IN-61
97837

NASA Technical Memorandum 4367

P. 34

Software Architecture for a Distributed Real-Time System in Ada, With Application to Telerobotics

Douglas R. Olsen, Steve Messiora,
and Stephen Leake

JUNE 1992

NASA

NASA Technical Memorandum 4367

Software Architecture for a Distributed Real-Time System in Ada, With Application to Telerobotics

Douglas R. Olsen
Hughes STX
Lanham, Maryland

Steve Messiora
Fairchild Space
Germantown, Maryland

Stephen Leake
Goddard Space Flight Center
Greenbelt, Maryland

NASA

National Aeronautics an
Space Administration

Office of Management

Scientific and Technical
Information Program

1992

Table of Contents

1. INTRODUCTION

1.1 Purpose

This document describes the architecture structure and software design methodology for the Engineering Test Bed (ETB). As part of the Flight Telerobotic Servicer (FTS) program, Martin Marietta Astronautics Group (MMAG) has been developing the Developmental Test Flight - One (DTF-1) manipulator under contract to the GSFC. To support this effort, the Development, Integration, and Test Facility (DITFAC) at GSFC was chartered to develop a system, namely the ETB, to be an operating prototype of the DTF-1 in the functional and operational sense. The ETB was to be used to evaluate the performance of the MMAG system in teleoperated, autonomous, and shared control modes under a range of operating conditions; this evaluation would include design and development of alternative teleoperated robotic control approaches. To meet the needs of the MMAG test and evaluation process, the ETB was being designed as an integrated, modular system with multiple real-time selectable algorithms. The ETB was mainly funded from the FTS. Since that project has been canceled, the ETB has lost most of its funding. Therefore, the software architecture described here has not yet been fully implemented and tested. The architecture is based on experience with two prototype systems in Ada, and on general software engineering principles.

The purpose of this document has been to help ensure a consistent software design throughout the phases of ETB software development. The intent was to publish this document only after a complete system based on this design had been fully implemented, after any final problems had been worked out. Since completion of the ETB is now delayed indefinitely, this document is offered in the hope that others may benefit.

The architecture presented is described in the context of a telerobotic application in Ada. However, the nature of the architecture is such that it has applications to any multi-processor distributed real-time system.

1.2 Scope

Section 1 describes the purpose, scope, and approach taken in defining the ETB architecture and software methodology.

Section 2 describes the architecture of the ETB telerobot system from a logical point of view, independent of most implementation issues such as specific robot control algorithms, software languages, hardware platform, or development environment. An overview introduces terms and concepts that are then expanded on in detail.

Section 3 provides methodology guidelines for implementation using Ada, and defines the ETB architecture more completely using Ada constructs and terminology.

Section 4 provides a glossary of terms in order to ensure consistency of meaning throughout the document. A term defined in the glossary is italicized the first time it is used in the document.

Section 5 lists references to other documents.

1.3 Approach

Establishment of a software architecture for a distributed real-time system is a step that historically has been ignored [1]. Typically, a set of system requirements is established, followed directly by software design. For complex systems such as real-time robotics, especially those that are distributed over multiple processors, this can result in serious inter-processor communication problems and more importantly, result in a fragile system that is costly to change and update.

Development of the ETB architecture and methodology was based on experience gained on the Multi-Algorithm Robot Control System (MARCS) and the Hierarchical Ada Robot Programming System (HARPS), both of which were in-house engineering efforts developed within the DITFAC, and software prototype systems (ETB Build 0 and Build 0.5). The MARCS development provided a system design that incorporated most NASA/NBS Standard Reference Model (NASREM) concepts [2], and described an overall system hardware and software architecture for achieving a multiple-algorithm robot control system. HARPS provided a plan expression language, a world model with a friendly user interface, and E-Move and Primitive NASREM modules [2] for a manipulator and vision system .

The approach to defining the ETB architecture started with the MARCS System Design Document (SDD) [3], and folded into it the best of Build 0 and the lessons learned from that effort. As Build 0 code was not documented, PAMELA [4] diagrams of the Build 0 were developed, and these, together with the actual code, provided baseline documentation of the Build 0 effort. The MARCS System Design Document (SDD) [2] and MARCS PAMELA diagrams provided baseline documentation of the MARCS effort. Also, to provide an example implementation using NASREM, several versions of Ada code from robotics efforts at the National Institute of Standards and Technology (NIST) were reviewed. The teams responsible for MARCS, ETB Builds 0.0 and 0.5, and HARPS met in a combined effort to produce the current document.

The ETB architecture in general is derived from NASREM; however the ETB development effort highlighted areas where NASREM is not necessarily practical when dealing with all the functional needs of an extensive multi-algorithm teleoperated system such as the ETB. NASREM provides an architecture that is applicable to many different robot algorithms, but not all algorithms being investigated in the ETB can be mapped directly into the Sensory Processing/World Modeling/Task Decomposition paradigm that is the hallmark of NASREM. As a result, the ETB architecture allows for some conceptual differences from NASREM.

Care was taken to define specific nomenclature for the ETB architecture and to be precise in the use of it; in some instances, existing NASREM terminology was not used expressly to avoid ambiguity. Some of the more specific technical or nomenclature differences between NASREM and the ETB are footnoted in the document. Taken as a whole, though, the NASREM and ETB architectures are fairly similar.

Throughout the ETB architecture definition process, emphasis was placed on establishing a well-structured software methodology into which the architecture could be directly mapped. A message passing communication protocol and a world model database server were defined that would support the multi-processor configuration and would isolate hardware interfaces. Finally, packaging guidelines were described for implementing the ETB architecture in Ada.

1.4 Acknowledgements

This document was developed under the auspices of NASA Goddard Space Flight Center, Code 714.2, and was made possible through the team efforts of a number of personnel involved in the Engineering Test Bed. The authors wish to acknowledge all ETB development participants, and to recognize those who contributed directly to the concepts in this document:

Maureen Bartholomew, NASA
John Bjorge, Fairchild Space
Kevin Cleary, Hughes STX
Dana Miller, NASA

2 ETB ARCHITECTURE

2.1 Architecture Overview

The ETB architecture defines a hierarchy for representing a telerobot system. Within this hierarchy, a *Module* [1] is a logical entity consisting of the software associated with a hardware component (or set of related components) in the robot system. Typical system components include, but are not limited to, servo manipulators, master hand controllers, joystick controllers, cameras, laser sensors, etc. Modules are generally organized into hierarchical levels, with *Supervisor* Modules providing a hierarchical operator interface.

A Module is comprised of *submodules,* which are *cyclically executing* processes that each perform a specific set of functions (e.g., *Job Assigner, Executor).* The different submodules in a Module can run on separate processors. Within a Module, the *Job Assigner (JA)* acts as coordinator for the Module. The submodules in the system communicate via *Command/Status (C/S) interface channels,* which are used to send *commands* down and relay *status* back up the system hierarchy. Submodules may also communicate via *setpoint data links,*[2] which are used to transfer control data (e.g., desired positions, velocities, accelerations, wrench feedback, etc.) from one submodule to another.

A *system algorithm* is a specific execution configuration of *Module algorithms* and system interfaces (C/S channels and setpoint data links). A submodule invokes *submodule algorithms (SMAs)* to perform algorithmic operations. A Module algorithm is a specific pattern of execution of SMAs by submodules in a Module. The JA submodule coordinates the activation of a selected Module algorithm.

