msg;
        type Out_Msg is new Msg;

end Many_To_One;

-- Package Remote is not to be used for general purpose communication
-- or for partition to partition communication.  It is intended to be used
-- used by RTS,  terface Agents, and in other special cases (IOS, OSS)
-- for communication across the lan.

-- All messages sent to the receiver are queued in FIFO order.  Each
-- receiver effectivly has its own queue.

-- When the receiver registers to receive
-- a message a queue size must be specified.  The queue size should be
-- determined based upon two factors.  First, the number of possible senders
-- and second, the relative execution rates of the senders and the receiver.
-- If the receiver is executing faster than or at the same rate as the
```

```
-- senders, the queue size must be at least as large as two times the number
-- of senders.  If the receiver is executing slower than the senders the
-- following formula can be used to calculate the queue size:
--                    ((senders rate / receivers rate) x 2) x #senders.
-- For example, if the receiver is executing at 10Hz with three senders
-- executing at 40 Hz the queue size should be ((40/10) x 2) x 3 = 24.

package Remote is  -- Special Case Communication

      package Mit renames Message_Internal_Types;

      type Out_Msg is limited private;
      type In_Msg is limited private;

      -- 512 has been chosen as the max queue size, no particular reason, just
      -- a guess for now.
      subtype Queue_Sizes is Mit.Internal_Queue_Sizes range 0 .. 512;

      ---------------------------  Exceptions  ----------------------------
      -- The following exception is raised by Put when the queue is full
      Queue_Full : exception;

      -- The following exception is raised by Get when there are no messages
      No_Messages : exception;

      -----------------   called by producers of message   ------------------

      -- This operation must be called by the producers during the
      -- Register_I/O submode for each message that is to be sent.
      procedure Register_To_Send_Msg
                     (Out_Msg_Id : in out Out_Msg;
                      Partition_Prefix : in Dis.Component_Id;
                      Msg_Dis_Id : in Dis.Message_Id;
                      Msg_Bit_Size : in Natural;
                      Queue_Size : in Queue_Sizes;
                      Execution_Rate : in Rts_Types.Execution_Rate;
                      Msg_Ptr_Addr : in System.Address);

      -- These operations are called to unregister messages.  They
      -- provide the capability for multiple senders and receivers
      -- to register to send or receive a message without knowing
      -- who the true sender or recieve will be.  Once the true sender
      -- or receiver is determined, the others unregister there messages.
      --
      -- NOTE: These operations must be called before the Join_Session
      -- operation is called!!!  They are not to be used after inter-asset
      -- communcation has been established.

      -- Unregister an out message.  (Un-Register_To_Send_Msg)
      procedure Unregister_Msg (Msg_Id : in out Out_Msg);

      -- Unregister an in message.  (Un-Register_To_Recv_Msg)
      procedure Unregister_Msg (Msg_Id : in out In_Msg);

      -- These two operations provide the capability for a sender
      -- or receiver to re-register to send or receive a message
      -- after un-registering it with one of the above Unregister
      -- operations.

      -- NOTE: These operations will not allow the sender or
      -- reciever to send or receive messages across the LAN.
      -- It will only let them start sending or receiving the
      -- message locally.  It is intended to be used by an asset
```

```
        -- after it has been dropped and is standalone and therefore
        -- does not wish to send or receive message across the lan
        -- but does need to send or receive them now that it is
        -- standalone.

        -- Reregister an out message.  (Re-Register_To_Send_Msg)
        procedure Reregister_Msg (Msg_Id : in out Out_Msg);

        -- Reregister an in message.  (Re-Register_To_Recv_Msg)
        procedure Reregister_Msg (Msg_Id : in out In_Msg);

        -- This operation must be called by the producers during the
        -- Create_Data submode for each message that is to be sent.
        procedure Create_Msg (Out_Msg_Id : in out Out_Msg);

        -- This operation will return true if the queue for Out_Msg_Id
        -- is full
        function Queue_Is_Full (Out_Msg_Id : in Out_Msg) return Boolean;

        -- The Put operation is called by the producers to send a message
        procedure Put (Out_Msg_Id : in out Out_Msg);

        -----------------   called by receiver of message   ------------------

        -- This operation must be called by the receiver during the
        -- Register_I/O submode for each message that is to be received.
        -- See the description at the begining of the Many_To_One package
        -- to determine the queue size.
        procedure Register_To_Recv_Msg (In_Msg_Id : in out In_Msg;
                                                    Partition_Prefix : in Dis.Component_Id;
                                                    Msg_Dis_Id : in Dis.Message_Id;
                                                    Msg_Bit_Size : in Natural;
                                                    Queue_Size : in Queue_Sizes;
                                                    Msg_Ptr_Addr : in System.Address);

        -- This operation must be called by the receiver during the
        -- Create_Data submode for each message that is to be received.
        procedure Create_Msg (In_Msg_Id : in out In_Msg);

        -- This operation may be called by the receiver to determine the
        -- the number of messages available to get.
        function Number_Of_Msgs_To_Get (In_Msg_Id : In_Msg) return Natural;

        -- The Get operations will retrieve the next message in the FIFO queue.
        procedure Get (In_Msg_Id : in out In_Msg);

        procedure Get (In_Msg_Id : in out In_Msg;
                       Msg_Time : out Simulation_Clock.Time);

        -- ---------------------------
        -- -- private
        -- --

    private

        pragma Inline (Get);
        pragma Inline (Put);
        pragma Inline (Number_Of_Msgs_To_Get);

        -- Record for SGI
        type Msg is
            record
                Buffer_Ptr : Mit.Msg_Buffer_Ptr;
```

```
                Desc_Ptr : Mit.Remote_Msg_Desc_Ptr;
                Tag_Ptr : Mit.Remote_Msg_Tag_Ptr;
                Partition_Ptr_Addr : System.Address;
                Routing_Table_Index : Integer := 0;
                Queue_Size : Mit.Internal_Queue_Sizes := 0;
                My_Ready_Remove_Index : Positive := 1;
            end record;

        type In_Msg is new Msg;
        type Out_Msg is new Msg;

end Remote;

-- This package contains controlling operations only to be used by rts.
-- The router needs special interfaces because it cannot create a Msg_Id
-- for each message that it puts and gets from the software backplane.
-- It is just a pass thru from the RTSN to the software backplane and
-- visa versa.
package Control is

        -- The status values for Join_Session and Drop_From_Session
        type Config_Status is (Success, Pending, Error);

        ---------------------------  Exceptions  ----------------------------
        -- The following exception is raised by Put when the queue is full
        Queue_Full : exception;  -- raised by Put when the queue is full

        -- The following exception is raised by Get when there are no messages
        No_Messages : exception;  -- raised by Get when there are no messages

        -- This procedure is to be used by the router to create a
        -- reflected message.
        procedure Create_Msg (Msg_Dis_Id : in Dis.Message_Id);

        -- This operation will return true if the queue for Msg_Dis_Id
        -- is full
        function Queue_Is_Full (Msg_Dis_Id : in Dis.Message_Id) return Boolean;

        -- This procedure is called by the Router to put a remote message
        -- into the swbp after receiveing if from the lan.
        procedure Put (Msg_Dis_Id : in Dis.Message_Id;
                       Msg_Addr : in System.Address);

        -- This procedure is to be used by the router to determine
        -- the number of remote messages in a message queue to be
        -- sent out over the lan.
        function Number_Of_Msgs_To_Get
                       (Msg_Dis_Id : Dis.Message_Id) return Natural;

        -- This procedure is called by the Router to get a remote message
        -- from the swbp to send it out over the lan.
        procedure Get (Msg_Dis_Id : in Dis.Message_Id;
                       Msg_Addr : in System.Address);

        -- This command instructs the software backplane to set up
        -- communication with a session.  It should be called after
        -- the Setup and Create Data submodes to establish inter-asset
        -- communication.
        procedure Join_Session (Session : in Std_Eng_Units.Sessions);

        -- This command instructs the software backplane to drop
        -- communication with the session.
        procedure Drop_From_Session;
```

```
        -- This operation will return the status of the last Join_Session
        -- or Drop_From_Session command.
        function Status return Config_Status;

        -- This operation will determine if the message can be a reflected
        -- message.  It will check the size of the message agianst the
        -- the amount of reflected memory left.  It will also check the
        -- type of the message.  Currently only One-TO-Many messages can
        -- be reflecterd.
        function This_Message_Can_Be_Reflected
                        (Msg_Dis_Id : in Dis.Message_Id) return Boolean;

        -- This operation is called by the Router to prepare a remote message
        -- to be sent remote.  It should be called for a specific remote
        -- message when the first asset that receives the remote message
        -- joins the session.
        procedure Setup_To_Send_Msg_Remote (Msg_Dis_Id : in Dis.Message_Id);

        -- This operation is called by the Router when a remote message
        -- that is being sent remote no longer needs to be send remote.
        -- It should be called for a specific remote message when the
        -- last asset that requires the message drops from the session.
        procedure Stop_Sending_Msg_Remote (Msg_Dis_Id : in Dis.Message_Id);

        -- This command shutsdown the software backplane
        procedure Shutdown;

private

        pragma Inline (Get);
        pragma Inline (Put);
        pragma Inline (Number_Of_Msgs_To_Get);
        pragma Inline (Status);
        pragma Inline (This_Message_Can_Be_Reflected);

end Control;

-- This package contains the communication interface for black boxes on
-- the RTSN.  It will provide operations to register, create, send, and
-- possibly receive messages.  These operations are different than those
-- in One-To-Many and Many-To-One because there may be special formats
-- required in order to communicate with a black box.
package Black_Box is

        package Mit renames Message_Internal_Types;

        type Command_Statuses is (Success, Pending, Bad_Id, Busy, Error);
        type Command_Ids is private;
        type Interface_States is (Enabled, Disabled);

        -- 512 has been chosen as the max queue size, no particular reason, just
        -- a guess for now.
        subtype Queue_Sizes is Mit.Internal_Queue_Sizes range 0 .. 512;

        type Comm_Types is (Stream, Dgram);
        type Networks is (Rtsn, Gp);

        type Out_Msg is limited private;
        type In_Msg is limited private;

        ----------------   called by the producer   ------------------

        -- This operation must be called by the producer during the
        -- Register_I/O submode for each message that is to be sent.
```

```
-- It will register the message with the software backplane
-- and establish the communication link with the black box.
procedure Register_To_Send_Msg
                (Out_Msg_Id : in out Out_Msg;
                 Partition_Prefix : in Dis.Component_Id;
                 Msg_Dis_Id : in Dis.Message_Id;
                 Msg_Bit_Size : in Natural;
                 Execution_Rate : in Rts_Types.Execution_Rate;
                 Msg_Ptr_Addr : in System.Address;
                 Receiving_Node : in Std_Eng_Units.Nodes;
                 Comm_Type : in Comm_Types;
                 Network : in Networks := Rtsn);

-- This operation must be called by the producer during the
-- Create_Data submode for each message that is to be sent.
-- It will allocate the buffers in the software backplane for
-- the message.
procedure Create_Msg (Out_Msg_Id : in out Out_Msg);

-- The Put operation is called by the producer to send a message
procedure Put (Out_Msg_Id : in out Out_Msg);

-----------------   called by receiver of message   ------------------

-- This operation must be called by the receiver during the
-- Register_I/O submode for each message that is to be received.
procedure Register_To_Recv_Msg (In_Msg_Id : in out In_Msg;
                                Partition_Prefix : in Dis.Component_Id;
                                Msg_Dis_Id : in Dis.Message_Id;
                                Msg_Bit_Size : in Natural;
                                Queue_Size : in Queue_Sizes;
                                Msg_Ptr_Addr : in System.Address;
                                Sending_Node : in Std_Eng_Units.Nodes;
                                Comm_Type : in Comm_Types;
                                Network : in Networks := Rtsn);

-- This operation must be called by the receiver during the
-- Create_Data submode for each message that is to be received.
procedure Create_Msg (In_Msg_Id : in out In_Msg);

-- This operation may be called by the receiver to determine the
-- the number of messages available to get.
function Number_Of_Msgs_To_Get (In_Msg_Id : In_Msg) return Natural;

-- The Get operations will retrieve the next message in the FIFO queue.
procedure Get (In_Msg_Id : in out In_Msg);

-----------------------   called by eithor   -------------------------

-- These commands can be called to command the software backplane to
-- perform certian operations associated with communication with a
-- black box.  The commands return command id which uniqely identify
-- the command.  Status of the command can be obtained by calling
-- the satatus function and passing it the id of the command on
-- which status is desired.  The status will be one of the following:
--      Success - the command has completed successfully
--      Pending - the command is in progress
--      Bad_Id - no command associated with this id
--      Busy - a command is already in progress to this node
--      Error - the command has not completed successuflly
-- Only one command can be outstanding to a node at a time.  If
-- multiple commands are issued all but the first will be ignored
-- and status calls will return with busy.  These commands must
-- be re-issued after the previous commands complete.
```

```
        -- This command will open and configure the communication link with
        -- the black box. It should not be called until the back box is ready
        -- to communicate.
        function Open_Comm (Node : in Std_Eng_Units.Nodes) return Command_Ids;

        -- This command will close the communication link with the
        -- black box.
        function Close_Comm (Node : in Std_Eng_Units.Nodes) return Command_Ids;

        -- This operation will return the status of the command associated
        -- with the command id.
        function Command_Status (Command_Id : in Command_Ids)
                                        return Command_Statuses;

        -- These commands return the state of the black box node interface
        function Interface_State
                        (Node : in Std_Eng_Units.Nodes) return Interface_States;

        function Interface_Is_Disabled
                        (Node : in Std_Eng_Units.Nodes) return Boolean;

        function Interface_Is_Enabled
                        (Node : in Std_Eng_Units.Nodes) return Boolean;

    private

        pragma Inline (Get);
        pragma Inline (Put);
        pragma Inline (Number_Of_Msgs_To_Get);
        pragma Inline (Open_Comm);
        pragma Inline (Close_Comm);
        pragma Inline (Command_Status);
        pragma Inline (Interface_State);
        pragma Inline (Interface_Is_Disabled);
        pragma Inline (Interface_Is_Enabled);

        type Command_Ids is new Natural;

        type Msg is
            record
                    Buffer_Ptr : Mit.Msg_Buffer_Ptr;
                    Desc_Ptr : Mit.Mto_Msg_Desc_Ptr;
                    Tag_Ptr : Mit.Mto_Msg_Tag_Ptr;
                    Partition_Ptr_Addr : System.Address;
                    Routing_Table_Index : Integer := 0;
                    Period : Rts_Types.Execution_Rate :=
                       Rts_Types.Execution_Rate'First;
            end record;

        type In_Msg is new Msg;
        type Out_Msg is new Msg;

    end Black_Box;

end Message;
-------------------------------------------------------------------------------
--| Abstract: This package provides the types and operations necessary to
--|           interface with the messaging system.
--|
--| Warnings: This package depends on the use of shared memory and shared
--|           semaphores.  The semaphores are only used during initialization,
--|           not during runtime.
--|
```

```
--          This package depends upon compatibility between System.Address
--          and the value of an access type.  It uses Unchecked_Conversion
--          to convert from x'Address to an access type.
-------------------------------------------------------------------------------
```

8.3. Mailbox

```
with Message_Internal_Types, Dis, System, Enter_Mailbox,
     Safestore_Mailbox, Malfunction_Mailbox, Mega_Mailbox, Std_Eng_Types;

package Mailbox is

    package Mit renames Message_Internal_Types;
    package Set renames Std_Eng_Types;

    ---------------------------- Constants  ---------------------------------

    -- The size of a mail message
    -- 2k storage units long - just a guess for now
    Max_Mailbox_Msg_Size : constant Natural := 2048;

    ---------------------------- Exceptions  --------------------------------

    -- Not used
    Not_A_Prefix : exception;

    -- Raised by Register_Mailbox if an exception occurs.  Or if the
    -- Component_Id supplied for the parameter 'My_Partition_Prefix'
    -- is not a partition prefix.  That is, the Component_Id was not
    -- registered (Dis.Register_Component) with Prefix set to True.
    -- When ever possible a message will be logged giving details as
    -- to why this exception was raised.
    Register_Mailbox_Error : exception;

    -- Raised by Get_User_Defined_Msg_Type, Get_User_Defined_Msg, and
    -- Get_Next_Msg_Type if they are called on an empty mailbox
    Mailbox_Empty : exception;

    -- Raised by Put_User_Defined_Msg if the desination mailbox is not
    -- found (not registered).
    Mailbox_Not_Found : exception;

    -- Raised by Put_User_Defined_Msg if a user defined mail message
    -- is too large.
    Mailbox_Message_Too_Large : exception;

    -- Raised by Put_User_Defined_Msg if the mailbox does not have enough
    -- memory to send the message.
    Mailbox_System_Out_Of_Memory : exception;

    -- Raised if the mailbox system cannot startup correctly.  If this is
    -- raised the mailbox system may not function correctly.
    Mailbox_Startup_Error : exception;

    -- Raised if the mailbox system cannot shutdown correctly.
    Mailbox_Shutdown_Error : exception;

    -- Raised by Register_Mailbox if there is an uncrcoverable internal
    -- error in the mailbox system.  If this error is raised, the mailbox
    -- system should be considerd erronous.
    Mailbox_Internal_Error : exception;

    --------------------   Types   ---------------------

    type Msg_Types is (Router, Return_To_Safestore, Return_To_Datastore,
                       Malfunction, Enter, Mega, User_Defined);
    for Msg_Types use (Router => -7,
                       Return_To_Safestore => -6,
```

```
                        Return_To_Datastore => -5,
                        Malfunction => -4,
                        Enter => -3,
                        Mega => -2,
                        User_Defined => -1);
for Msg_Types'Size use 32;

type Internal_Msg_Type is limited private;
type Mailboxes is limited private;

--------------------   called by owner of mailbox   ---------------------

-- This operation registers a mailbox it must be called in order to
-- send or receive mail messages.  It should be called during the
-- Register I/O submode.
procedure Register_Mailbox (My_Partition_Prefix : in Dis.Component_Id;
                            My_Mailbox_Id : in out Mailboxes);

-- Returns true if mail messages are present, false if not.
function Mail_Is_Present (My_Mailbox_Id : in Mailboxes) return Boolean;

-- Returns the number of mail messages currently in the mailbox.
function Num_Mail_Msgs (My_Mailbox_Id : in Mailboxes) return Natural;

-- Gets the type of the next mail message in the mailbox.  This is
-- the first step in retrieving a mail message.  After the type has
-- been determined the appropriate Get operation can be called, or
-- if the type is User_Defined then the Get_User_Defined_Msg_Type
-- operation can be called.
function Get_Next_Msg_Type (My_Mailbox_Id : in Mailboxes) return Msg_Types;

-- Operations to get the next message from the mailbox
procedure Get_Safestore_Msg (Safestore_Msg : out
                                Safestore_Mailbox.Safestore_Msg;
                             My_Mailbox_Id : in out Mailboxes);

procedure Get_Malfunction_Msg (Malfunction_Msg : out
                                  Malfunction_Mailbox.Malfunction_Msg;
                               My_Mailbox_Id : in out Mailboxes);

procedure Get_Enter_Msg (Enter_Msg : out Enter_Mailbox.Enter_Msg;
                         My_Mailbox_Id : in out Mailboxes);

procedure Get_Mega_Msg (Mega_Msg : out Mega_Mailbox.Mega_Msg;
                        My_Mailbox_Id : in out Mailboxes);

-- This operation is to be used by the Router to get messages from its
-- mailbox.  Address_For_Msg is the address for the locatin at which
-- the message should be placed.  This location must be capable of
-- holding a mail message of Max_Mailbox_Msg_Size (declared in this
-- package).  Dest_Partition_Prefix is the original destination of
-- the mail message.  Msg_Type is the type of the mail message.
-- My_Mailbox_Id is the routers mailbox id.
procedure Get_Router_Msg (Address_For_Msg : in System.Address;
                          Dest_Partition_Prefix : out Dis.Component_Id;
                          Msg_Type : out Internal_Msg_Type;
                          My_Mailbox_Id : in out Mailboxes);

--  USER DEFINED MESSAGE SUPPORT  --
-- The Get_User_Defined_Msg_Type and Get_User_Defined_Msg operations
-- provide support for receiveing user defined mail messages.  When
-- possible the above predefined mail messages types should be used
-- because they ensure that the sender and receiver are using the
```

```
-- same message type.  There are also support packages provided to
-- aid in using mail message of the predefined types.  If this generic
-- routines are used it is up to the user to ensure that the sender and
-- receiver agree on the structure of the mail message.
--
-- If the Get_Next_Mail_Msg_Type function returns User_Defined as the
-- type of the next message in the mailbox and if it is possible for
-- the mailbox to receive more than one type of user defined message
-- then the Get_User_Defined_Msg_Type operation must be called to
-- determine which user defined message is in the mailbox.  After
-- this has been determined, then the appropriate instantiation of
-- Get_User_Defined_Msg can be called to retrieve the message from
-- the mailbox.

-- Gets the type of the user defined mail message.
generic
       type User_Defined_Msg_Types is (<>);
function Get_User_Defined_Msg_Type
                  (My_Mailbox_Id : in Mailboxes) return User_Defined_Msg_Types;

-- Gets the user defined mail message.
generic
       type User_Defined_Mail_Msg is private;
procedure Get_User_Defined_Msg
                   (User_Defined_Msg : out User_Defined_Mail_Msg;
                    My_Mailbox_Id : in out Mailboxes);

-----------------   called by sender of mail message   -----------------

-- Sends a mail message to the specified partition.
procedure Put_Safestore_Msg (Safestore_Msg : in
                                          Safestore_Mailbox.Safestore_Msg;
                                      Dest_Partition_Prefix : in Dis.Component_Id);

procedure Put_Malfunction_Msg (Malfunction_Msg : in
                                            Malfunction_Mailbox.Malfunction_Msg;
                                        Dest_Partition_Prefix : in Dis.Component_Id);

procedure Put_Enter_Msg (Enter_Msg : in Enter_Mailbox.Enter_Msg;
                                    Dest_Partition_Prefix : in Dis.Component_Id);

procedure Put_Mega_Msg (Mega_Msg : in Mega_Mailbox.Mega_Msg;
                                   Dest_Partition_Prefix : in Dis.Component_Id);

procedure Put_Ds_Msg (Ds_Msg : in Mega_Mailbox.Mega_Msg;
                              Dest_Partition_Prefix : in Dis.Component_Id);

-- This operation is to be used by the Router to send mailbox
-- message to partition's mailboxes.  Address_of_Msg is the address
-- of the mail message.  Max_Mailbox_Msg_Size storage units will be
-- taken from this address and send to the destination mailbox for
-- Dest_Partition_Prefix.  Mail_Msg_Type is the type of the mail
-- message that is being sent.
procedure Put_Router_Msg (Address_Of_Msg : in System.Address;
                                    Msg_Type : in Internal_Msg_Type;
                                    Dest_Partition_Prefix : in Dis.Component_Id);

--  USER DEFINED MESSAGE SUPPORT  --
-- The Put_User_Defined_Msg operation is used to send a user defined
-- mail message.  When possible the above predefined mail messages types
-- should be used because they ensure that the sender and receiver are
-- using the same message type.  There are also support packages provided
-- to aid in using mail message of the predefined types.  If this generic
```

```
    -- routine is used, it is up to the user to ensure that the sender and
    -- receiver agree on the structure of the mail message.
    generic
           type User_Defined_Mail_Msg is private;
           type User_Defined_Msg_Types is (<>);
    procedure Put_User_Defined_Msg
                        (Mail_Msg_Type : in User_Defined_Msg_Types;
                         User_Defined_Msg : in User_Defined_Mail_Msg;
                         Dest_Partition_Prefix : in Dis.Component_Id);

    -- Shutsdown the mailbox messaging system.  Not to be called by partitions.
    procedure Shutdown;

-- ---------------------------
-- -- private
-- --

private

    pragma Inline (Mail_Is_Present);
    pragma Inline (Num_Mail_Msgs);

    pragma Inline (Get_Safestore_Msg);
    pragma Inline (Get_Malfunction_Msg);
    pragma Inline (Get_Enter_Msg);
    pragma Inline (Get_Mega_Msg);
    pragma Inline (Get_Router_Msg);

    pragma Inline (Put_Safestore_Msg);
    pragma Inline (Put_Malfunction_Msg);
    pragma Inline (Put_Enter_Msg);
    pragma Inline (Put_Mega_Msg);
    pragma Inline (Put_Router_Msg);

    -- Storage_Units per word (32 bits)
    Word : constant := 32 / System.Storage_Unit;

    type Internal_Msg_Type is new Set.Integer_32;

    type Message is new Mit.Storage_Units (1 .. Max_Mailbox_Msg_Size);
    -- Message'Size = 2048 * 8 = 16384
    for Message'Size use Max_Mailbox_Msg_Size * System.Storage_Unit;

    type Headers is
           record
               Dest_Partition_Prefix : Dis.Component_Id;
               Msg_Type : Internal_Msg_Type;
               Msg_Size : Natural := 0;
           end record;

    for Headers use
           record at mod 4;
               Dest_Partition_Prefix at 0 * Word range 0 .. 63;
               Msg_Type at 2 * Word range 0 .. 31;
               Msg_Size at 3 * Word range 0 .. 31;
           end record;
    for Headers'Size use 64 + 32 + 32;                          --#VER

    type Mailbox_Message is
           record
               Header : Headers;
               Msg : Message;
           end record;
```

```
    for Mailbox_Message use
           record at mod 4;
               Header at 0 * Word range 0 .. 127;
               Msg at 4 * Word range 0 .. 16383;
           end record;
    for Mailbox_Message'Size use 128 * 16384;                --#VER

    type Mailbox_Message_Ptr is access Mailbox_Message;

    -- Record for SGI
    type Mailboxes is
           record
               Buffer_Ptr : Mit.Mailbox_Buffer_Ptr;
               Desc_Ptr : Mit.Mailbox_Desc_Ptr;
               Tag_Ptr : Mit.Mailbox_Tag_Ptr;
               Routing_Table_Index : Natural := 0;
           end record;

    -- -------------- -----
    -- Frequently us  sizes
    --

    -- Size allocated for mailbox messages
    Mailbox_Message_Size : constant Natural :=
       Message'Size / System.Storage_Unit;

    -- Size of a mailbox message plus its associated header
    Message_Plus_Header_Size : constant Natural :=
       Mailbox_Message_Size + Headers'Size / System.Storage_Unit;

    -- Size of the message header
    Msg_Header_Size : constant Natural := Headers'Size / System.Storage_Unit;

    -- Size of the message type in the header record (it's an integer)
    Msg_Type_Size : constant Natural := Integer'Size / System.Storage_Unit;

end Mailbox;

------------------------------------------------------------------------------
--| Abstract: This package provides the types and operations necessary to
--|           interface with the mailbox communication system.  Each mail
--|           message must be of a specific type (ie. Safestore, Malfunction,
--|           Enter, etc).  The sender and receiver use this type to identify
--|           the kind of message so that they know how to deal with it (how
--|           to build it and how to split it).  This package provides support
--|           for some predefined mail message types such as: Safestore,
--|           Malfunction, and Enter.  These are common messages that will be
--|           used frequently and by many partitions.  Senders and receivers that
--|           use these mail message types are guareented to be using the same
--|           data types for the messages.  This package also provides support
--|           for "user defined" messages that are not widely used or shared
--|           between many partitions.  There are three generic subroutines which
--|           provide support user defined messages: Build_User_Defined_Msg,
--|           Get_User_Defined_Msg, and Split_User_Defined_Msg.  They are
--|           instantiated with the user defined message types.  It is up to
--|           the users to ensure that the sender and receiver agree on the
--|           structure and data type of user defined mail messages.
--|
--| Warnings: This package depends on the use of shared memory and shared
--|           semaphores.  The semaphores are only used during initialization,
--|           not during runtime.
--|
--|           There are two restrictions placed on the type used for User_Def-
```

```
--          ined_Msg_Types.  First, the type must have a size of 32 bits.
--          To insure this a length clause should be used
--          (ex. for Type'Size use 32;).  Second, The values of the type
--          must be positive.  This means that if an enumeration type is
--          used its litterals must not be given negative values with a
--          representation clause.
--
--          This package depends upon compatibility between System.Address
--          and the value of an access type.  It uses Unckecked_Conversion
--          to convert from x'Address to an access type.
-------------------------------------------------------------------------------
```

8.3.1 Enter_Mailbox

```
with Dis, Std_Eng_Types;
package Enter_Mailbox is

    package Set renames Std_Eng_Types;

    -- Data type for IOS Enter mailbox messages

    type Enter_Msg is private;  -- Initialize and IOS Enter data

    generic
        type Data_Type is private;
    procedure Create (Msg : in out Enter_Msg;
                      Id : Dis.Term_Id;
                      Value : Data_Type;
                      Index : Integer := 0);

    procedure Create_R6 (Msg : in out Enter_Msg;
                         Id : Dis.Term_Id;
                         Value : Set.Real_6;
                         Index : Integer := 0);

    procedure Create_R15 (Msg : in out Enter_Msg;
                          Id : Dis.Term_Id;
                          Value : Set.Real_15;
                          Index : Integer := 0);

    procedure Create_I8 (Msg : in out Enter_Msg;
                         Id : Dis.Term_Id;
                         Value : Set.Integer_8;
                         Index : Integer := 0);

    procedure Create_I16 (Msg : in out Enter_Msg;
                          Id : Dis.Term_Id;
                          Value : Set.Integer_16;
                          Index : Integer := 0);

    procedure Create_I32 (Msg : in out Enter_Msg;
                          Id : Dis.Term_Id;
                          Value : Set.Integer_32;
                          Index : Integer := 0);

    procedure Create_String
                (Msg : in out Enter_Msg; Id : Dis.Term_Id; Value : String);

    function Id (Msg : Enter_Msg) return Dis.Term_Id;
    function Index (Msg : Enter_Msg) return Integer;

    generic
        type Data_Type is private;
    function Value (Msg : Enter_Msg) return Data_Type;

    function Value_R6 (Msg : Enter_Msg) return Set.Real_6;
    function Value_R15 (Msg : Enter_Msg) return Set.Real_15;
    function Value_I8 (Msg : Enter_Msg) return Set.Integer_8;
    function Value_I16 (Msg : Enter_Msg) return Set.Integer_16;
    function Value_I32 (Msg : Enter_Msg) return Set.Integer_32;
    function Value_String (Msg : Enter_Msg; Length : Natural) return String;

    procedure Poke (Msg : Enter_Msg);

    Id_Not_Found : exception;
    -- raised when Poke is called with an identifier that
    -- has not been registered at the local level.
```

```
    Id_Not_Connected : exception;
    -- raised when Poke is called with an identifier that
    -- has not been connected with an address.

    Too_Large : exception;
    -- raised by Create if the data type is too big to fit
    -- in the value buffer.

private
    -- secret
end Enter_Mailbox;
```

8.3.2 Malfunction_Mailbox

```
with Dis, Std_Eng_Types;
package Malfunction_Mailbox is

    package Set renames Std_Eng_Types;

    -- Data type for Malfunction mailbox messages

    type Malfunction_Msg is private;  -- Malfunction Messages

    procedure Create (Msg : in out Malfunction_Msg;
                      Id : Dis.Malfunction_Id;
                      On_Or_Off : Set.On_Off := Set.On;
                      Scale : Set.Real_15 := 0.0;
                      Bias : Set.Real_15 := 0.0;
                      Option_Value : Natural := 0);

    function Id (Msg : Malfunction_Msg) return Dis.Malfunction_Id;

    generic
        type Discrete_Type is (<>);
    function Option (Msg : Malfunction_Msg) return Discrete_Type;
    function Option_Value (Msg : Malfunction_Msg) return Natural;

    function P1 (Msg : Malfunction_Msg) return Set.Real_15;
    function P2 (Msg : Malfunction_Msg) return Set.Real_15;
    function State (Msg : Malfunction_Msg) return Set.On_Off;

    procedure Poke (Msg : Malfunction_Msg);

    Bad_Size : exception;
    -- raised by generic Selector or Discrete if the generic actual
    -- parameter (enumeration type) is not 8, 16, or 32 bits long

private
    -- protected from sight
end Malfunction_Mailbox;
```

8.3.3 Safestore_Mailbox

```
with Dis;
package Safestore_Mailbox is

    -- Data type for Return-to-Safestore mailbox messages

    type Safestore_Msg is private;  -- Return to Safestore data

    type Byte is range 0 .. 255;
    for Byte'Size use 8;
    type Value_Buffer is array (Positive range <>) of Byte;

    procedure Create (Msg : in out Safestore_Msg;
                      Id : Dis.Message_Id;
                      Value : Value_Buffer);

    function Id (Msg : Safestore_Msg) return Dis.Message_Id;

    generic
        type Data_Type is private;
    function Value (Msg : Safestore_Msg) return Data_Type;

    Too_Large : exception;
    -- raised by Create if the data type is too big to fit

private
    -- invisible
end Safestore_Mailbox;
```

8.3.4 Mega_Mailbox

```
with Dis, Std_Eng_Types;
package Mega_Mailbox is

    package Set renames Std_Eng_Types;

    Max_Entries : constant := 60;

    type Mega_Msg is private;  -- For many (Term_ID - value) sets at once.

    -- The sender of the Mega_Msg must call Create before appending anything
    -- to the Mega_Msg.  After sending it, the sender should call Clear.
    procedure Create (Msg : in out Mega_Msg);
    procedure Clear (Msg : in out Mega_Msg);

    generic
        type Data_Type is private;
    procedure Append (Msg : in out Mega_Ms,
                      Id : Dis.Term_Id;
                      Value : Data_Type);

    procedure Append_R6
                (Msg : in out Mega_Msg; Id : Dis.Term_Id; Value : Set.Real_6);

    procedure Append_R15
                (Msg : in out Mega_Msg; Id : Dis.Term_Id; Value : Set.Real_15);

    procedure Append_I8 (Msg : in out Mega_Msg;
                         Id : Dis.Term_Id;
                         Value : Set.Integer_8);

    procedure Append_I16 (Msg : in out Mega_Msg;
                          Id : Dis.Term_Id;
                          Value : Set.Integer_16);

    procedure Append_I32 (Msg : in out Mega_Msg;
                          Id : Dis.Term_Id;
                          Value : Set.Integer_32);

    procedure Append_String
                (Msg : in out Mega_Msg; Id : Dis.Term_Id; Value : String);

    -- to avoid the exception Too_Large
    function Appendable (Msg : Mega_Msg; Bits : Integer) return Boolean;

    -- most of the "query"/"selector" operations operate on the "current entry".

    -- All entries in a mega message arrive as "valid".  An entry is
    -- invalidated by poking it or asking for its value.
    -- "Poke_All" will only poke valid entries.
    procedure Poke (Msg : in out Mega_Msg; Only_If_Valid : Boolean := True);
    -- poke the current entry; if it is already invalid, it will not
    -- be poked, unless Only_If_Valid is set to false.  Then it will
    -- be poked anyway.
    procedure Poke_All (Msg : in out Mega_Msg);
    -- poke all valid entries

    -- invalidate an entry : you don't want it poked.
    -- Zero means the current entry.
    procedure Invalidate (Msg : in out Mega_Msg);

    function Number_Of_Entries (Msg : Mega_Msg) return Natural;
    procedure First (Msg : in out Mega_Msg);
```

```
    procedure Next (Msg : in out Mega_Msg);
    procedure Go_To (Msg : in out Mega_Msg;
                     Id : Dis.Term_Id;
                     Found : out Boolean);
    function At_End (Msg : in Mega_Msg) return Boolean;

    -- function Id does not invalidate an entry.
    function Id (Msg : Mega_Msg) return Dis.Term_Id;

    -- retrieving a value invalidates its entry. these will not be
    -- poked automatically by Poke or Poke_All.
    generic
        type Data_Type is private;
    procedure Value (Msg : in out Mega_Msg; Data : out Data_Type);

    procedure Value_R6 (Msg : in out Mega_Msg; Data : out Set.Real_6);
    procedure Value_R15 (Msg : in out Mega_Msg; Data : out Set.Real_15);
    procedure Value_I8 (Msg : in out Mega_Msg; Data : out Set.Integer_8);
    procedure Value_I16 (Msg : in out Mega_Msg; Data : out Set.Integer_16);
    procedure Value_I32 (Msg : in out Mega_Msg; Data : out Set.Integer_32);
    procedure Value_String (Msg : in out Mega_Msg; Data : out String);
    -- be sure the string variable is the correct length

    Not_Created : exception;
    -- raised by Append if the Mega_Msg has not yet been initialized
    -- using Create.

    Too_Large : exception;
    -- raised by an Append if the data type is too big to fit in the
    -- remaining portion of the Mega_Msg.

    Too_Many_Entries : exception;
    -- tried to append more than Max_Entries entries.

    End_Error : exception;
    -- tried to advance beyond the end of the Mega_Msg.

private
    -- ya can't touch this
end Mega_Mailbox;
```

8.4. DIS

```
with System;
with Bunuons;
with Std_Eng_Types;
package Dis is

    package Set renames Std_Eng_Types;

    -- The DIS package is an "object manager".  The managed object is
    -- the Distributed Identifier Spec (DIS) table.  The package
    -- provides a number of abstract data types for the objects which
    -- populate the DIS tree in the body.

    --        ADT                Definition                    Handle
    --
    --    Component_ID           private                 Component_Handle
    --    Term_ID                private                 Term_Handle
    --    Message_ID             private                 Message_Handle
    --    Type_ID                private                 Type_Handle
    --    Type_Tag               open (enumeration)
    --    Malfunction_ID         private                 Malfunction_Handle
    --
    --     The DIS must be searched once to get an identifier's handle,
    -- which points to the node where the data is located.  Then all
    -- access to that identifier's data must use the handle; this reduces
    -- the number of searches.  Dis identifiers (objects with '_ID' suffix)
    -- are unique and distributable among different main programs and
    -- network nodes; the handle may only be used in the context of the
    -- main program in which the conversion has been performed.
    --
    --     Here is an example of a hierarchy of identifiers that can be
    -- placed into the DIS.
    --
    -- Robotics                          (Component_ID)
    --     SPDM                          (Component_ID, prefix => true)
    --         Fail                      (Malfunction_ID)
    --     SSRMS                         (Component_ID, prefix => t  e)
    --         Joint_1_Yaw               (Term_ID)
    --         Joint_1_Roll              (Term_ID)
    --     MT                            (Component_ID)
    --     Roll_Type                     (Type_ID)
    --     Yaw_Type                      (Type_ID)
    -- Environment                       (Component_ID)
    -- USAD                              (Component_ID)
    --     GNC                           (Component_ID)
    --     PROP                          (Component_ID)
    --         PAD                       (Component_ID, prefix => true)
    --             Tank                  (Component_ID array, 6)
    --                 Temp_Sensor       (Term_ID array, 3)
    --                 Fail_Temp_Sensor  (Malfunction_ID array, 3)
    --                 Storage_Leak      (Malfunction_ID);
    --             Current_Pressure      (Term_ID)
    --         Valve_Module              (Component_ID)
    --         Rocket_Assembly           (Component_ID, prefix => true)
    --             Rocket_Engine         (Component_ID multiple, 6)
    --                 Cat_Bed_Fail      (Malfunction_ID array, 2)
    --                 Heater_Fail       (Malfunction_ID)
    --             Thruster              (Component_ID array, 13)
    --                 Cat_Bed           (Component_ID)
    --                 Prop_Valves       (Component_ID)
    --                 Chamber_Pressure  (Term_ID)
```

```
--
-- The identifiers are entered into the DIS by making packages
-- (called _Defs (pronounced "deafs" (sic)) packages, as in
-- USAF_Defs or Rocket_Engine_Defs) which conform to a set of
-- rules and call the Register functions.  See the examples for
-- how these packages look.
--

type Component_Id is private;
Null_Component : constant Component_Id;
type Component_Handle is private;
Null_Comp_Handle : constant Component_Handle;

type Term_Id is private;
Null_Term : constant Term_Id;
type Term_Handle is private;
Null_Term_Handle : constant Term_Handle;

Max_Total_Terms : constant := 100_000;
subtype Term_Index is Set.Natural_32 range 0..Max_Total_Terms;

type Type_Id is private;
Null_Type : constant Type_Id;
type Type_Handle is private;
Null_Type_Handle : constant Type_Handle;

type Message_Id is private;
Null_Message : constant Message_Id;
type Message_Handle is private;
Null_Msg_Handle : constant Message_Handle;

type Malfunction_Id is private;
Null_Malfunction : constant Malfunction_Id;
type Malfunction_Handle is private;
Null_Malf_Handle : constant Malfunction_Handle;

-- Type_Tag is used in Register_Type
type Type_Tag is (Null_Tag,       -- placeholder
                  Integer_Tag,    -- 32 bit integer
                  Short_Tag,      -- 16 bit integer
                  Byte_Tag,       -- 8 bit integer
                  Float_Tag,      -- 32 bit float (SET.Real_6)
                  Double_Tag,     -- 64 bit float (SET.Real_15)
                  Character_Tag, -- a single character
                  String_Tag,     -- a fixed-length string
                  Enum_Tag);      -- for enumeration types

type User is (Look,          -- IOS readable
              Look_Enter,  -- IOS readable & writable
              Initialize); -- datastored & initialized term
type User_List is array (Positive range <>) of User;
Look_Only : constant User_List := (1 => Look);
Look_Initialize : constant User_List := (Look, Initialize);
Look_Enter_Initialize : constant User_List := (Look_Enter, Initialize);

Null_Address : constant System.Address := Eunuchs.Null_Address;

type Address_Array is array (Positive range <>) of System.Address;
Null_Address_Array : constant Address_Array := (1 .. 0 => Null_Address);

type Value_List is array (Natural range <>) of Natural;
Null_Value_List : constant Value_List := (1 .. 0 => 0);
```

```
-- Static operations on the DIS.

-- The DIS is created with static information using the 'Register'
-- routines below.  This static information includes identifiers
-- for objects, types, and malfunctions, as well as descriptor
-- information associated with these entities (such as parameters
-- that are to be used for malfunction routines).  The static
-- information in the DIS is consistent on all platforms and in all
-- applications (including all off-line applications) as long as
-- everyone is using the same version of the DIS.  The DIS is
-- augmented with dynamic information at runtime through the use
-- of the 'Connect' facilities.

function Register_Component (Parent : Component_Id;
                             Name : String;  -- length <= Max_Comp_Name
                             Prefix : Boolean := False;
                             Length : Natural := 0;
                             Labels : String := "") return Component_Id;
-- The Register_Component operation creates a node for a new level
-- in the DIS tree at the position indicated by the Parent parameter.
-- The Length parameter must be used to register multiple components
-- which have the same contents.  This allows a single Defs package
-- to register multiple copies of a set of identifiers.  In this case,
-- the Defs package can be said to resemble a record definition, and
-- the Component_ID array can be said to resemble an array of records.
--
-- A 'prefix' is a Component_ID which is registered with the Prefix
-- parameter set to True.  A prefix is required in the 'ancestry' of
-- any Term_ID or Malfunction_ID.  Also, a prefix's 'descendants' may
-- not include any other prefix Component_IDs.  These rules are
-- enforced by the DIS through the exception Prefix_Error.  A prefix
-- identifies a single mailbox; all Terms_IDs and Malfunction_IDs are
-- delivered to their respective partitions via the mailbox that is
-- identified by the prefix under which they were registered.

function Register_Term (Parent : Component_Id;
                        Name : String;   -- length <= Max_Id_Name
                        The_Type : Type_Id;
                        Users : User_List := Look_Enter_Initialize;
                        Length : Natural := 0;
                        Labels : String := "") return Term_Id;
-- Register_Term requires a Type_ID to indicate how the data
-- is to be interpreted.  A Term_ID array is an aggregate of
-- Term_ID's which have the same type.  A Term_ID array
-- is registered by supplying a Length parameter > 0.  If a
-- labels parameter is supplied, the labels will be used to
-- index the Term_ID array.

function Register_Message (Parent : Component_Id;
                           Name : String;   -- length <= Max_Id_Name
                           Bits : Natural;
                           Safestore : Boolean := False) return Message_Id;
-- A Message_ID is very similar to an Term_ID but is
-- only used for software backplane messages.  This
-- routine must always be supplied with a number of
-- bits.  No type information is supplied.  A flag
-- indicates whether or not the item is to be retrieved
-- for safestore.  Bits is the size of the message in bits.

function Register_Type (Parent : Component_Id;
                        Name : String;    -- length <= Max_Id_Name
                        The_Tag : Type_Tag;
                        Size : Natural := 0;
```

```
                        Low_Bound : String := "";
                        High_Bound : String := "";
                        Values : Value_List := Null_Value_List;
                        Labels : String := "") return Type_Id;
--     Type_IDs provide the ability to interpret data accessed
-- with Term_IDs.  Each Register_Term must be accompanied by
-- a Type_ID parameter.  Each Type_ID is registered using a 'type
-- tag' which indicates which class of type it will belong to.
--
-- For tag:        Required:           Optional:
--
-- Null_Tag        (error)
-- Integer_Tag                         Low or High bound
-- Short_Tag                           Low or High bound
-- Byte_Tag                            Low or High bound
-- Float_Tag                           Low or High bound
-- Double_Tag                          Low or High bound
-- Character_Tag                       Low or High bound
-- String_Tag      Size
-- Enum_Tag        Labels              Size, Values
--
--     When registering a Type_ID for 8, 16, or 32 bit integers (Byte_
-- Tag, Short_Tag, and Integer_Tag respectively), single or double
-- precision floating points (Float_Tag and Double_Tag), or single
-- characters (Character_Tag), a Low or High bound may be supplied.
--
--     Low and high bounds must be in the proper numeric or character
-- order, and must have the correct format (which depends on the type
-- tag -- byte, integer, short, float, double, or character).
--
--     Labels must be supplied for Enum literals.  The Labels
-- parameter specifies a list of names conforming to Ada syntax
-- separated by commas.
--
--     Size must be supplied for String_Tag'd types.  It is
-- optional for enum tag'd types (the default is 8 bits).
-- Size is the number of characters for string tag'd items
-- and the number of bits for types with Enum_Tag.  For
-- strings, Size must not be greater than Max_String.
--
--     Values are the representational aspect of enumeration
-- types; if no list is provided, the default ('POS)
-- numbering (0,1,2) is used; otherwise, each enum value
-- is stored.
--
--     Registration of String_Tag'd Type_IDs requires a Size
-- (the number of characters in the string, as with registering
-- Term_IDs).
--
--     Type_IDs of Enum_Tag must be supplied with a Labels parameter
-- (the DIS wants to see Ada-like identifiers separated by commas for
-- all "labels" parameters).  Optionally, the number of bits that
-- objects of this type use can be specified (the default number is
-- eight bits), as well as a list of values that matches to the
-- enumeration representation of the Ada type (this is not necessary
-- for enumeration types with no rep spec).  These three parameters
-- (bitss, labels, and values) can be obtained by instantiating a
-- generic call Enum_Functions which accepts the Ada enumeration
-- type as an actual parameter.  This is provided so that the user
-- can avoid hard-coding lists of labels and values that duplicate
-- the ones provided in the type declaration and rep spec.  (Enum_
-- Functions can also be used to get a label list from an enumeration
```

```
-- type for calling other DIS functions that have label list para-
-- meters).
--
--     To register an array with the DIS, use the Register_Term
-- function with a Length parameter.  This will create a set
-- of DIS identifiers; a length = 0 creates only one.  All of
-- the terms thus created will have the same type or tag.
--
--     You can simulate record types with the DIS also.  In
-- situations that call for an "array of records", you can define
-- a component id array; the Term_IDs in the "_Defs" package
-- for that component would be analogous to Ada record components.
-- You can also create a generic package with Term_IDs in it; if the
-- package has a generic object parameter of type DIS.Component_ID,
-- and this is used as the "Parent" of the Register_* calls, then the
-- package can be instantiated in the "_Defs" packages anywhere in your
-- DIS hierarchy.
--
--     The Sstf_Defs package registers Type_IDs for the Ada types
-- that are declared in the Std_Eng_Types and Std_Eng_Units packages.
-- For partition data that is declared of these types, you may use
-- the Sstf_Defs Type_IDs directly, or you may register subtypes based
-- on those Type_IDs.  It is best to use subtypes with well-chosen
-- sub-ranges, so that the IOS user can easily manipulate values to
-- be entered.

-- Register an integer-based subtype
function Register_Subtype (Parent : Component_Id;
                           Base : Type_Id;
                           Name : String := "";
                           Low_Bound : Set.Integer_32;
                           High_Bound : Set.Integer_32) return Type_Id;

-- Register a float-based subtype
function Register_Subtype (Parent : Component_Id;
                           Base : Type_Id;
                           Name : String := "";
                           Low_Bound : Set.Real_15;
                           High_Bound : Set.Real_15) return Type_Id;

-- Register a character-based subtype
function Register_Subtype (Parent : Component_Id;
                           Base : Type_Id;
                           Name : String := "";
                           Low_Bound : Character;
                           High_Bound : Character) return Type_Id;

-- These functions are used to register new Type_Ids which are derived
-- from previously registered Type_Ids, which are called "base" Type_Ids.
-- Any Type_Id with tag Byte_Tag, Short_Tag, Integer_Tag, Float_Tag,
-- Double_Tag, or Character_Tag can be used as a base Type_Id.  Base
-- Type_Ids with inappropriate tags will raise Tag_Error.
--
-- Since the only reason to register a type using these functions is to
-- provide different bounds for a previously registered Type_Id, the
-- Low_Bound and High_Bound parameters are not optional.  The bit size
-- of the new Type_Id is the same as that of the base Type_Id.

func    Register_Malfunction (Parent : Component_Id;
                              Name : String;  -- length <= Max_Id_Name
                              Options : Type_Id := Null_Type;
                              P1_Name : String := "";
                              P1_Low : Set.Real_15 := 0.0;
```

```
                                P1_High : Set.Real_15 := 0.0;
                                P1_Type : Type_Id := Null_Type;
                                P2_Name : String := "";
                                P2_Low : Set.Real_15 := 0.0;
                                P2_High : Set.Real_15 := 0.0;
                                P2_Type : Type_Id := Null_Type;
                                Store : Boolean := True;
                                Length : Natural := 0;
                                Labels : String := "") return Malfunction_Id;
-- There are four kinds of malfunctions:
--
--    Simple:  (a.k.a. parameterless)  This is registered
--             by supplying no Options or P1/P2 related
--             parameters.
--
--    Options: This is registered by supplying a type id
--             for the Options parameter.  It must be Enum_Tag'd.
--
--    P1:      This is registered by supplying a string for
--             P1_Name and a type id for P1_Type.  P1_Low
--             and P1_High can be supplied to give different
--             bounds to the parameter that override the low
--             and high limits of P1_Type.  The name of the
--             P1_Type is used as the "units" displayed on IOS.
--
--    P1_P2:   This is registered in the same way as a P1
--             malfunction; rules and options for the P2_
--             parameters are the same as for a P1_ parameters.
--
-- The Store flag indicates whether or not a malfunction is
-- datastored & intitialized.  It defaults to true; do not
-- set it to false; in fact, do not set it at all, since this
-- parameter will be deleted in the near future.  Setting it
-- to False raises Registration_Error.
--
-- To register an array of malfunctions, set Length > 0.
-- Labels may be supplied as an optional parameter; if
-- present, the number of labels supplied must be equal to
-- the Length value.

generic
    type Enum is (<>);
package Enum_Functions is

    -- The Enum_Functions package can be instantiated with any
    -- Ada enumeration type, so that the information needed by
    -- the Register_Type function for Enum_Tag types can be
    -- retrieved automatically.  Instantiating this package does
    -- not modify the DIS table.

    function Labels return String;
    function Num_Labels return Natural;
    function Size return Natural;
    function Values return Value_List;

end Enum_Functions;

-- The Report procedure produces a file which divulges the
-- inner secrets of the entire DIS.  The Load procedure
-- brings such a file into the DIS, populating it without
-- elaborating "_Defs" packages.
```

```
procedure Report (To_File : String;
                  Users : User_List := (Look, Look_Enter, Initialize);
                  Expand : Boolean := False);

-- Load will fail if the file to be loaded has not been
-- created with a complete user list (all users specified),
-- or if the version number in the file does not match the
-- current version number (the DIS maintains a version number
-- for the Load/Report routines).  The exception is Load_Version_Error.
-- File_List => true means From_File is not itself a report
-- file, but contains a list of report files.  Load_Error is
-- raised if the Load file is corrupt or created improperly.
-- Load_Name_Error is raised if a load file does not exist.

procedure Load (From_File : String; File_List : Boolean := True);

-- supply the version number of the Load/Report routines.
function Report_Version return Natural;

-- Registration & general exceptions
Syntax_Error : exception;
-- An identifier name or a label name has improper Ada
-- syntax or exceeds the limit for number of characters
-- (Max_Comp_Name for components, Max_Id_Name for other IDs,
-- and Max_Label_Name for labels; these constants are defined
-- toward the end of the visible part of this package spec).
-- Raised by:
--     Register_ routines (the Name parameter)
--     Register_Component (arrays -- the Labels parameter)
--     Register_Term (arrays -- the Labels parameter)
--     Register_Malfunction_Array (arrays -- the Labels parameter)
--     Register_Type (for types with Labels)

Format_Error : exception;
-- An improperly formatted string was given for a low
-- or high bound (e.g. a low bound for an integer tag'd
-- id is given as "0.0"), or a Labels parameter has
-- improper format.
-- Raised by:
--     Register_Type

Tag_Error : exception;
-- A type tag was used incorrectly (for example,
-- no length parameter was supplied with String_Tag).
-- Raised by:
--     Register_Type
--     Type_ID query routines (String_Length, Label_Index, Values,
--             Label_Value, Value_Index, Low_Bound, High_Bound,
--             Number_Of_Labels)
--     Register_Malfunction (if Options parm is not Enum_Tag'd)
--     Register_Subtype (if the base type is incompatible with the
--             bounds parameters, e.g., the base type has a floating
--             point tag, but the bounds are integers).
--     Connect_Term (using Symbol parameter, if registered symbol
--             has a type incompatible than the DIS Type_ID's Tag).
--     Connect_Malfunction (using Symbol parameters, if a
--             registered symbol's type is not compatible with
--             the type tags required for the DIS Malfunction_ID).

Subtype_Error : exception;
-- The bounds given for the subtype are not a proper sub-range of
-- the bounds of the base type.
-- Raised by:
--     Register_Subtype
```

```
Registration_Error : exception;
-- A Malfunction_ID is being registered, but non-compatible parameters
-- are being supplied to it.  For example, an Options parameter is
-- being supplied as well as a P1_Name parameter.  Or a P2_Name
-- is being supplied but not a P1_Name.  Or Store is set to False.
-- Raised by:
--     Register_Malfunction

Enum_Size_Error : exception;
-- Enumeration objects are not 8, 16, or 32 bits.
-- Raised by:
--     Register_Type
--     Enum_Functions package instantiations

Size_Error : exception;
-- The size, in bits, retrieved from the symbol map for Connect_Term
-- of Connect_Malfunction is different than the size supplied to the
-- DIS (via Register_Type) for the data associated with the symbol.
-- Raised by:
--     Connect_Term        (using Symbol parameter)
--     Connect_Malfunction (using Symbols parameters)

Connect_Error : exception;
-- The parameters supplied as Symbol strings to Connect_Malfunction
-- do not match the parameters given to Register_Malfunction, e.g.,
-- the Malfunction_Id was registered as an Options malf, but a symbol
-- was supplied for the P1 parameter to the Connect routine.  Or a
-- Connect_Malfunction routine was called which required either an
-- array of Malfunction_IDs or a single one, and the other was
-- supplied to it.
-- Raised by:
--     Connect_Malfunction (using Symbols parameters)
--     Connect_Malfunction
--     Connect_Malf_Array

Id_Not_Found : exception;
-- An identifier specified as part of a request was not in the DIS.
-- When converting a string to an ID or handle, it is often caused
-- be a misspelling; it is also of the result of not "with"-ing the
-- "_Defs" package that contains the identifier, or not loading a
-- Dis report file.
-- Raised by:
--     Register_ routines  (the Parent was not found)
--     Handle
--     Convert
--     Prefix_Comp

No_Prefix : exception;
-- The identifier given to a Prefix_Comp function is not
-- associated with any Component_ID prefix.
-- Raised by:
--     Prefix_Comp

Length_Error : exception;
-- The number of labels given for a component,
-- term, or malfunction array registration
-- does not match the Length parameter.
-- Or the Size parameter used for registering
-- a String_Tag'd item is greater then Max_String.
-- Raised by:
--     Register_Component     (for arrays)
--     Register_Term          (for arrays)
--     Register_Malfunction   (for arrays)
--     Register_Type          (for string_tag)
```

```
Limit_Error : exception;
-- The maximum number of identifiers hass been registered
-- under the current Parent component.  The maximum is
-- different for each type of identifier, and the limits
-- are represented by the constants Max_Components, Max_Terms,
-- Max_Types, Max_Messages, and Max_Malfunctions.
-- Raised by:
--     Register_   functions

No_Labels : exception;
-- A label-query routine was called but no labels
-- were registered with the identifier.
-- Raised by:
--     Index functions (Comp_, Term_, & Malfunction_ Handles)
--     Label functions
--     Label_Index
--     Label_Value

Label_Not_Found : exception;
-- The requested label was not found in the list
-- of labels associated with the identifier.
-- Raised by:
--     Convert routines for Component, Term, and Malfunction identifiers
--     Index functions (Comp_, Term_, & Malfunction_ Handles)
--     Label functions
--     Label_Index
--     Label_Value

Value_Not_Found : exception;
-- The value given for an enumeration association
-- is not in the value list, or the index given
-- for an id array or a multiple component is not
-- in the proper range.
-- Raised by:
--     Value_Index
--     Label (Type_Handle)

Index_Error : exception;
-- An index given for a component, term, or malfunction
-- array or for an Enum_Tag'd type identifier, is out of bounds
-- Raised by:
--     lots of things

Not_Array : exception;
-- The operation requires the handle supplied to be
-- pointing to an identifer that has been registered as an
-- array (i.e., the Length parameter was registered > 0).
-- Also, raised by Connect_Term if a Term_Id was passed in
-- which does not represent the first element of a Term_Id
-- array, but Connect_All is True.
-- Raised by:
--    all routines which require a handle for a term, malf,
--        or component array.
--    Connect_Term

Prefix_Error : exception;
-- A Component_ID is being registered as a Prefix, but one
-- of it's ancestors is already a prefix; or, a Term_ID or
-- Malfunction_Id is being registered, but no ancestor
-- component in the Parent is a Prefix.
-- Raised by:
--    Register_Component
--    Register_Term
--    Register_Malfunction
```

```
Duplicate_Error : exception;
-- An identifier has been registered with the same name
-- as another under the same component parent.  It is not
-- permissable to have more than one Term_ID, for instance,
-- called "XYZ" registered under the same parent that already
-- has a Term_ID registered called "XYZ".  However, a Term_Id
-- an a Message_Id (e.g.) can both be registered under the
-- same component parent, and have the same name.  Also, a
-- Term_Id called "XYZ" may be registered even if another
-- "XYZ" Term_Id has already been registered under a different
-- parent.  Only the full name must be unique for a particular
-- kind of identifier.
-- Raise by:
--    Register_Component
--    Register_Term
--    Register_Type
--    Register_Message
--    Register_Malfunction

Load_Version_Error : exception;
-- A report file being read via the Load procedure has
-- a different version number than the current Report version
-- number which the Dis maintains internally.  This is the
-- number returned by the Report_Version function.
-- Raised by:
--    Load

Load_Name_Error : exception;
-- A load file (either the file name given to the Load procedure
-- or a file name in a list of files) does not exist.
-- Raised by:
--    Load

Load_Error : exception;
-- The Load procedure has detected that its input file has
-- an incomplete list of Users in its first line--the list
-- must contain all of the users in the correct order; or,
-- the file (or a file in the file list) does not exist; or,
-- the file has badly formatted lines or is incomplete in
-- some way.
-- Raised by:
--    Load

Null_Error : exception;
-- A null identifier or handle was supplied.
-- Raised by:
--    most query routines
--    Navigate.Next routines

-- Operations on Component_ID and Component_Handle objects.

procedure Create_Symbols
            (The_Component : in out Component_Id; Parent : String);

-- Convert's String argument must contain an Alphanumeric version
-- of the ID ("Robotics.SPDM.Arm(2)").
procedure Convert (String_Component : String;
                   The_Component : out Component_Id;
                   The_Handle : out Component_Handle);
function Convert (String_Component : String) return Component_Id;
function Convert (String_Component : String) return Component_Handle;
function Handle (Of_Component : Component_Id) return Component_Handle;
function Image (Of_Component : Component_Id) return String;
function Value (Of_String : String) return Component_Id;
```

```
function Full_Name (The_Component : Component_Id) return String;
function The_Name (The_Component : Component_Handle) return String;
function Prefix (The_Component : Component_Handle) return Boolean;
function Number_Of_Levels (The_Component : Component_Id) return Natural;
function Subcomponent
            (The_Component : Component_Id; Component_Num : Natural)
           return Component_Id;

-- Functions that work on component arrays.  If the Component
-- passed to these is not an array, then Not_Array is raised.
-- The function Id_Array tells whether a component is one or not.
function Id_Array (The_Component : Component_Handle) return Boolean;
function Length (The_Array : Component_Handle) return Natural;
function Label (The_Array : Component_Handle; Index : Natural := 0)
              return String;
function Index (Of_Array : Component_Handle; Label : String := "")
              return Positive;
function In_Array (The_Component : Component_Id; The_Array : Component_Id)
                  return Boolean;
function Component (Of_Array : Component_Id; Index : Positive)
                    return Component_Id;
function Component (Of_Array : Component_Id; Label : String)
                    return Component_Id;
function Component (Of_Array : Component_Handle; Index : Positive)
                    return Component_Handle;
function Component (Of_Array : Component_Handle; Label : String)
                    return Component_Handle;

type Comp_Id_List is array (Positive range <>) of Component_Id;
type Comp_Handle_List is array (Positive range <>) of Component_Handle;
function Get_Prefixes return Comp_Id_List;
function Get_Prefixes return Comp_Handle_List;

-- The Build function creates a Component_Id by "concatenation" of
-- related Component_Ids.  The following rules apply:
--    *) The components listed must be related as ancestor/decendant.
--    *) They must be in order of ancestor/descendant (e.g. great-
--       great-great-grandparent, grandparent, child).
--    *) Intermediate levels of the lineage may be skipped.
--    *) This function does not enforce these rules, since it
--       would be too expensive time-wise to look up the data.
function Build (Comp_List : Comp_Id_List) return Component_Id;

-- The prefix name is the name given to a prefix when Connect_Prefix
-- is called.  The Partition_Id is added to a prefix by the Dis when
-- Connect_Prefix is called.  Partition_Id returns 0 if Connect_Prefix
-- has not been called for the component.
function Prefix_Name (The_Component : Component_Handle) return String;
function Partition_Id (The_Component : Component_Handle)
    return Set.Integer_32;

Subcomponent_Error : exception;
-- if the Component_Num argument is larger than the
-- number of levels that make up a component or there
-- are no subcomponents.

-- Operations on Term_ID objects.

procedure Create_Symbols (The_Term : in out Term_Id; Parent : String);

-- Convert's String argument must contain an Alphanumeric version
-- of the ID ("Robotics.SPDM.Arm(2).Joint_1_Yaw").
```

```
procedure Convert (String_Term : String;
                   The_Term : out Term_Id;
                   The_Handle : out Term_Handle);
function Convert (String_Term : String) return Term_Id;
function Convert (String_Term : String) return Term_Handle;
function Handle (Of_Term : Term_Id) return Term_Handle;
function Image (Of_Term : Term_Id) return String;
function Value (Of_String : String) return Term_Id;

function Build (Comp : Component_Id; Term : Term_Id) return Term_Id;

function The_Component (The_Term : Term_Id) return Component_Id;
function Full_Name (The_Term : Term_Id) return String;
function The_Name (The_Term : Term_Handle) return String;
function The_Type (The_Term : Term_Handle) return Type_Id;
function The_Type (The_Term : Term_Handle) return Type_Handle;

-- Lookable returns True if the term was registered with Look
-- or Look_Enter in the user list.  Enterable returns True if
-- the Term was registered with Look_Enter in the user list.
function Users (The_Term : Term_Handle) return User_List;
function Lookable (The_Term : Term_Handle) return Boolean;
function Enterable (The_Term : Term_Handle) return Boolean;
function Initializable (The_Term : Term_Handle) return Boolean;

-- operations on Term_ID arrays
function Id_Array (The_Term : Term_Handle) return Boolean;
function Length (The_Term_Array : Term_Handle) return Natural;
function Label (The_Term_Array : Term_Handle; Index : Natural := 0)
                return String;
function Index (The_Term_Array : Term_Handle; Label : String := "")
                return Positive;
function In_Array
             (The_Term : Term_Id; The_Term_Array : Term_Id) return Boolean;
function Term (The_Term_Array : Term_Id; Index : Positive) return Term_Id;
function Term (The_Term_Array : Term_Id; Label : String) return Term_Id;
function Term (The_Term_Array : Term_Handle; Index : Positive)
                return Term_Handle;
function Term (The_Term_Array : Term_Handle; Label : String)
                return Term_Handle;

-- Index operations.
procedure Add_Index (The_Term : Term_Handle; Index : Set.Natural_32);
function The_Index (The_Term : Term_Handle) return Term_Index;

-- If the Term_ID has not been "Connect"-ed, the Read_Address
-- function returns Null_Address.
function Read_Address (The_Term : Term_Handle) return System.Address;
function Prefix_Comp (The_Term : Term_Id) return Component_Id;

-- Operations on Message_ID objects.

procedure Create_Symbols (The_Message : in out Message_Id; Parent : String);

-- Convert's String argument must contain an Alphanumeric version
-- of the ID ("Robotics.SPDM.Arm(2).IF_Packet1").
procedure Convert (String_Message : String;
                   The_Message : out Message_Id;
                   The_Handle : out Message_Handle);
function Convert (String_Message : String) return Message_Id;
function Convert (String_Message : String) return Message_Handle;
function Handle (Of_Message : Message_Id) return Message_Handle;
function Build (Comp : Component_Id; Msg : Message_Id) return Message_Id;
```

```
function Image (Of_Message : Message_Id) return String;
function Value (Of_String : String) return Message_Id;

function The_Component (The_Message : Message_Id) return Component_Id;
function Full_Name (The_Message : Message_Id) return String;
function The_Name (The_Message : Message_Handle) return String;
function Size (The_Message : Message_Handle) return Natural;
function Safestore (The_Message : Message_Handle) return Boolean;

function Prefix_Comp (The_Message : Message_Id) return Component_Id;

-- Operations on Type_ID objects.

procedure Create_Symbols (The_Type : in out Type_Id; Parent : String);

-- Convert's String argument must contain an Alphanumeric version
-- of the ID ("Robotics.Position_Vector").
procedure Convert (String_Type : String;
                   The_Type : out Type_Id;
                   The_Handle : out Type_Handle);
function Convert (String_Type : String) return Type_Id;
function Convert (String_Type : String) return Type_Handle;
function Handle (Of_Type : Type_Id) return Type_Handle;
function Build (Comp : Component_Id; Typ : Type_Id) return Type_Id;
function Image (Of_Type : Type_Id) return String;
function Value (Of_String : String) return Type_Id;

function The_Component (The_Type : Type_Id) return Component_Id;
function Full_Name (The_Type : Type_Id) return String;
function The_Name (The_Type : Type_Handle) return String;
function The_Tag (The_Type : Type_Handle) return Type_Tag;
function Is_Subtype (The_Type : Type_Handle) return Boolean;
function String_Length (The_Type : Type_Handle) return Natural;
function Size (The_Type : Type_Handle) return Natural;

-- The following functions are useful for Enum_Tag'd Type_IDs; they
-- provide access to the information related to Labels, 'Pos-like
-- indexes, and representation values.  A "Label" is a string that
-- stands for an enumeration literal.  An "Index" is a numeral that
-- represents the position of a literal within the enumeration list
-- [the kind of value returned by Enum_Type'Pos(Literal)].  A "Value"
-- is the representation numeral supplied with an enumeration
-- representation clause.  Because Enum_Type'Pos starts with zero (0),
-- the DIS uses zero as the index to the first element of the
-- enumeration type_id; the label list array starts its index as zero
-- also.  This contrasts with the indexes for labels of term, component
-- and malfunction identifier arrays, which start at one.
--
-- o Label returns a String value given an number which (1) is
--   used as an Index into a list of label strings, if no Value_List
--   was supplied during Register_Type; or (2) is used as a Value
--   if a Value_List was supplied.
-- o Label_Index returns a 'Pos-like Index for enum tag'd types.
-- o Label_Value returns a 'Pos-like Index if the Type_ID
--   was registered without a Value_List parameter, or the appropriate
--   representation value id there is an associated Value_List.
-- o Value_Index returns a 'Pos-like Index given a Value.
-- o Values returns a Value_List entity which is indexed from zero.
--   The enum type's 'Pos value directly accesses the corresponding
--   representation value.
function Number_Of_Labels (The_Type : Type_Handle) return Natural;
function Label (The_Type : Type_Handle; Index : Natural) return String;
function Label_Index
```

```
            (The_Type : Type_Handle; Label : String) return Natural;
function Label_Value
            (The_Type : Type_Handle; Label : String) return Natural;
function Value_Index
            (The_Type : Type_Handle; Value : Natural) return Natural;
function Values (The_Type : Type_Handle) return Value_List;

function Low_Bound (The_Type : Type_Handle) return Set.Integer_32;
function High_Bound (The_Type : Type_Handle) return Set.Integer_32;
function Low_Bound (The_Type : Type_Handle) return Set.Integer_16;
function High_Bound (The_Type : Type_Handle) return Set.Integer_16;
function Low_Bound (The_Type : Type_Handle) return Set.Integer_8;
function High_Bound (The_Type : Type_Handle) return Set.Integer_8;
function Low_Bound (The_Type : Type_Handle) return Set.Real_6;
function High_Bound (The_Type : Type_Handle) return Set.Real_6;
function Low_Bound (The_Type : Type_Handle) return Set.Real_15;
function High_Bound (The_Type : Type_Handle) return Set.Real_15;
function Low_Bound (The_Type : Type_Handle) return Character;
function High_Bound (The_Type : Type_Handle) return Character;

-- Operations on Malfunction_ID objects.

procedure Create_Symbols
            (The_Malf : in out Malfunction_Id; Parent : String);

-- Convert's String argument must contain an Alphanumeric version
-- of the ID ("Robotics.SPDM.Fail").
procedure Convert (String_Malf : String;
                   The_Malf : out Malfunction_Id;
                   The_Handle : out Malfunction_Handle);
function Convert (String_Malf : String) return Malfunction_Id;
function Convert (String_Malf : String) return Malfunction_Handle;
function Handle (Of_Malf : Malfunction_Id) return Malfunction_Handle;
function Build (Comp : Component_Id; Malf : Malfunction_Id)
               return Malfunction_Id;
function Image (Of_Malf : Malfunction_Id) return String;
function Value (Of_String : String) return Malfunction_Id;

function The_Component (The_Malf : Malfunction_Id) return Component_Id;
function Full_Name (The_Malf : Malfunction_Id) return String;
function The_Name (The_Malf : Malfunction_Handle) return String;
type Malf_Kind is (Simple_Malf, Options_Malf, P1_Malf, P1_P2_Malf);

function Kind (The_Malf : Malfunction_Handle) return Malf_Kind;

function Options_Type (The_Malf : Malfunction_Handle) return Type_Handle;
function Options_Address
            (The_Malf : Malfunction_Handle) return System.Address;

function P1_Name (The_Malf : Malfunction_Handle) return String;
function P1_Low (The_Malf : Malfunction_Handle) return Set.Real_15;
function P1_High (The_Malf : Malfunction_Handle) return Set.Real_15;
function P1_Type (The_Malf : Malfunction_Handle) return Type_Handle;
function P1_Address (The_Malf : Malfunction_Handle) return System.Address;

function P2_Name (The_Malf : Malfunction_Handle) return String;
function P2_Low (The_Malf : Malfunction_Handle) return Set.Real_15;
function P2_High (The_Malf : Malfunction_Handle) return Set.Real_15;
function P2_Type (The_Malf : Malfunction_Handle) return Type_Handle;
function P2_Address (The_Malf : Malfunction_Handle) return System.Address;

function Active_Address
            (The_Malf : Malfunction_Handle) return System.Address;
```

```
function Stored (The_Malf : Malfunction_Handle) return Boolean;

function Id_Array (The_Malf : Malfunction_Handle) return Boolean;
function Length (The_Malf_Array : Malfunction_Handle) return Positive;
function Label (The_Malf_Array : Malfunction_Handle; Index : Natural := 0)
              return String;
function Index (The_Malf_Array : Malfunction_Handle; Label : String := "")
              return Positive;
function In_Array
            (The_Malf : Malfunction_Id; The_Malf_Array : Malfunction_Id)
            return Boolean;
function Malf (The_Malf_Array : Malfunction_Id; Index : Positive)
              return Malfunction_Id;
function Malf (The_Malf_Array : Malfunction_Id; Label : String)
              return Malfunction_Id;
function Malf (The_Malf_Array : Malfunction_Handle; Index : Positive)
              return Malfunction_Handle;
function Malf (The_Malf_Array : Malfunction_Handle; Label : String)
              return Malfunction_Handle;

function Prefix_Comp (The_Malf : Malfunction_Id) return Component_Id;

-- Connect facilities (Dynamic augmentation of the DIS)

-- The DIS is augmented with dynamic information at run time.  This
-- includes such things as the addresses of data items that
-- are to be associated with identifiers and locations of models
-- in the network.

-- This adds address information to the identifier.  Connect_Term adds
-- an address to a single term registered with the DIS.  However,
-- these Connects are also used for Term_Id arrays that map to an
-- Ada array.  If Connect_All is True (default) and Term represents
-- an ID array, only the address of the first element in the Ada array
-- need be supplied.  The rest of the addresses will be calculated
-- by the DIS.  This will only work if the Ada array is contiguous in
-- memory and the component addresses can be calculated using the
-- address of the first.  Connect_All is ignored if the term supplied
-- does not represent an ID array.  If it is False, even if the Term
-- ID supplied represents an ID array, only one address & term will be
-- connected by the routine.
procedure Connect_Term (Term : Term_Id;
                        Address : System.Address;
                        Connect_All : Boolean := False);

-- An alternate version of Connect_Term uses the symbol map
-- string to derive the address.  It works exactly like the straight
-- address version, including its behavior for Term_ID arrays.
procedure Connect_Term (Term : Term_Id;
                        Symbol : String;
                        Connect_All : Boolean := False);

-- Use this routine when the Term_ID array maps to a set of Ada terms
-- that are not contiguous in memory in such a way that Connect_Term
-- can simply calculate all of the appropriate addresses using the
-- first one.  The array of addresses must be the same length as the
-- previously registered term array.
procedure Connect_Term_Array
             (Term_Array : Term_Id; Addresses : Address_Array);

-- Connecting a malfunction means supplying the addresses for the
-- parameters associated with the malf.  These are the addresses of:
--     1) the P1 parameter,
```

```
--      2) the P2 parameter,
--      3) the discrete options parameter, and
--      4) the malfunction active flag
-- Use the version of Connect_Malfunction call below which permits
-- the use of symbols.  An address or symbol must be supplied for
-- all the parameters that apply to a particular malfunction.  For
-- example, if there is a P1 parameter but no P2 parameter associated
-- with a malfunction, and a Connect call is made for that malfunction,
-- there must be an address or symbol for the first element in the
-- array and a Null_Address and null symbol string for the second
-- element.  In all cases, an address or symbol is required for the
-- active flag.  Connect_Error is raised if these rules are violated.

type Malf_Addressable is (P1_Addr, P2_Addr, Options_Addr, Active_Addr);
type Malf_Addresses is array (Malf_Addressable) of System.Address;

-- This procedure is obsolete.  Phase it out and
-- use the version below instead.
procedure Connect_Malfunction
             (The_Malf : Malfunction_Id; Addresses : Malf_Addresses);

-- You can connect any malfunction parameter using a symbol or
-- using an address.  If you supply a symbol for a particular
-- parameter, you may not supply an address, and vice versa.
-- However, you may supply a symbol for one parameter and an
-- address for a different parameter.
procedure Connect_Malfunction
             (The_Malf : Malfunction_Id;
              Active_Symbol : String := "";
              P1_Symbol : String := "";
              P2_Symbol : String := "";
              Options_Symbol : String := "";
              Active_Address : System.Address := Null_Address;
              P1_Address : System.Address := Null_Address;
              P2_Address : System.Address := Null_Address;
              Options_Address : System.Address := Null_Address);

-- This call is not very useful; it doesn't seem to save much
-- coding over the Connect_Malfunction call, is probably error-prone,
-- and doesn't have the advantage of checking against data in the
-- symbol table.  It may be phased out in the future.
procedure Connect_Malf_Array
             (The_Malf_Array : Malfunction_Id;
              Active_Addresses : Address_Array;
              Options_Addresses : Address_Array := Null_Address_Array;
              P1_Addresses : Address_Array := Null_Address_Array;
              P2_Addresses : Address_Array := Null_Address_Array);

-- Adds location information (node and process ID)
-- to the Component identifier.
procedure Connect_Prefix (To_Comp : Component_Id; Prefix_Name : String);

-- Constants and magic numbers.

-- The maximum String length for String_Tag'd Type_IDs.
Max_String : constant := 40;

-- The maximum number of component levels.
Max_Levels : constant := 7;

-- The maximum number of identifiers that can be registered
-- per level.
Max_Components : constant := 255;
```

```
Max_Terms : constant := 65535;
Max_Types : constant := 255;
Max_Messages : constant := 255;
Max_Malfunctions : constant := 255;

-- The DIS Identifier String length constants.
--
--     Each DIS identifier can be represented in two ways: as a
-- string or as a set of integer values.  The Register_ functions
-- provide the integer-set version to the registering application.
-- Each identifer type has a Convert function which takes a String
-- value and produces an identifier type value (the integer-set).
-- The Full_Name function takes an identifier value and produces a
-- String value.  Examples of identifier string formats are:
--
-- comp_id.comp_id.term_id                    (simplest form)
-- comp_id.comp_id(3).type_id                 (indexed by 1..Length)
-- comp_id.comp_id.term_id(Left_Engine)       (indexed, user-def labels)
-- comp_id.comp_id(2).comp_id(NW).term_id     (mix 'n' match 'em)
-- comp_id.comp_id.comp_id....msg_id          (up to Max_Level Component_IDs)
--
--     Each Component_ID in the string can be up to Max_Comp_Name
-- characters long.  Each of the other identifiers can be up to
-- Max_ID_Name characters long.  Subscripts can be as long as
-- Max_Label_Name characters.

Max_Label_Name : constant := 30; -- max length for label names
-- this includes enumeration labels, and subscript labels for
-- identifier arrays (component, term, and malfunction types).

Max_Comp_Name : constant := 20;  -- max length for component id names

Max_Id_Name : constant := 40;    -- max length for other id names
Max_Malf_Name : constant := Max_Id_Name - 2;
-- Register_Malfunction tacks on a '_x' suffix to malf-related terms
Max_Subtype_Name : constant := Max_Id_Name - 3;
-- Register_Subtype tacks on a '_xx' suffix to the base type name

Max_Full_Name : constant :=
   (Max_Levels)    (Max_Comp_Name + 1) + (Max_Label_Name + 2)) +
      Max_Id_Name + (Max_Label_Name + 2);
-- Max length for fully qualified identifier names.
-- Leave enough room for max levels of components.  Each
-- component can be Max_Comp_Name + 1 (for the period) plus
-- the Max_Label_Name + 2 (for the parentheses).  And then
-- Max_ID_Name chars for the lowest level of identifier plus
-- Max_Label_Name + 2.