Data that describes or models a physical component of the system is stored as *objects* in the *World Model (WM).* The WM is a system-wide distributed database that is accessible to submodules in all Modules of the system for creating, reading, and writing objects.

2.1.1 System Hierarchy

Figure 1 shows a representative teleoperated/autonomous manipulator system with an operator-assisted vision capability. Each Module in the system is a logical grouping of the software processes

[1]Both NASREM and the ETB group together all software that is associated with a hardware component of the system, which the ETB refers to as a Module. NASREM establishes hierarchical levels (i.e., Servo, Prim, E-Move, etc.) into which all software in the system must fall. However, the ETB uses the labels of Servo, Prim, E-Move, etc. only in the context of manipulator control — not for camera control, sensors, vision processing, etc.

[2]NASREM uses the term attractor sets to refer to setpoint control data [5].

PRECEDING PAGE BLANK NOT FILMED

4 INTENTIONALLY BLANK

required to control a hardware component of the system. Each level in the system consists of a set of Modules interconnected in a vertical hierarchy by C/S interface channels, which are used to send commands down the hierarchy and relay status back up the hierarchy. (For clarity, the WM is not shown in Figure 1).

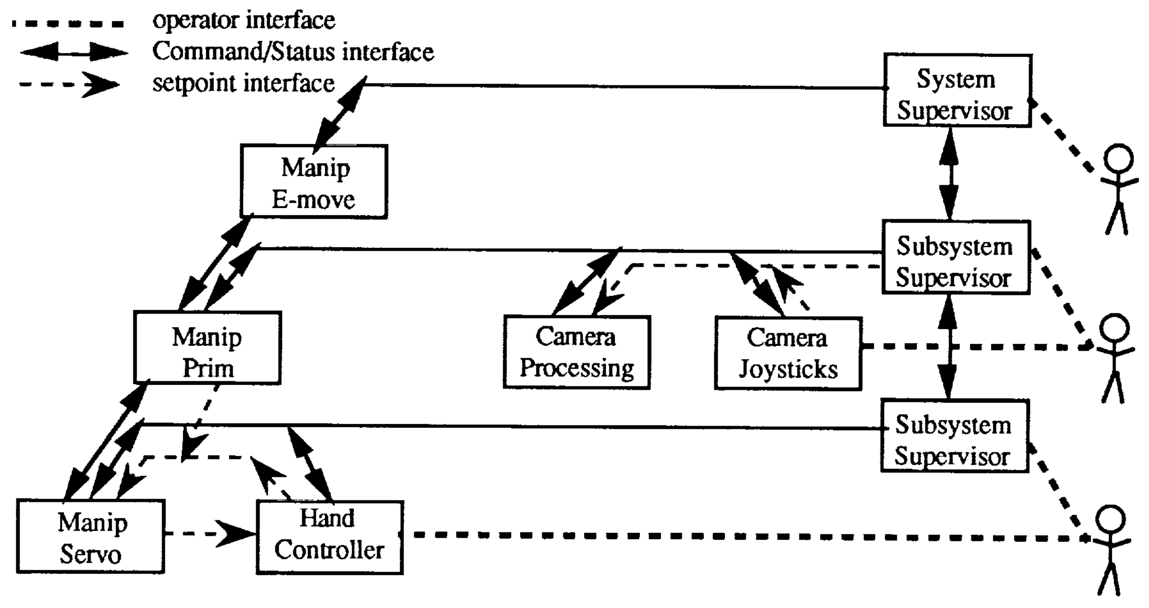


Figure 1. Module Hierarchy in a Telerobot System.

The ETB architecture adopts the NASREM hierarchical levels of Servo, Prim, E-Move, and Task [2] to describe levels of control for manipulators only. Teleoperative (master/slave) transformations are performed at the Servo level; autonomous systems use the higher levels as required. Each level sends commands to the next lower level to select the desired algorithm, coordinate systems, and subsystem state commands as appropriate, and processes status reported by the lower level. For reference, the first four NASREM hierarchical levels are summarized from [2] as:

- The Servo level, which directly interfaces with manipulator and teleop control (e.g., hand controllers, joysticks) hardware, collecting data, performing kinematic and coordinate transformations as necessary, and generating servo commands for the actuators.
- The Prim level, which generates smooth manipulator trajectory motion and force commands and sends setpoint data to the Servo level accordingly.
- The E-Move level, which decomposes elementary move commands into a series of command primitives.
- The Task level, which decomposes task actions performed on objects into sequences of manipulator motion elemental moves.

In a teleoperated system, the human operator must be capable of interfacing with the system. As shown in Figure 1, the operator interface has been defined as a hierarchy of *subsystem supervisor* Modules, which process keyboard (or other input device) commands from a human operator (or a higher level supervisor) and provides status back to the operator display (or higher level supervisor). A *subsystem* is defined as the subset of Modules and interfaces in the system that are subsidiaries to a subsystem supervisor Module. The highest subsystem supervisor in the system is known as the *system supervisor Module*. Subsystem supervisors are allowed for each level in the hierarchy, but are not required[3].

2.1.2 Interface Mechanisms

Two primary interface mechanisms are defined for the ETB architecture; C/S channels and setpoint data links. For both of these mechanisms, information is transferred asynchronously between a sending submodule and a receiving submodule via *messages* across a *message link*. Messages are treated as events. Messages received but not processed are not overwritten, rather they are queued so all will be processed.

Additionally, the WM can be considered to be an interface mechanism between submodules, in that a WM object can be written to by one submodule and read by another. The WM can be used as an interface mechanism for data whose transmittal is not to be treated as an event, which is not time critical in nature, and that meets the definition of a WM object. Otherwise, the data must be communicated via setpoint data links. The WM is described further in a later section.

C/S Channels and setpoint data links are used for both system-level interfaces (between submodules in different Modules) and Module-level interfaces (between submodules in the same Module).

2.1.2.1 C/S Channels

C/S channels consist of a paired set of links, a command link and a status link, between two submodules. C/S channels transfer command messages from a sending to a receiving submodule, and transfer status messages in reply. Command and status messages can be sent at any time. Command messages are used to command an algorithm change or to request a status. Status messages are used to report current state, either in response to a request for status or as a result of an internal state change. State changes can result from health monitoring or from normal events that require a response, such as the need to report a command completion.

[3]NASREM requires a separate Operator Interface for each level in its hierarchy [2]. In contrast with NASREM, to perform the operator interface function, the ETB establishes System Supervisors, but does not require a separate Supervisor for each level of manipulator control. The ETB allows a robot system to have just one System Supervisor, or as many Subsystem Supervisors as desired or required.

2.1.2.2 Setpoint Data Links

A setpoint data link is primarily used for data that is time critical, but can be used whenever the act of transmitting data from one submodule to another needs to treated as an event. Setpoint data is likely to change value every control cycle, but is not required to. Setpoint data is sent independently of C/S channels to eliminate going through the JA, which would add unnecessary data latencies.

In a teleoperative system, a typical setpoint data message would be one containing manipulator control data, such as desired slave end-effector state (position, force, etc.) generated by a master hand controller for use by a slave manipulator. A setpoint data link cannot be used to send commands or report status.