-- Package Navigate is used by tools that need to traverse the
-- memory-resident DIS tree.

package Navigate is

    -- Get the head DIS handle.
    function Head return Component_Handle;

    -- Get the handles for the child lists.
    function Comp_Children (Comp : Component_Handle)
                           return Component_Handle;
    function Type_Children (Comp : Component_Handle) return Type_Handle;
    function Term_Children (Comp : Component_Handle) return Term_Handle;
    function Msg_Children (Comp : Component_Handle) return Message_Handle;
    function Malf_Children (Comp : Component_Handle)
                           return Malfunction_Handle;
```

```
        -- Get the "next" handle in list.  "Next" looks at only the first
        -- handle of an ID array; "Next1" looks at all handles.
        function Next (Comp : Component_Handle) return Component_Handle;
        function Next1 (Comp : Component_Handle) return Component_Handle;
        function Next (Typ : Type_Handle) return Type_Handle;
        function Next (Term : Term_Handle) return Term_Handle;
        function Next1 (Term : Term_Handle) return Term_Handle;
        function Next (Msg : Message_Handle) return Message_Handle;
        function Next (Malf : Malfunction_Handle) return Malfunction_Handle;
        function Next1 (Malf : Malfunction_Handle) return Malfunction_Handle;

        -- Don't try to go past the end of a list, or you'll get the
        -- Null_Error.  Compare handles returned from Navigate's routines
        -- to the Null_<ID_Type>_Handle constants declared at the top
        -- of the DIS package spec.

    end Navigate;

private
    -- hidden from sight
end Dis;
--
--  Abstract: The DIS is used to create a set of logical names for
--            off-line and inter-model data references.  The DIS
--            internal data-base can be loaded in one of two ways:
--            through the elaboration of packages defining identifiers,
--            or by loading a file which contains the previously-
--            stored state of the DIS.  The first method is for
--            real-time models, the second is for off-line tools.
--            Models connect their data variables to the DIS logical
--            names with the "Connect" routines.
--
--  Warnings: Be sure to follow all the rules for DIS "_Defs" package
--            creation.  The DIS assigns identifier values in a strict
--            order indepentent of the elaboration order of the _Defs
--            packages; this scheme only works if the rules are followed.
--
```

8.5. SSTF_Defs

```
with Dis, Std_Eng_Types, Std_Eng_Units;
package Sstf_Defs is

    package Set renames Std_Eng_Types;

    -- The top-level "Component_IDs"

    Robotics : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "Robotics");
    Environment : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "Environment");
    Usad : constant Dis.Component_Id := Dis.Register_Component
                                          (Dis.Null_Component, "Distributed");
    Usav : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "USAV");
    Obcs : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "OBCS");
    Visual : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "Visual");
    Nts : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "NTS");
    Rts : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "RTS");
    Ios : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "IOS");
    Ops_Tools : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "Ops_Tools");
    Sac : constant Dis.Component_Id :=
       Dis.Register_Component (Dis.Null_Component, "SAC");

    -- The top-level "Type_IDs" (and type tags renamed from DIS).

    package Bf is new Dis.Enum_Functions (Boolean);
    Boolean : constant Dis.Type_Id :=
       Dis.Register_Type (Dis.Null_Component, "Boolean", D    um_Tag,
                          Labels => Bf.Labels,
                          Size => Bf.Size);

    Character : constant Dis.Type_Id :=
       Dis.Register_Type (Dis.Null_Component, "Character", Dis.Character_Tag,
                          Low_Bound => (1 => Ascii.Nul),    -- ascii 0
                          High_Bound => (1 => Ascii.Del)); -- ascii 127

    Graphic_Character : constant Dis.Type_Id :=
       Dis.Register_Subtype (Dis.Null_Component,
                             Base => Character,
                             Name => "Graphic_Character",
                             Low_Bound => ' ',     -- blank
                             High_Bound => '~');  -- tilde

    -- Logical types for the Fortran guys; use the Boolean labels.

    Logical_1 : constant Dis.Type_Id :=
       Dis.Register_Type (Dis.Null_Component, "Logical_1", Dis.Enum_Tag,
                          Labels => Bf.Labels,
                          Size => 8);
    Logical_2 : constant Dis.Type_Id :=
       Dis.Register_Type (Dis.Null_Component, "Logical_2", Dis.Enum_Tag,
                          Labels => Bf.Labels,
                          Size => 16);
    Logical_4 : constant Dis.Type_Id :=
```

```
   Dis.Register_Type (Dis.Null_Component, "Logical_4", Dis.Enum_Tag,
                      Labels => Bf.Labels,
                      Size => 32);

-- Standard Engineering Types

Integer_8 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Integer_8", Dis.Byte_Tag);
Natural_8 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Natural_8", Dis.Byte_Tag,
                      Low_Bound => "0");
Positive_8 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Positive_8", Dis.Byte_Tag,
                      Low_Bound => "1");

Integer_16 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Integer_16", Dis.Short_Tag);
Natural_16 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Natural_16", Dis.Short_Tag,
                      Low_Bound => "0");
Positive_16 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Positive_16", Dis.Short_Tag,
                      Low_Bound => "1");

Integer_32 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Integer_32", Dis.Integer_Tag);
Natural_32 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Natural_32", Dis.Integer_Tag,
                      Low_Bound => "0");
Positive_32 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Positive_32", Dis.Integer_Tag,
                      Low_Bound => "1");

Real_6 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Real_6", Dis.Float_Tag);
Real_15 : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Real_15", Dis.Double_Tag);

-- other types from SET

package Af is new Dis.Enum_Functions (Set.Active_Inactive);
Active_Inactive : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Active_Inactive", Dis.Enum_Tag,
                      Labels => Af.Labels,
                      Size => Af.Size);

package Ast is new Dis.Enum_Functions (Set.Actual_Sensed);
Actual_Sensed : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Actual_Sensed", Dis.Enum_Tag,
                      Labels => Ast.Labels,
                      Size => Ast.Size);

package Aut is new Dis.Enum_Functions (Set.Attached_Unattached);
Attached_Unattached : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Attached_Unattached", Dis.Enum_Tag,
                      Labels => Aut.Labels,
                      Size => Aut.Size);

package Avf is new Dis.Enum_Functions (Set.Available_Unavailable);
Availabile_Unavailable : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Availabile_Unavailable", Dis.Enum_Tag,
```

```
                    Labels => Avf.Labels,
                    Size => Avf.Size);

package Bzf is new Dis.Enum_Functions (Set.Busy_Not_Busy);
Busy_Not_Busy : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Busy_Not_Busy", Dis.Enum_Tag,
                    Labels => Bzf.Labels,
                    Size => Bzf.Size);

package Cnct is new Dis.Enum_Functions (Set.Closed_Not_Closed);
Closed_Not_Closed : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Closed_Not_Closed", Dis.Enum_Tag,
                    Labels => Cnct.Labels,
                    Size => Cnct.Size);

package Unt is new Dis.Enum_Functions (Set.Connected_Unconnected);
Connected_Unconnected : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                    "Connected_Unconnected", Dis.Enum_Tag,
                    Labels     nt.Labels,
                    Size =     .Size);

package Cf is new Dis.Enum_Functions (Set.Complete_Incomplete);
Complete_Incomplete : constant Dis.Type_Id :=
   Dis.Register_Type
      (Dis.Null_Component, "Complete_Incomplete", Dis.Enum_Tag,
       Labels => Cf.Labels,
       Size => Cf.Size);

package Ef is new Dis.Enum_Functions (Set.Empty_Not_Empty);
Empty_Not_Empty : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Empty_Not_Empty", Dis.Enum_Tag,
                    Labels => Ef.Labels,
                    Size => Ef.Size);

package Ent is new Dis.Enum_Functions (Set.Enabled_Disabled);
Enabled_Disabled : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Enabled_Disabled", Dis.Enum_Tag,
                    Labels => Ent.Labels,
                    Size => Ent.Size);

package Gnf is new Dis.Enum_Functions (Set.Go_No_Go);
Go_No_Go : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Go_No_Go", Dis.Enum_Tag,
                    Labels => Gnf.Labels,
                    Size => Gnf.Size);

package Tf is new Dis.Enum_Functions (Set.In_Tune_Not_In_Tune);
In_Tune_Not_In_Tune : constant Dis.Type_Id :=
   Dis.Register_Type
      (Dis.Null_Component, "In_Tune_Not_In_Tune", Dis.Enum_Tag,
       Labels => Tf.Labels,
       Size => Tf.Size);

package Iot is new Dis.Enum_Functions (Set.Input_Output);
Input_Output : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Input_Output", Dis.Enum_Tag,
                    Labels => Iot.Labels,
                    Size => Iot.Size);

package Int is new Dis.Enum_Functions (Set.Ios_Nominal);
Ios_Nominal : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Ios_Nominal", Dis.Enum_Tag,
```

```
                        Labels => Int.Labels,
                        Size => Int.Size);

package Onf is new Dis.Enum_Functions (Set.On_Off);
On_Off : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "On_Off", Dis.Enum_Tag,
                        Labels => Onf.Labels,
                        Size => Onf.Size);

package Opf is new Dis.Enum_Functions (Set.Open_Closed);
Open_Closed : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Open_Closed", Dis.Enum_Tag,
                        Labels => Opf.Labels,
                        Size => Opf.Size);

package Ont is new Dis.Enum_Functions (Set.Open_Not_Open);
Open_Not_Open : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Open_Not_Open", Dis.Enum_Tag,
                        Labels => Ont.Labels,
                        Size => Ont.Size);

package Ovt is new Dis.Enum_Functions (Set.Override_Not_Override);
Override_Not_Override : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                        "Override_Not_Override", Dis.Enum_Tag,
                        Labels => Ovt.Labels,
                        Size => Ovt.Size);

package Pdf is new Dis.Enum_Functions (Set.Pending_Not_Pending);
Pending_Not_Pending : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                        "Pending_Not_Pending", Dis.Enum_Tag,
                        Labels => Pdf.Labels,
                        Size => Pdf.Size);

package Pf is new Dis.Enum_Functions (Set.Present_Absent);
Present_Absent : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Present_Absent", Dis.Enum_Tag,
                        Labels => Pf.Labels,
                        Size => Pf.Size);

package Rst is new Dis.Enum_Functions (Set.Reset_Not_Reset);
Reset_Not_Reset : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Reset_Not_Reset", Dis.Enum_Tag,
                        Labels => Rst.Labels,
                        Size => Rst.Size);

package Rf is new Dis.Enum_Functions (Set.Right_Wrong);
Right_Wrong : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Right_Wrong", Dis.Enum_Tag,
                        Labels => Rf.Labels,
                        Size => Rf.Size);

package Sbt is new Dis.Enum_Functions (Set.Scale_Bias);
Scale_Bias : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Scale_Bias", Dis.Enum_Tag,
                        Labels => Sbt.Labels,
                        Size => Sbt.Size);

package Syt is new Dis.Enum_Functions (Set.Sync_Not_Sync);
Sync_Not_Sync : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Sync_Not_Sync", Dis.Enum_Tag,
                        Labels => Syt.Labels,
                        Size => Syt.Size);
```

```
package Ult is new Dis.Enum_Functions (Set.Unlocked_Locked);
Unlocked_Locked : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Unlocked_Locked", Dis.Enum_Tag,
                      Labels => Ult.Labels,
                      Size => Ult.Size);

-- String types

Assets : constant Dis.Type_Id :=
   Dis.Register_Type
      (Dis.Null_Component, "Assets", Dis.String_Tag, Size => 12);

Nodes : constant Dis.Type_Id :=
   Dis.Register_Type
      (Dis.Null_Component, "Nodes", Dis.String_Tag, Size => 12);

Sessions : constant Dis.Type_Id :=
   Dis.Register_Type
      (Dis.Null_Component, "Sessions", Dis.String_Tag, Size => 12);

-- Standard Engineering Units
--
-- Mostly consists of "renames" of earlier Type_IDs.  Note that the
-- string displayed for such renamed entities is the name given for
-- the original type identifier, which is not the same as the Ada
-- name below.

--    Time types

Seconds : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Seconds", Dis.Float_Tag);
Microseconds : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Microseconds", Dis.Float_Tag);
Milliseconds : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Milliseconds", Dis.Float_Tag);
Minutes : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Minutes", Dis.Float_Tag);
Hours : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Hours", Dis.Float_Tag);
Days : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Days", Dis.Float_Tag);
Days_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Days_DP", Dis.Double_Tag);

--    Physical types  (are you the ... ?)

Meters : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Meters", Dis.Float_Tag);
Meters_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Meters_DP", Dis.Double_Tag);
Square_Meters : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Square_Meters", Dis.Float_Tag);
Cubic_Meters : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Cubic_Meters", Dis.Float_Tag);
Meters_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Meters_Per_Second", Dis.Float_Tag);
Meters_Per_Second_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Meters_Per_Second_DP", Dis.Double_Tag);
Meters_Per_Second_Squared : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Meters_Per_Second_Squared", Dis.Float_Tag);
```

```
Meters_Per_Second_Squared_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Meters_Per_Second_Squared_DP", Dis.Double_Tag);
Reciprocal_Meters : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Reciprocal_Meters", Dis.Float_Tag);

Kilograms : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Kilograms", Dis.Float_Tag);
Kilograms_Kelvin : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Kilograms_Kelvin", Dis.Float_Tag);
Kilograms_Square_Meter : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Kilograms_Square_Meter", Dis.Float_Tag);
Kilograms_Per_Cubic_Meter : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Kilograms_Per_Cubic_Meter", Dis.Float_Tag);
Kilograms_Per_Cubic_Meter_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Kilograms_Per_Cubic_Meter_DP", Dis.Double_Tag);
Cubic_Meters_Per_Kilogram : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Cubic_Meters_Per_Kilogram", Dis.Float_Tag);

Joules : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Joules", Dis.Float_Tag);
Joules_Per_Kelvin : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Joules_Per_Kelvin", Dis.Float_Tag);
Joules_Per_Kilogram_Kelvin : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Joules_Per_Kilogram_Kelvin", Dis.Float_Tag);

Moles : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Moles", Dis.Float_Tag);
Moles_Per_Cubic_Meter : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Moles_Per_Cubic_Meter", Dis.Float_Tag);

Newtons : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Newtons", Dis.Float_Tag);
Newtons_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Newtons_DP", Dis.Double_Tag);
Newton_Meters : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Newton_Meters", Dis.Float_Tag);
Newton_Meters_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Newton_Meters_DP", Dis.Double_Tag);
Newton_Square_Meter : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Newtons_Square_Meter", Dis.Float_Tag);
Newtons_Per_Meter : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Newtons_Per_Meter", Dis.Float_Tag);

Pascals : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Pascals", Dis.Float_Tag);
Pascals_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Pascals_Per_Second", Dis.Float_Tag);

Degrees : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Degrees", Dis.Float_Tag);
```

```
Degrees_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Degrees_DP", Dis.Double_Tag);
Degrees_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Degrees_Per_Second", Dis.Float_Tag);
Degrees_Per_Second_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Degrees_Per_Second_DP", Dis.Double_Tag);
Degrees_Per_Second_Squared : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Degrees_Per_Second_Squared", Dis.Float_Tag);
Degrees_Per_Second_Squared_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Degrees_Per_Second_Squared_DP", Dis.Double_Tag);
Radians : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Radians", Dis.Float_Tag);
Radians_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Radians_DP", Dis.Double_Tag);
Radians_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Radians_Per_Second", Dis.Float_Tag);
Radians_Per_Second_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Radians_Per_Second_DP", Dis.Double_Tag);
Radians_Per_Second_Squared : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Radians_Per_Second_Squared", Dis.Float_Tag);
Radians_Per_Second_Squared_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Radians_Per_Second_Squared_DP", Dis.Double_Tag);
Steradians : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Steradians", Dis.Float_Tag);
Steradians_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Steradians_Per_Second", Dis.Float_Tag);
Steradians_Per_Second_Squared : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Steradians_Per_Second_Squared", Dis.Float_Tag);

Watts : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Watts", Dis.Float_Tag);
Kilowatts : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Kilowatts", Dis.Float_Tag);
Watts_Per_Square_Meter : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Watts_Per_Square_Meter", Dis.Float_Tag);
Watts_Per_Steradian : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Watts_Per_Steradian", Dis.Float_Tag);
Watts_Per_Square_Meter_Steradian : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Watts_Per_Square_Meter_Steradian", Dis.Float_Tag);

--     English types

Inches : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Inches", Dis.Float_Tag);
Inches_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Inches_DP", Dis.Double_Tag);
Square_Inches : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Square_Inches", Dis.Float_Tag);
Cubic_Inches : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Cubic_Inches", Dis.Float_Tag);
```

```
Feet : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Feet", Dis.Float_Tag);
Feet_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Feet_DP", Dis.Double_Tag);
Square_Feet : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Square_Feet", Dis.Float_Tag);
Square_Feet_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Square_Feet_DP", Dis.Double_Tag);
Cubic_Feet : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Cubic_Feet", Dis.Float_Tag);
Cubic_Feet_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Cubic_Feet_DP", Dis.Double_Tag);
Feet_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Feet_Per_Second", Dis.Float_Tag);
Feet_Per_Second_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Feet_Per_Second_DP", Dis.Double_Tag);
Feet_Per_Second_Squared : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Feet_Per_Second_Squared", Dis.Float_Tag);
Feet_Per_Second_Squared_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Feet_Per_Second_Squared_DP", Dis.Double_Tag);
Miles : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Miles", Dis.Float_Tag);
Miles_Per_Hour : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Miles_Per_Hour", Dis.Float_Tag);
Nautical_Miles : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Nautical_Miles", Dis.Float_Tag);
Nautical_Miles_Per_Hour : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Nautical_Miles_Per_Hour", Dis.Float_Tag);

Gallons : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Gallons", Dis.Float_Tag);
Gallons_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Gallons_Per_Second", Dis.Float_Tag);
Quarts : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Quarts", Dis.Float_Tag);

Pounds_Mass : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Pounds_Mass", Dis.Float_Tag);
Pounds_Mass_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Pounds_Mass_Per_Second", Dis.Float_Tag);
Slugs : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Slugs", Dis.Float_Tag);
Slugs_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Slugs_DP", Dis.Double_Tag);
Tons : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Tons", Dis.Float_Tag);

Pounds_Mass_Per_Cubic_Inch : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Pounds_Mass_Per_Cubic_Inch", Dis.Float_Tag);
Pounds_Mass_Per_Cubic_Foot : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Pounds_Mass_Per_Cubic_Foot", Dis.Float_Tag);
Pounds_Mass_Per_Cubic_Foot_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Pounds_Mass_Per_Cubic_Foot_DP", Dis.Double_Tag);
```

```
Pounds_Mass_Square_Foot : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Pounds_Mass_Square_Foot", Dis.Float_Tag);
Slugs_Square_Foot : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Slugs_Square_Foot", Dis.Float_Tag);
Slugs_Square_Foot_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Slugs_Square_Foot_DP", Dis.Double_Tag);

Pounds_Force : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Pounds_Force", Dis.Float_Tag);
Pounds_Force_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Pounds_Force_DP", Dis.Double_Tag);

Btus : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "BTUs", Dis.Float_Tag);
Btus_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "BTUs_Per_Second", Dis.Float_Tag);
Btus_Per_Square_Foot : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "BTUs_Per_Square_Foot", Dis.Float_Tag);
Btus_Per_Square_Foot_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "BTUs_Per_Square_Foot_DP", Dis.Double_Tag);
Btus_Per_Square_Foot_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "BTUs_Per_Square_Foot_Per_Second", Dis.Float_Tag);
Btus_Per_Square_Foot_Per_Second_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "BTUs_Per_Square_Foot_Per_Second_Dp", Dis.Double_Tag);

Calories : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Calories", Dis.Float_Tag);
Calories_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Calories_Per_Second", Dis.Float_Tag);
Foot_Pounds_Force : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Foot_Pounds_Force", Dis.Float_Tag);
Foot_Pounds_Force_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Foot_Pounds_Force_DP", Dis.Double_Tag);
Foot_Pounds_Force_Seconds : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Foot_Pounds_Force_Seconds", Dis.Float_Tag);
Foot_Pounds_Force_Seconds_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Foot_Pounds_Force_Seconds_DP", Dis.Double_Tag);
Horsepower : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Horsepower", Dis.Float_Tag);
Atmospheres : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Atmospheres", Dis.Float_Tag);
Atmospheres_Per_Second : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                      "Atmospheres_Per_Second", Dis.Float_Tag);
Inches_Mercury : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Inches_Mercury", Dis.Float_Tag);
Psi : constant Dis.Type_Id := Dis.Register_Type
                              (Dis.Null_Component, "PSI", Dis.Float_Tag);
Pounds_Force_Per_Square_Foot : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
```

```
                    "Pounds_Force_Per_Square_Foot", Dis.Float_Tag);
Pounds_Force_Per_Square_Foot_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                    "Pounds_Force_Per_Square_Foot_DP", Dis.Double_Tag);

--      Temperature types

Kelvin : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Kelvin", Dis.Float_Tag);
Celsius : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Celsius", Dis.Float_Tag);
Fahrenheit : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Fahrenheit", Dis.Float_Tag);
Rankine : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Rankine", Dis.Float_Tag);

--      Luminance types

Candelas : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Candelas", Dis.Float_Tag);
Candelas_Per_Meter_Squared : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                    "Candelas_Per_Meter_Squared", Dis.Float_Tag);
Lumens : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Lumens", Dis.Float_Tag);
Lux : constant Dis.Type_Id := Dis.Register_Type
                              (Dis.Null_Component, "Lux", Dis.Float_Tag);
Radiance : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Radiance", Dis.Float_Tag);
Radiant_Intensity : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                    "Radiant_Intensity", Dis.Float_Tag);

--      Electromagnetic types

Amps : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Amps", Dis.Float_Tag);
Amps_Per_Square_Meter : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component,
                    "Amps_Per_Square_Meter", Dis.Float_Tag);
Amps_Per_Meter : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Amps_Per_Meter", Dis.Float_Tag);
Columbs : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Columbs", Dis.Float_Tag);
Frequency : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Frequency", Dis.Float_Tag);
Frequency_Dp : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Frequency_DP", Dis.Double_Tag);
Farads : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Farads", Dis.Float_Tag);
Henries : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Henries", Dis.Float_Tag);
Hertz : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Hertz", Dis.Float_Tag);
Impedeance : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Impedeance", Dis.Float_Tag);
Ohms : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Ohms", Dis.Float_Tag);
Siemens : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Seimens", Dis.Float_Tag);
Tesla : constant Dis.Type_Id :=
   Dis.Register_Type (Dis.Null_Component, "Tesla", Dis.Float_Tag);
Weber : constant Dis.Type_Id :=
```

```
      Dis.Register_Type (Dis.Null_Component, "Weber", Dis.Float_Tag);
   Volts : constant Dis.Type_Id :=
      Dis.Register_Type (Dis.Null_Component, "Volts", Dis.Float_Tag);
   Kvolts : constant Dis.Type_Id :=
      Dis.Register_Type (Dis.Null_Component, "Kvolts", Dis.Float_Tag);

   --     Miscelaneous types

   Decibels : constant Dis.Type_Id :=
      Dis.Register_Type (Dis.Null_Component, "Decibels", Dis.Float_Tag);
   Decibels_Dp : constant Dis.Type_Id :=
      Dis.Register_Type (Dis.Null_Component, "Decibels_DP", Dis.Double_Tag);
   Rpm : constant Dis.Type_Id := Dis.Register_Type
                                 (Dis.Null_Component, "Rpm", Dis.Float_Tag);
   Non_Dimensional : constant Dis.Type_Id :=
      Dis.Register_Type (Dis.Null_Component, "Non_Dimensional", Dis.Float_Tag);
   Non_Dimensional_Dp : constant Dis.Type_Id :=
      Dis.Register_Type (Dis.Null_Component,
                         "Non_Dimensional_DP", Dis.Double_Tag);

   Kilobytes : constant Dis.Type_Id :=
      Dis.Register_Type (Dis.Null_Component, "Kilobytes", Dis.Float_Tag);
   Megabytes : constant Dis.Type_Id :=
      Dis.Register_Type (Dis.Null_Component, "Megabytes", Dis.Float_Tag);

   -- Enumeration types

   package Spt is new Dis.Enum_Functions (Std_Eng_Units.Shapes);
   Shapes : constant Dis.Type_Id :=
      Dis.Register_Type (Dis.Null_Component, "Shapes", Dis.Enum_Tag,
                         Labels => Spt.Labels,
                         Size => Spt.Size);

end Sstf_Defs;
```

8.6. Timer_Services_Class

The Timer_Services_Class is a service package that is used to support classes that wish to run at a slower, harmonic rate of the partition. Timer_Services_Class must be used as shown in the Class Template with Computed Period (7.2). Within a given period, the service procedure Update must be called before the selector functions Time_To_Execute and Actual_Delta_Time are valid. Time_To_Execute must be true to get a valid, non-zero, time from the Actual_Delta_Time function. The current implementation of Timer_Services_Class is listed below.

```
with Std_Eng_Units;
package Timer_Services_Class is

   package Seu renames Std_Eng_Units;  --Simplifies parameter names.

   type Rates is (Full, Half, Quarter, Eighth, Sixteenth, Thirty_Second, Sixty_Fourth);

   subtype Period_Offsets is Natural range 1..64;

   type Object is limited private;     --Limited private is preferred.

-- ****************************  Modifiers  *********************************

   procedure Create (Timer           : in out Object;
                     Subrate         : in     Rates          := Full;
                     Period_Base_Time : in    Seu.Seconds    := 0.0;
                     Period_Offset   : in     Period_Offsets := 1);

   procedure Update (Timer      : in out Object;
                     Delta_Time : in     Seu.Seconds);

-- ****************************  Selectors  *********************************

   function Time_To_Update (Timer : Object) return Boolean;
   function Actual_Delta_Time (Timer : Object) return Seu.Seconds;
   function Get_Child_Rate
                (Parent_Rate         : Rates;
                 Rate_Off_The_Parent : Rates) return Rates;

private
   type Object is
      record
         Period_Time    : Seu.Seconds := 0.0;
         Timer          : Seu.Seconds := 0.0;
         Timer_Offset   : Seu.Seconds := 0.0;
         Delta_Time     : Seu.Seconds := 0.0;
         Time_To_Update : Boolean     := False;
      end record;
end Timer_Services_Class;
-------------------------------------------------------------------------------
--| Abstract    : This class provides the timer services needed to run class
--|               instances at a slower, harmonic relative rate of the partition
--|
--| Warnings    : None.
-------------------------------------------------------------------------------
```

9. APPENDIX III – QUESTIONS AND ANSWERS:

9.1. Ada Structural Components:

1. Why is SSVTF defining and encouraging the use of a defined and consistent Ada structures (classes, partitions)?

Experience has proven that for large object–oriented real–time Ada programs, it is important to clearly define and consistently implement the software to assure success. Ada is a general purpose language that can be used in a variety of ways. Haphazard use of Ada constructs in a so–called object–oriented design methodology can easily defeat the point of what makes OOD a benefit. Adding in real–time and simulation constraints can worsen the impact. The most logical approach is to define what Ada structures support the requirements and design methodology and then consistently apply them over the entire project. These structures provide the basis for the design. However, they are not meant to be overly restrictive – if there are good reasons to move slightly outside the standard and the overall design goals are not corrupted, then no problem. In most cases, the design standard will be improved in these cases.

2. Why are there partitions?

Regardless of what your design is, at some point, the final class instances must be created somewhere – usually in an Ada main program or ASM. For a real–time, distributed system, large programs must also be divided into self–contained manageable chunks that can be easily configured into the system. Those chunks are ~equivalent to CIs on SSVTF. Ada mains could be the chunks, but mains are usually too large, restrictive, and brittle. Non–robust communication methods usually require modelers to assume certain build configurations (what model is on what cpu). SVM was built to provide a seamless interface to support distributed processing so models could run on a "virtual single machine". The best level to provide this service was at the partition level – the level below the Ada main. In addition, Ada 9x will support partitions and this will make SSVTF closer to the future Ada standard. Partitions therefore are not an SVM particular requirement – they are a reality.

3. When do you make an Ada main? How many partitions in an Ada main?

For the target machine, Ada mains will be constructed out of 1 or more partitions. The number of partitions within an Ada main depends on the RMS rate and execution time requirements of each partition. In general, modelers do not need to worry about this.

4. What is the criteria for choosing partitions for a CI? Can there be too many or too few? How large can a partition be (lines of code and execution time)?

A CI can be made of 1 or more partitions, a partition could represent several CIs, or a set of CI requirements could be divided amongst many partitions. The first situation is the most common. A better way to think about partitions is that they represent a logical assembly of objects representing something in the real world. They may be real–world objects themselves like a star tracker, or they are a collection of objects that perform a real–world capability like a hydraulics system. The determination on what is too many or too few depends on the system itself and a compromise between the various real–time constraints. As a note, the more partitions there are, the more overhead processing is required to schedule and to send messages. Eventually, there will be timing values measuring the overhead. Also note that partitions, in general, should assume a time lag of one period for data. There are options to sequentially data between partitions running on the same cpu and same rate (see messaging system), but this is not the norm. Large partitions on the other hand could result in very large, monolithic models that are hard to manage and cumbersome to use. Basically, a CI should be designed such that is is decoupled and easy to work with. The modeler must strike a good balance.

A partition must be able to execute on a single cpu. The current estimate is 10 SMIPS /processor. As for size, it varies widely but a good number is between 5–20 KSLOCS per partition (average).

5. How is a class represented in Ada?

Classes are constructed as Ada abstract data type packages. See appendix I.

6. Do all class structures need create, request_state_change, update, and selectors?

Update is required since the models usually do something over time. Request_State_Change is needed for object initialization or aperitif change state requests (like malfunctions). Selectors are usually needed to extract state data from the object unless the OUT parameters are used in the Update routine. It is preferred to use selectors. Create is optional since it supports 1 time instance configuration. Using Create to configure a class is equivalent to creating an Ada generic class. Create is called during program elaboration.

NOTE: Using these specific names (Update, Request_State_Change) are not critical (but recommended). Other names (such as Initialize for Request_State_Change) are ok. The important point is the meaning of these procedures – class structures should be consistent in implementation. For example, the Initialization procedure must support singular state changes as the Request_State_Change procedure does. There can be more (or less) procedures actually implemented as long as the structural meaning is retained.

7. Why are local variables in a partition's body? Why are there no parameters in partition body routines?

Local variables in the partition body are either the state of the partition (along with the class instances declared there), or they are temporary values that are used to hold intermediate transitions of external data to class data parameters or to hold OUT parameters from classes. In many cases, it is unavoidable not to have these parameters. Using selectors in class structures reduces the number of these variables.

There can be procedures with parameters in the body of the partition. The "parameterless" procedures are simply groupings of object connections for maintainability and modularity concerns. It appears (and is true) that these procedures are working off of common data in the partition body, but placing the logic of these routines within a single update routine can make for a very long procedure. Modeler discretion is required.

8. Can lone variables be declared in a partition's spec or class specs or bodies (outside the private type)?

No, never. Doing so would cause bad side–effects in that global data would be created in the system. In partitions, lone variables in the spec imply that someone is going to read them. Since partitions never WITH (reference) other partitions directly, this would invalidate the whole point of building a seamless distributed system (creating a full–fledged mess). Lone variables outside the class private type (but defined in the class) is equivalent to creating global data. Doing this invalidates the whole point of object–oriented design. Side–effects would include models that work stand–alone but fail on the integrated sim or exhibit peculiarities during integrated operations (data being changed arbitrarily). (Note that local data variables declared in procedure declaration sections are valid since they exist only during the life of the procedure call.)

9. Why are there DIS objects in interface definition packages? Aren't only "types" allowed here?

True, types are only allowed here. The DIS object however provides a unique ID for a message. A unique ID is required to dynamically connect partition messages during set–up (senders must say which message they are outputting, and receivers must state what message they want to receive. The DIS id is the "way" they identify the specific message). This object is not global data since it is never read or written to – rather, it is used to set up a unique ID for the message during set–up only.

10. Should message records have default values in interface definition packages?

No. Multiple distributed elaboration will cause the messages to reset multiple times. Also, the messaging system design cannot take advantage of default values (since pointers are never allocated). Messages are first initialized during the self-init mode by the defining partition (the sender).

11. Why are there access types in Interface_Defn packages?

Two reasons. (1) The software backplane is a general messaging system. It is not (and should not) be dependent on the data types sent by partitions. Pointers of the access types are used to point to the memory of the partition's message area as an address. Knowing the message size, the backplane can send messages generically (bytes) from one partition to another. (2) For efficiency reasons, the pointers defined in the partition body can be manipulated (in some cases) instead of actually copying data. Messages are truly sent, but the most optimal method can be performed by the backplane.

Note that pointers of the access types are never allocated (using the NEW construct). SVM uses the pointers as address pointers only.

12. Can class data types be WITHed into Interface_Defn packages, or do all interface types have to be in the Interface_Defn package or SEU/SET?

Classes should not be WITHed in Interface_Defn type packages. This would tie the whole system together from the top to the bottom causing compilation problems and would be very non-resilient to change. Try to keep interface data atomic using SEU types. If a class defines its own type, let the partition convert it if it ends up in the interface.

13. Why are access pointers used in non real-time and records used in real-time in defining class abstract data types?

The advantage of using access types for the limited private type is that the attributes of the object can be deferred to the body. The disadvantage is that the memory for the variables are declared on the heap during startup and are therefore not available in the symbol map for real-time debugging. Since r/t debugging is necessary, access types cannot be used.

14. Why doesn't the standard allow non-partition ASMs as a legal construct in addition to the class structure?

This standard discourages the use of ASMs as a general programming structure for the following reasons. (1) ASMs are Ada structures not normal in other OO languages. In these languages, classes are like ADTs, period. (2) Developers tend to make ASMs talk to ASMs directly (which is not possible with ADT class structures). This will eliminate a possible reuse without code modification and results in a more brittle design. (3) If designers first think in terms of ADT classes, it has been proven that better OO designs materialize. If the mind set is ASMs, then resulting design usually isn't as robust as it could have been.

There are cases however where ASMs are the best solution. In those cases, the standard does not discourage their use.

15. Why aren't class structures defined as generic ASM's?

Generic ASMs do make good class structures except that they are Ada-specific (non-standard class structures). In addition, there is a certain inefficiency with Ada generics that can be significant if used too much. The standard discourages generic ASM classes primarily to stick with current OO technology and to give us a fighting chance to fit the processing of the trainer into the purchased computer systems.

Note that generic ADTs are reasonable structures if classes are to be generalized. Again, generics should be used judiciously (don't go overboard). Note that it takes longer to develop a generic class than a specific class (experience has shown) – make sure they are worth it.

16. Can Ada tasking be used for real–time applications?

No (unless there is an extremely good reason). Ada tasking conflicts with the RMS real–time scheduling activities. SVM will not be able to control the task and sequence other models correctly. If tasks are used, they most likely will have to run in their own UNIX process (further complicating RMS and the load configuration). Tasking in general should be used only when truly necessary – using the construct for the sake of using it will only lead to unnecessarily complex designs.

17. How do we test a partition in standalone? Can we use the Rational?

Not yet, but there is an effort underway to support partition testing on the Rational.

9.2. Executive Sequencing and Moding:

1. How do I run my model in real time?

Section 4.1 describes how a model runs in real–time under SVM. Basically, a partition instantiates a package called "Periodic" from the "Generic_Model" package (appendix II, pg 66) and provides the update rate (in hz) and mode routines applicable to the partition (run, freeze, hold, etc.). From that point on, the thread executive calls the partition's mode routines at the appropriate time and at the proper rate.

2. How does a model (partition) get the time or delta time to be used when updating the state? What is the unit of time (seconds, microsecs, millisecs)?

The unit for time is defined in SEU as "seconds" (real number). Classes and partitions should use this type for defining update times. Time itself can be obtained two ways. When a partition instantiates a thread executive from the package "Periodic" in the "generic model" (see appendix II,), a variable "Delta_Time" becomes available. This is the delta time based on the period time of the thread executive (i.e. 0.100 for 10hz, 0.033_333 for 30hz). It will always equal this interval. If SGMT or GMT is needed, the partition may call the selectors G_M_T or S_G_M_T also in the "Periodic" package specification. This returns the period–relative GMT/SGMT time for the partition. Period–relative means that the clock with respect to the using partition ticks at the rate of the partition (i.e. 100ms or 33.33ms increments for 10hz or 30hz respectively). Note that reading the actual time (i.e. to the current microsecond) from the partition is not generally provided since with RMS, the model execution may slide around in the period producing inconsistent time values. Period relative time is the only consistent clock for the partition. For special cases, however, instantaneous time can be made available.

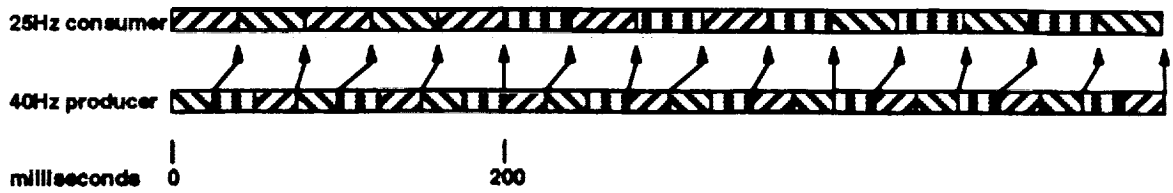
3. How do the mode routines execute?

The instantiation of the "Periodic" package of the "Generic_Model" takes a generic formal parameters the partition's mode routines. The thread executive created from the instantiation coordinates its activities with a SVM master executive and then executes the correct mode procedure at the correct time.

4. How do I chose the rate the partition runs at?

The rate of a partition is based on the response required by the model. Response should be the minimum rate (slowest) based on how the student will perceive the real–time behavior. If the student can't perceive the difference between 30 hz and 10 hz for the particular model, use 10. Response requirements also depend on the hardware. The model must run fast enough to service hardware

(otherwise, the hardware may enter a fault mode). The modeler should be aware of the execution rate differences (and its implication to data flow) between producer partitions and consumer partitions. Execution rates for the partitions do not have to be harmonic, but data may appear to be irregular between non–harmonic partitions. In the example below, a 25Hz partition consumes data produced by a 40Hz partition; the data is taken from the producer's period that has completed immediately before the beginning of the consumer's period. The producer executes eight times to every five executions of the consumer, and so produced data is skipped in a pattern repeating every 200 milliseconds.



5. How do I determine how much fidelity to build into a model?

Fidelity is dependent on what the student or instructor want to see in the way of data and the amount of functionality required. The fidelity of the model should be the minimum required to support the training. For example, if a student cannot detect the effects of a valve in transition (opening or closing), then a full fidelity valve simulation is not needed. A simple open/closed model is adequate. If the transition causes effects based on the transition that are detectable, a higher fidelity model may be needed. Strive for the minimum required fidelity. With a good objectized design, the model fidelity can always be easily increased.

6. Can objects run slower than partition?

Yes. However, the intent is to have an entire partition execute at a given rate (as a whole). For special circumstances (due to execution cost), modelers may slow down objects defined by a partition – the reduced rate must be a harmonic (1/2, 1/4, 1/8, etc.) of the partition rate. Modelers will have to balance the partition across the partition periods and execution must stay within partition period intervals (all execution must complete within the partition worst case RMS time allocation per period). Reference Appendix I, 7.2 (class template with computed period) for an example structure. Modelers may use other means to sub–schedule (internal jump–lists), but the above approach is the recommended approach

7. Why is the partitions interface definition WITHed in the spec instead of the body?

By convention only. If you look at the partition spec, you can see what interfaces it externally references. If the external interfaces are WITHed in the body, they will mix with all the WITHs of the internal classes and other packages.

9.3. Messaging:

1. How do partition's (model) communicate?

Partitions communicate only via the software backplane via the SVM interface packages Message and Mailbox. Partitions define message structures in "Interface_Defn" packages.

2. When do I use one–to–many?

Most real–world interfaces (electrical wiring, pipes, busses, electrical signals, etc.) use the 1–to–many messaging approach. This approach should be used when the producer outputs a message that any receiver can receive. 1–to–many can be 1–to–1.

3. When do I use many–to–one?

Many–to–1 is used when a partition has many inputs all of the same message type. For example, many producers supply load back to the electrical system. The electrical system defines the type and the producers use that type to send the message to the electrical system partition. The messages are queued so none will be missed.

4. When do I use mailbox?

The mailbox should only be used for command and control (non real–world interfaces). Normally, partitions will only read their mail, not send it. The IOS will use mail extensively to send malfunction requests.

5. Can transaction processing run under SVM and RMS?

Not easily (actually no). RMS assumes that models execute at a given periodic rate for a set time. RMS theory can then guarantee the periodic updates. A transaction process has an inconsistent cycle time (event based) and runs till it's done (not abiding by the periodic cycle rules). This will cause the other RMS models to miss their deadlines. Merging the two execution models (RMS, transaction) together is very non trivial, opens up cpu performance, cpu allocation, and UNIX process and OS issues, and results in a brittle software design. It can be done, but we'd rather avoid it.

9.4. Generic Partition:

1. If a model is designed as a partition and is a generic system, what options are there for reducing the duplication of code?

In special cases, the modeler can define a partition using a generic. This allows multiple partitions to be created from a single code module. Reference the generic partition write–up in this document.

9.5. DIS

1. Why is DIS necessary?

In order to "see" data on the IOS or gather data for datastore, there must be a capability to map logical names to physical variables and a way to uniquely id the variables. In addition, there is a requirement for no off–line tools. The DIS provides this capability and a few more features like providing unique message ids and partition ids. DIS is "part of the code" therefore no off–line tools are needed – the symbol mapping will always be updated with the loads.

2. How does IOS get access to data variables in system?

Via the DIS identifiers and a DIS feature called "look" that can gather data via its address. The IOS maps IOS page variables to DIS names, sends a request to SVM to see the symbol, and SVM returns the bytes that make up the symbol.

3. How do we read and set variables for engineering debug? Are DIS variables used for this type of engineering analysis?

The RTSC Ada compiler vendor will provide a real–time debugger. DIS variables can be used for engineering analysis, but modelers should not add non–IOS DIS parameters just for this purpose. The DIS has a reasonable limit on the number of variables it can register.

9.6. Datastore:

1. Why is data extracted using peek operations but brought back by using mailbox? The mailbox approach is cumbersome, why not poke the data back into the addresses?

Peek operations will not harm the model and the model does not have to do anything extra except for registering datastore data in the DIS to support this approach. The mailbox is used for inputs be-

cause not all the state of the objects is datastored. Updating a partial state via a poke operation (back-door) will cause problems. The mailbox allows the model to logically apply datastore data via "Request_State_Change" class operations.

There are still some issues with the approach that are being worked out. This area will be tuned and simplified in the future.

2. Why can't partitions define records for datastore terms instead of each individual term?

The instructor and RECON need to see the ASCII names of the datastore terms (and other associated information). Extracting binary records will not support this activity. In addition, if the load changes, datastore binary records may be altered causing an odd failure on return to datastore (if the partition overlays the record representation onto the binary datastore file).

3. What data should be datastored?

Only independent variables. Variables that can be obtained or derived from other partitions' output data or internal independent data should not be in the datastore. Models should keep the terms at a minimum. Currently, SSVTF is limited to around 40,000 terms. This issue still needs work.

9.7. Interface Agent:

1. SVM allows transparent connections between partitions, but between assets an interface agent is required. Why?

The software backplane can route messages to other assets without interface agents. However, there are special requirements on SSVTF regarding assets because they can be added, dropped, simulated, or stimulated. Something must exist to provide these modeling functions. Also, when connected to heterogeneous platforms, the interface agent must make sure the bytes are ordered correctly. Moding of the asset must also be managed. See the write-up on interface agents.

10. APPENDIX IV –EXAMPLE CODE (NON–REAL–TIME)