2.1.3 Health Monitoring

Status responses are used to report current state, either in response to a request for status or as a result of an internal state change. Internal state changes can result from detected error conditions or from normal events. System health monitoring involves detecting failures and error conditions and causing a resultant internal state change. State changes are reported as status responses to allow other parts of the system to respond appropriately. Various degrees of health monitoring can be implemented depending on the requirements of the system. Health monitoring for the ETB involves checking for three kinds of errors:

- Hardware errors. To protect laboratory personnel and equipment, the ETB needs to be fail-safe. Status responses must indicate when a hardware failure is detected so that a safe mode can be automatically commanded. However, no fault isolation or fault-tolerant operations are required for the ETB.
- Software errors. To maintain fail-safe operations, status responses must be provided for software failures. However, the ETB software will not be tested to a flight-qualified level of reliability.
- Operator errors. Status messages must be generated for invalid mode commands and parameter ranges. Certain lab experiments may require off-nominal combinations of system modes and algorithms; thus, validity checks will not be made to be too restrictive.

2.2 Module Definition

2.2.1 Module Properties

A Module is defined to have the following properties:

- Is a logical entity consisting of the software associated with a hardware component (or set of related components) in the robot system.
- Can implement more than one Module algorithm, and executes the Module algorithm that is appropriate for the currently selected system algorithm.
- Can be implemented on a single processor or across multiple processors.
- Is comprised of *submodules*, which are *cyclically executing* processes that each perform a functional activity of the Module. A single JA submodule is defined for each Module. At least one Executor submodule is defined for each Module, with additional submodules defined as needed[4]. (Submodules are further described in a later section.)

Modules and their interfaces (C/S channels and setpoint data links) can be dynamically configured during runtime, as shown in Figure 2 (WM not shown for clarity).

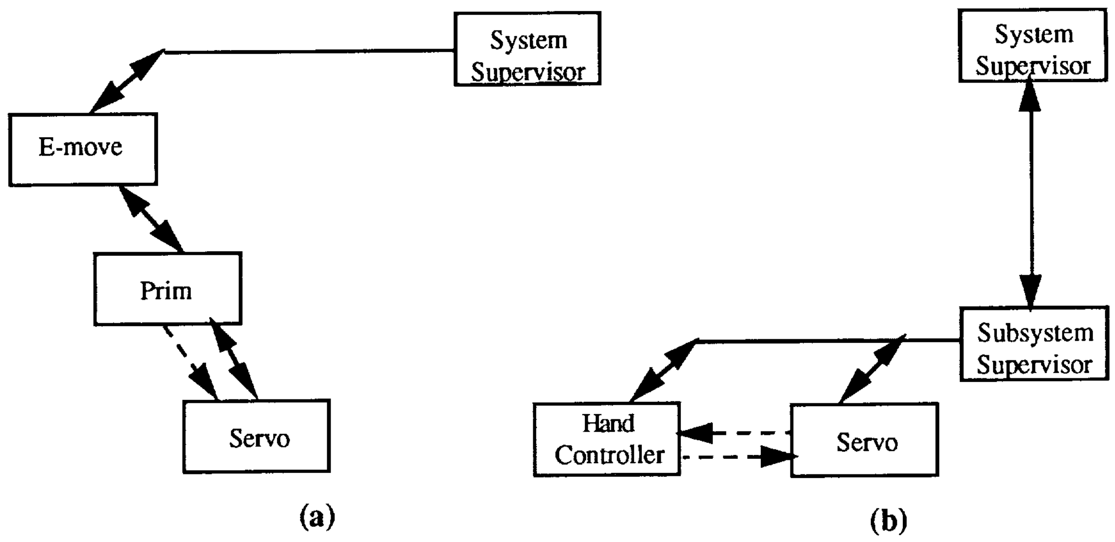


Figure 2. System Algorithm Configurations: (a) Autonomous, (b) Teleoperated.

Having separate Modules for each hardware component facilitates system modularity. Modularity allows an operator to mix and match between using hardware components (e.g., hand controller,

[4]The ETB defines a Job Assigner and an Executor, but does not refer to them together as performing "task decomposition" in the NASREM sense (also, the ETB does not define a Planner).

joysticks) to control more than one other piece of hardware (e.g., cameras, manipulators). Through the selection of system algorithms, Figure 2a shows a fully autonomous system algorithm, with the System Supervisor Module supervising a single manipulator at the E-move level. Figure 2b illustrates another system algorithm, this with a fully teleoperative system algorithm with wrench feedback. In this case, a Subsystem Supervisor Module directly supervises a Master Module and a Servo Module.

2.2.2 System-Level Interfaces

C/S channels are used to implement the hierarchical nature of Modules. Two Modules at the same hierarchical level (such as a hand controller Module and a manipulator Servo Module) cannot send commands and status to and from each other. Rather, some higher level Module in the hierarchy, such as a Prim or a subsystem supervisor, will send commands or queries about the current state of the system down to them. Similarly, each Module in the hierarchy collects status from lower level Modules and sends appropriate status information up to higher level Modules (or, if a supervisor Module, to the operator).

Setpoint data links can only be defined between Modules if they are part of the same subsystem; i.e., have a common, higher level subsystem supervisor. Setpoint data cannot be used to change algorithms. Setpoint links are dynamically configured according to the system algorithm currently selected, which is the primary rationale for requiring a common supervisor Module.

2.3 World Model

The WM is a system-wide distributed database for creating, reading, writing, and deleting objects. Objects in the WM quantifiably describe, or model, a physical entity of the system[5]. Typical WM objects include manipulator link lengths (or other physical characteristics of the system hardware), reference frames, Jacobians, or the currently measured arm pose. However, a current commanded pose would not be a WM object as it does not represent a physical characteristic of the system[6].

At the lowest level, a WM object consists of an identification, an individual data item or items (that are all written to by the same submodule at the same time), and a timetag. Higher level *object groups*

[5]NASREM uses the term World Model to refer both to a database of objects and to the cyclically executing computational processes that manipulate the objects. The ETB restricts the use of the term World Model to describing just the distributed, globally accessible database.

[6]NASREM uses the World Model for all communication between software modules and thus places everything – algorithm commands and status messages, state data, setpoint data, and WM objects – in the World Model [6].

can be built out of lower level objects according to some common characteristic, such as all those that correspond with a hardware component.

Access to WM objects is via a single writer/multiple reader shared-access protocol using semaphore lockout protection, which allows WM accesses to happen anytime within an execution cycle[7]. Single writer does not imply that just one submodule writes to the object throughout the object's lifetime; rather that only one submodule may write at any one time. The WM also supports locking an object for read or write indefinitely, to allow a read/compute/write type operation. Different system algorithms may require different submodules to write to an object. Higher level object groups can implement semaphore lockouts on the entire group. Using semaphore lockouts on groups should only be done relatively infrequently to reduce excessive contentions for access to individual data items when reading and writing to object groups. Object group semaphore lockouts are required when editing the group structure, or if a system algorithm requires strict time correlation between objects in the group.

WM data is, by definition, globally visible to the entire system. However, the Modules in the system may reside on more than one processor. This implies the implementation of the WM must allow for distribution and update of the WM across multiple processors. Updates of objects across multiple processors will take a finite minimum time, which is acceptable since, by definition, receipt of a new value of a WM object cannot be treated as an event (i.e., it is not time critical). If an application requires a data item to be updated in less time than can be guaranteed by the WM distribution mechanism, then the data item should be sent via a setpoint data link.