```
------------------------------------------------------------------------------
--  Unit Name   : Lesson_Class
--
--  Abstract    : Controls a set of Lectures, Data_Recordings, Control_Laws,
--                and an Exit_Test to achieve one or more learning objectives.
--                Uses an Interactive Control_Panel to permit the student to
--                control the Lesson progress.
--
--  Exceptions  :
--                Requestor_not_Authorized
--
--  Warnings    : None.
--
--  Author      : Gary Young
--
--  Department  : TSC.SSVTF.Computer_Systems.Software_Engineering
--
--  Revisions   : Date      Author
--                4-30-92   Bill Wessale
--                --Added Header
--
--                6-1-92    Gary Young
--                Added Selector and Modifier sections
--                Implemented Selectors : Script_Id, Version,
--                Active_Object_ID, Next_Object_ID, Current_Step,
--                Percent_Complete, Prerequisites
--
--  O-Spec      : JSC-32xxx, Section 5
--
--  Copyright   : Developed by CAE-Link under the Training Systems Contract
--                for the Johnson Space Center (JSC) of the National
--                Aeronautics and Space Administration (NASA).  All rights
--                reserved.
--
------------------------------------------------------------------------------
with Std_Eng_Types;
package Lesson_Class is
   type Object is limited private;
   type Object_Id is (Tbd);
   type Prerequisite_List is (Tbd);
   package Set renames Std_Eng_Types;
   subtype Steps is Set.Positive range 1 .. 500;
-- ***************** Modifiers ************************
   procedure Create (Instance : in out Object);
   procedure Destroy (Instance : in out Object);
   procedure Revise (Instance : in out Object);
   procedure Browse (Instance : in Object);
   procedure Export (Instance : in Object; File_Name : in String);
   procedure Import (Instance : in out Object; File_Name : in String);
   procedure Start (Instance : in out Object);
   procedure Pause (Instance : in out Object);
   procedure Resume (Instance : in out Object);
   procedure Backspace (Instance : in out Object; Number_Of_Steps : in Steps);
   procedure Skipahead (Instance : in out Object; Number_Of_Steps : in Steps);
   procedure Kill (Instance : in out Object);
   procedure Report_Status (Instance : in Object);
   procedure Activity_Complete (Instance : in out Object);
-- ***************** Selectors ************************
   function Script_Id (Instance : in Object) return String;
   function Version (Instance : in Object) return String;
```

```
   function Active_Object_Id (Instance : in Object) return Object_Id;
   function Next_Object_Id (Instance : in Object) return Object_Id;
   function Current_Step (Instance : in Object) return Positive;
   function Percent_Complete (Instance : in Object) return Positive;
   function Prerequisites (Instance : in Object) return Prerequisite_List;
private
   type State;
   type Object is access State;
end Lesson_Class;

------------------------------------------------------------------------------
--| Unit Name    : Lesson_Class
--|
--| Author       : Gary Young
--|
--| Department   : TSC.SSVTF.Computer_Systems.Software_Engineering
--|
--| Revisions    : Date      Author
--|                4-30-92   Bill Wessale
--|                --Added Header
--|
--| O-Spec       : JSC-32xxx, Section 5
--|
--| Copyright    : Developed by CAE-Link under the Training Systems Contract
--|   1992         for the Johnson Space Center (JSC) of the National
--|                Aeronautics and Space Administration (NASA).  All rights
--|                reserved.
--|
------------------------------------------------------------------------------
with Training_Script_Class;
package body Lesson_Class is
   type Script_Designator is (Tbd);
   type Version_Designator is (Tbd);
   type State is
      record
         Script            : Training_Script_Class.Object;
         Current_Mode      : Set.Mode           := Set.Initialize;
         Script_Id         : Script_Designator  := Tbd;
         Version           : Version_Designator := Tbd;
         Active_Object_Ids : Object_Id          := Tbd;
         Next_Object_Id    : Object_Id          := Tbd;
         Current_Step      : Steps              := 1;
         Percent_Complete  : Set.Percent        := 0;
         Prerequisites     : Prerequisite_List  := Tbd;
      end record;
   procedure Create (Instance : in out Object) is
   begin
      Training_Script_Class.Create (Instance => Instance.Script);
      -- Initialize the Current_Mode component
   end Create;
   procedure Destroy (Instance : in out Object) is
   begin
      Training_Script_Class.Destroy (Instance => Instance.Script);
   end Destroy;
   procedure Revise (Instance : in out Object) is
   begin
      Training_Script_Class.Revise (Instance => Instance.Script);
      -- Revise the Current_Mode component
   end Revise;
   procedure Browse (Instance : in Object) is
   begin
      Training_Script_Class.Browse (Instance => Instance.Script);
```

```
   -- Initialize the Current_Mode component
end Browse;
procedure Export (Instance : in Object; File_Name : in String) is
begin
   Training_Script_Class.Export
      (Instance => Instance.Script, File_Name => File_Name);
   -- Append the Current_Mode to the file
end Export;
procedure Import (Instance : in out Object; File_Name : in String) is
begin
   Training_Script_Class.Import
      (Instance => Instance.Script, File_Name => File_Name);
   -- Input the Current_Mode
end Import;
procedure Start (Instance : in out Object) is
begin
   Training_Script_Class.Start (Instance => Instance.Script);
   -- Change the Current_Mode component to the appropriate mode
end Start;
procedure Pause (Instance : in out Object) is
begin
   Training_Script_Class.Pause (Instance => Instance.Script);
   -- Change the Current_Mode component to be Paused
end Pause;
procedure Resume (Instance : in out Object) is
begin
   Training_Script_Class.Resume (Instance => Instance.Script);
   -- Change the Current_Mode component to the appropriate mode
end Resume;
procedure Backspace (Instance : in out Object; Number_Of_Steps : in Steps) is
begin
   Training_Script_Class.Backspace (Instance        => Instance.Script,
                                    Number_Of_Steps => Number_Of_Steps);
end Backspace;
procedure Skipahead (Instance : in out Object; Number_Of_Steps : in Steps) is
begin
   Training_Script_Class.Skipahead (Instance        => Instance.Script,
                                    Number_Of_Steps => Number_Of_Steps);
end Skipahead;
procedure Kill (Instance : in out Object) is
begin
   Training_Script_Class.Kill (Instance => Instance.Script);
   -- Change the Current_Mode component to Killed
end Kill;
procedure Report_Status (Instance : in Object) is
begin
   Training_Script_Class.Report_Status (Instance => Instance.Script);
   -- Report status on the Current_Mode here
end Report_Status;
procedure Activity_Complete (Instance : in out Object) is
begin
   Training_Script_Class.Activity_Complete (Instance => Instance.Script);
   -- Change the Current_Mode component to be Completed.
end Activity_Complete;
function Script_Id (Instance : in Object) return String is
begin
   return Script_Designator'Image (Instance.Script_Id);
end Script_Id;
function Version (Instance : in Object) return String is
begin
   return Version_Designator'Image (Instance.Version);
end Version;
```

```
   function Active_Object_Id (Instance : in Object) return Object_Id is
   begin
      return Instance.Active_Object_Ids;
   end Active_Object_Id;
   function Next_Object_Id (Instance : in Object) return Object_Id is
   begin
      return Instance.Next_Object_Id;
   end Next_Object_Id;
   function Current_Step (Instance : in Object) return Positive is
   begin
      return Instance.Current_Step;
   end Current_Step;
   function Percent_Complete (Instance : in Object) return Positive is
   begin
      return Instance.Percent_Complete;
   end Percent_Complete;
   function Prerequisites (Instance : in Object) return Prerequisite_List is
   begin
      return Instance.Prerequisites;
   end Prerequisites;
end Lesson_Class;
```

11 Appendix V – Hydraulic System Example

The following is an example of a real world hydraulic system and its representation as a simulated software system. This example will include design considerations based upon the SSVTF architecture as outlined in this document.

11.1 Real World Hydraulic System

The hydraulic system provides pressurized hydraulic fluid to actuators that move the control surfaces and raise and lower the landing gear. The system is controlled via the hydraulic control panel which provides switches to control the system. The system sends signals to the hydraulic control panel so that the control panel can display the status of the system. The system receives power from the electrical system

Refer to figure 1 for a pictorial representation of the hydraulic system and related components. Notice that although the actuators, control surfaces, landing gear, hydraulic control panel and electrical system are shown in the figure, they are modeled as separate entities. The modeling of these external components will not be done here. This example will, however, model the interface to these entities.

Therefore, the hydraulic system includes the following components:

- Two fluid pressurization assemblies that each include one motor, one gear box and one pump
- Two valves
- Two accumulators
- One reservoir
- One reservoir quantity sensor
- Two pressure sensors
- A fluid distribution system
- Fluid return lines

11.1.1 Fluid Pressurization Assembly

A fluid pressurization assembly is a collection of mechanical devices that convert electrical power to hydraulic pressure. This assembly includes a motor, gear box and pump, each described as follows:

11.1.1.1 Motor

The motor uses electrical power to turn a shaft that drives the gear box. In providing power to the gear box, the motor in turn produces a load on the electrical system.

The motor is powered on and off via a switch on the hydraulic control panel. The motor provides an indication of whether it is on or off back to the hydraulic control panel.

11.1.1.2 Gear Box

The gear box transfers torque from the motor to the pump.

11.1.1.3 Pump

The pump pressurizes the hydraulic fluid provided by the reservoir. It also sends its operational status to the hydraulic control panel.

11.1.2 Valve

A valve allows the isolation of the pressurization system from the distribution system. Since this valve is electrically actuated it produces a load on the electrical system.

The valves in the hydraulic system are controlled via the hydraulic control panel. The valves provide the hydraulic control panel with an indication of their position, ranging from open to closed.

11.1.3 Accumulator

An accumulator is type of damper that helps the hydraulic system maintain a constant pressure. It is divided into a fluid side and a gas side separated by a movable diaphragm. Hydraulic pressure is absorbed by the accumulator by allowing the fluid from the distribution system to push the diaphragm and increase the gas pressure by lowering its volume. When the pressure in the distribution system lowers, the pressure of the gas in the accumulator pushes fluid back into the distribution system.

11.1.4 Reservoir

A reservoir is a storage container for hydraulic fluid.

11.1.5 Reservoir Quantity Sensor

A reservoir quantity sensor provides an indication of the level of hydraulic fluid in the reservoir.

The quantity sensor in the hydraulic system is electrically powered. It receives power from, and in turn place a load on the electrical system. The quantity sensor provides a signal to the hydraulic control panel so that the quantity of fluid in the reservoir can be displayed.

11.1.6 Pressure Sensor

A pressure sensor provides an indication of the hydraulic pressure in the distribution system.

Like the reservoir quantity sensor, the pressure sensors in the hydraulic system are electrically powered. The pressure sensors provide signals to the hydraulic control panel so that the distribution system pressure can be displayed.

11.1.7 Distribution System

A distribution system is a network of hydraulic tubing that distributes pressurized hydraulic fluid to the actuators.

11.1.8 Return Lines

Return lines return excess hydraulic fluid from the actuators and distribution system to the reservoir.

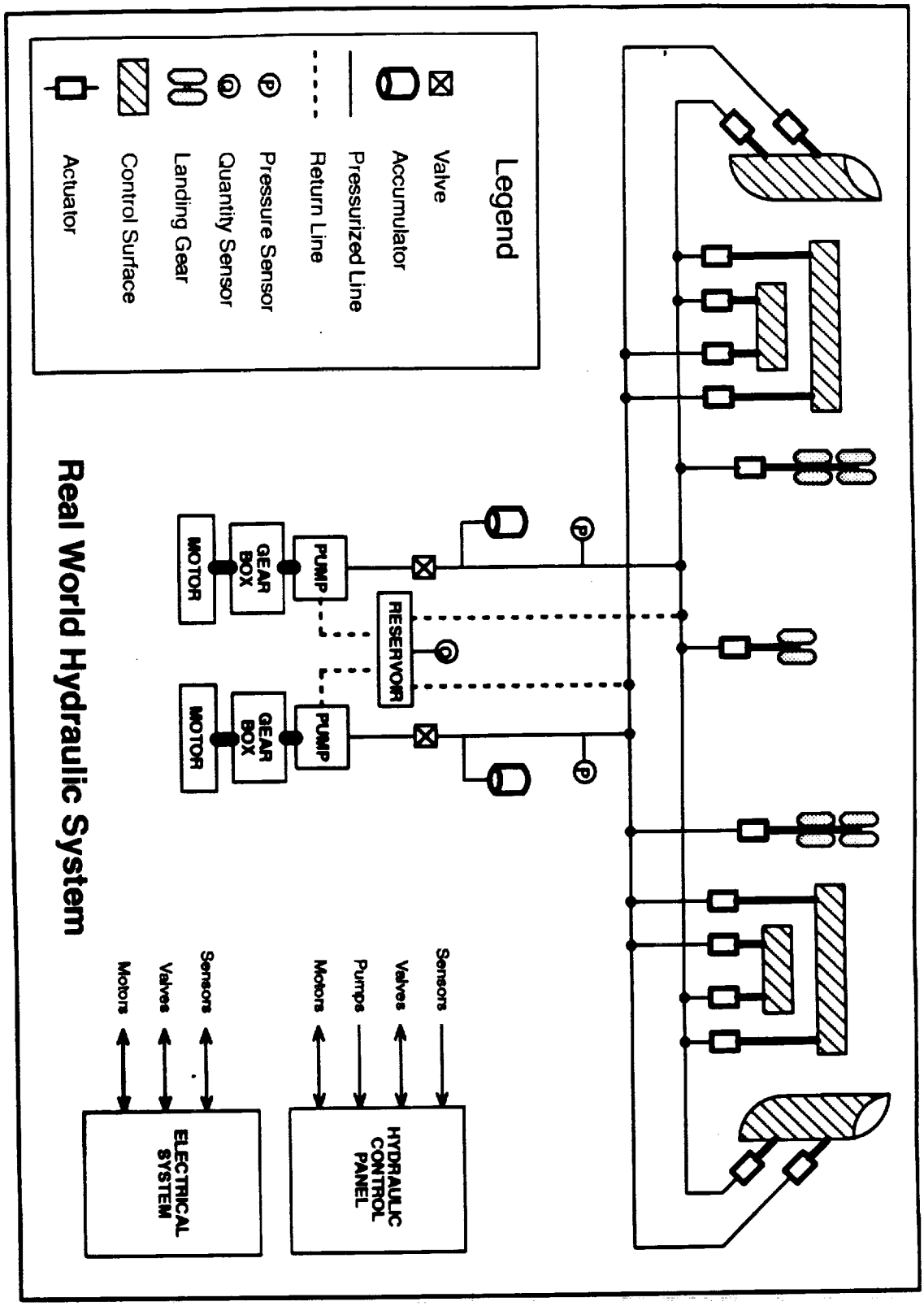


Figure 1 Real World Hydraulic System

11.2 Specification of the Software System

In order to create a software simulation of a real world hydraulic system, all relevant components of the real world system must be modeled, as well as any additional components to support the simulation. Two such support components in this example are the IOS and the aural cue. More details on these two components will be given later.

11.2.1 External Components

At this early point in the analysis, the system can be defined in terms of components that are internal (accumulators, distribution system, reservoir, etc.) and components that are external. The externals are as follows:

- Control surfaces (includes control surface actuators)
- Landing gear (includes landing gear actuators)
- Electrical system
- Hydraulic control panel
- IOS
- Aural cue

Although this example will not give the details of the external models, it does specify the interfaces to these externals. Figure 2 shows the associations of the hydraulic system and its externals.

11.2.1.1 Control Surfaces

The real world control surfaces are moved by actuators which are connected to the hydraulic system. The control surface actuators consume fluid based on the pressure of the fluid provided by the hydraulic system. The actuators also return fluid via the return lines. The interface between the hydraulic system and the control surfaces will therefore provide a mechanism by which the hydraulic system can provide an indication of the available pressure and the control surfaces can provide the actual pressure flow (in–flow) and the return flow.

11.2.1.2 Landing Gear

Although the landing gear model would be quite different than the control surfaces model, the interface between the hydraulic system and the landing gear is similar to the interface between the hydraulic system and the control surfaces. The interface must provide a mechanism by which the hydraulic system can provide an indication of the available pressure and by which the landing gear can provide the actual pressure flow (in–flow) and the return flow.

11.2.1.3 Electrical System

In addition to providing power to other systems, the real world electrical system provides power to the sensors, valves and motors in the hydraulic system. This is modeled in the software system via an interface by which the electrical system provides the status (on or off) of each of the relevant circuit breakers. The interface must also allow the hydraulic system to provide the electrical system with the load that it places on each of the corresponding circuits.

11.2.1.4 Hydraulic Control Panel

The real world hydraulic control panel commands the motor to power on and off and commands the valves to open and close. The hydraulic control panel also displays the status of the hydraulic system via a selected set of parameters. These parameters include pump and motor status (on or off), valve position, indicated pressure and indicated reservoir quantity. The interface between the simulated hydraulic system and the simulated hydraulic control panel must provide a mechanism by which these parameters are passed between the two.

11.2.1.5 IOS

The IOS allows an operator to control the overall simulation. For this example, the operator may insert malfunctions and display and modify certain object state variables.

11.2.1.6 Malfunctions

Table 1 presents a list of simulated malfunctions which effect the Hydraulic System CSCI.

Malfunction name	Description	Allocation
PUMP–#1 FAILURE	Pump #1 does not produce fluid flow when prime mover is providing RPM.	Hydraulic Pump.
PUMP–#2 FAILURE	Pump #2 does not produce fluid flow when prime mover is providing RPM.	Hydraulic Pump.
PRESSURE COMPENSATION FAILURE #1	Pump #1 cannot regulate pressure. Pressure varies wildly with demanded flow. Possible water hammer transients in circuit #1.	Hydraulic Pump.
PRESSURE COMPENSATION FAILURE #2	Pump #2 cannot regulate pressure. Pressure varies wildly with demanded flow. Possible water hammer transients in circuit #2.	Hydraulic Pump.
VALVE #1 FAILURE	Isolation Valve #1 is stuck in position it was in at the time malfunction was inserted.	Hydraulic Distribution System.
VALVE #2 FAILURE	Isolation Valve #2 is stuck in position it was in at the time malfunction was inserted.	Hydraulic Distribution System.
PRESSURE SENSOR #1 FAILURE	Pressure Sensor in circuit #1 fails to indicate zero (0) psi.	Hydraulic Distribution System.
PRESSURE SENSOR #2 FAILURE	Pressure Sensor in circuit #2 fails to indicate over pressure.	Hydraulic Distribution System,.
MOTOR #1 FAILURE	Electric Motor #1 fails to produce RPM, but is not jammed.	Hydraulic Pump Drive Unit.
MOTOR #2 FAILURE	Electric Motor #2 fails to produce RPM and is jammed.	Hydraulic Pump Drive Unit.
CIRCUIT #1 LEAK	1 GPM leak in circuit #1.	Hydraulic Distribution System
CIRCUIT #2 LEAK	5 GPM leak in circuit #2.	Hydraulic Distribution System.

Table 1 Hydraulic System Malfunctions

11.2.1.7 Look and Enter Data

Table 2 presents a list of the Hydraulic System parameters which will be displayed or modified at the instructor's station or recorded by the Session Manager Subsystem for any purpose. Of these parameters, reservoir quantity and pump flow may be modified by the operator.

Parameter name	Description	Allocation
MOTOR #1 ON\OFF	Report commanded on\off status of motor #1.	Hydraulic Pump Drive Unit.
MOTOR #2 ON\OFF	Report commanded on\off status of motor #2.	Hydraulic Pump Drive Unit.

MOTOR #1 RPM	Report current RPM of motor #1.	Hydraulic Pump Drive Unit.
MOTOR #2 RPM	Report current RPM of motor #2.	Hydraulic Pump Drive Unit.
FLUID LEVEL	Report fluid level in reservoir.	Hydraulic Distribution System.
PRESSURE #1	Report current pressure in circuit #1.	Hydraulic Distribution System.
PRESSURE #2	Report current pressure in circuit #2.	Hydraulic Distribution System.
FLOW #1	Report current flow generated by pump #1.	Hydraulic Pump.
FLOW #2	Report current flow generated by pump #2.	Hydraulic Pump.
VALVE #1	Report current open\close status of isolation valve #1.	Hydraulic Distribution System.
VALVE #2	Report current open\close status of isolation valve #2.	Hydraulic Distribution System.

Table 2 IOS Display parameter list for Hydraulic System

11.2.1.8 Aural Cue

The aural cue produces audio sounds of the mechanical devices that are being simulated. For this simulation, the aural cue will produce motor sounds when a motor is on and pump sounds when a pump is on. The interface between the hydraulic system and the aural cue therefore must provide a mechanism to transfer these commands from the hydraulic system to the aural cue.

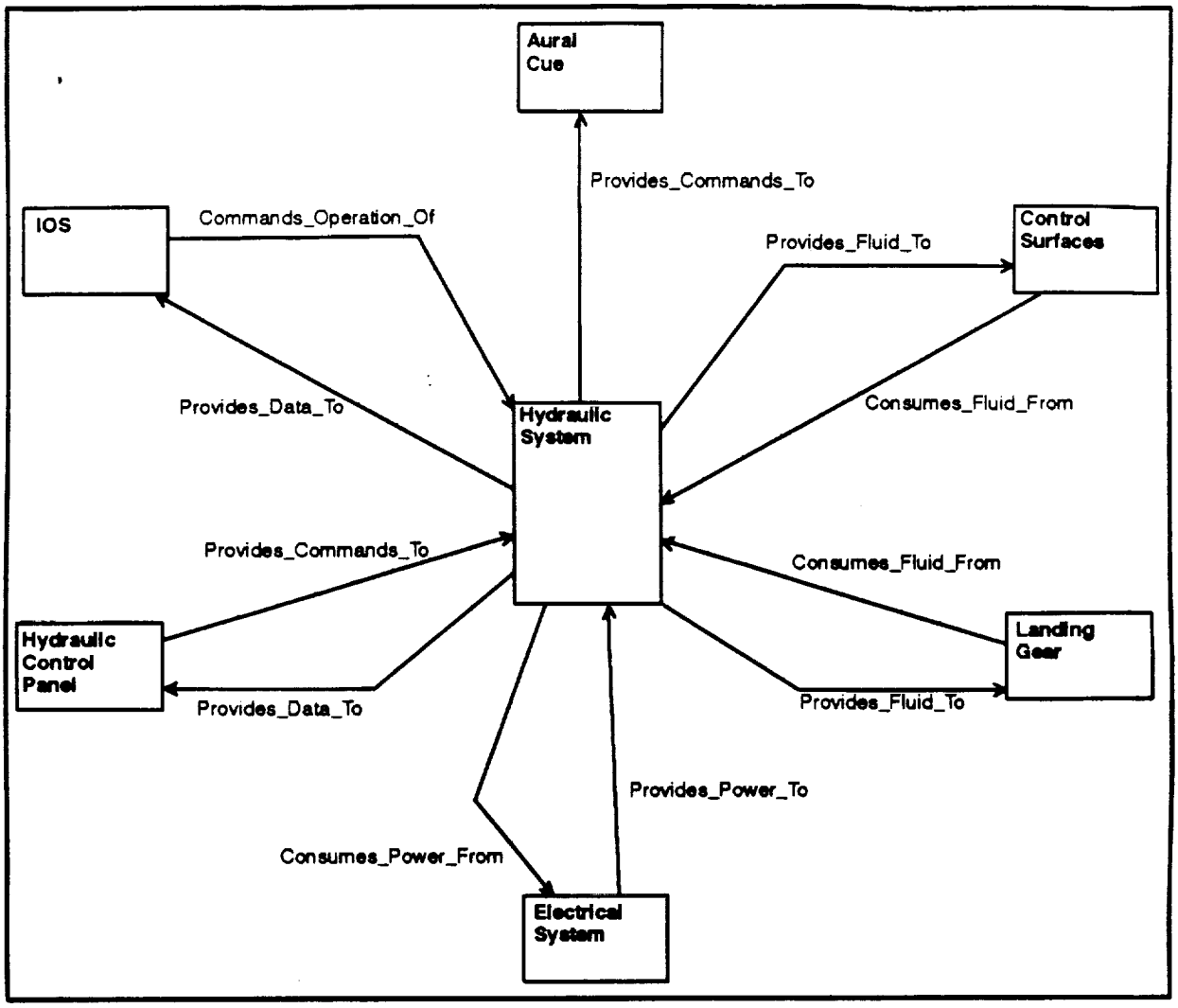


Figure 3 Hydraulic System External Association Diagram

11.2.2 Internal Components

In terms of the object oriented analysis, the hydraulic system is viewed as an object composed of a collection of lower order components that parallel the composition of the real world system. The decomposition of the hydraulic system into these components is is shown in figure 4.

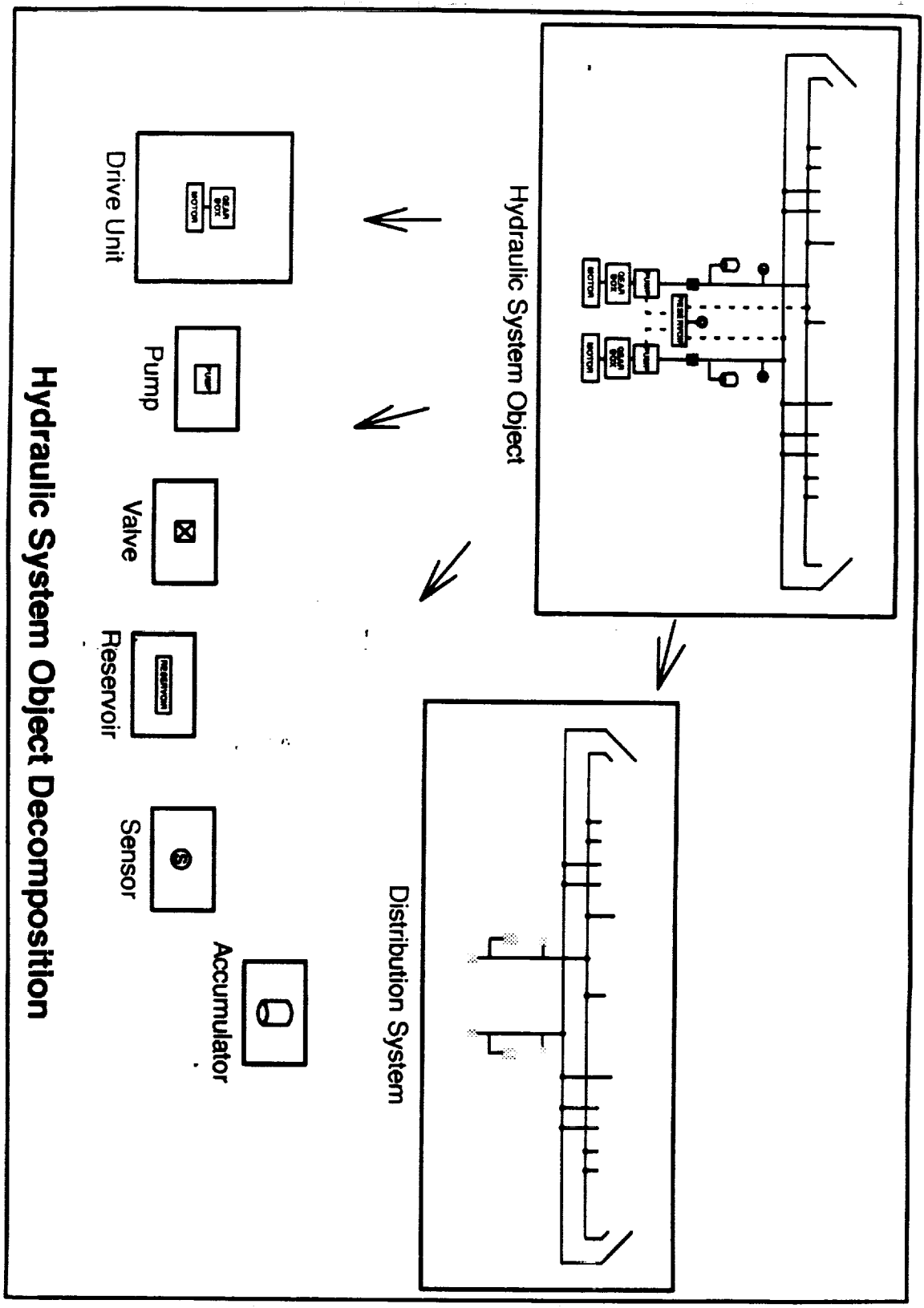


Figure 4 Hydraulic System Object Decomposition

11.3 Transition to Design

In review, classes may be composed of 1) other classes, 2) attributes and operations or 3) a combination of both. Objects may be 1) an instance of one or more classes 2) composed of other objects instantiated at a lower level or 3) a combination of both.

Furthermore, since an object is an instance of a class, one or more objects can be created from a class. An object instantiated from a class in effect creates a copy of the attributes so that the newly created object takes on its own identity (i.e. state), independent of all other instances of the same class. This supports the reuse principle. A class is said to be reused if more than one object is instantiated from the class.

After the components of the hydraulic system are identified, they are allocated as objects or classes, or further broken down into sub–components. Generally, these components should modeled as a class to facilitate reuse. If the abstraction doesn't already exist as a class, a new class is created and added to the reuse pool. If the abstraction (or something reasonably close) does exist as a class, then the class is reused and attributes and operations are added, if necessary.

11.3.1 Sensor Class

The simulation must support two pressure sensors and one quantity sensor. Since the two types of sensors are similar, a general sensor class is created. The pressure sensors and quantify sensor are then created by instantiating the sensor class with the relevant load units and sensed units.

11.3.2 Reservoir Class

A hydraulic reservoir class is created from a generic reservoir class much like the quantity and pressure sensors are created from a generic sensor class. The hydraulic reservoir class is created by instantiating the sensor class with the desired volume, volume rate units and time units.

11.3.3 Drive Unit Class

A drive unit class is constructed using lower level classes much the same way that the hydraulic pump class was constructed. For this simulation the motor of the drive unit is a DC type motor. Suppose that in the reuse pool there exists an electric motor class with attributes of nominal_speed, nominal_torque, shaft_speed and shaft_fail (boolean). Since a DC motor is a more specialized type of electric motor, inheritance is used to create a DC motor class. The DC motor class inherits the attributes and operations of the electric motor class and adds the attributes load, minimum_voltage, maximum_voltage, nominal_load, nominal_speed and nominal_torque. Refer to the Elec_Motor_Class package specification on page V–17 and the DC_Motor_Class package specification on page V–26.

11.3.4 Hydraulic Pump Class

A specialized pressure compensating hydraulic pump class is created by combining the attributes and operations of an axial piston pump class, an actuator class and a centrifugal pump class. Because the resulting hydraulic pump class is very specialized in nature, it serves to show that by using inheritance and composition, it can be constructed with basic building block classes. The composition of a hydraulic pump is shown in figure 6. Notice that an axial piston pump class is created by inheriting the attributes and operations of a positive displacement pump class and defining additional necessary attributes and operations. The attributes of the positive displacement pump class are summarized as follows:

- Displacement (Gallons)
- Efficiency (No Units)
- Flow Rate (Gallons Per Second)
- Total Piston Area (Square_Feet)

The axial piston pump class attributes include those inherited from the positive displacement pump class plus the following:

- Bias (Gallons Per Second)
- Delta_Flow (Gallons Per Second)
- Loss_Flow (Gallons Per Second)
- Pressure (Psi)
- Scale (No Units)
- Torque (Foot_Pound_Force)

In this example an additional flow attribute is added to improve the fidelity of the simulation since the flow rate attribute provided by the positive displacement pump class is overly simplified. Refer to the Positive_Displacement_Pump_Class package specification on page V–36 and the Axial_Piston_Pump_Class package specification on page V–39.

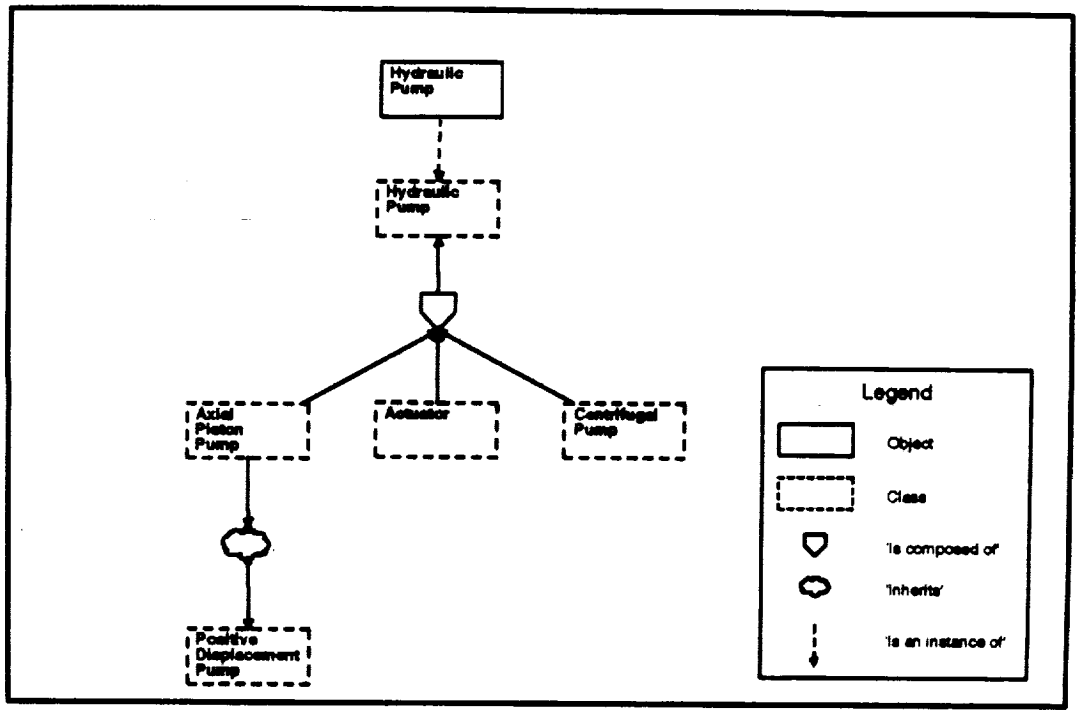


Figure 5 Hydraulic Pump Composition Diagram

11.3.5 Other Classes

For this example, the accumulator, distribution system, valve and gear box classes will all be of the normal (non–generic) class variety.

In this design, all control (interpreting messages, updating objects, etc.) is handled on the partition level. Figure 6 shows the real world hydraulic system as an abstract state machine (ASM) and its decomposition into classes (abstract data types or ADTs)

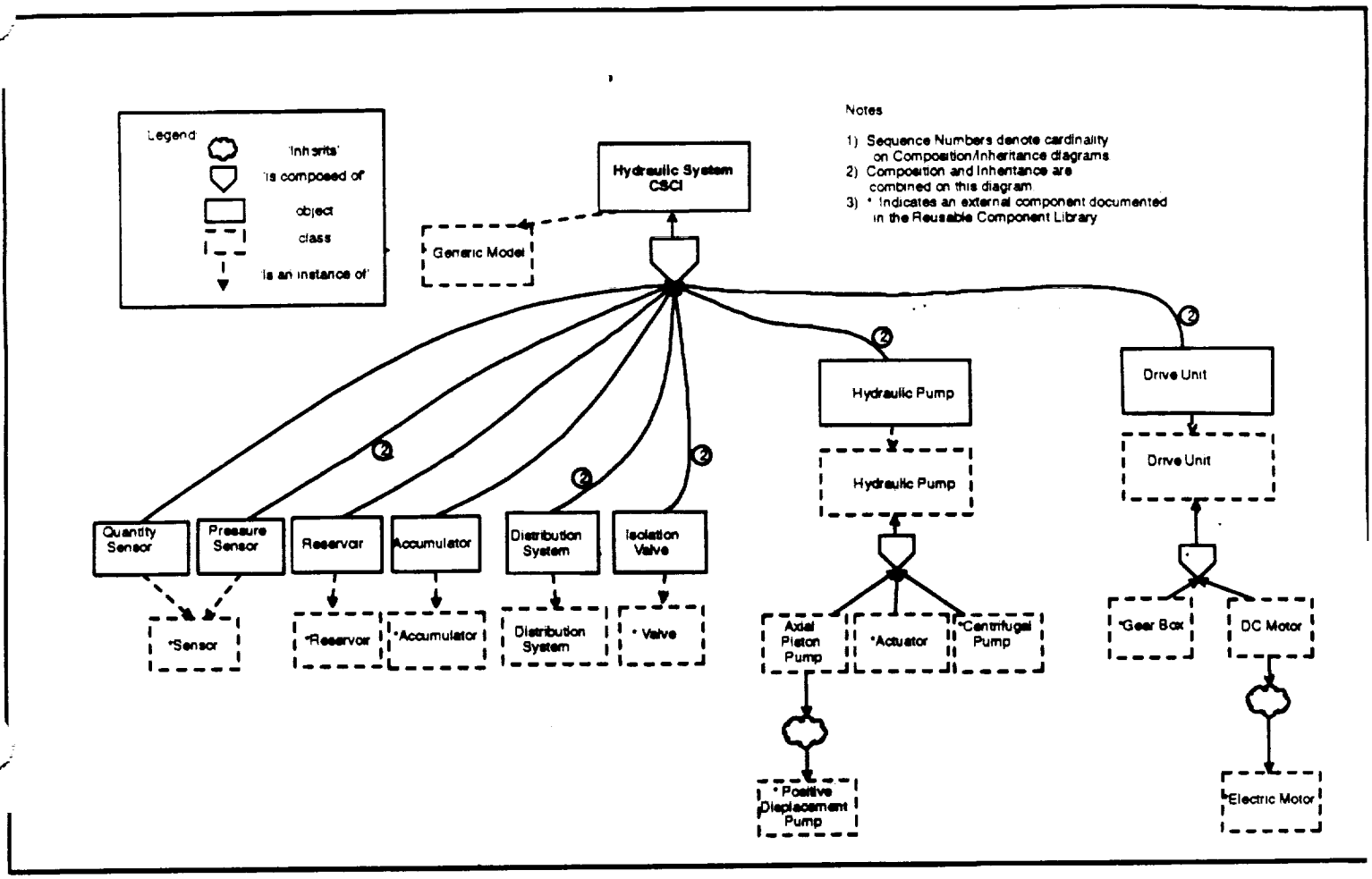


Figure 6 Hydraulic System Composition Diagram

11.4 Class Specification

A class specification is implemented as an Ada package. A typical class specification for the SSVTF project contains the following:

- Attributes in the form of a limited private record type
- Type declarations to support the modifiers
- Modifier specifications
- Selector specifications
- A textual description of the class

11.4.1 Attributes

Attributes define the state of the object. The attributes are collected in a single Ada record type and are made unavailable outside of the package by declaring the record as 'limited private'. This allows an object to be declared of the record type, but none of the attributes defined in the record may be directly accessed where the object is declared. Instead, the attributes are accessed via modifiers and selectors.

11.4.2 Type Declarations

Types are declared in order to specify parameters that are passed into the modifier operations. An example for a switch class is as follows:

```
type Commands is (Initialize, On, Off, Fail);
```

An input parameter is then specified using this type, such as:

```
procedure Request_State_Change (Instance : in out Object;
                                Command  : in     Commands);
```

Request_State_Change can then be used to initialize the switch, change the position attribute to 'On' or 'Off', or cause the switch to fail.

11.4.3 Modifier Specifications

Modifiers are Ada procedures that allow the state of the object instantiated from that class to be changed. Modifier names should be in the form of an action verb. Request_State_Change (above) is an example of a modifier. Note that since one or more instances of the class may exist, the specific instance of the class is passed to the procedure as an 'in out' parameter. This allows the modifier to have access to all attributes defined for the class and also allows the modifier to change any attributes of the class (hence the name 'modifier').

11.4.3.1 Default Modifiers

The following modifiers should be specified for each class:

- Update
- Request_State_Change
- Create

11.4.3.2 Update

The Update modifier is called periodically to update the state of the instance of the class. The period of the call is passed in as delta time since the previous update.

11.4.3.3 Request_State_Change

The Request_State_Change modifier serves a variety of purposes. The associated enumeration type Commands allows the caller of Request_State_Change request that the instance of the class change its internal state. For example, a valve may allow its position to be changed, or may allow a malfunction to be inserted.

11.4.3.4 Create

The Create modifier is typically called once upon partition initialization. It serves to allow the instance to be tailored in some way. In the case of an accumulator object, minimum and maximum pressures and volumes may be set. Note that the Create operation is analogous to Ada elaboration. Both are done once upon initialization.

11.4.4 Selector Specifications

Selectors are Ada functions that return the current value of an attribute associated with the object instantiated from that class. Selector names should be in the form of a noun. An example of a selector follows:

```
function Position (Instance : in Object) return Set.On_Ness;
```

Again, since one or more instances of the class may exist, the specific instance of the class is passed to the function. Since a selector cannot change the state of an object, the parameter 'Instance' is passed as a read–only parameter using the 'in' qualifier. The selector returns the position of the switch (on or off).

11.4.5 Textual Description

The textual description of the class are in the form of Ada comments and contain items such as identification, description, author, revision history. Refer to the SSVTF Ada coding standards document for more information.

11.5 Class Examples

11.5.1 The Accumulator Class

Based on the real-world characteristics of an accumulator as identified in the object oriented requirements analysis the attributes and operations of the accumulator may be listed as shown below.

Attributes	**Units**	**Operations**
Flow Rate	Gallons/Second	**Modifiers**
Gas Pressure	PSI	Create
Gas Volume	Cubic Feet	Request_State_Change
Fluid Volume	Cubic Feet	Update_State
Quantity Held	Gallons	
Minimum Gas Pressure	PSI	**Selectors**
Minimum Gas Volume	Cubic Feet	Flow_Rate
Maximum Gas Volume	Cubic Feet	Quantity_Held
Minimum Fluid Volume	Cubic Feet	
Maximum Fluid Volume	Cubic Feet	

Table 3 Accumulator Attributes and Operations

From this listing of attributes and operations, a class specification (Ada package specification) is created, as shown in Ada Unit 1 on page V-17.

11.5.2 The Pressure and Quantity Sensor Class

The simulation of the pressure sensor and quantity sensor are very similar in this example. Each sensor has an actual and nominal load placed on the electrical system. Each may be failed by the IOS and each makes the sensed value available. The sole difference between the pressure sensor and the quantity sensor is the units (i.e. type) of the sensed value (pressure in PSI and quantity in gallons). To take advantages of these similarities, a generic class is specified. In order to instantiate the generic, the instantiation must supply the specific sensed value type (PSI or gallons) to create a more specific class. For the purposes of this example, the actual and nominal load type will also be specified as a generic parameter to the generic class.

Note that in this example the generic sensor class is used to create a new class for the pressure sensor and a new class for the quantity sensor. While in Ada terms the generic class is instantiated with the generic actual parameters, an instance of the class in object oriented terms does not yet exist.

Attributes	**Units**	**Operations**
Bias	Generic sensed units	**Modifiers**
Load	Generic load units	Create
Nominal Load	Generic load units	Request_State_Change
Output Value	Generic sensed units	Update
Sensor Failed	Boolean	
Scale	None	**Selectors**

		Elec_Load
		Sensed_Output

Table 4 Generic Sensor Attributes and Operations

From this listing of generic attributes and operations, a generic class specification (Ada generic package specification) is created, as shown in Ada Unit 4 on page V-20.

11.6 The Hydraulic System Partition

The hydraulic system partition is an abstract state machine (ASM) implemented as an Ada package.

The partition can be thought of as the 'glue' or 'smarts' that holds the hydraulic system simulation together. It is at this level that messages are received, interpreted and acted upon. Logic at the partition level is responsible for routing relevant data to the lower levels. Likewise, the partition must provide data to the outside world.

11.6.1 Hydraulic System Partition Interfaces

The hydraulic system partition communicates with the hydraulic control panel, electrical system, landing gear, control surfaces, aural cue and IOS. In this example, the hydraulic control panel partition and the electrical system partition provide interface definition packages for their respective interfaces. Interface definitions between the hydraulic system partition and the other external systems are provided by the hydraulic system partition and maintained in the Hyd_Sys_Intfc_Defn package.

11.6.2 Hydraulic_System_Partition Package Specification

Since the partition does not export (in the Ada sense) any operations or data, the content of the partition package specification is minimal. By SVM convention, all of the partition's interface definitions, both internal and external are 'withed' into the package specification. Although these packages could technically be 'withed' into the package body, they are 'withed' here so that the partition's interfaces become more apparent.

11.6.3 Hydraulic_System_Partition Package Body

The hydraulic system partition package body contains the declarations that allocate memory for the hydraulic system and instantiates the generic thread executive. It also creates instances of generic classes and defines local types.

11.6.3.1 Generic Class Instantiations

Three new classes are created via generic class instantiations. A pressure sensor class and quantity sensor class are instances of the generic sensor class. Also, a hydraulic reservoir class is instantiated from a generic reservoir class.

11.6.3.2 Local Type Definitions

Since it is convenient to manage the components of the dual-redundant hydraulic system using two element arrays, several array types are created. The index for these array types is an enumeration type defined as:

```
type Sys_1_Sys_2 is (Sys_1, Sys_2);
```

These array types are used for creating class instances as well as defining data internal to the partition. Note that since anonymous arrays are not allowed by the coding standards, these array types are necessary to declare an array.

11.6.3.3 Message Pointers

Each message that is to be sent or received must have a unique identifier. The memory for these identifiers is allocated (i.e. the data objects are declared) in the hydraulic system package body and are later initialized in the Create_Data mode routine.

11.6.3.4 Class Instances

Each major component (object) modeled in the simulation is an instance of a class. In Ada terms an object is created by declaring a data object using the 'Object' type provided by the corresponding class package. The major components of the hydraulic system are the accumulators (2), distribution systems (2), drive units (2), isolation valves (2), pressure sensors (2), pumps (2), reservoir and reservoir quantity sensor.

Using the accumulators as an example of dual components, the 'accumulator' data object is of the array type 'accumulators'. The array type 'accumulators' is declared as:

```
type Accumulators is array (Sys_1_Sys_2) of Accumulator_Class.Object;
```

The data object 'Accumulator' is declared as:

```
Accumulator : Accumulators;
```

Using these conventions, the accumulator objects take the form:

```
System 1 Accumulator:        Accumulator (Sys_1)
System 2 Accumulator:        Accumulator (Sys_2)
```

Using the reservoir as an example of a single component, the 'reservoir' data object is declared as:

```
Reservoir : Hyd_Reservoir_Class.Object;
```

11.6.3.5 Internal Data

Since a partition controls all of the objects contained within it, it is often necessary to create partition–internal data to manage the manipulation of of the objects.

This data may include temporary storage of data that links two or more objects. For example, total return flow is internal data that is computed by summing the return flows from the landing gear and control surfaces as received in messages from these external systems. The total return flow is later used to update the reservoir quantity.

Internal data may also include identifiers used to help manage the partition in the simulation environment. These identifiers include the partition name as a string constant, and the partition ID as a natural number.

A design decision was made to model the motor relays of the hydraulic system internal to the partition rather that creating a separate relay class due to their trivial nature. This is done by creating the following data object:

```
Motor_Relay_Power : On_Off_A := (others=> Off);
```

where On_Off_A has been previously declared as (in the Orvc_Common_Types package):

```
type On_Off_A is array (Sys_1_Sys_2) of Set.On_Ness;
```

This demonstrates another use of internal partition data.

11.6.3.6 Creating Thread_Exec

Since the partition represents a single executive thread, it must instantiate the generic package Generic_Model.Periodic to register the partition name, frequency and all of the partition's mode routines with the SVM executive.

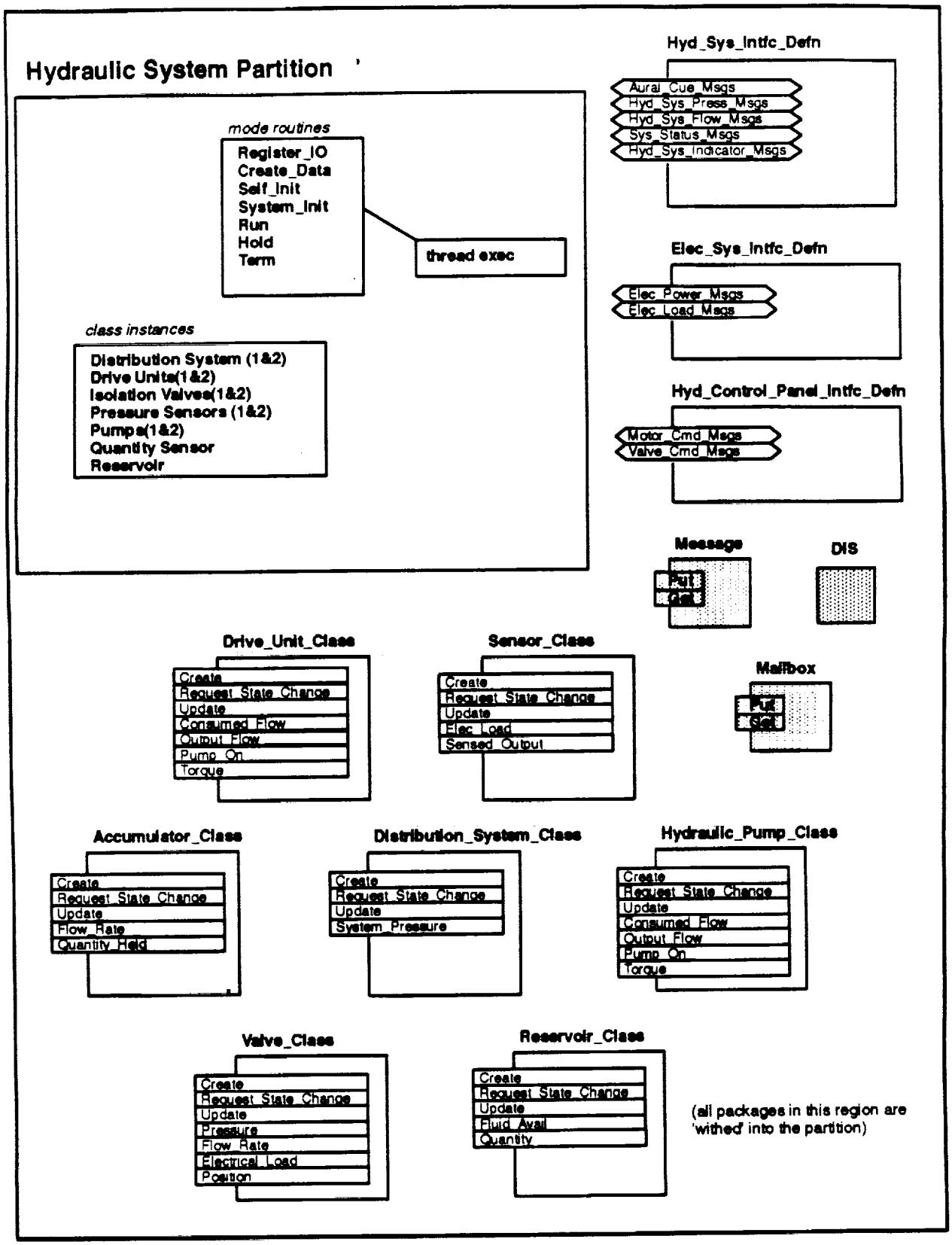


Figure 7 Hydraulic System Partition

Ada Unit 1 Accumulator_Class Package Specification

```
with Std_Eng_Units;

use Std_Eng_Units;

package Accumulator_Class is

   package Seu renames Std_Eng_Units;

   type Object is limited private;

   type Commands is (Initialize, No_Pressure);

-- ********************** Modifiers ********************** --

   procedure Create (Instance      : in out Object;
                     Parent_Name   : in     String := "";
                     Init_Press    : in     Seu.Psi;
                     Min_Gas_Press : in     Seu.Psi;
                     Min_Gas_Vol   : in     Seu.Cubic_Feet;
                     Max_Gas_Vol   : in     Seu.Cubic_Feet;
                     Min_Fluid_Vol : in     Seu.Cubic_Feet;
                     Max_Fluid_Vol : in     Seu.Cubic_Feet);

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean;
                                   Pressure : in     Seu.Psi := 4000.0);

   procedure Update (Instance   : in out Object;
                     Pressure   : in     Seu.Psi;
                     Delta_Time : in     Seu.Seconds);

-- ********************** Selectors ********************** --

   function Flow_Rate (Instance : in Object) return Seu.Gallons_Per_Second;

   function Quantity_Held (Instance : in Object) return Seu.Gallons;

private

   type Object is
      record
         Flow_Rate     : Seu.Gallons_Per_Second := 0.0;
         Gas_Pressure  : Seu.Psi                := 0.0;
         Gas_Volume    : Seu.Cubic_Feet         := 0.0;
         Fluid_Volume  : Seu.Cubic_Feet         := 0.0;
         Quantity_Held : Seu.Gallons            := 0.0;
         Min_Gas_Press : Seu.Psi                := 0.0;
         Min_Gas_Vol   : Seu.Cubic_Feet         := 0.0;
         Max_Gas_Vol   : Seu.Cubic_Feet         := 0.0;
         Min_Fluid_Vol : Seu.Cubic_Feet         := 0.0;
         Max_Fluid_Vol : Seu.Cubic_Feet         := 0.0;
      end record;

end Accumulator_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of hydraulic accumulator.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Commands for Request_State_Change procedure.

Allows instance constants to be set.

Used to aperiodically request a state change.

Called periodically to update the state.

Selectors to get state values maintained by object.

The list of attributes for this class. Outside of this class, these attributes can be modified only via the given modifiers and selectors specified above. This is enforced by declaring the record type as limited private.

Ada Unit 2 Accumulator_Class Package Body

```
package body Accumulator_Class is

-- ***********************************************

-- Report_Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;

-- ***************** Modifiers ********************

   procedure Create (Instance      : in out Object;
                     Parent_Name   : in     String := "";
                     Init_Press    : in     Seu.Psi;
                     Min_Gas_Press : in     Seu.Psi;
                     Min_Gas_Vol   : in     Seu.Cubic_Feet;
                     Max_Gas_Vol   : in     Seu.Cubic_Feet;
                     Min_Fluid_Vol : in     Seu.Cubic_Feet;
                     Max_Fluid_Vol : in     Seu.Cubic_Feet) is

   begin

      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

      Instance.Gas_Pressure  := Init_Press;
      Instance.Min_Gas_Press := Min_Gas_Press;
      Instance.Min_Gas_Vol   := Min_Gas_Vol;
      Instance.Max_Gas_Vol   := Max_Gas_Vol;
      Instance.Min_Fluid_Vol := Min_Fluid_Vol;
      Instance.Max_Fluid_Vol := Max_Fluid_Vol;

-- Need function here to convert gas pressure to gas volume &
-- fluid volume

   end Create;

-- ***********************************************

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean;
                                   Pressure : in     Seu.Psi := 4000.0) is
   begin
      case Command is

         when Initialize =>

            Instance.Gas_Pressure := Pressure;

-- Add function here to determine gas volume & fluid for this new gas
-- pressure

         when No_Pressure =>

            null;

      end case;

   end Request_State_Change;

-- ***********************************************

   procedure Update (Instance   : in out Object;
                     Pressure   : in     Seu.Psi;
                     Delta_Time : in     Seu.Seconds) is

   begin

-- NOTE: This procedure is greatly simplified.

      Instance.Gas_Pressure := Pressure;
      Instance.Flow_Rate    := 0.025;

   end Update;

-- ***************** Selectors ************

   function Flow_Rate (Instance : in Object) return Seu.Gallons_Per_Second is
   begin
```

Local to package body. This is used to enter class attributes into the symbol table. The IOS then has direct ("backdoor") access to the attributes.

```
      return Instance.Flow_Rate;
   end Flow_Rate;
-- ***************************************************
   function Quantity_Held (Instance : in Object) return Gallons is
   begin
      return Instance.Quantity_Held;
   end Quantity_Held;
end Accumulator_Class;
```

Ada Unit 3 Accumulator_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;
separate (Accumulator_Class)
procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is
begin
-- No symbols to report
   null;
end Report_Symbols;
-------------------------------------------------------------------------
--|  Abstract: This separate reports symbols to the symbol map.
--|
--|  Warnings: If no symbols are to be reported, this separate could
--|            be deleted.
-------------------------------------------------------------------------
```

IOS does not access any attributes of the accumulator class. Report_Symbols can be removed when the partition is optimized. See the valve class for an example of reported attributes.

Ada Unit 4 Generic_Sensor_Class Package Specification

```
generic
   type Load_Units     is digits <>;
   type Non_Dim_Units  is digits <>;
   type Sensed_Units   is digits <>;

package Generic_Sensor_Class is

   type Object is limited private;

   type Commands is (Sensor_Fail, Sensor_Incorrect);

-- ********************** Modifiers ********************** --

   procedure Create (Instance     : in out Object;
                     Parent_Name  : in     String := "";
                     Nominal_Load : in     Load_Units);

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean;
                                   Scale    : in     Non_Dim_Units := 1.0;
                                   Bias     : in     Sensed_Units  := 0.0);

   procedure Update (Instance     : in out Object;
                     Power_Avail  : in     Boolean;
                     Sensed_Input : in     Sensed_Units);

-- ********************** Selectors ********************** --

   function Elec_Load (Instance : in Object) return Load_Units;

   function Sensed_Output (Instance : in Object) return Sensed_Units;

private

   type Object is
      record
         Bias          : Sensed_Units  := 0.0;
         Load          : Load_Units    := 0.0;
         Nominal_Load  : Load_Units    := 0.0;
         Output_Value  : Sensed_Units  := 0.0;
         Scale         : Non_Dim_Units := 1.0;
         Sensor_Failed : Boolean       := False;
      end record;

end Generic_Sensor_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of generic sensors.
--|
--|  Warnings: pragma Inline used in body.
------------------------------------------------------------------------
```

The generic formal parameters. The actual parameters are specified when this package is instantiated.

The sensor may be commanded to fail or read incorrectly.

Note that the remaining portion of the package specification is identical to a normal (non-generic) package specification.

Ada Unit 5 Generic_Sensor_Class Package Body

```
package body Generic_Sensor_Class is

-- ************************************************************

-- Overloaded Operators

   function *** (Left : in Non_Dim_Units; Right : in Sensed_Units)
                return Sensed_Units is

   begin
      return (Sensed_Units (Left) * Right);
   end ***;

   pragma Inline (***);

-- ************************************************************

-- Report_Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;

-- ********************** Modifiers *********************

   procedure Create (Instance     : in out Object;
                     Parent_Name  : in     String := "";
                     Nominal_Load : in     Load_Units) is

   begin

      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

      Instance.Nominal_Load := Nominal_Load;

   end Create;

-- ************************************************************

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean;
                                   Scale    : in     Non_Dim_Units := 1.0;
                                   Bias     : in     Sensed_Units  := 0.0) is

   begin

      case Command is

         when Sensor_Fail =>

            Instance.Sensor_Failed := Apply;

         when Sensor_Incorrect =>

            Instance.Bias  := Bias;
            Instance.Scale := Scale;

      end case;

   end Request_State_Change;

-- ************************************************************

   procedure Update (Instance     : in out Object;
                     Power_Avail  : in     Boolean;
                     Sensed_Input : in     Sensed_Units) is

   begin

      if Power_Avail and not Instance.Sensor_Failed then

         Instance.Output_Value := Instance.Scale * Sensed_Input + Instance.Bias;
         Instance.Load         := Instance.Nominal_Load;

      else

         Instance.Output_Value := 0.0;
         Instance.Load         := 0.0;

      end if;

   end Update;
```

Default initialization allows user to only pass in necessary data.

After next update sensor will output zero.

After next update, sensor will output according to this new scale and bias.

Sensor value affected by values from Request_State_Change.

```
-- ************************ Selectors ************************

   function Elec_Load (Instance : in Object) return Load_Units is
   begin

      return (Instance.Load);

   end Elec_Load;

-- ************************************************************

   function Sensed_Output (Instance : in Object) return Sensed_Units is
   begin

      return (Instance.Output_Value);

   end Sensed_Output;

end Generic_Sensor_Class;
```

Ada Unit 6 Generic_Sensor_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;

separate (Generic_Sensor_Class)

procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is

begin

-- No symbols to report

   null;

end Report_Symbols;

------------------------------------------------------------------------
--|  Abstract: This separate reports symbols to the symbol map.
--|
--|  Warnings: If no symbols are to be reported, this separate could
--|            be deleted.
------------------------------------------------------------------------
```

Ada Unit 7 Elec_Motor_Class Package Specification

```
with Std_Eng_Units;
with Std_Eng_Types;

use Std_Eng_Units;
use Std_Eng_Types;

package Elec_Motor_Class is

   package Seu renames Std_Eng_Units;
   package Set renames Std_Eng_Types;

   type Object is limited private;

   type Commands is (Motor_Fail);

-- ********************** Modifiers ********************** --

   procedure Create (Instance       : in out Object;
                     Parent_Name    : in     String := "";
                     Nominal_Speed  : in     Seu.Radians_Per_Second;
                     Nominal_Torque : in     Seu.Foot_Pound_Force);

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean);

   procedure Update (Instance    : in out Object;
                     Torque      : in     Seu.Foot_Pound_Force;
                     Power_Avail : in     Boolean);

-- ********************** Selectors ********************** --

   function Shaft_Output (Instance : in Object) return Seu.Radians_Per_Second;

private

   type Object is
      record
         Nominal_Speed  : Seu.Radians_Per_Second := 0.0;
         Nominal_Torque : Seu.Foot_Pound_Force   := 0.0;
         Shaft_Speed    : Seu.Radians_Per_Second := 0.0;
         Shaft_Fail     : Boolean                := False;
      end record;

end Elec_Motor_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|             of electric motor.
--|
--|  Warnings: pragma Inline used in body.
------------------------------------------------------------------------
```

Ada Unit 8 Elec_Motor_Class Package Body

```
package body Elec_Motor_Class is

-- ************************************************************

--  Overloaded operators

   function "*" (Left : Seu.Radians_Per_Second; Right : Seu.Non_Dimensional)
                 return Seu.Radians_Per_Second is
   begin
      return Seu.Radians_Per_Second (Set.Real_6 (Left) * Set.Real_6 (Right));
   end "*";

   pragma Inline ("*");

-- ************************************************************

-- Report_Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;

-- ********************* Modifiers  ****************************

   procedure Create (Instance       : in out Object;
                     Parent_Name    : in     String := "";
                     Nominal_Speed  : in     Seu.Radians_Per_Second;
                     Nominal_Torque : in     Seu.Foot_Pound_Force) is
   begin
      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

-- Initialize state to motor off and not failed

      Instance.Nominal_Speed  := Nominal_Speed;
      Instance.Nominal_Torque := Nominal_Torque;

   end Create;

-- ************************************************************

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean) is
   begin
      case Command is

         when Motor_Fail =>

            Instance.Shaft_Fail := Apply;

      end case;

   end Request_State_Change;

-- ************************************************************

   procedure Update (Instance    : in out Object;
                     Torque      : in     Seu.Foot_Pound_Force;
                     Power_Avail : in     Boolean) is

      Sf_1 : Seu.Non_Dimensional;

   begin

-- Based on torque load, available power and shaft status, determine shaft speed

      if (Torque <= Instance.Nominal_Torque) and
         Power_Avail and (not Instance.Shaft_Fail) then

         Sf_1 := 1.0 - Seu.Non_Dimensional (Torque / Instance.Nominal_Torque);

      else

         Sf_1 := 0.0;

      end if;

      Instance.Shaft_Speed := Instance.Nominal_Speed * Sf_1;
```

```
   end Update;
-- ********************** Selectors  ************************
   function Shaft_Output (Instance : in Object) return Seu.Radians_Per_Second is
   begin
      return (Instance.Shaft_Speed);
   end Shaft_Output;
end Elec_Motor_Class;
```

Ada Unit 9 Elec_Motor_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;
separate (Elec_Motor_Class)
procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is
begin
-- No symbols to report
   null;
end Report_Symbols;
------------------------------------------------------------------------
--|  Abstract: This separate reports symbols to the symbol map.
--|
--|  Warnings: If no symbols are to be reported, this separate could
--|            be deleted.
------------------------------------------------------------------------
```

Ada Unit 10 Dc_Motor_Class Package Specification

```
with Elec_Motor_Class;
with Std_Eng_Units;

use Std_Eng_Units;

package Dc_Motor_Class is

   package Em_Cls renames Elec_Motor_Class;
   package Seu    renames Std_Eng_Units;

   type Object is limited private;

   type Commands is (Motor_Fail);

-- ********************** Modifiers ********************** --

   procedure Create (Instance      : in out Object;
                     Parent_Name   : in     String := "";
                     Max_Voltage   : in     Seu.Volts;
                     Min_Voltage   : in     Seu.Volts;
                     Nominal_Load  : in     Seu.Amps;
                     Nominal_Speed : in     Seu.Radians_Per_Second;
                     Nominal_Torque : in    Seu.Foot_Pound_Force);

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean);

   procedure Update (Instance    : in out Object;
                     Delta_Time  : in     Seu.Seconds;
                     Torque      : in     Seu.Foot_Pound_Force;
                     Avail_Power : in     Seu.Volts);

-- ********************** Selectors ********************** --

   function Load (Instance : in Object) return Seu.Amps;

   function Shaft_Output (Instance : in Object) return Seu.Radians_Per_Second;

private

   type Object is
      record
         Elec_Load    : Seu.Amps  := 0.0;
         Elec_Motor   : Em_Cls.Object;
         Max_Voltage  : Seu.Volts := 0.0;
         Min_Voltage  : Seu.Volts := 0.0;
         Nominal_Load : Seu.Amps  := 0.0;
         Power_On     : Boolean   := False;
      end record;

end Dc_Motor_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of DC motors based on a simple electric motor.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Class used within this class. The DC Motor class inherits the attributes and operations of the Electric Motor class

Ada Unit 11 Dc_Motor_Class Package Body

```
package body Dc_Motor_Class is
-- ***********************************************************

-- Report Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String := "") is separate;

-- ************************* Modifiers ***************************

   procedure Create (Instance       : in out Object;
                     Parent_Name    : in     String := "";
                     Max_Voltage    : in     Seu.Volts;
                     Min_Voltage    : in     Seu.Volts;
                     Nominal_Load   : in     Seu.Amps;
                     Nominal_Speed  : in     Seu.Radians_Per_Second;
                     Nominal_Torque : in     Seu.Foot_Pound_Force) is
   begin

      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

      Instance.Max_Voltage  := Max_Voltage;
      Instance.Min_Voltage  := Min_Voltage;
      Instance.Nominal_Load := Nominal_Load;

-- Create electric motor instance

      Em_Cls.Create (Instance       => Instance.Elec_Motor,
                     Parent_Name    => Parent_Name & ".Motor",
                     Nominal_Speed  => Nominal_Speed,
                     Nominal_Torque => Nominal_Torque);

   end Create;

-- ***********************************************************

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean) is
   begin
      case Command is

         when Motor_Fail =>

            Em_Cls.Request_State_Change (Instance => Instance.Elec_Motor,
                                         Command  => Em_Cls.Motor_Fail,
                                         Apply    => Apply);

      end case;

   end Request_State_Change;

-- ***********************************************************

   procedure Update (Instance    : in out Object;
                     Delta_Time  : in     Seu.Seconds;
                     Torque      : in     Seu.Foot_Pound_Force;
                     Avail_Power : in     Seu.Volts) is

   begin

-- Motor is operational when power is: (min volts <= avail_power <= max volts)

      Instance.Power_On := (Instance.Min_Voltage <= Avail_Power) and
                           (Avail_Power <= Instance.Max_Voltage);

-- Determine shaft speed based on power status and torque load

      Em_Cls.Update (Instance    => Instance.Elec_Motor,
                     Torque      => Torque,
                     Power_Avail => Instance.Power_On);

-- Return constant load if powered, otherwise return 0.0
-- NOTE: This process could be replaced by a specific function which could
--       vary the returned load based on the input voltage, torque load and
--       shaft speed.
```

Create provides user specified constants to the instance of the DC motor.

```
      if Instance.Power_On then
         Instance.Elec_Load := Instance.Nominal_Load;
      else
         Instance.Elec_Load := 0.0;
      end if;
   end Update;
-- ************************ Selectors  ******************************
   function Load (Instance : in Object) return Seu.Amps is
   begin
      return (Instance.Elec_Load);
   end Load;
-- ************************************************************
   function Shaft_Output (Instance : in Object) return Seu.Radians_Per_Second is
   begin
      return (Em_Cls.Shaft_Output (Instance.Elec_Motor));
   end Shaft_Output;
end Dc_Motor_Class;
```

Shaft_Output is an attribute of the Electric Motor class and is made available at the DC Motor class via this pass through selector.

Ada Unit 12 Dc_Motor_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;
separate (Dc_Motor_Class)
procedure Report_Symbols (Instance    : in out Object;
                          Parent_Name : in     String := "") is
begin
   null;
end Report_Symbols;
-------------------------------------------------------------------------
--|  Abstract: This separate reports symbols to the symbol map.
--|
--|  Warnings: If no symbols are to be reported, this separate could
--|            be deleted.
-------------------------------------------------------------------------
```

Ada Unit 13 Gear_Box_Class Package Specification

```
with Std_Eng_Units;

use Std_Eng_Units;

package Gear_Box_Class is

   package Seu renames Std_Eng_Units;

   type Object is limited private;

   type Commands is (Gear_Seizure);

-- ********************** Modifiers ********************** --

   procedure Create (Instance    : in out Object;
                     Parent_Name : in     String := "";
                     Max_Torque  : in     Seu.Foot_Pound_Force);

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean);

   procedure Update (Instance     : in out Object;
                     Delta_Time   : in     Seu.Seconds;
                     Torque       : in     Seu.Foot_Pound_Force;
                     Supply_Speed : in     Seu.Radians_Per_Second);

-- ********************** Selectors ********************** --

   function Torque_Load (Instance : in Object) return Seu.Foot_Pound_Force;

   function Shaft_Output (Instance : in Object) return Seu.Radians_Per_Second;

private

   type Object is
      record
         Max_Torque_Load : Seu.Foot_Pound_Force    := 0.0;
         Torque_Load     : Seu.Foot_Pound_Force    := 0.0;
         Shaft_Speed     : Seu.Radians_Per_Second := 0.0;
         Seized          : Boolean                 := False;
      end record;

end Gear_Box_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of gear box used for transmission of rotation speed.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 14 Gear_Box_Class Package Body

```
package body Gear_Box_Class is
-- ***********************************************************
-- Report_Symbols (used by Create)
   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;
-- ******************** Modifiers ***************************
   procedure Create (Instance    : in out Object;
                     Parent_Name : in     String := "";
                     Max_Torque  : in     Seu.Foot_Pound_Force) is
   begin
      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);
      Instance.Max_Torque_Load := Max_Torque;
   end Create;
-- ***********************************************************
   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean) is
   begin
      case Command is
         when Gear_Seizure =>
            Instance.Seized := Apply;
      end case;
   end Request_State_Change;
-- ***********************************************************
   procedure Update (Instance     : in out Object;
                     Delta_Time   : in     Seu.Seconds;
                     Torque       : in     Seu.Foot_Pound_Force;
                     Supply_Speed : in     Seu.Radians_Per_Second) is

   begin
-- Based on shaft status, determine shaft speed and torque load
      if not Instance.Seized then
         Instance.Shaft_Speed := Supply_Speed;
         Instance.Torque_Load := Torque;
      else
         Instance.Shaft_Speed := 0.0;
         Instance.Torque_Load := Instance.Max_Torque_Load;
      end if;
   end Update;
-- ***************************** Selectors ********************
   function Torque_Load (Instance : in Object) return Seu.Foot_Pound_Force is
   begin
      return (Instance.Torque_Load);
   end Torque_Load;
-- ***********************************************************
   function Shaft_Output (Instance : in Object) return Seu.Radians_Per_Second is
   begin
      return (Instance.Shaft_Speed);
   end Shaft_Output;
end Gear_Box_Class;
```

Ada Unit 15 Gear_Box_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;
separate (Gear_Box_Class)
procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is
begin
-- No symbols to report
   null;
end Report_Symbols;
-------------------------------------------------------------------------
--  Abstract: This separate reports symbols to the symbol map.
--
--  Warnings: If no symbols are to be reported, this separate could
--            be deleted.
-------------------------------------------------------------------------
```

Ada Unit 16 Drive_Unit_Class Package Specification

```
with Std_Eng_Types;
with Std_Eng_Units;

with Dc_Motor_Class;
with Gear_Box_Class;

use Std_Eng_Types;
use Std_Eng_Units;

package Drive_Unit_Class is

   package Set renames Std_Eng_Types;
   package Seu renames Std_Eng_Units;

   type Object is limited private;

   type Commands is (Gearbox_Seizure,    -- No output from gearbox.
                     Motor_Fail);        -- No output from motor

-- ********************** Modifiers ********************** --

   procedure Create (Instance           : in out Object;
                     Parent_Name        : in     String := "";
                     Gearbox_Max_Torque : in     Seu.Foot_Pound_Force);

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean);

   procedure Update (Instance    : in out Object;
                     Avail_Power : in     Seu.Volts;
                     Delta_Time  : in     Seu.Seconds;
                     Torque      : in     Seu.Foot_Pound_Force);

-- ********************** Selectors ********************** --

   function Elec_Load (Instance : in Object) return Seu.Amps;

   function Motor_On (Instance : in Object) return Boolean;

   function Shaft_Speed (Instance : in Object) return Seu.Radians_Per_Second;

private

   type Object is
      record
         Motor        : Dc_Motor_Class.Object;
         Gear_Box     : Gear_Box_Class.Object;
         Motor_Status : Set.On_Off := Set.Off;
      end record;

end Drive_Unit_Class;

-------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of components consisting of an electric motor and a
--|            gear box.
--|
--|  Warnings: None.
-------------------------------------------------------------------
```

This class consists of more than one class.

Ada Unit 17 Drive_Unit_Class Package Body

```
package body Drive_Unit_Class is

-- Motor Data from NASA document NASA-91-1135, "Spec for Acme Elec Motor
-- Co. Type XYZ-123A Electric Motor".

   Motor_Load       : constant Seu.Amps               := 1.0;
   Motor_Max_Speed  : constant Seu.Radians_Per_Second := 628.0;   -- 6000 rpm
   Motor_Max_Torque : constant Seu.Foot_Pound_Force   := 300.0;
   Motor_Max_Volts  : constant Seu.Volts              := 15.0;
   Motor_Min_Volts  : constant Seu.Volts              := 8.0;

-- ************************************************************

-- Report_Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;

-- ************************* Modifiers **************************

   procedure Create (Instance           : in out Object;
                     Parent_Name        : in     String := "";
                     Gearbox_Max_Torque : in     Seu.Foot_Pound_Force) is

   begin

      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

      Dc_Motor_Class.Create (Instance       => Instance.Motor,
                             Parent_Name    => Parent_Name & ".dc_motor",
                             Max_Voltage    => Motor_Max_Volts,
                             Min_Voltage    => Motor_Min_Volts,
                             Nominal_Load   => Motor_Load,
                             Nominal_Speed  => Motor_Max_Speed,
                             Nominal_Torque => Motor_Max_Torque);

      Gear_Box_Class.Create (Instance    => Instance.Gear_Box,
                             Parent_Name => Parent_Name & ".gear_box",
                             Max_Torque  => Gearbox_Max_Torque);

   end Create;

-- ************************************************************

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean) is

   begin

      case Command is

         when Gearbox_Seizure =>

            Gear_Box_Class.Request_State_Change
              (Instance => Instance.Gear_Box,
               Command  => Gear_Box_Class.Gear_Seizure,
               Apply    => Apply);

         when Motor_Fail =>

            Dc_Motor_Class.Request_State_Change
              (Instance => Instance.Motor,
               Command  => Dc_Motor_Class.Motor_Fail,
               Apply    => Apply);

      end case;

   end Request_State_Change;

-- ************************************************************

   procedure Update (Instance    : in out Object;
                     Avail_Power : in     Seu.Volts;
                     Delta_Time  : in     Seu.Seconds;
                     Torque      : in     Seu.Foot_Pound_Force) is separate;

-- ************************* Selectors **************************
```

Class specific data used to create other classes during elaboration.

Data provided to class during elaboration to create other class.

Constant data provided by class.

Data provided by class create procedure.