The ETB needs to be able to add and delete WM objects during runtime. Providing create and delete operations with garbage collection allows the system to run forever, while permitting temporary objects to be used. Typically this occurs when the operator needs to add an arbitrary frame for the Jacobian, or a vision system may detect a new object that has been added to the robot workspace. While all types of objects in the WM are known at compile time, specific objects can be added or deleted during runtime.

To maintain inherent system safety, WM data needs to include a timestamp, and all readers of the data need to check for appropriate timeouts. In addition, timestamps are used for other functions, such as for data correlation during data logging.

[7]NASREM performs WM accesses only at the beginning of the cycle.

2.4 Submodules

A Module consists of a JA submodule, one or more Executor submodules, and other submodules as necessary. The internal structure of a Module, internal Module interfaces, and system interfaces between Modules are shown in Figure 3.

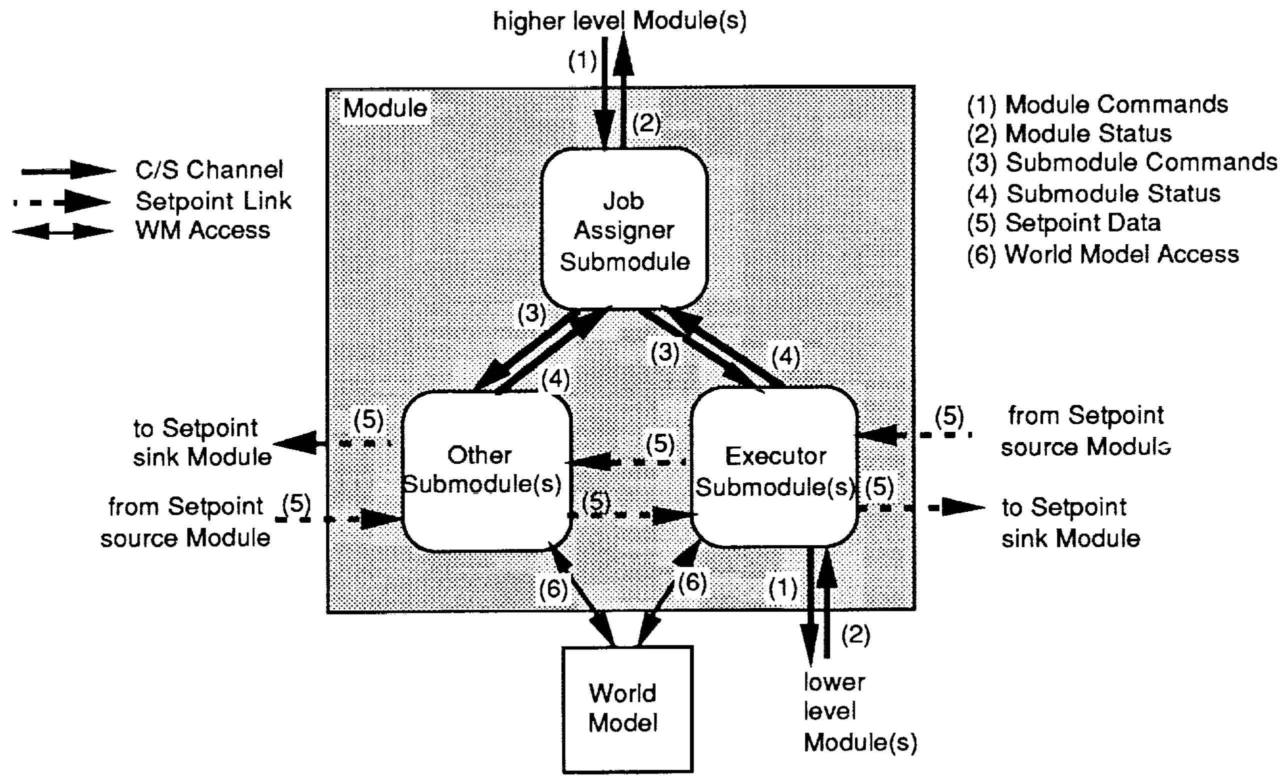


Figure 3. Module Structure and Interfaces.

2.4.1 Submodule Properties

A submodule is defined to have the following properties[8]:

- Performs a functional activity of the Module (or a group of functional activities that have similar properties, such as cycle time requirements).
- Is cyclically executing.
- May run *asynchronously* or *synchronously* with other submodules within the subsystem. If it is the latter, synchronization is done via C/S or setpoint messages.

[8]The NASREM terms 'atomic box' and 'process' [7] are somewhat analogous to ETB submodules, though sufficient differences exist so that the terms should not be used interchangeably.

- Can have the following kinds of system-level interfaces (to submodules in other Modules) :
  - C/S channels for receiving Module commands and sending status (JA only).
  - C/S channels for sending Module commands and receiving status (Executor only).
  - External setpoint data links (all except JA[9]).
- Can have the following kinds of Module-level interfaces (to submodules in the same Module) :
  - C/S channels for sending submodule commands and receiving status (JA only).
  - C/S channels for receiving submodule commands and sending status (all except JA).
  - Internal setpoint data links (all except JA).
  - WM accesses (all except JA).
- Except for the JA, may invoke one or more *submodule algorithms (SMAs)*, which are inactive units of code called by the submodule to perform an algorithmic operation. (SMAs are further defined in a later section.)
- May contain *state data* [10]for internal use by the submodule, for storing state data on behalf of SMAs, or for interfacing between SMAs. State data that is used only by the submodule is called *submodule state data* ; state data stored on behalf of an SMA is called *SMA state data.*
- Must be implemented entirely on a single processor. However, more than one submodule can reside on each processor (this implies a tasking model when implemented).

A Module algorithm is a particular execution configuration of submodules in a Module. In general, different Module algorithms are created by executing different submodules, or by calling different SMAs from submodules. However, changing the scheduling patterns (process rates, priorities, etc..) of submodules also results in different Module algorithms.

Incorporating multiple Module algorithms may require more than one configuration of submodule interfaces. For example, in one Module algorithm, data may be exchanged between submodules via the WM, while in another Module algorithm, the same data may be implemented as state data contained in a single submodule. The latter implementation would be appropriate if the highest priority of the design was to optimize for minimum cycle time (such as was done for DTF-1).

Each submodule is responsible for monitoring safety critical interfaces in that a timeout check needs to be made against the timestamp on all WM data that is accessed by the submodule. If a timeout expires,

[9]The ETB Job Assigner only processes commands and status; it does not process any desired manipulator motion data (e.g., setpoints) as is done by a NASREM Job Assigner.

[10]NASREM uses the term "process variables" to refer to state data [7].

the submodule needs to take appropriate internal safeing actions and send a corresponding status message to the JA submodule.

2.4.2 Module-Level Interfaces

C/S channels are used to support the hierarchical nature of submodules within the Module. Any two submodules at the same level (such as two Executors, or an Executor and some other submodule) cannot send commands and status to and from each other. Rather, the JA will send commands or queries about the current state of the submodules down to them. Similarly, each submodule in the Module sends appropriate status information up to the JA.

Setpoint links can be defined between any two submodules in the Module, except for the JA, which has no function capable of generating or processing setpoint data. Setpoint links are dynamically configured according to the algorithm currently selected.

When a submodule receives a command, status, or setpoint message, it is queued by the message handler and treated as an event. The submodule is required to respond to each, rather than allowing old messages to be overwritten. Though submodules in a Module are typically asynchronous to each other, command and setpoint messages can provide synchronous event-driven execution.

2.4.3 Job Assigner

Each Module in the system is required to have a JA submodule, which coordinates operations of all submodules within the Module. Because of its role as Module coordinator, a JA is defined to have several unique properties in comparison with other submodules:

- Is the only submodule with a C/S interface to higher level Modules (if any).
- Communicates with other submodules in the Module via C/S interfaces.
- Cannot send/receive setpoint data messages.
- Cannot execute SMAs.
- Cannot create, delete, read, or write objects in the WM.
- Cannot interface with hardware.

The JA receives messages from higher level modules to change the Module algorithm. Upon receipt of a message, the JA needs to determine whether the command is valid for the current state of the Module. Once a command is validated, the JA coordinates the operations of submodules by sending command messages to each that are appropriate for the commanded Module algorithm.

Status messages are used either to report current state information or to report detected errors (hardware failures, software failures, and operator errors). Each submodule is responsible for generating a status of its operations (and for the Executor, a status of any lower level Modules as well) and reporting to the JA. The JA forwards a status of the whole Module to higher level Modules (or to the operator if it is a supervisor JA) when requested or when an error is detected. This status may include errors detected by the JA itself, such as an invalid operator command or a timeout on a command to a submodule. As an additional response to a reported error, the JA must take action as applicable to maintain a fail-safe condition; e.g., commanding a safe Module algorithm, such as 'limp'.

2.4.4 Executor Submodules

In general, an *Executor submodule* performs the most time-critical operations of each Module. Functions that are not time-critical, or that can be performed in parallel (such as sensory processing[11]), should be separated out into another submodule unless timing requirements of the algorithm preclude it. A Module may have more than one Executor submodule if the selected algorithm is so designed (e.g., use of separate Executors for each manipulator joint).

For Modules at the lowest level in the hierarchy, the Executor submodule sends commands directly to hardware components, and may also read data from the hardware. For manipulator Servo Modules, the Executor is responsible for computing, through calling of SMAs, the servo control law for manipulators.

For higher level Modules, the Executor submodule is the only submodule with a C/S interface to lower level Modules. As such, it is responsible for interpreting a command from its own JA and commanding the lower Module JA appropriately. Similarly, the Executor must forward a summary status of the lower level Modules up to its own JA (or if an Executor for a supervisor Module, to a display).

Executor submodules receive and send event-based data messages via setpoint data links, such as a currently commanded pose from a hand controller Executor to a slave Executor. Setpoint messages cannot be used to command an algorithm change or report status. However, setpoint messages can contain data that exceeds some limit check. To maintain fail-safe operations, the receiving submodule may have to take a reflex action, such as changing to a 'limp' algorithm on its own. Such a change is reported via a status message to the JA so that a complete algorithm change can be coordinated.

---

11 NASREM requires all algorithms to have a separate Sensory Processing component. While the ETB allows an algorithm to perform Sensory Processing functions independent from Executor functions, it does not require all algorithms to make this delineation. Separating the two can be difficult with some hardware component interfaces, and is essentially trivial with some sensory systems.

2.4.5 Other Submodules

Depending on the requirements of the algorithm, a Module may have just an Executor and a JA submodule, or may have many other submodules. Implementing an algorithm using separate submodules allows an algorithm to run functions in parallel, reducing computations in the algorithm critical path. Computations must be allocated to other submodules if they are to run at different rates, or are to run on different processors. In a master/slave telerobot, computations that typically can be separated into other submodules include Jacobians, kinematics, gravity compensation, and sensory data processing.

As with Executors, other submodules can have setpoint data links with any other submodule (except the JA) and can access the WM. They have Module-level C/S channel interfaces with the JA, but do not have system-level C/S channels with submodules in other Modules.

2.5 Submodule Algorithms

An SMA is an inactive unit of code that is called by a submodule to perform algorithmic operations, typically of a mathematical nature. Individual SMAs should be as independent of each other as possible. An SMA may be associated with (called by) just one submodule or by multiple submodules in the Module, or even by submodules from different Modules. An SMA has no C/S channel or setpoint link interfaces with the rest of the system, and does not access the WM. Any outside interfaces or WM accesses are done by the submodule from which the SMA is called. Although an SMA declares state data types, the submodules that call the SMA need to store the actual data. This approach is used since two submodules may be executing the same SMA on the same processor, necessitating multiple copies of the state data. State data, setpoint data, and WM data are passed to and from the SMA as parameters in the subprogram call.

SMAs can have a range of function complexity. *High-level SMAs* change their behavior based on the input parameters passed to them (e.g., conditional branches on a Boolean input). An example would be a teleoperation SMA that chooses between position-rate and rate-rate motion mapping modes. *Low-level SMAs* do not change their behavior based on the input parameter(s). Examples of low-level algorithms include forward kinematics, Jacobians, and matrix math operations.

3. Ada IMPLEMENTATION

3.1 Overview

The previous section describes the architecture for a hierarchical telerobot system. An architecture, however, can be applied to applications with a number of different languages and runtime environments. This section will map the ETB architecture into Ada constructs [8], taking into account the physical aspects of a distributed Ada runtime environment.

The following implementation details are covered in this section:

1) General Packaging Requirements
2) Distributed Communication Requirements
3) Message Handler implementation
4) World Model implementation
5) Hardware Interface implementation
6) Submodule implementation
7) Submodule Algorithm implementation

Implementation of C/S channels and setpoint data links requires the capability to perform distributed communications within Ada programs; these requirements are restated in Section 3.1.2. An implementation of these requirements, with alternative variations, is provided by the Message_Handler package, which is described in Section 3.2. An implementation of the World Model requirements as previously defined in the ETB architecture is provided by the World_Model package, which is described in Section 3.3. While both the Message_Handler and World_Model packages described here have been partially prototyped to some extent, they have not yet been implemented in a complete system of submodules. Thus, while it is felt the implementation described here is internally consistent and has validity, it has yet to be fully verified under real-world conditions.

3.1.1 General Packaging Requirements

In general, subprograms could be grouped into packages in many different ways. The following list contains a set of criteria for grouping subprograms into packages:

- Functionality. Each package should perform a clearly defined function or set of closely related functions; i.e., that can be designed and implemented largely independent of the rest of the software system.

- Interfaces. All data types specifically associated with a physical object should be packaged together. Conversely, if two subprograms don't share any types and/or operators, then they shouldn't be grouped.
- Maturity. Subprograms or packages that are considered mature should be separated from those that are likely to change as algorithms are further developed.
- Safety. Subprograms that perform dangerous operations (e.g., writing to actuators) should be packaged separately from non-dangerous ones (e.g., computing Jacobians).
- Size. A package should not be of excessive size, where size is measured primarily by compile time, but also by the amount of time required for code review.
- Visibility. Unnecessary visibility should be minimized. However, if grouping two subprograms into separate packages will result in the packages always being with-ed together, then they should be grouped into the same package.

3.1.2 Distributed Communications Requirements

A complete set of requirements for a distributed communications package has been derived during a workshop on real-time Ada issues [9]. The ETB architecture requires only a subset of these requirements. (If this architecture is implemented in an environment that already provides utilities that satisfy these requirements, the Message_Handler package should encapsulate such a utility.) Specifically, for the ETB the distributed communications utility must:

1. Provide a connection-based message passing scheme, where both the sender and receiver must explicitly make a connection.