```
   function Elec_Load (Instance : in Object) return Seu.Amps is
   begin

      return (Dc_Motor_Class.Load (Instance.Motor));

   end Elec_Load;
-- ************************************************************

   function Motor_On (Instance : in Object) return Boolean is
   begin

      return (Instance.Motor_Status = Set.On);

   end Motor_On;
-- ************************************************************

   function Shaft_Speed (Instance : in Object) return Seu.Radians_Per_Second is
   begin

      return (Gear_Box_Class.Shaft_Output (Instance.Gear_Box));

   end Shaft_Speed;

end Drive_Unit_Class;
```

Ada Unit 18 Drive_Unit_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;

separate (Drive_Unit_Class)

procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is

begin

-- No symbols to report

   null;

end Report_Symbols;

------------------------------------------------------------------------------
--|  Abstract: This separate reports symbols to the symbol map.
--|
--|  Warnings: If no symbols are to be reported, this separate could
--|            be deleted.
------------------------------------------------------------------------------
```

Ada Unit 19 Drive_Unit_Class.Update Separate Procedure

```
separate (Drive_Unit_Class)

procedure Update (Instance    : in out Object;
                  Avail_Power : in     Seu.Volts;
                  Delta_Time  : in     Seu.Seconds;
                  Torque      : in     Seu.Foot_Pound_Force) is

begin

--  Update electric motor

   Dc_Motor_Class.Update
     (Instance    => Instance.Motor,
      Delta_Time  => Delta_Time,
      Torque      => Gear_Box_Class.Torque_Load (Instance.Gear_Box),
      Avail_Power => Avail_Power);

-- Set motor Status flag

   if Avail_Power >= 0.1 then

      Instance.Motor_Status := Set.On;

   else

      Instance.Motor_Status := Set.Off;

   end if;
```

```
-- Update Gear box

   Gear_Box_Class.Update
      (Instance,     => Instance.Gear_Box,
       Delta_Time    => Delta_Time,
       Torque        => Torque,
       Supply_Speed  => Dc_Motor_Class.Shaft_Output (Instance.Motor));

end Update;

-------------------------------------------------------------------------
--  Abstract: This package contains the Drive Unit Class Update
--            procedure.
--
--  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 20 Positive_Displacement_Pump_Class Package Specification

```
with Std_Eng_Units;
with Std_Eng_Types;

use Std_Eng_Units;
use Std_Eng_Types;

package Positive_Displacement_Pump_Class is

   package Seu renames Std_Eng_Units;
   package Set renames Std_Eng_Types;

   type Object is limited private;

   type Commands is (Set_Efficiency);

-- ********************** Modifiers ************************ --

   procedure Create (Instance          : in out Object;
                     Parent_Name       : in     String              := "";
                     Efficiency        : in     Seu.Non_Dimensional := 1.0;
                     Number_Of_Pistons : in     Integer;
                     Piston_Area       : in     Seu.Square_Feet);

   procedure Request_State_Change (Instance   : in out Object;
                                   Command    : in Commands;
                                   Apply      : in Boolean;
                                   Efficiency : in Seu.Non_Dimensional := 1.0);

   procedure Update (Instance : in out Object;
                     Stroke   : in     Seu.Feet;
                     Rotation : in     Seu.Radians_Per_Second);

-- ********************** Selectors ********************** --

   function Flow (Instance : in Object) return Seu.Gallons_Per_Second;

private

   type Object is
      record
         Displacement      : Seu.Gallons            := 0.0;
         Efficiency        : Seu.Non_Dimensional    := 1.0;
         Flow_Rate         : Seu.Gallons_Per_Second := 0.0;
         Total_Piston_Area : Seu.Square_Feet        := 0.0;
      end record;

end Positive_Displacement_Pump_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of a hydraulic positive dispacement pump.
--|
--|  Warnings: pragma Inline in body.
------------------------------------------------------------------------
```

Ada Unit 21 Positive_Displacement_Pump_Class Package Body

```
package body Positive_Displacement_Pump_Class is

   type Revs_Per_Second is new Seu.Real_6;

-- ************************************************************

-- Overloaded operators

   function "*" (Left : in Seu.Gallons; Right : in Revs_Per_Second)
                return Seu.Gallons_Per_Second is

   begin
      return (Seu.Gallons_Per_Second (Seu.Real_6 (Left) * Seu.Real_6 (Right)));
   end "*";

   function "*" (Left  : in Seu.Gallons_Per_Second;
                 Right : in Seu.Non_Dimensional)
                return Seu.Gallons_Per_Second is
   begin
      return (Seu.Gallons_Per_Second (Seu.Real_6 (Left) * Seu.Real_6 (Right)));

   end "*";

   function "*" (Left : in Seu.Square_Feet; Right : in Integer)
                return Seu.Square_Feet is
   begin
      return (Seu.Square_Feet (Seu.Real_6 (Left) * Seu.Real_6 (Right)));

   end "*";

   function "*" (Left : in Seu.Square_Feet; Right : in Seu.Feet)
                return Seu.Cubic_Feet is
   begin
      return (Seu.Cubic_Feet (Seu.Real_6 (Left) * Seu.Real_6 (Right)));

   end "*";

   pragma Inline ("*");

-- ************************************************************

-- Report_Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;

-- ********************* Modifiers ********************************

   procedure Create (Instance          : in out Object;
                     Parent_Name       : in     String              := "";
                     Efficiency        : in     Seu.Non_Dimensional := 1.0;
                     Number_Of_Pistons : in     Integer;
                     Piston_Area       : in     Seu.Square_Feet) is

   begin
      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

      Instance.Efficiency        := Efficiency;
      Instance.Total_Piston_Area := Piston_Area * Number_Of_Pistons;

   end Create;

-- ************************************************************

   procedure Request_State_Change
               (Instance   : in out Object;
                Command    : in     Commands;
                Apply      : in     Boolean;
                Efficiency : in     Seu.Non_Dimensional := 1.0) is
   begin
      case Command is

         when Set_Efficiency =>
```

```
            if Apply then
               Instance.Efficiency := Efficiency;
            end if;
      end case;
   end Request_State_Change;
-- ************************************************************
   procedure Update (Instance : in out Object;
                     Stroke   : in     Seu.Feet;
                     Rotation : in     Seu.Radians_Per_Second) is
      Speed_Rps             : Revs_Per_Second;
      Displacement_Ft_Cubed : Seu.Cubic_Feet;
   begin
-- Calculate Displacement in gallons (1 gallon = 0.133681 cubic_feet)
      Instance.Displacement :=
        Seu.Gallons (Instance.Piston_Area * Stroke) / 0.133681);
-- Convert rotation to revs_per_second ->
--         rps = (rads/sec) * (1 rev/ 2(pi) rads)
      Speed_Rps := Revs_Per_Second (Rotation * 0.159155);
-- Based on displacement, rotaional speed, and efficiency, determine flow rate
      Instance.Flow_Rate :=
        (Instance.Displacement * Speed_Rps) * Instance.Efficiency;
   end Update;
-- ************************ Selectors ******************************
   function Flow (Instance : in Object) return Seu.Gallons_Per_Second is
   begin
      return (Instance.Flow_Rate);
   end Flow;
end Positive_Displacement_Pump_Class;
```

Ada Unit 22 Positive_Displacement_Pump_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;
separate (Positive_Displacement_Pump_Class)
procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is
begin
-- No symbols to report
   null;
end Report_Symbols;
-------------------------------------------------------------------------
--|  Abstract: This separate reports symbols to the symbol map.
--|
--|  Warnings: If no symbols are to be reported, this separate could
--|            be deleted.
-------------------------------------------------------------------------
```

Ada Unit 23 Axial_Piston_Pump_Class Package Specification

```
with Std_Eng_Types;
with Std_Eng_Units;

with Positive_Displacement_Pump_Class;

use Std_Eng_Types;
use Std_Eng_Units;

package Axial_Piston_Pump_Class is

   package Set renames Std_Eng_Types;
   package Seu renames Std_Eng_Units;

   type Object is limited private;

   type Commands is (Modify_Efficiency, Set_Delta_Flow);

-- ********************** Modifiers ********************** --

   procedure Create (Instance          : in out Object;
                     Parent_Name       : in     String := "";
                     Number_Of_Pistons : in     Integer;
                     Piston_Area       : in     Seu.Square_Feet);

   procedure Request_State_Change
               (Instance    : in out Object;
                Command     : in     Commands;
                Delta_Flow  : in     Seu.Gallons_Per_Second := 0.0;
                Efficiency  : in     Seu.Non_Dimensional    := 1.0);

   procedure Update (Instance      : in out Object;
                     Press         : in     Seu.Psi;
                     Rotation_Rate : in     Seu.Radians_Per_Second;
                     Stroke        : in     Seu.Feet);

-- ********************** Selectors ********************** --

   function Flow (Instance : in Object) return Seu.Gallons_Per_Second;

   function Pressure (Instance : in Object) return Seu.Psi;

   function Torque (Instance : in Object) return Seu.Foot_Pound_Force;

private

   type Object is
      record
         Bias       : Seu.Gallons_Per_Second := 0.0;
         Delta_Flow : Seu.Gallons_Per_Second := 0.0;
         Flow       : Seu.Gallons_Per_Second := 0.0;
         Loss_Flow  : Seu.Gallons_Per_Second := 0.0;
         Pd_Pump    : Positive_Displacement_Pump_Class.Object;
         Pressure   : Seu.Psi                := 0.0;
         Scale      : Seu.Non_Dimensional    := 1.0;
         Torque     : Seu.Foot_Pound_Force   := 0.0;
      end record;

end Axial_Piston_Pump_Class;

------------------------------------------------------------------------
--|  Abstract: This package models a hydraulic pump which uses an
--|            axial piston arrangement to generate hydraulic pressure
--|            based on rotational speed.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Commands used to modify state data with Request_State_Change.

Request_State_Change to provide malfunction and reset capability.

Use of another class within this class.

Ada Unit 24 Axial_Piston_Pump_Class Package Body

```
package body Axial_Piston_Pump_Class is

-- ************************************************************

-- Overloaded Operators

   function "*" (Left  : in Seu.Non_Dimensional;
                 Right : in Seu.Gallons_Per_Second)
                return Seu.Gallons_Per_Second is
   begin
      return (Seu.Gallons_Per_Second (Set.Real_6 (Right) * Set.Real_6 (Left)));
   end "*";

   function "*" (Left : in Seu.Non_Dimensional; Right : in Seu.Psi)
                return Seu.Psi is
   begin
      return (Seu.Psi (Set.Real_6 (Right) * Set.Real_6 (Left)));
   end "*";

   pragma Inline ("*");

-- ************************************************************

-- Loss Flow Rate Function

   function Calc_Loss_Flow (Pressure  : in Seu.Psi;
                            Flow_Rate : in Seu.Gallons_Per_Second)
                           return Seu.Gallons_Per_Second is

      Flow_Sf : Seu.Non_Dimensional;

   begin

-- Need function here to produce the following flow_sf's:
--              Press        Flow_sf
--              400.0          0.001
--             1680.0          0.01
--             2800.0          0.10
-- NOTE: For now, use hard coded value of 0.05 (5% loss)

      Flow_Sf := 0.05;

      return (Flow_Sf * Flow_Rate);

   end Calc_Loss_Flow;

-- ************************************************************

-- Calculate Torque Function

   function Calc_Torque (Speed : in Seu.Radians_Per_Second;
                         Flow  : in Seu.Gallons_Per_Second)
                        return Seu.Foot_Pound_Force is
   begin
-- Need some sort of function to obtain torque based on flow rate & speed.
-- For now, use constant value.

      return (15.0);

   end Calc_Torque;

-- ************************************************************

-- Report_Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String := "") is separate;

-- ********************** Modifiers ***************************

   procedure Create (Instance          : in out Object;
                     Parent_Name       : in     String := "";
                     Number_Of_Pistons : in     Integer;
                     Piston_Area       : in     Seu.Square_Feet) is
   begin
      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);
```

```
      Positive_Displacement_Pump_Class.Create
        (Instance          => Instance.Pd_Pump,
         Parent_Name       => Parent_Name & ".PD_Pump",
         Number_Of_Pistons => Number_Of_Pistons,
         Piston_Area       => Piston_Area);

   end Create;
-- ************************************************************

   procedure Request_State_Change
               (Instance   : in out Object;
                Command    : in     Commands;
                Delta_Flow : in     Seu.Gallons_Per_Second := 0.0;
                Efficiency : in     Seu.Non_Dimensional    := 1.0) is

   begin
      case Command is

         when Modify_Efficiency =>

            Positive_Displacement_Pump_Class.Request_State_Change
              (Instance   => Instance.Pd_Pump,
               Apply      => True,
               Command    => Positive_Displacement_Pump_Class.Set_Efficiency,
               Efficiency => Efficiency);

         when Set_Delta_Flow =>

            Instance.Delta_Flow := Delta_Flow;

      end case;

   end Request_State_Change;
-- ************************************************************

   procedure Update (Instance      : in out Object;
                     Press         : in     Seu.Psi;
                     Rotation_Rate : in     Seu.Radians_Per_Second;
                     Stroke        : in     Seu.Feet) is

      Loss_Flow : Seu.Gallons_Per_Second;

   begin
-- Calculate pump flow based on stroke and speed.  Calculate loss flow
-- based on pump flow and system pressure.  Total flow rate consists of
-- pump flow minus loss flow plus any IOS commanded flow delta.

      Positive_Displacement_Pump_Class.Update (Instance => Instance.Pd_Pump,
                                               Stroke   => Stroke,
                                               Rotation => Rotation_Rate);

      Instance.Flow := Positive_Displacement_Pump_Class.Flow (Instance.Pd_Pump);

      Loss_Flow := Calc_Loss_Flow (Pressure  => Press,
                                   Flow_Rate => Instance.Flow);

      Instance.Flow := Instance.Flow - Loss_Flow + Instance.Delta_Flow;

-- Determine torque from total flow rate.

      Instance.Torque := Calc_Torque
                           (Speed => Rotation_Rate, Flow => Instance.Flow);

-- Output pressure equals input pressure.

      Instance.Pressure := Press;

   end Update;
-- ********************* Selectors ********************** --

   function Flow (Instance : in Object) return Seu.Gallons_Per_Second is
   begin
      return (Instance.Flow);
   end Flow;
-- ************************************************************
```

Create performed for class contained within this class.

Malfunction passed to class contained in this class.

Update other class from within this class.

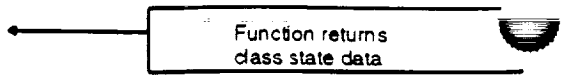
Use state data from other class to calculate state data for this class.

Function returns class state data

```
   function Pressure (Instance : in Object) return Seu.Psi is
   begin
      return (Instance.Pressure);
   end Pressure;
-- ***************************************************************

   function Torque (Instance : in Object) return Seu.Foot_Pound_Force is
   begin
      return (Instance.Torque);
   end Torque;

end Axial_Piston_Pump_Class;
```



Ada Unit 25 Axial_Piston_Pump_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;

separate (Axial_Piston_Pump_Class)

procedure Report_Symbols (Instance    : in out Object;
                          Parent_Name : in     String := "") is

begin

-- No symbols to report

   null;

end Report_Symbols;

-------------------------------------------------------------------------
--|  Abstract: This separate reports symbols to the symbol map.
--|
--|  Warnings: If no symbols are to be reported, this separate could
--|            be deleted.
-------------------------------------------------------------------------
```

Ada Unit 26 Actuator_Class Package Specification

```
with Std_Eng_Units;
with Std_Eng_Types;

use Std_Eng_Units;
use Std_Eng_Types;

package Actuator_Class is

   package Seu renames Std_Eng_Units;
   package Set renames Std_Eng_Types;

   type Object is limited private;

   type Commands is (Set_Leak);

-- ******************** Modifiers ************************** --

   procedure Create (Instance : in out Object; Parent_Name : in String := "");

   procedure Request_State_Change (Instance  : in out Object;
                                   Command   : in     Commands;
                                   Leak_Rate : in     Seu.Gallons_Per_Second);

   procedure Update (Instance   : in out Object;
                     Delta_Time : in     Seu.Seconds;
                     Pressure   : in     Psi);

-- ********************** Selectors ********************** --

   function Stroke (Instance : in Object) return Seu.Feet;

private

   type Inches_Per_Second_Squared is new Set.Real_6;

   type Object is
      record
         Leak_Rate          : Seu.Gallons_Per_Second     := 0.0;
         Spool_Force        : Seu.Pounds_Force           := 0.0;
         Friction           : Seu.Pounds_Force           := 0.0;
         Spring_Force       : Seu.Pounds_Force           := 0.0;
         Accel              : Inches_Per_Second_Squared  := 0.0;
         Velocity           : Seu.Feet_Per_Second        := 0.0;
         Position_Lim       : Seu.Feet                   := 0.0;
         Swash_Plate_Angle  : Seu.Radians                := 0.0;
         Stroke             : Seu.Feet                   := 0.0;
      end record;

end Actuator_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of hydraulic actuators.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 27 Actuator_Class Package Body

```
package body Actuator_Class is
-- ***********************************************************
-- Report Symbols (used by Create)
   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;
-- ****************** Modifiers ****************************
   procedure Create (Instance : in out Object; Parent_Name : in String := "") is
   begin
      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);
   end Create;
-- ***********************************************************
   procedure Calculate_Spool_Force (Instance : in out Object) is
   begin
-- NOTE: Operational details have been omitted.
      null;
   end Calculate_Spool_Force;
-- ***********************************************************
   procedure Calculate_Accel (Instance : in out Object) is
   begin
-- NOTE: Operational details have been omitted.
      null;
   end Calculate_Accel;
-- ***********************************************************
   procedure Calculate_Friction
             (Instance : in out Object; Press_Input : Seu.Psi) is
   begin
-- NOTE: Operational details have been omitted.
      null;
   end Calculate_Friction;
-- ***********************************************************
   procedure Calculate_Velocity (Instance : in out Object) is
   begin
-- NOTE: Operational details have been omitted.
      null;
   end Calculate_Velocity;
-- ***********************************************************
   procedure Calculate_Position (Instance : in out Object) is
   begin
-- NOTE: Operational details have been omitted.
      null;
   end Calculate_Position;
-- ***********************************************************
   procedure Calculate_Swash_Plate_Angle (Instance : in out Object) is
   begin
-- NOTE: Operational details have been omitted.
```

```
      null;
   end Calculate_Swash_Plate_Angle;
-- ***********************************************************

   procedure Calculate_Stroke (Instance : in out Object) is
   begin
-- NOTE: Operational details have been omitted.

      null;
   end Calculate_Stroke;
-- ***********************************************************

   procedure Calculate_Adjustment_Spring (Instance : in out Object) is
   begin
-- NOTE: Operational details have been omitted.

      null;
   end Calculate_Adjustment_Spring;
-- ***********************************************************

   procedure Request_State_Change (Instance  : in out Object;
                                   Command   : in     Commands;
                                   Leak_Rate : in     Seu.Gallons_Per_Second) is
   begin
      case Command is

         when Set_Leak =>

            Instance.Leak_Rate := Leak_Rate;

      end case;
   end Request_State_Change;
-- ***********************************************************

   procedure Update (Instance   : in out Object;
                     Delta_Time : in     Seu.Seconds;
                     Pressure   : in     Psi) is

      Total_Force : Seu.Pounds_Force;

   begin
      Calculate_Friction (Instance => Instance, Press_Input => Pressure);

      Calculate_Adjustment_Spring (Instance => Instance);

      Calculate_Spool_Force (Instance => Instance);

      Total_Force := Instance.Spool_Force -
                        Instance.Spring_Force - Instance.Friction;

      Calculate_Accel (Instance => Instance);

      Calculate_Velocity (Instance => Instance);

      Calculate_Position (Instance => Instance);

      Calculate_Swash_Plate_Angle (Instance => Instance);

      Calculate_Stroke (Instance => Instance);

   end Update;
-- ************************ Selectors ********************

   function Stroke (Instance : in Object) return Seu.Feet is
   begin
      return (Instance.Stroke);
   end Stroke;

end Actuator_Class;
```

Ada Unit 28 Actuator_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;
separate (Actuator_Class)
procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is
begin
-- No symbols to report
   null;
end Report_Symbols;
-------------------------------------------------------------------------
--|  Abstract: This separate reports symbols to the symbol map.
--|
--|  Warnings: If no symbols are to be reported, this separate could
--|            be deleted.
-------------------------------------------------------------------------
```

Ada Unit 29 Centrifugal_Pump_Class Package Specification

```
with Std_Eng_Units;
with Std_Eng_Types;

use Std_Eng_Units;
use Std_Eng_Types;

package Centrifugal_Pump_Class is

   package Seu renames Std_Eng_Units;
   package Set renames Std_Eng_Types;

   type Object is limited private;

-- ********************** Modifiers ************************ --

   procedure Create (Instance : in out Object; Parent_Name : in String);

   procedure Update (Instance      : in out Object;
                     Delta_Time    : in     Seu.Seconds;
                     Supply_Speed  : in     Seu.Radians_Per_Second;
                     Fluid_Avail   : in     Boolean;
                     Consumed_Flow : in     Seu.Gallons_Per_Second);

-- ********************** Selectors ********************** --

   function Torque (Instance : in Object) return Seu.Foot_Pound_Force;

   function Pressure (Instance : Object) return Seu.Psi;

   function Consumed_Flow (Instance : Object) return Seu.Gallons_Per_Second;

private

   type Object is
      record
         Torque : Seu.Foot_Pound_Force   := 0.0;
         Press  : Seu.Psi                := 0.0;
         Flow   : Seu.Gallons_Per_Second := 0.0;
      end record;

end Centrifugal_Pump_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of hydraulic centrifugal pumps which pressurize fluid
--|            based on input shaft rotation speed.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 30 Centrifugal_Pump_Class Package Body

```
package body Centrifugal_Pump_Class is
-- ************************************************************

-- Report Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String := "") is separate;

-- ********************** Modifiers **************************

   procedure Create (Instance : in out Object; Parent_Name : in String) is
   begin

      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

   end Create;

-- ************************************************************

   procedure Update (Instance      : in out Object;
                     Delta_Time    : in     Seu.Seconds;
                     Supply_Speed  : in     Seu.Radians_Per_Second;
                     Fluid_Avail   : in     Boolean;
                     Consumed_Flow : in     Seu.Gallons_Per_Second) is
   begin

-- Set flow rate consumed for output

      Instance.Flow := Consumed_Flow;

-- Function needed to convert supply speed, fluid availability and
-- delta time into pressure.

      Instance.Press := 0.0;

-- Function needed to convert supply speed, fluid availability and
-- delta time into torque.

      Instance.Torque := 0.0;

   end Update;

-- ************************ Selectors ************************

   function Torque (Instance : in Object) return Seu.Foot_Pound_Force is
   begin

      return (Instance.Torque);

   end Torque;

-- ************************************************************

   function Pressure (Instance : Object) return Seu.Psi is
   begin

      return (Instance.Press);

   end Pressure;

-- ************************************************************

   function Consumed_Flow (Instance : Object) return Seu.Gallons_Per_Second is
   begin

      return (Instance.Flow);

   end Consumed_Flow;

end Centrifugal_Pump_Class;
```

Ada Unit 31 Centrifugal_Pump_Class.Report_Symbols Separate Procedure

```
with Symbol_Map;

separate (Centrifugal_Pump_Class)

procedure Report_Symbols (Instance    : in out Object;
                          Parent_Name : in     String := "") is

begin
```

```
-- No symbols to report

   null;

end Report_Symbols;

------------------------------------------------------------------------
--    Abstract: This separate reports symbols to the symbol map.
--
--    Warnings: If no symbols are to be reported, this separate could
--              be deleted.
------------------------------------------------------------------------
```

Ada Unit 32 Hydraulic_Pump_Class Package Specification

```
with Actuator_Class;
with Axial_Piston_Pump_Class;
with Centrifugal_Pump_Class;
with Random;

with Std_Eng_Types;
with Std_Eng_Units;

use Std_Eng_Types;
use Std_Eng_Units;

package Hydraulic_Pump_Class is

   package Set renames Std_Eng_Types;
   package Seu renames Std_Eng_Units;

   type Object is limited private;

   type Commands is (Compensator_Fail,   -- Erratic pressure flow from pump
                     Modify_Flow_Rate,   -- Scale pump flow rate
                     Pump_Fail);         -- No flow when pump is driven

-- ********************** Modifiers ********************** --

   procedure Create (Instance : in out Object; Parent_Name : in String := "");

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in Commands;
                                   Apply    : in Boolean;
                                   Bias     : in Seu.Non_Dimensional := 0.0;
                                   Scale    : in Seu.Non_Dimensional := 1.0);

   procedure Update (Instance        : in out Object;
                     Delta_Time      : in     Seu.Seconds;
                     Fluid_Avail     : in     Boolean;
                     Shaft_Speed     : in     Seu.Radians_Per_Second;
                     System_Pressure : in     Seu.Psi);

-- ********************** Selectors ********************** --

   function Consumed_Flow (Instance : in Object) return Seu.Gallons_Per_Second;

   function Output_Flow (Instance : in Object) return Seu.Gallons_Per_Second;

   function Pump_On (Instance : in Object) return Boolean;

   function Torque (Instance : in Object) return Seu.Foot_Pound_Force;

private

   type Object is
      record
         Actuator            : Actuator_Class.Object;
         Axial_Pump          : Axial_Piston_Pump_Class.Object;
         Centrifugal_Pump    : Centrifugal_Pump_Class.Object;
         Compensator_Fail    : Boolean                    := False;
         Consumed_Flow       : Seu.Gallons_Per_Second     := 0.0;
         Flow_Bias           : Seu.Gallons_Per_Second     := 0.0;
         Flow_Out            : Seu.Gallons_Per_Second     := 0.0;
         Flow_Scale          : Seu.Non_Dimensional        := 1.0;
         Press_Delta         : Seu.Psi                    := 0.0;
         Pump_Sensor_Failure : Boolean                    := False;
         Pump_Status         : Set.On_Off                 := Off;
         Random_Id           : Random.Handle;
         Shaft_Fail          : Boolean                    := False;
         Shaft_Speed         : Seu.Radians_Per_Second     := 0.0;
         Torque              : Seu.Foot_Pound_Force       := 0.0;
      end record;

end Hydraulic_Pump_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of hydraulic pumps composed of an axial piston pump, an
```

This class uses three other classes.

```
--              actuator and a centrifugal pump.
--
--   Warnings: pragma Inline used in body.
------------------------------------------------------------------------
```

Ada Unit 33 Hydraulic_Pump_Class Package Body

```
package body Hydraulic_Pump_Class is

-- Pump Data From NASA document NASA-91-10761, "Spec for Acme Hydraulic Pump
-- Co. Type abc-456b Hydraulic Pump".

   Num_Of_Pistons     : constant Integer         := 6;
   Single_Piston_Area : constant Seu.Square_Feet := 5.45413E-3;  -- 1" diam

-- ************************************************************

-- Overloaded Operators

   function "*" (Left  : in Seu.Non_Dimensional;
                 Right : in Seu.Gallons_Per_Second)
                return Seu.Gallons_Per_Second is
   begin
      return Seu.Gallons_Per_Second (Set.Real_6 (Left) * Set.Real_6 (Right));
   end "*";

   function "*" (Left : in Seu.Psi; Right : in Seu.Non_Dimensional)
                return Seu.Psi is
   begin
      return Seu.Psi (Set.Real_6 (Left) * Set.Real_6 (Right));
   end "*";

   pragma Inline ("*");

-- ************************************************************

-- Report_Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;

-- ********************** Modifiers ****************************

   procedure Create (Instance : in out Object; Parent_Name : in String := "") is

   begin

      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

      Actuator_Class.Create (Instance    => Instance.Actuator,
                             Parent_Name => Parent_Name & ".accuat");

      Axial_Piston_Pump_Class.Create
        (Instance          => Instance.Axial_Pump,
         Parent_Name       => Parent_Name & ".axial_pump",
         Number_Of_Pistons => Num_Of_Pistons,
         Piston_Area       => Single_Piston_Area);

      Centrifugal_Pump_Class.Create (Instance    => Instance.Centrifugal_Pump,
                                     Parent_Name => Parent_Name & ".cent_pump");

      Random.Initialize (The_Handle => Instance.Random_Id);

   end Create;

-- ************************************************************

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in Commands;
                                   Apply    : in Boolean;
                                   Bias     : in Seu.Non_Dimensional := 0.0;
                                   Scale    : in Seu.Non_Dimensional := 1.0) is

   begin

      case Command is

         when Compensator_Fail =>

            Instance.Compensator_Fail := A

         when Modify_Flow_Rate =>

            Instance.Flow_Scale := Sca
            Instance.Flow_Bias  := Seu       s_Per_Second (Bias);
```

```
         when Pump_Fail =>
            Instance.Shaft_Fail := Apply;
      end case;
   end Request_State_Change;
-- ************************************************************
   procedure Update (Instance        : in out Object;
                     Delta_Time      : in     Seu.Seconds;
                     Fluid_Avail     : in     Boolean;
                     Shaft_Speed     : in     Seu.Radians_Per_Second;
                     System_Pressure : in     Seu.Psi) is separate;
-- ***************************** Selectors **********************
   function Consumed_Flow (Instance : in Object)
                           return Seu.Gallons_Per_Second is
   begin
      return Centrifugal_Pump_Class.Consumed_Flow (Instance.Centrifugal_Pump);
   end Consumed_Flow;
-- ************************************************************
   function Output_Flow (Instance : in Object) return Seu.Gallons_Per_Second is
   begin
      return (Instance.Flow_Out);
   end Output_Flow;
-- ************************************************************
   function Pump_On (Instance : in Object) return Boolean is
   begin
      return (Instance.Pump_Status = Set.On);
   end Pump_On;
-- ************************************************************
   function Torque (Instance : in Object) return Seu.Foot_Pound_Force is
   begin
      return (Instance.Torque);
   end Torque;
end Hydraulic_Pump_Class;
```

Ada Unit 34 Hydraulic_Pump_Class.Report_Symbols Separate Procedure

```
with Symbols;
separate (Hydraulic_Pump_Class)
procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is
begin
   Symbols.Register (Name         => Parent_Name & ".Flow_Out",
                     Base_Type    => Symbols.Real
                     Tick_Address => Instance.Flow_Out'Address,
                     Tick_Size    => Instance.Flow_Out'Size);
end Report_Symbols;
------------------------------------------------------------------------
--|  Abstract: This procedure reports the flow attribute of the
--|            hydraulic pump class to the symbol map.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 35 Hydraulic_Pump_Class.Update Separate Procedure

```
separate (Hydraulic_Pump_Class)

procedure Update (Instance        : in out Object;
                  Delta_Time      : in     Seu.Seconds;
                  Fluid_Avail     : in     Boolean;
                  Shaft_Speed     : in     Seu.Radians_Per_Second;
                  System_Pressure : in     Seu.Psi) is

begin

-- Determine shaft speed based on suppled shaft speed and shaft status.

   if not Instance.Shaft_Fail then

      Instance.Shaft_Speed := Shaft_Speed;

   else

      Instance.Shaft_Speed := 0.0;

   end if;

-- Update Centrifugal Pump

   Centrifugal_Pump_Class.Update
     (Instance      => Instance.Centrifugal_Pump,
      Delta_Time    => Delta_Time,
      Supply_Speed  => Instance.Shaft_Speed,
      Fluid_Avail   => Fluid_Avail,
      Consumed_Flow => Axial_Piston_Pump_Class.Flow (Instance.Axial_Pump));

-- Calculate pressure difference between scavenge pump output pressure and
-- system pressure.  Include effects of pressure compensator failure (if
-- required).

   Instance.Press_Delta := System_Pressure - Centrifugal_Pump_Class.Pressure
                                               (Instance.Centrifugal_Pump);

   if Instance.Press_Delta <= 0.1 then

      Instance.Press_Delta := 0.0;

   end if;

   if Instance.Compensator_Fail then

      Instance.Press_Delta :=
        Instance.Press_Delta *
          Seu.Non_Dimensional (2.0 * Random.Float_Value (Instance.Random_Id));

   end if;

-- Update pressure compensation actuator

   Actuator_Class.Update (Instance   => Instance.Actuator,
                          Delta_Time => Delta_Time,
                          Pressure   => Instance.Press_Delta);

-- Update Axial Piston Pump

   Axial_Piston_Pump_Class.Update
     (Instance      => Instance.Axial_Pump,
      Press         => Instance.Press_Delta,
      Rotation_Rate => Instance.Shaft_Speed,
      Stroke        => Actuator_Class.Stroke (Instance.Actuator));

   Instance.Flow_Out := Axial_Piston_Pump_Class.Flow (Instance.Axial_Pump);

   Instance.Flow_Out :=
     Instance.Flow_Scale * Instance.Flow_Out + Instance.Flow_Bias;

   Update ouput variables

   if Shaft_Speed >= 0.1 then

      Instance.Pump_Status := Set.On;
   else

      Instance.Pump_Status := Set.Off;

   end if;
```

Update of class uses state variables inputs and output of state from other class.

```
-- Pump torque includes centrifugal pump and axial piston pump torque terms

   Instance.Torque := Centrifugal_Pump_Class.Torque
                        (Instance.Centrifugal_Pump) +
                      Axial_Piston_Pump_Class.Torque (Instance.Axial_Pump);

end Update;

-------------------------------------------------------------------------
--   Abstract: This procedure provides the periodic update of the
--             Hydraulic Pump Class.
--
--   Warnings: None.
-------------------------------------------------------------------------
```

Use of state output functions from other classes to determine state data.

Ada Unit 36 Distribution_System_Class Package Specification

```
with Std_Eng_Units;
with Std_Eng_Types;

use Std_Eng_Units;
use Std_Eng_Types;

package Distribution_System_Class is

   package Seu renames Std_Eng_Units;
   package Set renames Std_Eng_Types;

   type Object is limited private;

   type Commands is (Set_Leak);

-- ********************** Modifiers ********************** --

   procedure Create (Instance    : in out Object;
                     Parent_Name : in     String := "";
                     Press_Const : in     Seu.Non_Dimensional);

   procedure Request_State_Change (Instance  : in out Object;
                                   Command   : in     Commands;
                                   Apply     : in     Boolean;
                                   Leak_Rate : in     Seu.Gallons_Per_Second :=
                                     0.25);  -- 15 gal per min

   procedure Update (Instance      : in out Object;
                     Delta_Time    : in     Seu.Seconds;
                     Consumed_Flow : in     Seu.Gallons_Per_Second;
                     Supply_Flow   : in     Seu.Gallons_Per_Second);

-- ********************** Selectors ********************** --

   function System_Pressure (Instance : in Object) return Seu.Psi;

private

   type Object is
      record
         Leak_Rate    : Seu.Gallons_Per_Second := 0.0;
         Sys_Constant : Seu.Non_Dimensional    := 0.0;
         Sys_Pressure : Seu.Psi                := 0.0;
      end record;

end Distribution_System_Class;

------------------------------------------------------------------------
--|  Abstract: This package models the system of pipes which distribute
--|            hydraulic fluid between components.
--|
--|  Warnings: pragma Inline used in body
------------------------------------------------------------------------
```

Ada Unit 37 Distribution_System_Class Package Body

```
package body Distribution_System_Class is

-- ******************** Overloaded Operators *********************

   function *** (Left  : in Seu.Non_Dimensional;
                 Right : in Seu.Gallons_Per_Second)
                return Seu.Gallons_Per_Second is
   begin
      return Seu.Gallons_Per_Second (Set.Real_6 (Left) * Set.Real_6 (Right));
   end ***;

   pragma Inline ("***");

-- ************************************************************

-- Report Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String := "") is separate;

-- ************************** Modifiers ********************************

   procedure Create (Instance    : in out Object;
                     Parent_Name : in     String := "";
                     Press_Const : in     Seu.Non_Dimensional) is
   begin
      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

      Instance.Sys_Constant := Press_Const;

   end Create;

-- ************************************************************

   procedure Request_State_Change
               (Instance  : in out Object;
                Command   : in     Commands;
                Apply     : in     Boolean;
                Leak_Rate : in     Seu.Gallons_Per_Second := 0.25) is
   begin
      case Command is

         when Set_Leak =>

            if Apply then

               Instance.Leak_Rate := Leak_Rate;

            else

               Instance.Leak_Rate := 0.0;

            end if;

      end case;

   end Request_State_Change;

-- ************************************************************

   procedure Update (Instance      : in out Object;
                     Delta_Time    : in     Seu.Seconds;
                     Consumed_Flow : in     Seu.Gallons_Per_Second;
                     Supply_Flow   : in     Seu.Gallons_Per_Second) is

      Total_Flow  : Seu.Gallons_Per_Second;
      Delta_Press : Seu.Psi;

   begin

-- Determine pressure change by flow rate into/out of system multiplied by
-- system constant.

      Total_Flow := Supply_Flow - Consumed_Flow - Instance.Leak_Rate;

      Delta_Press := Seu.Psi (Instance.Sys_Constant * Total_Flow);
```

Register state data with symbol map for use by IOS.

```
-- Calculate new system pressure.
      Instance.Sys_Pressure := Instance.Sys_Pressure + Delta_Press;
   end Update;
-- ************************* Selectors ************************ --
   function System_Pressure (Instance : in Object) return Seu.Psi is
   begin
      return (Instance.Sys_Pressure);
   end System_Pressure;
end Distribution_System_Class;
```

Ada Unit 38 Distribution_System_Class.Report_Symbols Separate Procedure

```
with Symbols;
separate (Distribution_System_Class)
procedure Report_Symbols (Instance    : in out Object;
                          Parent_Name : in     String := "") is
begin
   Symbols.Register (Name         => Parent_Name & ".System_Pressure",
                     Base_Type    => Symbols.Real,
                     Tick_Address => Instance.Sys_Pressure'Address,
                     Tick_Size    => Instance.Sys_Pressure'Size);
end Report_Symbols;
-------------------------------------------------------------------------
--|  Abstract: This procedure reports the system pressure attribute of
--|            the distribution system class to the symbol map.
--|
--|  Warnings: None
-------------------------------------------------------------------------
```

Class state data registered with symbol map for use by IOS.

Ada Unit 39 Generic_Reservoir_Class Package Specification

```
generic
   type Volume_Units    is digits <>;
   type Vol_Rate_Units  is digits <>;
   type Time_Units      is digits <>;

   Max_Leak_Rate : in Vol_Rate_Units;

package Generic_Reservoir_Class is

   type Object is limited private;

   type Commands is (Leak_Malfunction, Set_Qty);

-- ************************ Modifiers ************************ --

   procedure Create (Instance    : in out Object;
                     Parent_Name : in     String := "";
                     Init_Qty    : in     Volume_Units);

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Quantity : in     Volume_Units := 0.0);

   procedure Update (Instance      : in out Object;
                     Delta_Time    : in     Time_Units;
                     Consumed_Rate : in     Vol_Rate_Units;
                     Returned_Rate : in     Vol_Rate_Units);

-- ********************** Selectors ********************** --

   function Fluid_Avail (Instance : in Object) return Boolean;

   function Quantity (Instance : in Object) return Volume_Units;

private

   type Object is
      record
         Fluid_Avail : Boolean        := False;
         Qty         : Volume_Units   := 0.0;
         Leak_Rate   : Vol_Rate_Units := 0.0;
      end record;

end Generic_Reservoir_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of hydraulic reservoirs.
--|
--|  Warnings: This class should be instantiated with compatiable units
--|            (i.e. gallons, gallons_per_sec & seconds) to prevent
--|            incorrect calculations.
------------------------------------------------------------------------
```

Totally generic class. No dependence on SEU/SET. This could be instantiated in English or metric units. User needs to ensure units are compatible.

Ada Unit 40 Generic_Reservoir_Class Package Body

```
package body Generic_Reservoir_Class is

-- ************************************************************

-- Overloaded Operators

   function *** (Left : Vol_Rate_Units; Right : Time_Units)
                return Volume_Units is
   begin
      return (Volume_Units (Left) * Volume_Units (Right));
   end ***;

-- ************************************************************

-- Report_Symbols (used by Create)

   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;

-- ******************** Modifiers *************************

   procedure Create (Instance    : in out Object;
                     Parent_Name : in     String := "";
                     Init_Qty    : in     Volume_Units) is
   begin
      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);

      Instance.Qty         := Init_Qty;
      Instance.Fluid_Avail := (Instance.Qty > 0.0);

   end Create;

-- ************************************************************

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Quantity : in     Volume_Units := 0.0) is
   begin
      case Command is

         when Leak_Malfunction =>

            if Instance.Leak_Rate >= 0.0 then

               Instance.Leak_Rate := Max_Leak_Rate;

            else

               Instance.Leak_Rate := 0.0;

            end if;

         when Set_Qty =>

            Instance.Qty := Quantity;

      end case;

   end Request_State_Change;

-- ************************************************************

   procedure Update (Instance      : in out Object;
                     Delta_Time    : in     Time_Units;
                     Consumed_Rate : in     Vol_Rate_Units;
                     Returned_Rate : in     Vol_Rate_Units) is

      Delta_Qty : Volume_Units;

   begin
      Delta_Qty := (Returned_Rate - Consumed_Rate) * Delta_Time;

      Instance.Qty := Instance.Qty + Delta_Qty;

      if Instance.Qty <= 0.0 then

         Instance.Qty := 0.0;
```

State variable based on input parameters from this procedure.

```
      end if;

      Instance.Fluid_Avail := not (Instance.Qty <= 0.0);

   end Update;
-- ************************ Selectors ****************************

   function Fluid_Avail (Instance : in Object) return Boolean is
   begin
      return (Instance.Fluid_Avail);
   end Fluid_Avail;
-- ************************************************************

   function Quantity (Instance : in Object) return Volume_Units is
   begin

      return (Instance.Qty);

   end Quantity;

end Generic_Reservoir_Class;
```

Ada Unit 41 Generic_Reservoir_Class.Report_Symbols Separate Procedure

```
with Symbols;

separate (Generic_Reservoir_Class)

procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is

begin

   Symbols.Register (Name         => Parent_Name & ".Quantity",
                     Base_Type    => Symbols.Real,
                     Tick_Address => Instance.Qty'Address,
                     Tick_Size    => Instance.Qty'Size);

end Report_Symbols;

------------------------------------------------------------------------
--|  Abstract: This procedure reports the quantity attribute of the
--|            reservoir class to the symbol map.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 42 Valve_Class Package Specification

```
with Std_Eng_Types;
with Std_Eng_Units;

use Std_Eng_Types;
use Std_Eng_Units;

package Valve_Class is

   package Set renames Std_Eng_Types;
   package Seu renames Std_Eng_Units;

   type Object is limited private;

   type Commands is (Initialize, Freeze_Valve);

-- ********************** Modifiers ********************** --

   procedure Create (Instance : in out Object; Parent_Name : in String := "");

   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean);

   procedure Update (Instance  : in out Object;
                     Close_Cmd : in     Set.On_Off;
                     Open_Cmd  : in     Set.On_Off;
                     Pressure  : in     Seu.Psi;
                     Power     : in     Seu.Volts;
                     Flow_Rate : in     Seu.Gallons_Per_Second);

-- ********************** Selectors ********************** --

   function Pressure (Instance : in Object) return Seu.Psi;

   function Flow_Rate (Instance : in Object) return Seu.Gallons_Per_Second;

   function Electrical_Load (Instance : in Object) return Seu.Amps;

   function Full_Closed (Instance : in Object) return Boolean;

   function Full_Open (Instance : in Object) return Boolean;

private

   type Positions is (Open, In_Transition, Closed);

   type Object is
      record
         Electrical_Load     : Seu.Amps                := 0.0;
         Flow_Rate           : Seu.Gallons_Per_Second  := 0.0;
         Movement_Efficiency : Seu.Non_Dimensional     := 1.0;
         Power               : Seu.Volts               := 0.0;
         Position            : Positions               := Closed;
         Pressure            : Seu.Psi                 := 0.0;
      end record;

end Valve_Class;

------------------------------------------------------------------------
--|  Abstract: This package provides a real time simulation of a class
--|            of hydraulic valves.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 43 Valve_Class Package Body

```
package body Valve_Class is
-- ************************************************************
-- Report_Symbols (used by Create)
   procedure Report_Symbols (Instance    : in out Object;
                             Parent_Name : in     String) is separate;
-- ********************** Modifiers ********************** --
   procedure Create (Instance : in out Object; Parent_Name : in String := "") is
   begin
      Report_Symbols (Instance => Instance, Parent_Name => Parent_Name);
   end Create;
-- ************************************************************
   procedure Request_State_Change (Instance : in out Object;
                                   Command  : in     Commands;
                                   Apply    : in     Boolean) is
   begin
      case Command is
         when Initialize =>
            Instance.Pressure        := 0.0;
            Instance.Flow_Rate       := 0.0;
            Instance.Power           := 0.0;
            Instance.Electrical_Load := 0.0;
            Instance.Position        := Closed;
         when Freeze_Valve =>
            if Apply then
               Instance.Movement_Efficiency := 0.0;  -- freeze valve
            else
               Instance.Movement_Efficiency := 1.0;  -- unfreeze valve
            end if;
      end case;
   end Request_State_Change;
-- ************************************************************
   procedure Update (Instance  : in out Object;
                     Close_Cmd : in     Set.On_Off;
                     Open_Cmd  : in     Set.On_Off;
                     Pressure  : in     Seu.Psi;
                     Power     : in     Seu.Volts;
                     Flow_Rate : in     Seu.Gallons_Per_Second) is
   begin
      -- NOTE: Valve operation details have been omitted
      Instance.Pressure  := Pressure;
      Instance.Power     := Power;
      Instance.Flow_Rate := Flow_Rate;
      Instance.Position  := Open;
   end Update;
-- ********************** Selectors ******************** --
   function Pressure (Instance : in Object) return Seu.Psi is
   begin
      return Instance.Pressure;
   end Pressure;
-- ************************************************************
```

```
   function Flow_Rate (Instance : in Object) return Seu.Gallons_Per_Second is
   begin
      return Instance.Flow_Rate;
   end Flow_Rate;
-- ************************************************************
   function Electrical_Load (Instance : in Object) return Seu.Amps is
   begin
      return Instance.Electrical_Load;
   end Electrical_Load;
-- ************************************************************
   function Full_Closed (Instance : in Object) return Boolean is
   begin
      return (Instance.Position = Closed);
   end Full_Closed;
-- ************************************************************
   function Full_Open (Instance : in Object) return Boolean is
   begin
      return (Instance.Position = Open);
   end Full_Open;
end Valve_Class;
```

Ada Unit 44 Valve_Class.Report_Symbols Separate Procedure

```
with Symbols;
separate (Valve_Class)
procedure Report_Symbols (Instance : in out Object; Parent_Name : in String) is
begin
   Symbols.Register (Name         => Parent_Name & ".Position",
                     Base_Type    => Symbols.Enum,
                     Tick_Address => Instance.Position'Address,
                     Tick_Size    => Instance.Position'Size);
end Report_Symbols;
------------------------------------------------------------------------
--|  Abstract: This procedure reports the position attribute of the
--|            the valve class to the symbol map.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 45 Elec_Sys_Intfc_Defs Package Specification

```
with Dis;
with Orvc_Defs;
with Std_Eng_Types;
with Std_Eng_Units;

use Std_Eng_Types;
use Std_Eng_Units;

package Elec_Sys_Intfc_Defs is

   package Set renames Std_Eng_Types;
   package Seu renames Std_Eng_Units;

-- Circuit Breaker idents from NASA Space Station System Schematic,
-- Document NASA-SS-911-1234.

   type Cb_Ids is (Cb_1021_001,  -- Hyd Sys Motor Power sys 1
                   Cb_1021_002,  -- Hyd Sys Motor Power sys 2
                   Cb_1021_003,  -- Hyd Sys Motor Relay Power sys 1
                   Cb_1021_004,  -- Hyd Sys Motor Relay Power sys 2
                   Cb_1022_001,  -- Hyd Sys Isolation Valve Power sys 1
                   Cb_1022_002,  -- Hyd Sys Isolation Valve Power sys 2
                   Cb_1023_001,  -- Hyd Sys Pressure Sensor Power sys 1 & sys 2
                   Cb_1023_002,  -- Hyd Sys Quantity Sensor Power
                   Cb_1031_001,  -- Landing Light Power
                   Cb_1032_001,  -- Windshield Wiper power
                   Cb_1033_001,  -- UHF Radio Power
                   Cb_1033_002,  -- VHF Radio Power
                   Cb_1033_003); -- Radio Control Panel Indicator power

   type Elec_Power is
      record
         Power   : On_Off;
         Voltage : Volts;
      end record;

   type Cb_Power is array (Cb_Ids) of Elec_Power;

-- ************************************************************************
-- Electric Power Messages -> Power provided to consumers - one to many
-- ************************************************************************

   type Elec_Power_Msgs is
      record
         Cb : Cb_Power;
      end record;

   type Elec_Power_Msg_Ptrs is access Elec_Power_Msgs;

   Elec_Power_Msg_Size : constant Integer := Elec_Power_Msgs'Size;

-- message identifiers

   Elec_Power_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Electrical_System,
                           Name   => "Elec_Power_Msg_ID",
                           Bits   => Elec_Power_Msg_Size);

-- ************************************************************************
-- Electric Load Messages -> Load returned from consumers - many to one
-- ************************************************************************

   type Elec_Load_Msgs is
      record
         Cb   : Cb_Ids;
         Load : Seu.Amps;
      end record;

   type Elec_Load_Msg_Ptrs is access Elec_Load_Msgs;

   Elec_Load_Msg_Size : constant Integer := Elec_Load_Msgs'Size;

-- message identifiers

   Elec_Load_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Electrical_System,
```

```
                              Name    => "Elec_Load_Msg_ID",
                              Bits    => Elec_Load_Msg_Size);

end Elec_Sys_Intfc_Defs;

------------------------------------------------------------------------
--|  Abstract: This package contains the Electrical System Interface
--|            Definition types.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 46 Hyd_Control_Panel_Intfc_Defs Package Specification

```
with Dis;
with Orvc_Defs;

with Std_Eng_Types;
with Std_Eng_Units;

use Std_Eng_Types;
use Std_Eng_Units;

package Hyd_Control_Panel_Intfc_Defs is

   package Set renames Std_Eng_Types;
   package Seu renames Std_Eng_Units;

-- ************************************************************
-- Motor Command Messages -> Output to Hyd Sys Electric Motors
-- ************************************************************

   type Motor_Cmd_Msgs is
      record
         Motor_Cmd : Set.On_Off;
      end record;

   type Motor_Cmd_Msg_Ptrs is access Motor_Cmd_Msgs;

   Motor_Cmd_Msg_Size : constant Integer :=
     Motor_Cmd_Msgs'Size;

-- message identifiers

   Sys_1_Motor_Cmd_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_Control_Panel,
                           Name   => "Sys_1_Motor_Cmd_Msg_ID",
                           Bits   => Motor_Cmd_Msg_Size);

   Sys_2_Motor_Cmd_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_Control_Panel,
                           Name   => "Sys_2_Motor_Cmd_Msg_ID",
                           Bits   => Motor_Cmd_Msg_Size);

-- ************************************************************
-- Valve Command Messages -> Output to Hyd Sys Isolation Valves
-- ************************************************************

   type Valve_Cmd_Msgs is
      record
         Vlv_Close_Cmd : Set.On_Off;
         Vlv_Open_Cmd  : Set.On_Off;
      end record;

   type Valve_Cmd_Msg_Ptrs is access Valve_Cmd_Msgs;

   Valve_Cmd_Msg_Size : constant Integer :=
     Valve_Cmd_Msgs'Size;

-- message identifiers

   Sys_1_Valve_Cmd_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_Control_Panel,
                           Name   => "Sys_1_Valve_Cmd_Msg_ID",
                           Bits   => Valve_Cmd_Msg_Size);

   Sys_2_Valve_Cmd_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_Control_Panel,
```

```
                              Name    => "Sys_2_Valve_Cmd_Msg_ID",
                              Bits    => Valve_Cmd_Msg_Size);

end Hyd_Control_Panel_Intfc_Defs;
------------------------------------------------------------------------
--|  Abstract: This package contains the Hydraulic Control Panel
--|            interface type definitions.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 47 Hyd_Sys_Intfc_Defs Package Specification

```
with Dis;
with Orvc_Defs;
with Std_Eng_Types;
with Std_Eng_Units;

use Std_Eng_Types;
use Std_Eng_Units;

package Hyd_Sys_Intfc_Defs is

   package Set renames Std_Eng_Types;
   package Seu renames Std_Eng_Units;

-- ************************************************************
-- Aural Cue Messages -> output to Aural cue system for sounds
-- ************************************************************

   type Aural_Cue_Msgs is
      record
         Pump_Noise_Sys_1  : On_Off;
         Pump_Noise_Sys_2  : On_Off;
         Motor_Noise_Sys_1 : On_Off;
         Motor_Noise_Sys_2 : On_Off;
      end record;

   type Aural_Cue_Msg_Ptrs is access Aural_Cue_Msgs;

   Aural_Cue_Msg_Size : constant Integer :=
     Aural_Cue_Msgs'Size;

-- message identifiers

   Aural_Cue_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_System,
                           Name   => "Aural_Cue_Msg_ID",
                           Bits   => Aural_Cue_Msg_Size);

-- ********************************************************************
-- Hyd Sys Pressure Messages -> System Pressure provided to consumers
-- ********************************************************************

   type Hyd_Sys_Press_Msgs is      -- one to many -> output to consumers
      record
         Press : Seu.Psi;
      end record;

   type Hyd_Sys_Press_Msg_Ptrs is access Hyd_Sys_Press_Msgs;

   Hyd_Sys_Press_Msg_Size : constant Integer :=
     Hyd_Sys_Press_Msgs'Size;

-- message identifiers

   Sys_1_Press_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_System,
                           Name   => "Sys_1_Press_Msg_ID",
                           Bits   => Hyd_Sys_Press_Msg_Size);

   Sys_2_Press_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_System,
                           Name   => "Sys_2_Press_Msg_ID",
                           Bits   => Hyd_Sys_Press_Msg_Size);

-- ************************************************************
-- Hyd Sys Flow Messages -> Flow rates returned from consumers
-- ************************************************************
```

```
   type Hyd_Sys_Flow_Msgs is        -- many to one -> returned by consumers
      record
         Press_Flow  : Seu.Gallons_Per_Second;
         Return_Flow : Seu.Gallons_Per_Second;
      end record;

   type Flow_Msg_Ptrs is access Hyd_Sys_Flow_Msgs;

   Flow_Msg_Size : constant Integer :=
     Hyd_Sys_Flow_Msgs'Size;

-- message identifiers

   Sys_1_Flow_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_System,
                           Name   => "Sys_1_Flow_Msg_ID",
                           Bits   => Flow_Msg_Size);

   Sys_2_Flow_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_System,
                           Name   => "Sys_2_Flow_Msg_ID",
                           Bits   => Flow_Msg_Size);

-- ****************************************************************************
-- Hyd Status Message -> Pump, Motor & Valve Status Returned to Hyd Control Panel
-- ****************************************************************************

   type Hyd_Status_Msgs is          -- returned to hyd control pnl by hyd sys
      record
         Motor_Status                  : Set.On_Off;
         Pump_Status                   : Set.On_Off;
         Valve_Sensed_Not_Full_Open    : Boolean;
         Valve_Sensed_Not_Full_Closed  : Boolean;
      end record;

   type Status_Msg_Ptrs is access Hyd_Status_Msgs;

   Status_Msg_Size : constant Integer :=
     Hyd_Status_Msgs'Size;

-- message identifiers

   Sys_1_Status_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_System,
                           Name   => "Sys_1_Status_Msg_ID",
                           Bits   => Status_Msg_Size);

   Sys_2_Status_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_System,
                           Name   => "Sys_2_Status_Msg_ID",
                           Bits   => Status_Msg_Size);

-- ********************************************************************
-- Hyd Sys Indicator Messages -> Status returned to Hyd Control Panel
-- ********************************************************************

   type Hyd_Sys_Indicator_Msgs is  -- returned to hyd control pnl by hyd sys
      record
         Sys_1_Press_Indicated : Seu.Psi;
         Sys_2_Press_Indicated : Seu.Psi;
         Sys_Qty_Indicated     : Seu.Gallons;
      end record;

   type Hyd_Sys_Indicator_Msg_Ptrs is access Hyd_Sys_Indicator_Msgs;

   Hyd_Sys_Indicator_Msg_Size : constant Integer :=
     Hyd_Sys_Indicator_Msgs'Size;

-- message identifiers

   Hyd_Sys_Indicator_Msg_Id : constant Dis.Message_Id :=
     Dis.Register_Message (Parent => Orvc_Defs.Hydraulic_System,
                           Name   => "Hyd_Sys_Indicator_Msg_ID",
                           Bits   => Hyd_Sys_Indicator_Msg_Size);

end Hyd_Sys_Intfc_Defs;

------------------------------------------------------------------------
--|  Abstract: This package contains the Hydraulic System Interface
```

```
--|            definitions.
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 48 Hydraulic_System_Partition Package Specification

```
with Hyd_Control_Panel_Intfc_Defs;
with Elec_Sys_Intfc_Defs;
with Hyd_Sys_Intfc_Defs;

package Hydraulic_System_Partition is

end Hydraulic_System_Partition;

------------------------------------------------------------------------
--|  Abstract: This package contains the Hydraulic System Partition
--|            specification.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

External partition interface definitions

Ada Unit 49 Hydraulic_System_Partition Package Body

```
with Dis;
with Hydraulic_System_Defs;
with Symbols;

with Message;
with Mailbox;
with Generic_Model;

with Std_Eng_Types;
with Std_Eng_Units;
with Orvc_Common_Types;

with Generic_Reservoir_Class;
with Generic_Sensor_Class;

with Accumulator_Class;
with Distribution_System_Class;
with Drive_Unit_Class;
with Hydraulic_Pump_Class;
with Valve_Class;

use Std_Eng_Types;
use Std_Eng_Units;

package body Hydraulic_System_Partition is

-- ---------------
-- Package Renames
-- ---------------

   package Set renames Std_Eng_Types;
   package Seu renames Std_Eng_Units;
   package Oct renames Orvc_Common_Types;

-- ----------------------
-- Generic Instantiations
-- ----------------------

   package Pressure_Sensor_Class is
     new Generic_Sensor_Class (Load_Units     => Seu.Amps,
                               Non_Dim_Units => Seu.Non_Dimensional,
                               Sensed_Units  => Seu.Psi);

   package Quantity_Sensor_Class is
     new Generic_Sensor_Class (Load_Units     => Seu.Amps,
                               Non_Dim_Units => Seu.Non_Dimensional,
                               Sensed_Units  => Seu.Gallons);

   package Hyd_Reservoir_Class is
     new Generic_Reservoir_Class
           (Volume_Units    => Seu.Gallons,
            Vol_Rate_Units => Seu.Gallons_Per_Second,
            Time_Units     => Seu.Seconds,
            Max_Leak_Rate  => 0.0333333);     -- 2.0 gallons per min

-- -----------
-- Local Types
-- -----------
```

SVM references

Class references

Instantiation of generic classes for data types

Generic class instantiation with configuration object.

```
   type On_Off_A is array (Oct.Sys_1_Sys_2) of Set.On_Off;
   type Volt_A   is array (Oct.Sys_1_Sys_2) of Seu.Volts;
   type Rpm_A    is array (Oct.Sys_1_Sys_2) of Seu.Rpm;
   type Gps_A    is array (Oct.Sys_1_Sys_2) of Seu.Gallons_Per_Second;

   type Accumulators is array (Oct.Sys_1_Sys_2) of Accumulator_Class.Object;
   type Distribution_Systems is array (Oct.Sys_1_Sys_2) of
                                Distribution_System_Class.Object;
   type Drive_Units is array (Oct.Sys_1_Sys_2) of Drive_Unit_Class.Object;
   type Isolation_Valves is array (Oct.Sys_1_Sys_2) of Valve_Class.Object;
   type Pumps is array (Oct.Sys_1_Sys_2) of Hydraulic_Pump_Class.Object;
   type Press_Sensors is array (Oct.Sys_1_Sys_2) of
                            Pressure_Sensor_Class.Object;

-- ----------------
-- Message Pointers
-- ----------------

-- Outputs

   Cb_1021_001_Load_Msg_Id : Message.Out_Msg;
   Cb_1021_001_Load_Msg    : Elec_Sys_Intfc_Defs.Elec_Load_Msg_Ptrs;

   Cb_1021_002_Load_Msg_Id : Message.Out_Msg;
   Cb_1021_002_Load_Msg    : Elec_Sys_Intfc_Defs.Elec_Load_Msg_Ptrs;

   Cb_1021_003_Load_Msg_Id : Message.Out_Msg;
   Cb_1021_003_Load_Msg    : Elec_Sys_Intfc_Defs.Elec_Load_Msg_Ptrs;

   Cb_1021_004_Load_Msg_Id : Message.Out_Msg;
   Cb_1021_004_Load_Msg    : Elec_Sys_Intfc_Defs.Elec_Load_Msg_Ptrs;

   Cb_1022_001_Load_Msg_Id : Message.Out_Msg;
   Cb_1022_001_Load_Msg    : Elec_Sys_Intfc_Defs.Elec_Load_Msg_Ptrs;

   Cb_1022_002_Load_Msg_Id : Message.Out_Msg;
   Cb_1022_002_Load_Msg    : Elec_Sys_Intfc_Defs.Elec_Load_Msg_Ptrs;

   Cb_1023_001_Load_Msg_Id : Message.Out_Msg;
   Cb_1023_001_Load_Msg    : Elec_Sys_Intfc_Defs.Elec_Load_Msg_Ptrs;

   Cb_1023_002_Load_Msg_Id : Message.Out_Msg;
   Cb_1023_002_Load_Msg    : Elec_Sys_Intfc_Defs.Elec_Load_Msg_Ptrs;

   Aural_Cue_Msg_Id : Message.Out_Msg;
   Aural_Cue_Msg    : Hyd_Sys_Intfc_Defs.Aural_Cue_Msg_Ptrs;

   Sys_1_Press_Msg_Id : Message.Out_Msg;
   Sys_1_Press_Msg    : Hyd_Sys_Intfc_Defs.Hyd_Sys_Press_Msg_Ptrs;

   Sys_2_Press_Msg_Id : Message.Out_Msg;
   Sys_2_Press_Msg    : Hyd_Sys_Intfc_Defs.Hyd_Sys_Press_Msg_Ptrs;

   Sys_1_Status_Msg_Id : Message.Out_Msg;
   Sys_1_Status_Msg    : Hyd_Sys_Intfc_Defs.Status_Msg_Ptrs;

   Sys_2_Status_Msg_Id : Message.Out_Msg;
   Sys_2_Status_Msg    : Hyd_Sys_Intfc_Defs.Status_Msg_Ptrs;

   Hyd_Sys_Indicator_Msg_Id : Message.Out_Msg;
   Hyd_Sys_Indicator_Msg    : Hyd_Sys_Intfc_Defs.Hyd_Sys_Indicator_Msg_Ptrs;

-- Inputs

   Sys_1_Flow_Msg_Id : Message.In_Msg;
   Sys_1_Flow_Msg    : Hyd_Sys_Intfc_Defs.Flow_Msg_Ptrs;

   Sys_2_Flow_Msg_Id : Message.In_Msg;
   Sys_2_Flow_Msg    : Hyd_Sys_Intfc_Defs.Flow_Msg_Ptrs;

   Elec_Pwr_Msg_Id : Message.In_Msg;
   Elec_Pwr_Msg    : Elec_Sys_Intfc_Defs.Elec_Power_Msg_Ptrs;

   Sys_1_Motor_Cmd_Msg_Id : Message.In_Msg;
   Sys_1_Motor_Cmd_Msg    : Hyd_Control_Panel_Intfc_Defs.Motor_Cmd_Msg_Ptrs;

   Sys_2_Motor_Cmd_Msg_Id : Message.In_Msg;
   Sys_2_Motor_Cmd_Msg    : Hyd_Control_Panel_Intfc_Defs.Motor_Cmd_Msg_Ptrs;

   Sys_1_Valve_Cmd_Msg_Id : Message.In_Msg;
   Sys_1_Valve_Cmd_Msg    : Hyd_Control_Panel_Intfc_Defs.Valve_Cmd_Msg_Ptrs;
```

Types used internally only by this partition

Declaration of the partition's external input/output messages

```
   Sys_2_Valve_Cmd_Msg_Id : Message.In_Msg;
   Sys_2_Valve_Cmd_Msg    : Hyd_Control_Panel_Intfc_Defs.Valve_Cmd_Msg_Ptrs;

-- ---------------------------------
-- Internal Partition Class Instances
-- ---------------------------------

   Accumulator  : Accumulators;
   Dist_Sys     : Distribution_Systems;
   Drive_Unit   : Drive_Units;
   Iso_Vlv      : Isolation_Valves;
   Press_Sensor : Press_Sensors;
   Pump         : Pumps;
   Qty_Sensor   : Quantity_Sensor_Class.Object;
   Reservoir    : Hyd_Reservoir_Class.Object;

-- ----------------------
-- Internal Partition Data
-- ----------------------

   Delt_Time      : Seu.Seconds;
   Elap_Time      : Seu.Seconds      := 0.0;
   Hyd_Sys_Mbox   : Mailbox.Mailboxes;
   Partition_Name : constant String := "Hydraulic_System";
   Pid            : Natural          := 42;
   Stabilized     : Boolean          := False;

   Motor_Power       : Volt_A   := (others => 0.0);
   Motor_Relay_Power : On_Off_A := (others => Set.Off);
   Iso_Valve_Power   : Volt_A   := (others => 0.0);

   Motor_Cmd     : On_Off_A := (others => Set.Off);
   Vlv_Close_Cmd : On_Off_A := (others => Set.Off);
   Vlv_Open_Cmd  : On_Off_A := (others => Set.Off);

   Motor_Speed : Rpm_A := (others => 0.0);

   Motor_Status : On_Off_A := (others => Set.Off);
   Pump_Status  : On_Off_A := (others => Set.Off);

   Tot_Return_Flow : Gps_A := (others => 0.0);
   Tot_Press_Flow  : Gps_A := (others => 0.0);

   Num_Mbox_Empty_Exceptions : Natural := 0;

-- vvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvvv

   procedure Initialize_Outputs is separate;
   procedure Report_Symbols (Parent_Name : in String) is separate;

   procedure Initialize_Model is separate;
   procedure Process_Mailbox is separate;
   procedure Register_Io is separate;
   procedure Update_Inputs is separate;
   procedure Update_Supply_Components is separate;
   procedure Update_Press_Components is separate;
   procedure Update_Outputs is separate;
   procedure Update_Hydraulic_System is separate;

   procedure Set_Up;
   procedure Create_Data;
   procedure Self_Init;
   procedure System_Init;
   procedure Run;
   procedure Hold;
   procedure Term;

   package Thread_Exec is new Generic_Model.Periodic
                              (Name => Partition_Name,
                               Rate => Generic_Model.P10hz,
                               Execute_Set_Up_Model => Set_Up,
                               Execute_Create_Data_Model => Create_Data,
                               Execute_Self_Init_Model => Self_Init,
                               Execute_System_Init_Model => System_Init,
                               Execute_Run_Model => Run,
                               Execute_Freeze_Model => Run,
```

Instances of classes – the model's (partition's) state.

Temporary values used to transform class values to external interfaces.

Partition mode routines.

SVM thread e… instantiation.

```
                                    Execute_Hold_Model => Hold,
                                    Execute_Terminate_Model => Term);

   procedure Set_Up is separate;
   procedure Create_Data is separate;
   procedure Self_Init is separate;
   procedure System_Init is separate;
   procedure Run is separate;
   procedure Hold is separate;
   procedure Term is separate;

end Hydraulic_System_Partition;
```

Ada Unit 50 Hydraulic_System_Partition.Create_Data Separate Procedure

```
separate (Hydraulic_System_Partition)
procedure Create_Data is
begin
-- ---------------------------------------
-- For each message, CREATE_MSG is required
-- ---------------------------------------
-- Create each many-to-one output message
   Message.Many_To_One.Create_Msg (Out_Msg_Id => Cb_1021_001_Load_Msg_Id);
   Message.Many_To_One.Create_Msg (Out_Msg_Id => Cb_1021_002_Load_Msg_Id);
   Message.Many_To_One.Create_Msg (Out_Msg_Id => Cb_1021_003_Load_Msg_Id);
   Message.Many_To_One.Create_Msg (Out_Msg_Id => Cb_1021_004_Load_Msg_Id);
   Message.Many_To_One.Create_Msg (Out_Msg_Id => Cb_1022_001_Load_Msg_Id);
   Message.Many_To_One.Create_Msg (Out_Msg_Id => Cb_1022_002_Load_Msg_Id);
   Message.Many_To_One.Create_Msg (Out_Msg_Id => Cb_1023_001_Load_Msg_Id);
   Message.Many_To_One.Create_Msg (Out_Msg_Id => Cb_1023_002_Load_Msg_Id);
-- Create each one-to-many output message
   Message.One_To_Many.Create_Msg (Out_Msg_Id => Aural_Cue_Msg_Id);
   Message.One_To_Many.Create_Msg (Out_Msg_Id => Sys_1_Press_Msg_Id);
   Message.One_To_Many.Create_Msg (Out_Msg_Id => Sys_2_Press_Msg_Id);
   Message.One_To_Many.Create_Msg (Out_Msg_Id => Sys_1_Status_Msg_Id);
   Message.One_To_Many.Create_Msg (Out_Msg_Id => Sys_2_Status_Msg_Id);
   Message.One_To_Many.Create_Msg (Out_Msg_Id => Hyd_Sys_Indicator_Msg_Id);
-- Initialize first output messages sent to other partitions
   Initialize_Outputs;
-- Create each many_to_one input message
   Message.Many_To_One.Create_Msg (In_Msg_Id => Sys_1_Flow_Msg_Id);
   Message.Many_To_One.Create_Msg (In_Msg_Id => Sys_2_Flow_Msg_Id);
-- Create each one-to-many input message
   Message.One_To_Many.Create_Msg (In_Msg_Id => Elec_Pwr_Msg_Id);
   Message.One_To_Many.Create_Msg (In_Msg_Id => Sys_1_Motor_Cmd_Msg_Id);
   Message.One_To_Many.Create_Msg (In_Msg_Id => Sys_2_Motor_Cmd_Msg_Id);
   Message.One_To_Many.Create_Msg (In_Msg_Id => Sys_1_Valve_Cmd_Msg_Id);
   Message.One_To_Many.Create_Msg (In_Msg_Id => Sys_2_Valve_Cmd_Msg_Id);
end Create_Data;
-------------------------------------------------------------------
--|  Abstract: This procedure is an SVM mode routine that creates the
```

```
--              input and output messages of the Hydraulic System
--              Partition and calls Initialize_Outputs.
--
--    Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 51 Hydraulic_System_Partition.Hold Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Hold is
begin

-- In Hold, suspend normal processing.
-- Other special actions (ie Step_Ahead) would be performed here.

   null;

end Hold;

-------------------------------------------------------------------------
--|  Abstract: This procedure is an SVM mode routine that is
--|            periodically called while the simulation in 'hold'
--|            mode.
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 52 Hydraulic_System_Partition.Initialize_Model Separate Procedure

```
separate (Hydraulic_System_Partition)
procedure Initialize_Model is

begin

-- Initialize partition data and all objects such that the partition
-- is in an initial conditions state.  The details are left out in
-- this example.

   null;

end Initialize_Model;

-------------------------------------------------------------------------
--|  Abstract: This procedure initializes the Hydraulic System
--|            including all of its components and partition data.
--|            This procedure is called by Set_Up and also by
--|            Self_Init during a full initialization (Full_IC =
--|            True).
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 53 Hydraulic_System_Partition.Initialize_Outputs Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Initialize_Outputs is

begin

-- This routine sets the output messages to a default value during
-- initializations

   Cb_1021_001_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1021_001;
   Cb_1021_002_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1021_002;
   Cb_1021_003_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1021_003;
   Cb_1021_004_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1021_004;
   Cb_1022_001_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1022_001;
   Cb_1022_002_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1022_002;
   Cb_1023_001_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1023_001;
   Cb_1023_002_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1023_002;

   Cb_1021_001_Load_Msg.Load := 0.0;
   Cb_1021_002_Load_Msg.Load := 0.0;
```

```
   Cb_1021_003_Load_Msg.Load := 0.0;
   Cb_1021_004_Load_Msg.Load := 0.0;
   Cb_1022_001_Load_Msg.Load := 0.0;
   Cb_1022_002_Load_Msg.Load := 0.0;
   Cb_1023_001_Load_Msg.Load := 0.0;
   Cb_1023_002_Load_Msg.Load := 0.0;

   Aural_Cue_Msg.Pump_Noise_Sys_1  := Set.Off;
   Aural_Cue_Msg.Pump_Noise_Sys_2  := Set.Off;
   Aural_Cue_Msg.Motor_Noise_Sys_1 := Set.Off;
   Aural_Cue_Msg.Motor_Noise_Sys_2 := Set.Off;

   Sys_1_Press_Msg.Press := 0.0;
   Sys_2_Press_Msg.Press := 0.0;

   Sys_1_Status_Msg.Motor_Status                 := Set.Off;
   Sys_1_Status_Msg.Pump_Status                  := Set.Off;
   Sys_1_Status_Msg.Valve_Sensed_Not_Full_Open   := True;
   Sys_1_Status_Msg.Valve_Sensed_Not_Full_Closed := False;
   Sys_2_Status_Msg.Motor_Status                 := Set.Off;
   Sys_2_Status_Msg.Pump_Status                  := Set.Off;
   Sys_2_Status_Msg.Valve_Sensed_Not_Full_Open   := True;
   Sys_2_Status_Msg.Valve_Sensed_Not_Full_Open   := True;

   Hyd_Sys_Indicator_Msg.Sys_1_Press_Indicated := 0.0;
   Hyd_Sys_Indicator_Msg.Sys_2_Press_Indicated := 0.0;
   Hyd_Sys_Indicator_Msg.Sys_Qty_Indicated     := 0.0;

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1021_001_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1021_002_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1021_003_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1021_004_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1022_001_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1022_002_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1023_001_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1023_002_Load_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Aural_Cue_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Sys_1_Press_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Sys_2_Press_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Sys_1_Status_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Sys_2_Status_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Hyd_Sys_Indicator_Msg_Id);

end Initialize_Outputs;

------------------------------------------------------------------------
--|  Abstract: This procedure initializes all Hydraulic System
--|            Partition output messages and sends them out once.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Set output messages

send output messages

Ada Unit 54 Hydraulic_System_Partition.Process_Mailbox Separate Procedure

```
with Mail_Msg_Types;
with Enter_Mailbox;
with Malfunction_Mailbox;
with Safestore_Mailbox;

separate (Hydraulic_System_Partition)

procedure Process_Mailbox is

-- The following are renames from the Hydraulic_System_Partition body:

-- package Set renames Std_Eng_Types;
-- package Seu renames Std_Eng_Units;
-- package Oct renames Orvc_Common_Types;
```

```
   package Hyd_Sys_Dis renames Hydraulic_System_Defs;

   package Enter       renames Enter_Mailbox;
   package Malfunction renames Malfunction_Mailbox;

-- The following provides visability to the Dis Term ID equality operator:

   function "=" (Left, Right : Dis.Term_Id) return Boolean renames Dis."=";

-- The following provides visability to the Dis Malf ID equality operator:

   function "=" (Left, Right : Dis.Malfunction_Id) return Boolean
      renames Dis."=";

   Msg_Type : Mailbox.Msg_Types;

   Enter_Msg : Enter.Enter_Msg;
   Malf_Msg  : Malfunction_Mailbox.Malfunction_Msg;
   Datastore_Msg  : Mega_Mailbox.Mega_Msg;

   Enter_Id : Dis.Term_Id;
   Malf_Id  : Dis.Malfunction_Id;

   Apply : Boolean;

   Invalid_Msg_Id      : exception;
   Invalid_Enter_Term  : exception;
   Invalid_Malfunction : exception;

   function Sys_Id is new Malfunction_Mailbox.Selector (Oct.Sys_1_Sys_2);

begin

   for I in 1 .. Mailbox.Num_Mail_Msgs (Mailbox_Id => Hyd_Sys_Mbox) loop

--    Get the type of the next message

      Msg_Type := Mailbox.Get_Next_Msg_Type(Hyd_Sys_Mbox);

--    Process the mailbox message

      case Msg_Type is

         when Mailbox.Enter =>

--          Process the IOS Enter message

            Mailbox.Get_Enter_Msg(Enter_MSg => Enter_Msg,
                                  My_Mailbox_Id => Hyd_Sys_Mbox);

            Enter_Id := Enter.Id (Msg => Enter_Msg);

            if Enter_Id = Hyd_Sys_Dis.Fluid_Level then

--             Request reservoir quantity change. Pass in new quantity.

               Hyd_Reservoir_Class.Request_State_Change
                 (Instance => Reservoir,
                  Command  => Hyd_Reservoir_Class.Set_Qty,
                  Quantity => Seu.Gallons (Enter.Value_R6 (Msg => Enter_Msg)));

            elsif Enter_Id = Hyd_Sys_Dis.Flow_Pump_1 then

--             Request pump 1 flow rate change. Pass in new rate.

               Hydraulic_Pump_Class.Request_State_Change
                 (Instance => Pump (Oct.Sys_1),
                  Command  => Hydraulic_Pump_Class.Modify_Flow_Rate,
                  Apply    => True,
                  Bias     => Seu.Non_Dimensional
                                (Enter.Value_R6 (Msg => Enter_Msg)));

            elsif Enter_Id = Hyd_Sys_Dis.Flow_Pump_2 then

--             Request pump 2 flow rate change. Pass in new rate.

               Hydraulic_Pump_Class.Request_State_Change
                 (Instance => Pump (Oct.Sys_2),
                  Command  => Hydraulic_Pump_Class.Modify_Flow_Rate,
                  Apply    => True,
                  Bias     => Seu.Non_Dimensional
                                (Enter.Value_R6 (Msg => Enter_Msg)));
```

call object to affect change.

Request rate change

```
            else
--             no other enter values are expected
               raise Invalid_Enter_Term;
            end if;
         when Mailbox.Malfunction =>
--          Process the malfunction message
            Mailbox.Get_Malfunction_Msg(Malfunction_Msg => Malf_Msg,
                                        My_Mailbox_Id => Hyd_Sys_Mbox);
--          Get the DIS malfunction identifier
            Malf_Id := Malfunction.Id (Msg => Malf_Msg);
--          Get the state of the malfunction (On or Off) and convert
--          to a boolean to pass to Request_State_Change procedures.
            Apply := Malfunction.State (Msg => Malf_Msg) = Set.On;
            if Malf_Id = Hyd_Sys_Dis.Pump_No_Flow then
--             Process the hydraulic pump malfunction.  Pass in which
--             pump (#1 or #2).
               Hydraulic_Pump_Class.Request_State_Change
                 (Instance => Pump (Sys_Id (Msg => Malf_Msg)),
                  Command  => Hydraulic_Pump_Class.Pump_Fail,
                  Apply    => Apply);
            elsif Malf_Id = Hyd_Sys_Dis.Press_Comp_Fail then
--             Process the pressure compensator malfunction.  Pass in which
--             compensator (#1 or #2).
               Hydraulic_Pump_Class.Request_State_Change
                 (Instance => Pump (Sys_Id (Msg => Malf_Msg)),
                  Command  => Hydraulic_Pump_Class.Compensator_Fail,
                  Apply    => Apply,
                  Bias     => 0.0,
                  Scale    => 1.0);
            elsif Malf_Id = Hyd_Sys_Dis.Iso_Valve_Freeze then
--             Request the valve to freeze.  Pass in which
--             valve (#1 or #2).
               Valve_Class.Request_State_Change
                 (Instance => Iso_Vlv (Sys_Id (Msg => Malf_Msg)),
                  Command  => Valve_Class.Freeze_Valve,
                  Apply    => Apply);
            elsif Malf_Id = Hyd_Sys_Dis.Pressure_Sensor_Fail then
--             Request the pressure sensor to read incorrectly.  Pass
--             in which sensor (#1 or #2), the error factor (scale) and
--             the offset (bias).
               Pressure_Sensor_Class.Request_State_Change
                 (Instance => Press_Sensor (Sys_Id (Msg => Malf_Msg)),
                  Command  => Pressure_Sensor_Class.Sensor_Incorrect,
                  Apply    => Apply,
                  Scale    => Seu.Non_Dimensional
                                (Malfunction.Scale (Msg => Malf_Msg)),
                  Bias     => Seu.Psi (Malfunction.Bias (Msg => Malf_Msg)));
            elsif Malf_Id = Hyd_Sys_Dis.Motor_Zero_Rpm then
--             Process the motor fail malfunction.  Pass in which motor
--             (#1 or #2).
               Drive_Unit_Class.Request_State_Change
                 (Instance => Drive_Unit (Sys_Id (Msg => Malf_Msg)),
                  Command  => Drive_Unit_Class.Motor_Fail,
                  Apply    => Apply);
            elsif Malf_Id = Hyd_Sys_Dis.Dist_Sys_Leak then
```

```
            Process the distribution system leak malfunction.  Pass
            in which distribution system (#1 or #2) and the leak
            rate.  Notice that the leak rate is passed in as the
            malfunction bias.

            Distribution_System_Class.Request_State_Change
               (Instance   => Dist_Sys (Sys_Id (Msg => Malf_Msg)),
               Command    => Distribution_System_Class.Set_Leak,
               Apply      => Apply,
               Leak_Rate  => Seu.Gallons_Per_Second
                             (Malfunction.Bias (Msg => Malf_Msg)));

         else
            raise exception, no other malfunctions are expected.
            raise Invalid_Malfunction;
         end if;
      when Mailbox.Return_To_Datastore =>
         Process the return to datastore message
         Mailbox.Get_Mega_Msg(Mega_MSg => Datastore_Msg,
                              My_Mailbox_Id => Hyd_Sys_Mbox);

         Get initiazation data from the mailbox and populate the model.
         The details are left out in this example.

      when others =>
         raise exception, no other message types are expected.
         raise Invalid_Msg_Id;

   end case;
end loop;
exception
   when Mailbox.Mailbox_Empty =>
      Num_Mbox_Empty_Exceptions := Num_Mbox_Empty_Exceptions + 1;

   when others =>
      null;  -- allow propagation of all other exceptions
end Process_Mailbox;
-------------------------------------------------------------------
--|  Abstract: This procedure receives and processes the mailbox
--|            messages.  The Hydraulic System Partition can receive
--|            ICS enter or malfunction mailbox messages.  The
--|            partition does not send any mailbox messages.
--|
--|  Warnings: The distribution system leak malfunction leak rate is
--|            received via the the bias attribute of the of the
--|            malfunction message.
--|
--|            The following exceptions are propagated beyond the
--|            scope of this procedure:
--|                 Invalid_Msg_Id
--|                 Invalid_Enter_Term
--|                 Invalid_Malfunction
-------------------------------------------------------------------
```

Ada Unit 55 Hydraulic_System_Partition.Register_Io Separate Procedure

```
with Orvc_Defs;
separate (Hydraulic_System_Partition)
procedure Register_Io is
   package Elec_Sys_If  renames Elec_Sys_Intfc_Defs;
   package Hyd_Sys_If   renames Hyd_Sys_Intfc_Defs;
   package Cntrl_Pnl_If renames Hyd_Control_Panel_Intfc_Defs;
```

```
   Receive_Queue_Size : constant := 50;

begin
-- Register Partition Mailbox

   Mailbox.Register_Mailbox (My_Partition_Prefix => Orvc_Defs.Hydraulic_System,
                             My_Mailbox_Id => Hyd_Sys_Mbox);

-- Register the messages sent from the Hydraulic System to
-- other partitions (output messages).