2. Provide the capability for runtime connection of links.

3. Provide the capability to send a message without waiting for an acknowledgement (i.e., an asynchronous send).

4. Provide a receive-message operation.

5. Provide the buffering of messages.

6. Provide communication error detection and report errors via exceptions.

In addition, a submodule must be capable of waiting for receipt of any one of several messages over a number of different links.

3.2 Message Handler Package

The Message_Handler package provides a message passing layer supporting a distributed Ada system. C/S channels and setpoint data links are supported. C/S Channels consist of a pair of links – a command link and a status link. Command links, status links, and setpoint links allow messages to be sent in one direction with a single receiver. Messages received are queued in buffers within the Message_Handler package. Whether the sending and receiving submodules are on the same processor or on different processors is transparent to the involved submodules, as is what specific protocol is used for processor-to-processor transfer (e.g., message passing co-processor (MPC) or common memory).

Submodules are allowed to wait at the beginning of an execution cycle for external event messages. This implies that they can wait for a message on several message links simultaneously, either command, status or setpoint. Thus the Message_Handler package must support this capability. One mechanism for this (not yet prototyped) is to have a selective wait in the submodule task with an alternative for each message link to be waited on. This implies that the message handler must be able to call the entries in the submodule task. The potential circular visibility problem is handled by letting Message_Handler define a generic package which takes a subprogram parameter that is a renaming of the task entry. The same generic package can take the type of the data on the link, thus providing a clean implementation. The kernel queueing mechanism for rendezvous might be used to queue messages; many compilers recommend this use for interrupt handlers. When Ada 9x is released, it may allow for selective entry calls [10] , which would then provide another alternative implementation.

An alternative (partially prototyped; see Figure 4) is for the Message_Handler to provide a generic procedure Receive that waits indefinitely for a message on a single link, and encapsulate the call to Receive in a task which then rendezvous with the main submodule task as above. This approach requires more than one task per submodule. Due to project cancelation, these two approaches have not been sufficiently evaluated as to the advantages and disadvantages of either; the latter alternative is described in more detail to illustrate the system design in context with a Message_Handler package.

Figure 4 shows the interfaces to the Message_Handler package (Appendix A describes the PAMELA diagram notation used in Figure 4). The visible subprograms in the Message_Handler package are Initialize, Create_Link, Delete_Link, Connect_Link, Disconnect_Link, Send, and Receive. The receiving submodule is responsible for creating (and eventually deleting) the link; the sending submodule must subsequently connect to it (and can later disconnect from it). All links are identified by a string name and a link ID; a name server maintains the list of link names and provides translation between names and IDs. Deletion of links is provided to allow for resource recovery.

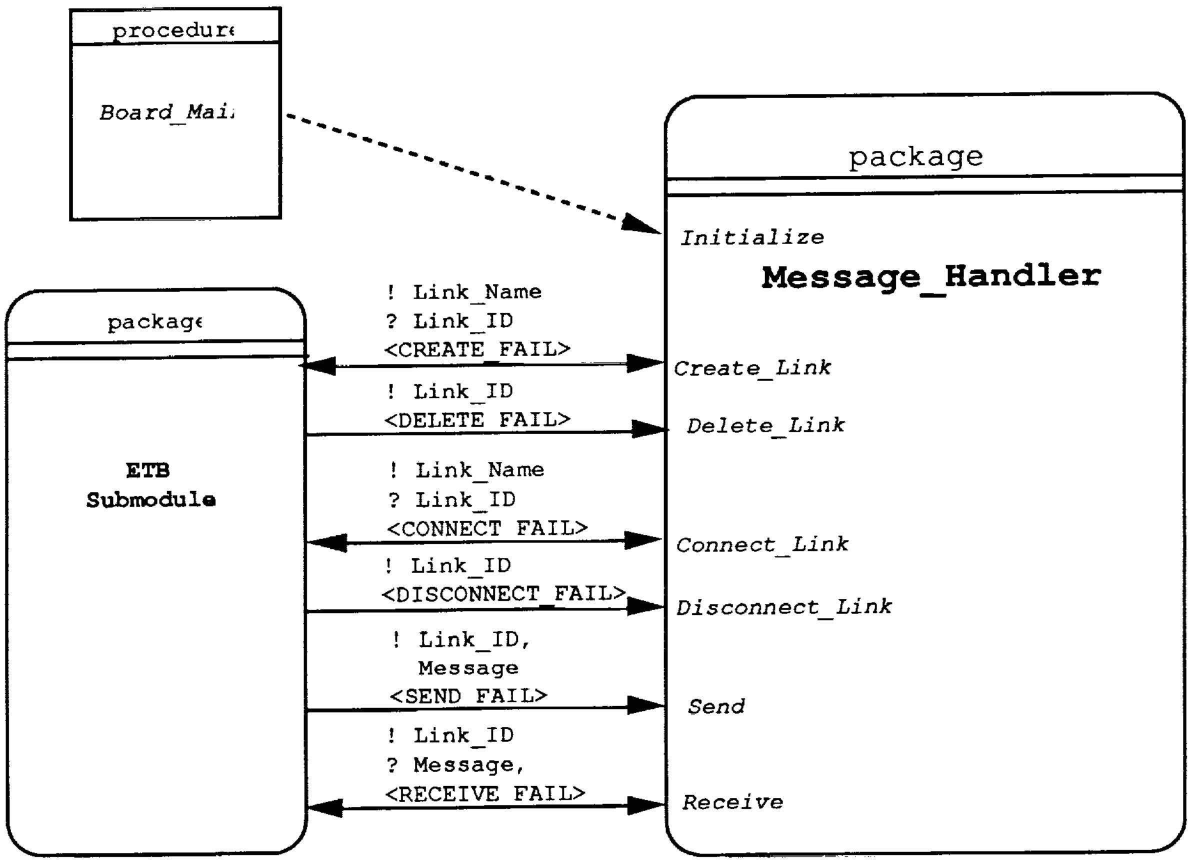


Figure 4. Message Handler Interface.

Links are generally established (i.e., created then connected) as soon as both the sending and receiving submodules are active in the system, although they can be established at any time. Establishment of a link implies foreknowledge of the need for a link by both the sending and receiving submodules. It is anticipated that a link will only be deleted when the receiving submodule (which created the link) is removed from the active system.

Figure 5 illustrates establishment of a link between a sending and a receiving submodule, followed by transmission of some messages via the link. The sender connects to a link by calling the Connect_Link subprogram using a specified Link_Name. As the receiver has not yet created the link, the sender waits for it to do so. The Create_Link subprogram is called by the receiver to create the link using a Link_Name (string); a Link_ID (private) is returned, which will be used by the receiver when calling the Receive subprogram. (If the link already existed, per the name server, a CREATE_FAIL exception would be raised.) As the Connect_Link call is waiting, once the receiver has created the link, the link is established and a Link_ID is returned to the sender, which will be used calling the Send subprogram.

(If the link was already created when Connect_Link was called, the Link_ID would be returned immediately; if the link was already connected, a CONNECT_FAIL exception would be raised.)

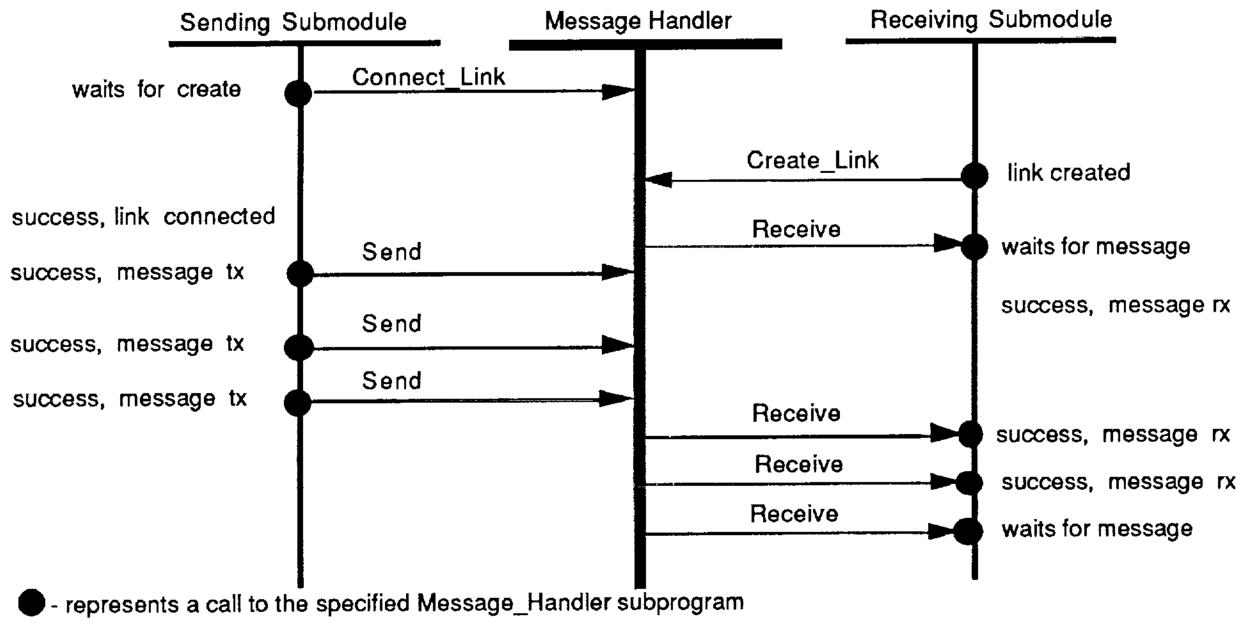


Figure 5. Illustration of a Message Link Establishment.

Both Send and Receive are generic subprograms that are instantiated in the submodule with the desired message type. The receiver calls the Receive subprogram to receive a message; these calls are successful as long as a message exists in the queue, otherwise the Receive waits for the message. The sender calls the Send subprogram with a specified Link_ID and the Message (generic parameter). The Receive subprogram returns the Message (generic parameter) from the queue or waits for a Message to be sent. The sender does not have to wait for the message to be received since the message is queued by the Message_Handler.

Any sender can connect to the created link, but only one can be connected at a time. A sender can call Disconnect_Link at any time, without loss of any messages that may remain in the queue. The receiver can call Delete_Link at any time; any messages remaining in the queue for the receiver on that link are discarded.

3.3 World Model Package

The requirements of a World Model for the ETB architecture were defined in Section 2.3. An implementation of a World_Model package has been specified (see Figure 6). Due to project cancelation, this implementation has not been completely evaluated in operation with ETB submodules. Nevertheless, it is described here in order to illustrate the ETB system design in context with a World_Model package.

Figure 6 shows the interfaces to the World_Model package (Appendix A describes the PAMELA diagram notation used in Figure 6). The visible subprograms in the World_Model package are Initialize, Create, Delete, Get_Object_ID, Set_Lock, Read, and Write.

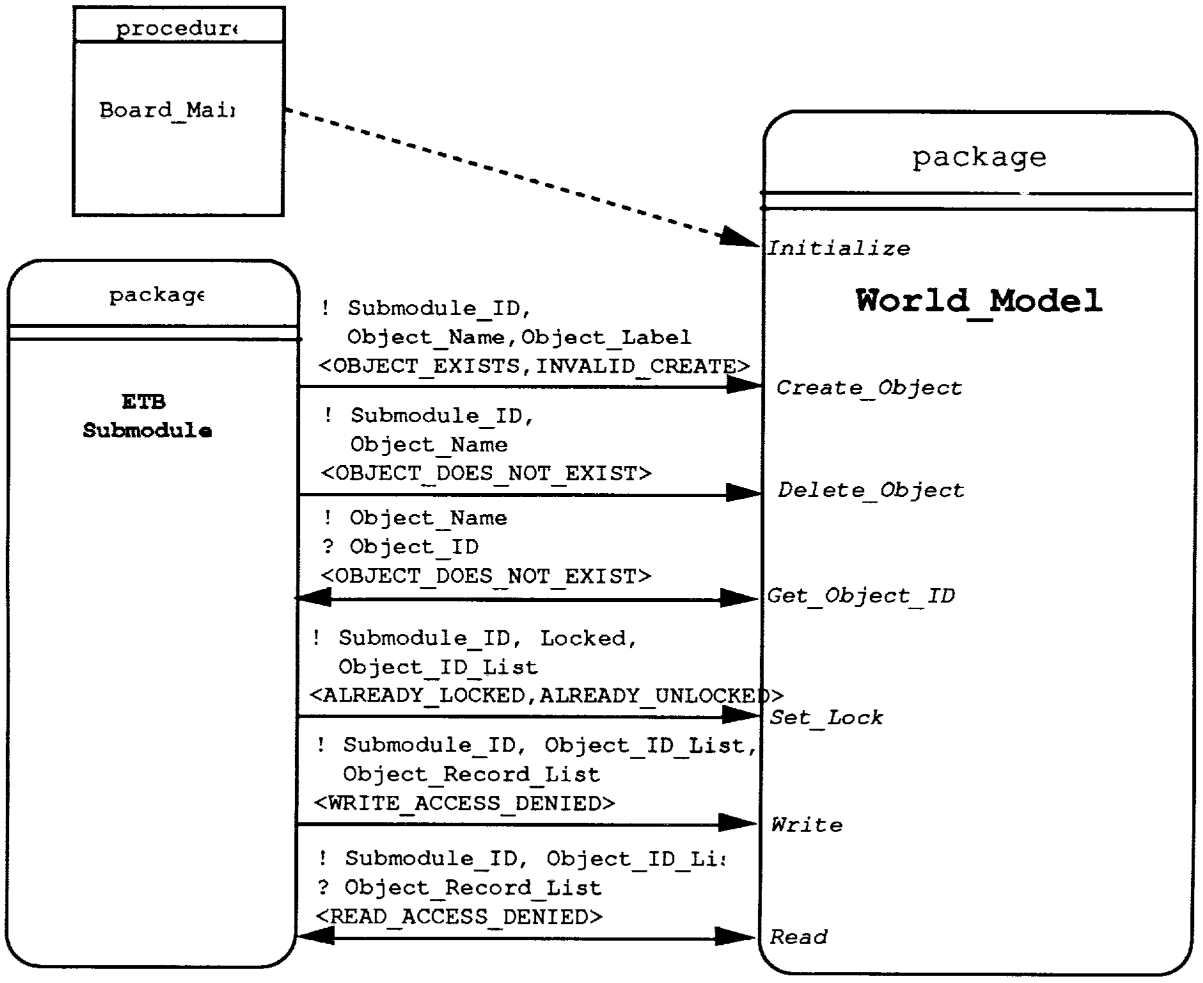

procedur
Board_Mai
package
Initialize
World_Model
package
ETB Submodule
! Submodule_ID, Object_Name, Object_Label <OBJECT_EXISTS, INVALID_CREATE>
Create_Object
! Submodule_ID, Object_Name <OBJECT_DOES_NOT_EXIST>
Delete_Object
! Object_Name ? Object_ID <OBJECT_DOES_NOT_EXIST>
Get_Object_ID
! Submodule_ID, Locked, Object_ID_List <ALREADY_LOCKED, ALREADY_UNLOCKED>
Set_Lock
! Submodule_ID, Object_ID_List, Object_Record_List <WRITE_ACCESS_DENIED>
Write
! Submodule_ID, Object_ID_Li ? Object_Record_List <READ_ACCESS_DENIED>
Read

Figure 6. World Model Interface.