   Message.Many_To_One.Register_To_Send_Msg
     (Out_Msg_Id            => Cb_1021_001_Load_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Electrical_System,
      Msg_Dis_Id            => Elec_Sys_If.Elec_Load_Msg_Id,
      Msg_Ptr_Addr          => Cb_1021_001_Load_Msg'Address);

   Message.Many_To_One.Register_To_Send_Msg
     (Out_Msg_Id            => Cb_1021_002_Load_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Electrical_System,
      Msg_Dis_Id            => Elec_Sys_If.Elec_Load_Msg_Id,
      Msg_Ptr_Addr          => Cb_1021_002_Load_Msg'Address);

   Message.Many_To_One.Register_To_Send_Msg
     (Out_Msg_Id            => Cb_1021_003_Load_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Electrical_System,
      Msg_Dis_Id            => Elec_Sys_If.Elec_Load_Msg_Id,
      Msg_Ptr_Addr          => Cb_1021_003_Load_Msg'Address);

   Message.Many_To_One.Register_To_Send_Msg
     (Out_Msg_Id            => Cb_1021_004_Load_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Electrical_System,
      Msg_Dis_Id            => Elec_Sys_If.Elec_Load_Msg_Id,
      Msg_Ptr_Addr          => Cb_1021_004_Load_Msg'Address);

   Message.Many_To_One.Register_To_Send_Msg
     (Out_Msg_Id            => Cb_1022_001_Load_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Electrical_System,
      Msg_Dis_Id            => Elec_Sys_If.Elec_Load_Msg_Id,
      Msg_Ptr_Addr          => Cb_1022_001_Load_Msg'Address);

   Message.Many_To_One.Register_To_Send_Msg
     (Out_Msg_Id            => Cb_1022_002_Load_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Electrical_System,
      Msg_Dis_Id            => Elec_Sys_If.Elec_Load_Msg_Id,
      Msg_Ptr_Addr          => Cb_1022_002_Load_Msg'Address);

   Message.Many_To_One.Register_To_Send_Msg
     (Out_Msg_Id            => Cb_1023_001_Load_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Electrical_System,
      Msg_Dis_Id            => Elec_Sys_If.Elec_Load_Msg_Id,
      Msg_Ptr_Addr          => Cb_1023_001_Load_Msg'Address);

   Message.Many_To_One.Register_To_Send_Msg
     (Out_Msg_Id            => Cb_1023_002_Load_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Electrical_System,
      Msg_Dis_Id            => Elec_Sys_If.Elec_Load_Msg_Id,
      Msg_Ptr_Addr          => Cb_1023_002_Load_Msg'Address);

   Message.One_To_Many.Register_To_Send_Msg
     (Out_Msg_Id            => Aural_Cue_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Aural_Cue,
      Msg_Dis_Id            => Hyd_Sys_If.Aural_Cue_Msg_Id,
      Msg_Bit_Size          => Hyd_Sys_If.Aural_Cue_Msg_Size,
      Execution_Rate        => Thread_Exec.Rate_Of_Execution,
      Msg_Ptr_Addr          => Aural_Cue_Msg'Address);

   Message.One_To_Many.Register_To_Send_Msg
     (Out_Msg_Id            => Sys_1_Press_Msg_Id,
      Partition_Prefix      => Orvc_Defs.Hydraulic_Systems,
      Msg_Dis_Id            => Hyd_Sys_If.Sys_1_Press_Msg_Id,
      Msg_Bit_Size          => Hyd_Sys_If.Hyd_Sys_Press_Msg_Size,
      Execution_Rate        => Thread_Exec.Rate_Of_Execution,
      Msg_Ptr_Addr          => Sys_1_Press_Msg'Address);
```

Output messages

```
Message.One_To_Many.Register_To_Send_Msg
   (Out_Msg_Id              => Sys_2_Press_Msg_Id,
    Partition_Prefix        => Orvc_Defs.Hydraulic_Systems,
    Msg_Dis_Id              => Hyd_Sys_If.Sys_2_Press_Msg_Id,
    Msg_Bit_Size            => Hyd_Sys_If.Hyd_Sys_Press_Msg_Size,
    Execution_Rate          => Thread_Exec.Rate_Of_Execution,
    Msg_Ptr_Addr            => Sys_2_Press_Msg'Address);

 Message.One_To_Many.Register_To_Send_Msg
   (Out_Msg_Id              => Sys_1_Status_Msg_Id,
    Partition_Prefix        => Orvc_Defs.Hydraulic_System,
    Msg_Dis_Id              => Hyd_Sys_If.Sys_1_Status_Msg_Id,
    Msg_Bit_Size            => Hyd_Sys_If.Status_Msg_Size,
    Execution_Rate          => Thread_Exec.Rate_Of_Execution,
    Msg_Ptr_Addr            => Sys_1_Status_Msg'Address);

  Message.One_To_Many.Register_To_Send_Msg
    (Out_Msg_Id             => Sys_2_Status_Msg_Id,
     Partition_Prefix       => Orvc_Defs.Hydraulic_System,
     Msg_Dis_Id             => Hyd_Sys_If.Sys_2_Status_Msg_Id,
     Msg_Bit_Size           => Hyd_Sys_If.Status_Msg_Size,
     Execution_Rate         => Thread_Exec.Rate_Of_Execution,
     Msg_Ptr_Addr           => Sys_2_Status_Msg'Address);

  Message.One_To_Many.Register_To_Send_Msg
    (Out_Msg_Id             => Hyd_Sys_Indicator_Msg_Id,
     Partition_Prefix       => Orvc_Defs.Hydraulic_System,
     Msg_Dis_Id             => Hyd_Sys_If.Hyd_Sys_Indicator_Msg_Id,
     Msg_Bit_Size           => Hyd_Sys_If.Hyd_Sys_Indicator_Msg_Size,
     Execution_Rate         => Thread_Exec.Rate_Of_Execution,
     Msg_Ptr_Addr           => Hyd_Sys_Indicator_Msg'Address);

-- Reigster the messages sent to the Hydraulic System from
-- other partitions (input messages).

    Message.Many_To_One.Register_To_Recv_Msg
      (In_Msg_Id            => Sys_1_Flow_Msg_Id,
       Partition_Prefix     => Orvc_Defs.Hydraulic_System,
       Msg_Dis_Id           => Hyd_Sys_If.Sys_1_Flow_Msg_Id,
       Msg_Bit_Size         => Hyd_Sys_If.Flow_Msg_Size,
       Queue_Size           => Receive_Queue_Size,
       Msg_Ptr_Addr         => Sys_1_Flow_Msg'Address);

    Message.Many_To_One.Register_To_Recv_Msg
      (In_Msg_Id            => Sys_2_Flow_Msg_Id,
       Partition_Prefix     => Orvc_Defs.Hydraulic_System,
       Msg_Dis_Id           => Hyd_Sys_If.Sys_2_Flow_Msg_Id,
       Msg_Bit_Size         => Hyd_Sys_If.Flow_Msg_Size,
       Queue_Size           => Receive_Queue_Size,
       Msg_Ptr_Addr         => Sys_2_Flow_Msg'Address);

    Message.One_To_Many.Register_To_Recv_Msg
      (In_Msg_Id            => Elec_Pwr_Msg_Id,
       Partition_Prefix     => Orvc_Defs.Electrical_System,
       Msg_Dis_Id           => Elec_Sys_If.Elec_Power_Msg_Id,
       Execution_Rate       => Receive_Hertz_Rate,
       Msg_Ptr_Addr         => Elec_Pwr_Msg'Address);

    Message.One_To_Many.Register_To_Recv_Msg
      (In_Msg_Id            => Sys_1_Motor_Cmd_Msg_Id,
       Partition_Prefix     => Orvc_Defs.Hydraulic_Control_Panel,
       Msg_Dis_Id           => Cntrl_Pnl_If.Sys_1_Motor_Cmd_Msg_Id,
       Execution_Rate       => Receive_Hertz_Rate,
       Msg_Ptr_Addr         => Sys_1_Motor_Cmd_Msg'Address);

    Message.One_To_Many.Register_To_Recv_Msg
      (In_Msg_Id            => Sys_2_Motor_Cmd_Msg_Id,
       Partition_Prefix     => Orvc_Defs.Hydraulic_Control_Panel,
       Msg_Dis_Id           => Cntrl_Pnl_If.Sys_2_Motor_Cmd_Msg_Id,
       Execution_Rate       => Receive_Hertz_Rate,
       Msg_Ptr_Addr         => Sys_2_Motor_Cmd_Msg'Address);

    Message.One_To_Many.Register_To_Recv_Msg
      (In_Msg_Id            => Sys_1_Valve_Cmd_Msg_Id,
```

Input messages

```
      Partition_Prefix     => Orvc_Defs.Hydraulic_Control_Panel,
      Msg_Dis_Id           => Cntrl_Pnl_If.Sys_1_Valve_Cmd_Msg_Id,
      Execution_Rate       => Receive_Hertz_Rate,
      Msg_Ptr_Addr         => Sys_1_Valve_Cmd_Msg'Address);

   Message.One_To_Many.Register_To_Recv_Msg
      In_Msg_Id            => Sys_2_Valve_Cmd_Msg_Id,
      Partition_Prefix     => Orvc_Defs.Hydraulic_Control_Panel,
      Msg_Dis_Id           => Cntrl_Pnl_If.Sys_2_Valve_Cmd_Msg_Id,
      Execution_Rate       => Receive_Hertz_Rate,
      Msg_Ptr_Addr         => Sys_2_Valve_Cmd_Msg'Address);

end Register_Io;

-------------------------------------------------------------------------
--|  Abstract: This procedure registers the input and output messages
--|            of the Hydraulic System Partition.
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 56 Hydraulic_System_Partition.Report_Symbols Separate Procedure

```
with Symbols;

separate (Hydraulic_System_Partition)

procedure Report_Symbols (Parent_Name : in String) is

begin

   -- register all complex variables needed by IOS that were created in this partition

   null -- this partition has only simple variables

end Report_Symbols;

-------------------------------------------------------------------------
--|  Abstract: This procedure reports the motor speed and motor status
--|            (as defined in the Hydraulic System Partition body) to
--|            the symbol map.
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 57 Hydraulic_System_Partition.Run Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Run is
begin

-- This routine provides normal partition updates.

   Delta_Time := Thread_Exec.Delta_Time;

   Update_Hydraulic_System;

end Run;

-------------------------------------------------------------------------
--|  Abstract: This procedure is an SVM mode routine that performs one
--|            iteration of the Hydraulic System Partition.  This mode
--|            routine is periodically called while the simulation is
--|            in 'run' mode.
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 58 Hydraulic_System_Partition.Self_Init Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Self_Init is

begin
```

```
   if Thread_Exec.A_Full_Ic_Is_Required then
      Initialize_Model;
   end if;
   Process_Mailbox;
   Initialize_Outputs;
   Stabilized := False;
   Elapsed_Time := 0.0;
   Thread_Exec.Ready_To_Transition;
end Self_Init;
-------------------------------------------------------------------
--|  Abstract: This procedure is an SVM mode routine that readies the
--|            Hydraulic System Partition for system intialization.
--|
--|            If the input parameter Full_Ic is true,
--|            Initialize_Model is called to initialize the partition
--|            data and all of its components.
--|
--|            The stabilized flag is set to False to allow
--|            model stabilization in the System_Init procedure.
--|
--|            Upon completion, the model nofifies the executive that
--|            it is ready to transition.
--|
--|            NOTE: This is a one pass initialization, no iterating!
--|
--|  Warnings: None.
-------------------------------------------------------------------
```

Ada Unit 59 Hydraulic_System_Partition.Set_Up Separate Procedure

```
separate (Hydraulic_System_Partition)
procedure Set_Up is

begin
   -- Create instances of classes.
      for I in 1 .. Oct.Sys_1_Sys_2 loop
         Accumulator_Class.Create
           (Instance      => Accumulator (Index),
            Parent_Name   => Partition_Name & ".accumulator(" &
                             Oct.Sys_1_Sys_2'Image (Index) & ")",
            Init_Press    => 4000.0,
            Min_Gas_Press => 1000.0,
            Min_Gas_Vol   => 1.0,
            Max_Gas_Vol   => 2.0,
            Min_Fluid_Vol => 2.0,
            Max_Fluid_Vol => 3.5);
         Distribution_System_Class.Create
           (Instance     => Dist_Sys (Index),
            Parent_Name => Partition_Name & ".dist_sys(" &
                           Oct.Sys_1_Sys_2'Image (Index) & ")",
            Press_Const => 0.15);
         Drive_Unit_Class.Create
              Instance          => Drive_Unit (Index),
              Parent_Name       => Partition_Name & ".drive_unit(" &
                                   Oct.Sys_1_Sys_2'Image (Index) & ")",
           Gearbox_Max_Torque => 250.0);
         Valve_Class.Create (Instance    => Iso_Vlv (Index),
                             Parent_Name => Partition_Name & ".iso_vlv(" &
                                            Oct.Sys_1_Sys_2'Image (Index) & ")");
         Pressure_Sensor_Class.Create
           (Instance     => Press_Sensor (Index),
```

```
         Parent_Name  => Partition_Name & ".press_sensor(" &
                         Oct.Sys_1_Sys_2'Image (Index) & ")",
         Nominal_Load => 0.67);

      Hydraulic_Pump_Class.Create
        (Instance    => Pump (Index),
         Parent_Name => Partition_Name & ".pump(" &
                        Oct.Sys_1_Sys_2'Image (Index) & ")");

   end loop;

   Quantity_Sensor_Class.Create (Instance     => Qty_Sensor,
                                 Parent_Name  => Partition_Name & ".qty_sensor",
                                 Nominal_Load => 0.5);

   Hyd_Reservoir_Class.Create (Instance    => Reservoir,
                               Parent_Name => Partition_Name & ".reservoir",
                               Init_Qty    => 5.25);

-- Initialize partition data and all objects.

   Initialize_Model;

-- Report Partition level symbols

   Report_Symbols (Parent_Name => Partition_Name);

-- Link variables to the DIS for reporting to IOS.

   -- Connect simple variables created locally by their address.

   Dis.Connect_Term (Term    => Hydraulic_System_Defs.Motor_1_On_Off,
                     Address => Motor_Speed(Oct.Sys_1));

   Dis.Connect_Term (Term    => Hydraulic_System_Defs.Motor_2_On_Off,
                     Address => Motor_Speed(Oct.Sys_2));

   Dis.Connect_Term (Term    => Hydraulic_System_Defs.Motor_1_Rpm,
                     Address => Motor_Status(Oct.Sys_1));

   Dis.Connect_Term (Term    => Hydraulic_System_Defs.Motor_2_Rpm,
                     Address => Motor_Status(Oct.Sys_2));

   -- Connect complex variables by their registered symbol name.

   Dis.Connect_Term (Term    => Hydraulic_System_Defs.Fluid_Level,
                     Symbol  => "Hydraulic_System.reservoir.quantity");

   Dis.Connect_Term
     (Term    => Hydraulic_System_Defs.Pressure_Sys_1,
      Symbol  => "Hydraulic_System.dist_sys(sys_1).system_pressure");

   Dis.Connect_Term
     (Term    => Hydraulic_System_Defs.Pressure_Sys_2,
      Symbol => "Hydraulic_System.dist_sys(sys_2).system_pressure");

   Dis.Connect_Term (Term    => Hydraulic_System_Defs.Flow_Pump_1,
                     Symbol  => "Hydraulic_System.pump(sys_1).flow_out");

   Dis.Connect_Term (Term    => Hydraulic_System_Defs.Flow_Pump_2,
                     Symbol  => "Hydraulic_System.pump(sys_2).flow_out");

   Dis.Connect_Term
     (Term    => Hydraulic_System_Defs.Iso_Valve_Sys_1,
      Symbol  => "Hydraulic_System.iso_valve(sys_1).position");

   Dis.Connect_Term
     (Term    => Hydraulic_System_Defs.Iso_Valve_Sys_2,
      Symbol  => "Hydraulic_System.iso_valve(sys_2).position");

-- Register input/output messages

   Register_Io;

-- Notify Thread_Exec that have completed setup

   Thread_Exec.Ready_To_Transition;

end Set_Up;

------------------------------------------------------------------------
--|  Abstract: This is an SVM mode procedure that calls the 'create'
```

```
--|            procedure for every instance of a class and 'connects'
--|            the specified class attributes to the corresponding DIS
--|            terms. After all objects are created, Initialize_Model
--|            is called to intialize objects and partition data.
--|            Register_Io is called to register the input and output
--|            messages of the Hydraulic Partition.  Upon completion,
--|            this procedure notifies the executive that the model
--|            is ready to transition.
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 60 Hydraulic_System_Partition.System_Init Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure System_Init is

begin

-- This routine is called after Self_Init is complete.
-- Partition is updated until stable, then reports in.

   Delta_Time := Thread_Exec.Delta_Time;

   if not Stabilized then

--    Update Hydraulic System for 5 seconds to allow components to
--    stabilize at new conditions before reporting ready to transition.

      if Elapsed_Time < 5.0 then

         Update_Hydraulic_System;

         Elapsed_Time := Elapsed_Time + Delta_Time;

      else

--    When stable, set flag and report in to executive.

         Stabilized := True;

         Thread_Exec.Ready_To_Transition;

      end if;

   else

--    Once stablized, continue normal updates

      Update_Hydraulic_System;

   end if;

end System_Init;

-------------------------------------------------------------------------
--|  Abstract: This is an SVM mode procedure that perform the
--|            Hydraulic System Partition system initialization by
--|            repeatedly cycling the model to stabilize it.
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 61 Hydraulic_System_Partition.Term Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Term is
begin

-- Actions required to terminate processes would be performed here.  This
-- could include deallocating devices and closing files.

   null;

end Term;

-------------------------------------------------------------------------
--|  Abstract: This procedure is an SVM mode routine that is called to
```

```
--              gracefully terminate the processing associated with the
--              Hydraulic System Partition.
--
--   Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 62 Hydraulic_System_Partition.Update_Hydraulic_System Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Update_Hydraulic_System is
begin
-- Process Mailbox commands

   Process_Mailbox;

-- Read input messages

   Update_Inputs;

-- Update Pressurization Components (reservoir, drive units, pumps, etc)

   Update_Press_Components;

-- Update Supply Components (valves, distribution plumbing, etc)

   Update_Supply_Components;

-- Set Output messages

   Update_Outputs;

end Update_Hydraulic_System;

-------------------------------------------------------------------------
--|  Abstract: This procedure performs the update of the Hydraulic
--|            System Partition
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Ada Unit 63 Hydraulic_System_Partition.Update_Inputs Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Update_Inputs is

begin
-- This routine gets the input messages

   Message.One_To_Many.Get (In_Msg_Id => Elec_Pwr_Msg_Id);

   Message.One_To_Many.Get (In_Msg_Id => Sys_1_Motor_Cmd_Msg_Id);

   Message.One_To_Many.Get (In_Msg_Id => Sys_2_Motor_Cmd_Msg_Id);

   Message.One_To_Many.Get (In_Msg_Id => Sys_1_Valve_Cmd_Msg_Id);

   Message.One_To_Many.Get (In_Msg_Id => Sys_2_Valve_Cmd_Msg_Id);

   -- Determine total flow rates consumed and returned by hyd components

   Tot_Press_Flow (Oct.Sys_1) := 0.0;
   Tot_Press_Flow (Oct.Sys_2) := 0.0;

   Tot_Return_Flow (Oct.Sys_1) := 0.0;
   Tot_Return_Flow (Oct.Sys_2) := 0.0;

   for I in 1 .. Message.Many_To_One.Number_Of_Msgs_To_Get
                   (In_Msg_Id => Sys_1_Flow_Msg_Id) loop

      Message.Many_To_One.Get (In_Msg_Id => Sys_1_Flow_Msg_Id);

      Tot_Press_Flow (Oct.Sys_1) :=
        Tot_Press_Flow (Oct.Sys_1) + Sys_1_Flow_Msg.Press_Flow;

      Tot_Return_Flow (Oct.Sys_1) :=
        Tot_Press_Flow (Oct.Sys_1) + Sys_1_Flow_Msg.Return_Flow;
```

```
   end loop;

   for I in 1 .. Message.Many_To_One.Number_Of_Msgs_To_Get
                    (In_Msg_Id => Sys_2_Flow_Msg_Id) loop

      Message.Many_To_One.Get (In_Msg_Id => Sys_2_Flow_Msg_Id);

      Tot_Press_Flow (Oct.Sys_2) :=
        Tot_Press_Flow (Oct.Sys_2) + Sys_2_Flow_Msg.Press_Flow;

      Tot_Return_Flow (Oct.Sys_2) :=
        Tot_Press_Flow (Oct.Sys_2) + Sys_2_Flow_Msg.Return_Flow;

   end loop;

end Update_Inputs;

------------------------------------------------------------------------
--|  Abstract: This procedure updates the inputs to the Hydraulic
--|            System Partition
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 64 Hydraulic_System_Partition.Update_Outputs Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Update_Outputs is

begin

-- Set Circuit breaker id for each load message

   Cb_1021_001_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1021_001;
   Cb_1021_002_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1021_002;
   Cb_1021_003_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1021_003;
   Cb_1021_004_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1021_004;
   Cb_1022_001_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1022_001;
   Cb_1022_002_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1022_002;
   Cb_1023_001_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1023_001;
   Cb_1023_002_Load_Msg.Cb := Elec_Sys_Intfc_Defs.Cb_1023_002;

-- Set elec load values for components.

   Cb_1021_001_Load_Msg.Load :=
     Drive_Unit_Class.Elec_Load (Drive_Unit (Oct.Sys_1));

   Cb_1021_002_Load_Msg.Load :=
     Drive_Unit_Class.Elec_Load (Drive_Unit (Oct.Sys_2));

   if Motor_Relay_Power (Oct.Sys_1) = Set.On then

      Cb_1021_003_Load_Msg.Load := 0.5;

   else

      Cb_1021_003_Load_Msg.Load := 0.0;

   end if;

   if Motor_Relay_Power (Oct.Sys_2) = Set.On then

      Cb_1021_004_Load_Msg.Load := 0.5;

   else

      Cb_1021_004_Load_Msg.Load := 0.0;

   end if;

   Cb_1022_001_Load_Msg.Load :=
     Valve_Class.Electrical_Load (Iso_Vlv (Oct.Sys_1));

   Cb_1022_002_Load_Msg.Load :=
     Valve_Class.Electrical_Load (Iso_Vlv (Oct.Sys_2));

   Cb_1023_001_Load_Msg.Load :=
     Pressure_Sensor_Class.Elec_Load (Press_Sensor (Oct.Sys_1)) +
       Pressure_Sensor_Class.Elec_Load (Press_Sensor (Oct.Sys_2));
```

```
   Cb_1023_002_Load_Msg.Load := Quantity_Sensor_Class.Elec_Load (Qty_Sensor);

-- Set system pressure (actual, not sensed) for use by hyd components

   Sys_1_Press_Msg.Press :=
     Distribution_System_Class.System_Pressure (Dist_Sys (Oct.Sys_1));
   Sys_2_Press_Msg.Press :=
     Distribution_System_Class.System_Pressure (Dist_Sys (Oct.Sys_2));

-- Set system pressure (sensed) for output to hydraulic control panel

   Hyd_Sys_Indicator_Msg.Sys_1_Press_Indicated :=
     Pressure_Sensor_Class.Sensed_Output (Press_Sensor (Oct.Sys_1));

   Hyd_Sys_Indicator_Msg.Sys_2_Press_Indicated :=
     Pressure_Sensor_Class.Sensed_Output (Press_Sensor (Oct.Sys_1));

-- Set sensed quantity value for output to hyd control panel

   Hyd_Sys_Indicator_Msg.Sys_Qty_Indicated :=
     Quantity_Sensor_Class.Sensed_Output (Qty_Sensor);

-- Set isolation valve position discretes for output to Hyd Control Panel

   Sys_1_Status_Msg.Valve_Sensed_Not_Full_Open :=
     not (Valve_Class.Full_Open (Iso_Vlv (Oct.Sys_1)));

   Sys_1_Status_Msg.Valve_Sensed_Not_Full_Closed :=
     (Valve_Class.Full_Closed (Iso_Vlv (Oct.Sys_1)));

   Sys_2_Status_Msg.Valve_Sensed_Not_Full_Open :=
     (Valve_Class.Full_Open (Iso_Vlv (Oct.Sys_2)));

   Sys_2_Status_Msg.Valve_Sensed_Not_Full_Closed :=
     (Valve_Class.Full_Closed (Iso_Vlv (Oct.Sys_2)));

-- Set motor and pump status discretes for output to hyd control panel

   Sys_1_Status_Msg.Motor_Status := Motor_Status (Oct.Sys_1);
   Sys_1_Status_Msg.Pump_Status  := Pump_Status (Oct.Sys_1);

   Sys_2_Status_Msg.Motor_Status := Motor_Status (Oct.Sys_2);
   Sys_2_Status_Msg.Pump_Status  := Pump_Status (Oct.Sys_2);

-- Set Discretes for output to aural cue (sound) system

   Aural_Cue_Msg.Pump_Noise_Sys_1  := Pump_Status (Oct.Sys_1);
   Aural_Cue_Msg.Pump_Noise_Sys_2  := Pump_Status (Oct.Sys_2);
   Aural_Cue_Msg.Motor_Noise_Sys_1 := Motor_Status (Oct.Sys_1);
   Aural_Cue_Msg.Motor_Noise_Sys_2 := Motor_Status (Oct.Sys_2);

-- Put messages

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1021_001_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1021_002_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1021_003_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1021_004_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1022_001_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1022_002_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1023_001_Load_Msg_Id);

   Message.Many_To_One.Put (Out_Msg_Id => Cb_1023_002_Load_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Aural_Cue_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Sys_1_Press_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Sys_2_Press_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Sys_1_Status_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Sys_2_Status_Msg_Id);

   Message.One_To_Many.Put (Out_Msg_Id => Hyd_Sys_Indicator_Msg_Id);

end Update_Outputs;

------------------------------------------------------------------------
--|  Abstract: This procedure updates the outputs of the Hydraulic
```

```
--              System Partition.
--
--   Warnings: None.
-------------------------------------------------------------------
```

Ada Unit 65 Hydraulic_System_Partition.Update_Press_Components Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Update_Press_Components is

begin
--   Update reservoir
   Hyd_Reservoir_Class.Update
     (Instance      => Reservoir,
      Delta_Time    => Delta_Time,
      Consumed_Rate => (Hydraulic_Pump_Class.Consumed_Flow (Pump (Oct.Sys_1)) +
                        Hydraulic_Pump_Class.Consumed_Flow (Pump (Oct.Sys_2))),
      Returned_Rate => (Tot_Return_Flow (Oct.Sys_1) +
                        Tot_Return_Flow (Oct.Sys_2)));

-- -- Update Quantity Sensor and return electrical load
   Quantity_Sensor_Class.Update
     (Instance     => Qty_Sensor,
      Power_Avail  =>
        (Elec_Pwr_Msg.Cb (Elec_Sys_Intfc_Defs.Cb_1023_002).Power = Set.On),
      Sensed_Input => Hyd_Reservoir_Class.Quantity (Reservoir));

-- Set Motor and Pump signals and power for components
   Motor_Cmd (Oct.Sys_1) := Sys_1_Motor_Cmd_Msg.Motor_Cmd;
   Motor_Cmd (Oct.Sys_2) := Sys_2_Motor_Cmd_Msg.Motor_Cmd;

   Motor_Power (Oct.Sys_1) :=
     Elec_Pwr_Msg.Cb (Elec_Sys_Intfc_Defs.Cb_1021_001).Voltage;
   Motor_Power (Oct.Sys_2) :=
     Elec_Pwr_Msg.Cb (Elec_Sys_Intfc_Defs.Cb_1021_002).Voltage;

   Motor_Relay_Power (Oct.Sys_1) := Elec_Pwr_Msg.Cb
                                      (Elec_Sys_Intfc_Defs.Cb_1021_003).Power;

   Motor_Relay_Power (Oct.Sys_2) := Elec_Pwr_Msg.Cb
                                      (Elec_Sys_Intfc_Defs.Cb_1021_004).Power;

   for Sys_Index in Oct.Sys_1_Sys_2 loop
   --   When relay power is not avail, or motor is commanded off, set motor power
   --   to off.  This replaces relay_1021_001  and relay_1021_002.
      if (Motor_Relay_Power (Sys_Index) /= Set.On) or
         (Motor_Cmd (Sys_Index) /= Set.On) then

         Motor_Power (Sys_Index) := 0.0;

      end if;
   --   Update Drive Unit and set motor status discrete
      Drive_Unit_Class.Update
        (Instance    => Drive_Unit (Sys_Index),
         Avail_Power => Motor_Power (Sys_Index),
         Delta_Time  => Delta_Time,
         Torque      => Hydraulic_Pump_Class.Torque (Pump (Sys_Index)));
      if Drive_Unit_Class.Motor_On (Drive_Unit (Sys_Index)) then

         Motor_Status (Sys_Index) := Set.On;

      else
         Motor_Status (Sys_Index) := Set.Off;

      end if;
   -- Convert drive unit speed from rad/sec to revs/min for IOS display
      Motor_Speed (Sys_Index) := Seu.Rpm (9.5493 * Drive_Unit_Class.Shaft_Speed
                                            (Drive_Unit (Sys_Index)));
```

Selector calls returning data from "connected" objects.

Relays not modeled as objects – design issue.

```
-- Update Hydraulic Pump and set pump status.

      Hydraulic_Pump_Class.Update
         (Instance => Pump (Sys_Index),
         Delta_Time => Delta_Time,
         Fluid_Avail => Hyd_Reservoir_Class.Fluid_Avail (Reservoir),
         Shaft_Speed => Drive_Unit_Class.Shaft_Speed (Drive_Unit (Sys_Index)),
         System_Pressure => Distribution_System_Class.System_Pressure
                            (Dist_Sys (Sys_Index)));

      if Hydraulic_Pump_Class.Pump_On (Pump (Sys_Index)) then

         Pump_Status (Sys_Index) := Set.On;

      else

         Pump_Status (Sys_Index) := Set.Off;

      end if;

   end loop;

end Update_Press_Components;

------------------------------------------------------------------------
--|  Abstract: This procedure updates the fluid pressurization
--|            components of the Hydraulic System Partition.
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 66 Hydraulic_System_Partition.Update_Supply_Components Separate Procedure

```
separate (Hydraulic_System_Partition)

procedure Update_Supply_Components is

begin

-- Set Valve signals and power

   Iso_Valve_Power (Oct.Sys_1) := Elec_Pwr_Msg.Cb
                                  (Elec_Sys_Intfc_Defs.Cb_1022_001).Voltage;
   Iso_Valve_Power (Oct.Sys_2) := Elec_Pwr_Msg.Cb
                                  (Elec_Sys_Intfc_Defs.Cb_1022_002).Voltage;

   Vlv_Close_Cmd (Oct.Sys_1) := Sys_1_Valve_Cmd_Msg.Vlv_Close_Cmd;
   Vlv_Open_Cmd (Oct.Sys_1)  := Sys_1_Valve_Cmd_Msg.Vlv_Open_Cmd;

   Vlv_Close_Cmd (Oct.Sys_2) := Sys_2_Valve_Cmd_Msg.Vlv_Close_Cmd;
   Vlv_Open_Cmd (Oct.Sys_2)  := Sys_2_Valve_Cmd_Msg.Vlv_Open_Cmd;

   for Sys_Index in Oct.Sys_1_Sys_2 loop

-- Update Isolation Valve

      Valve_Class.Update (Instance  => Iso_Vlv (Sys_Index),
                          Close_Cmd => Vlv_Close_Cmd (Sys_Index),
                          Open_Cmd  => Vlv_Open_Cmd (Sys_Index),
                          Pressure  => Distribution_System_Class.System_Pressure
                                       (Dist_Sys (Sys_Index)),
                          Power     => Iso_Valve_Power (Sys_Index),
                          Flow_Rate => Hydraulic_Pump_Class.Output_Flow
                                       (Pump (Sys_Index)));

--   Update distribution system (plumbing that connects components)

       Distribution_System_Class.Update
         (Instance      => Dist_Sys (Sys_Index),
          Delta_Time    => Delta_Time,
          Consumed_Flow => Tot_Press_Flow (Sys_Index),
          Supply_Flow   => Valve_Class.Flow_Rate (Iso_Vlv (Sys_Index)));

-- Update pressure sensor

       Pressure_Sensor_Class.Update
         (Instance     => Press_Sensor (Sys_Index),
          Power_Avail  =>
            (Elec_Pwr_Msg.Cb (Elec_Sys_Intfc_Defs.Cb_1023_001).Power = Set.On),
```

```
        Sensed_Input =>
          Distribution_System_Class.System_Pressure (Dist_Sys (Sys_Index)));

    end loop;
end Update_Supply_Components;
------------------------------------------------------------------------
--|  Abstract: This procedure updates the fluid supply components of
--|            the Hydraulic System Partition
--|
--|  Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 67 Orvc_Common_Types Package Specification

Similar to Std_Eng_Types, but applies to a particular subsystem.

```
package Orvc_Common_Types is

   type Sys_1_Sys_2 is (Sys_1, Sys_2);

   type On_Off is (On, Off);

end Orvc_Common_Types;

------------------------------------------------------------------------
--|   Abstract: This package provides the common data types used by the
--|             partitions of the ORVC.
--|
--|   Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 68 Orvc_Defs Package Specification

Orvc_Defs is like SSVTF's SSTF_Defs package.

Top–Level partitions

Dis versions of standard types

```
with Dis;
with Orvc_Common_Types;

package Orvc_Defs is

   -- The top-level "Component_IDs" in the ORVC system

   Aural_Cue : constant Dis.Component_Id :=
     Dis.Register_Component (Dis.Null_Component, "Aural_cue", Prefix => True);

   Hydraulic_Control_Panel : constant Dis.Component_Id :=
     Dis.Register_Component (Dis.Null_Component, "Hydraulic_Control_Panel", Prefix => True);

   Control_Surfaces : constant Dis.Component_Id :=
     Dis.Register_Component (Dis.Null_Component, "Control_Surfaces", Prefix => True);

   Electrical_System : constant Dis.Component_Id :=
     Dis.Register_Component (Dis.Null_Component, "Electrical_System", Prefix => True);

   Hydraulic_System : constant Dis.Component_Id :=
     Dis.Register_Component (Dis.Null_Component, "Hydraulic_System", Prefix => True);

   Landing_Gear : constant Dis.Component_Id :=
     Dis.Register_Component (Dis.Null_Component, "Landing_Gear", Prefix => True);

   Session_Manager : constant Dis.Component_Id :=
     Dis.Register_Component (Dis.Null_Component, "Session_Manager", Prefix => True);

   Real_6 : constant DIs.Type_Id :=
     Dis.Register_Type (Dis.Null_Component, "Real_6", Dis.Float_Tag);

   Real_15 : constant Dis.Type_Id :=
     Dis.Register_Type (Dis.Null_Component, "Real_15", Dis.Double_Tag);

   package O_O is new Dis.Enum_Functions (Orvc_Common_Types.On_Off);

   On_Off_Type_Id : constant Dis.Type_Id :=
     Dis.Register_Type (Parent      => Orvc_Defs.Hydraulic_System,
                        Name        => "On_Off",
                        The_Tag     => Dis.Enum_Tag,
                        Size        => O_O.Size,
                        Labels      => O_O.Labels);

end Orvc_Defs;

------------------------------------------------------------------------
--|   Abstract: This package contains the DIS definitions for the ORVC.
--|
--|   Warnings: None.
------------------------------------------------------------------------
```

Ada Unit 69 Hydraulic_System_Defs Package Specification

```
with Dis;
with Orvc_Defs;
with Orvc_Common_Types;

package Hydraulic_System_Defs is

   ---------------------------------------------
   -- Define and Register Types
   ---------------------------------------------
```

```
package Sys_1_2 is new Dis.Enum_Functions (Orvc_Common_Types.Sys_1_Sys_2);

Pounds_Per_Sq_In : constant Dis.Type_Id :=
  Dis.Register_Type (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Pounds_Per_Sq_In",
                     The_Tag   => Dis.Float_Tag,
                     Low_Bound => "0.0",
                     High_Bound => "340.0");

-----------------------------------------------
-- Register Terms
-----------------------------------------------

Motor_1_On_Off : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Motor_1_On_Off",
                     The_Type  => Orvc_Defs.On_Off_Type_Id,
                     Users     => (1 => Dis.Look));

Motor_2_On_Off : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Motor_2_On_Off",
                     The_Type  => Orvc_Defs.On_Off_Type_Id,
                     Users     => (1 => Dis.Look));

Motor_1_Rpm : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Motor_1_RPM",
                     The_Type  => Orvc_Defs.Real_6,
                     Users     => (1 => Dis.Look));

Motor_2_Rpm : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Motor_2_RPM",
                     The_Type  => Orvc_Defs.Real_6,
                     Users     => (1 => Dis.Look));

Fluid_Level : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Fluid_Level",
                     The_Type  => Orvc_Defs.Real_6
                     Users     => (1 => Dis.Look_Enter));

Pressure_Sys_1 : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Pressure_Sys_1",
                     The_Type  => Pounds_Per_Sq_In,
                     Users     => (1 => Dis.Look));

Pressure_Sys_2 : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Pressure_Sys_2",
                     The_Type  => Pounds_Per_Sq_In,
                     Users     => (1 => Dis.Look));

Flow_Pump_1 : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Flow_Pump_1",
                     The_Type  => Orvc_Defs.Real_6,
                     Users     => (1 => Dis.Look_Enter));

Flow_Pump_2 : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Flow_Pump_2",
                     The_Type  => Orvc_Defs.Real_6,
                     Users     => (1 => Dis.Look_Enter));

Iso_Valve_Sys_1 : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
                     Name      => "Iso_Valve_Sys_1",
                     The_Type  => Orvc_Defs.Real_6,
                     Users     => (1 => Dis.Look));

Iso_Valve_Sys_2 : constant Dis.Term_Id :=
  Dis.Register_Term (Parent    => Orvc_Defs.Hydraulic_System,
```

Data types used in definitions package.

Register IOS "Look only" term.

Register IOS Look/Enter term

```
                        Name     => "Iso_Valve_Sys_2",
                        The_Type => Orvc_Defs.Real_6,
                        Users    => (1 => Dis.Look));

   ---------------------------------------------
   -- Register Malfunctions
   ---------------------------------------------

   Pump_No_Flow : constant Dis.Malfunction_Id :=
     Dis.Register_Malfunction (Parent   => Orvc_Defs.Hydraulic_System,
                               Name     => "Pump_No_Flow",
                               Length   => 2,
                               Labels   => Sys_1_2.Labels);

   Press_Comp_Fail : constant Dis.Malfunction_Id :=
     Dis.Register_Malfunction (Parent   => Orvc_Defs.Hydraulic_System,
                               Name     => "Pressure_Comp_Fail",
                               Length   => 2,
                               Labels   => Sys_1_2.Labels);

   Iso_Valve_Freeze : constant Dis.Malfunction_Id :=
     Dis.Register_Malfunction (Parent   => Orvc_Defs.Hydraulic_System,
                               Name     => "Iso_Valve_Freeze",
                               Length   => 2,
                               Labels   => Sys_1_2.Labels);

   Pressure_Sensor_Fail : constant Dis.Malfunction_Id :=
     Dis.Register_Malfunction (Parent  => Orvc_Defs.Hydraulic_System,
                               Name    => "Pressure_Sensor_Fail",
                               Length  => 2,
                               Labels  => Sys_1_2.Labels,
                               P1_Name => "Scale",
                               P1_Low  => 0.0,
                               P1_High => 5.0,
                               P1_Type => Orvc_Defs.Real_6
                               P2_Name => "Offset",
                               P2_Low  => -4000.0,
                               P2_High => 4000.0,
                               P2_Type => Orvc_Defs.Real_6);

   Motor_Zero_Rpm : constant Dis.Malfunction_Id :=
     Dis.Register_Malfunction (Parent   => Orvc_Defs.Hydraulic_System,
                               Name     => "Motor_Zero_RPM",
                               Length   => 2,
                               Labels   => Sys_1_2.Labels);

   Dist_Sys_Leak : constant Dis.Malfunction_Id :=
     Dis.Register_Malfunction (Parent   => Orvc_Defs.Hydraulic_System,
                               Name     => "Dist_Sys_Leak",
                               Length   => 2,
                               Labels   => Sys_1_2.Labels,
                               P1_Name  => "Leak_Rate_GPM",
                               P1_Low   => 0.0,
                               P1_High  => 5.0,
                               P1_Type  => Orvc_Defs.Real_6);

end Hydraulic_System_Defs;

-------------------------------------------------------------------------
--|  Abstract: This package contains the Hydraulic System DIS
--|            definitions.
--|
--|  Warnings: None.
-------------------------------------------------------------------------
```

Register malfunction with no parameters.

Register malfunction with parameters.