WM objects can be created by all submodules (except JA - exclusion of JA submodules is implied throughout the remainder of this WM description), and are all created during runtime. All submodules are originally compiled with knowledge of all object types (Object_Label), and will create specific objects of those types when they are needed. Generally, all objects initially needed by the submodule are created as part of startup procedures. All objects are identified by a string name and an object ID; a name server maintains the list of object names and provides translation between names and IDs. This implies foreknowledge of the need for an object by all writing and reading submodules. Deletion of objects is provided to allow for resource recovery. Only the submodule that created the object (as identified by the Submodule_ID) is allowed to delete it. The World_Model package provides Read and Write subprograms for accessing an object individually, or collectively as a group. A Set_Lock subprogram allows a submodule to indefinitely lock out all access to an object or a group of objects (use of Set_Lock is not required to read or write to an object, and is normally not used).

A World_Model package is initialized on each processor, each of which becomes a *WM agent*. The body of the World_Model package contains the mechanism, as yet unspecified, for distributing and maintaining objects between WM agents across multiple processors. For systems with a small number of processors, a possible mechanism would be to maintain a copy of each object on each processor; however, this would only be viable for a system with a small number of processors, as distribution traffic would soon become excessive as a the system grew larger. Preferably, a means should be provided to indicate which processors need access to each object, which would keep distribution traffic to a minimum.

When a submodule writes to an unlocked object or object group, to maintain data integrity the local WM agent will still lock out all local read accesses, but does not lock out remote WM agent access. This approach speeds up the local writing process and reduces system-wide access contentions. Once the local write process is completed, the local WM agent will distribute the update to all remote WM agents that need it (this approach was taken, rather than distributing all object updates on a periodic basis, to minimize update latencies between processors).

For special cases, a submodule can prevent all other submodules from obtaining read or write access to an object or a group of objects; this would be done, for example, if it is important to maintain time continuity across all objects in a group. When an object or object group is locked, the WM agent lock out all except the locking submodule, which retains read and write access. If the locking submodule writes to the object or object group, its value is updated, but access by other submodules is denied. Only the locking submodule (as identified by the Submodule_ID) can unlock the object or object group. To lock or unlock an object or object group, a submodule calls Set_Lock with the Locked (Boolean) parameter either Locked or NOT Locked.

To illustrate use of the World_Model by submodules, Figure 7 shows the creation of a WM object, followed by the writing and reading of that object. The writing submodule calls Create, supplying the Submodule_ID (enumeration), Object_Name (string), and Object_Labels (enumeration). The Object_Label describes the type of the object, and the Object_Name distinguishes objects of the same type (if the object name already exists, an OBJECT_ALREADY_EXISTS exception will be raised). To access a World Model object, a submodule needs the Object_ID (private). The Object_ID is provided by calling the Get_Object_ID subprogram and supplying the Object_Name. The Get_Object_ID subprogram searches the World Model using the name provided and returns the Object_ID (if the object cannot be found, an OBJECT_DOES_ NOT_EXIST exception will be raised).

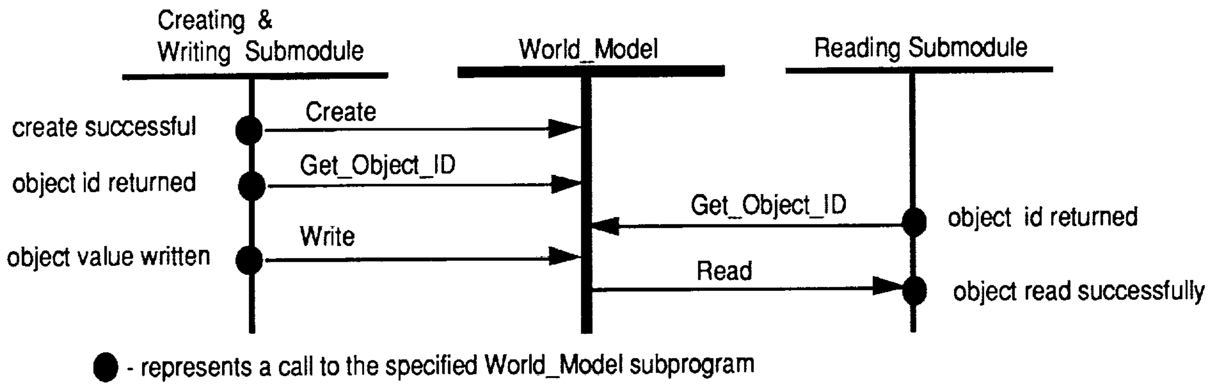


Figure 7. Illustration of a World_Model Access.

The Write subprogram uses the Object_ID_List and Object_Record_List (which, since this is not an object group, have a list length of one) to update the object. A WRITE_ACCESS_DENIED exception will be raised if the object is currently being read or written to by another submodule.

The Read subprogram uses the Object_ID_List (array of Object_IDs) to return the requested Object_Record_List (array of objects). An Object_Record contains the Object_Value and a Time_Stamp; the Time_Stamp indicates an appropriate time value for the object (e.g., when read by a sensor, when computed by an SMA, when written to by the writing submodule, etc). A READ_ACCESS_DENIED exception will be raised if the object is currently locked or is being updated by the local WM agent.

3.4 Hardware Interface Packages

Hardware Interface (HW) packages isolate the interfaces needed for a specific device to a single package. HW packages are used for a wide range of applications, from interfaces to ports on vendor-specific hardware (such as the Robotics Research Corporation's (RRCs) analog servo cards), to interfaces for generic serial ports (e.g., RS 232), or for compiler support services (such as real-time-clock packages or Multibus II MPC support). HW packages cannot access the WM, but they can store state data (such as resolver data for the RRC).

From the point of view of the rest of the system software, HW packages resemble SMAs in that they have visible subprograms (read, write, etc.), are called by submodules, and all data exchanges are done via parameter passing. Layering of HW packages is allowed, and is useful for isolating HW specific details and providing clean, efficient interfaces.

Some hardware ports produce data that must be read on interrupt. In these instances the HW package will either be used to install an interrupt handler, or else will declare a task to service the interrupt. In this sense, HW packages differ from SMAs in that they can have active tasks. The task will generally have a higher priority than other tasks, but will also be relatively quite fast. Passive HW packages are used for hardware ports that can be read at an arbitrary time and that don't require interrupt service.

Only one HW package exists for each device. However, more than one submodule can call each HW package for a given algorithm. This allows for algorithms where a sensory processing submodule calls the HW package to read data, and an Executor submodule calls the HW package to write data (debugging is also facilitated by allowing for more than one calling submodule). To improve runtime efficiency, no built-in semaphore protection is provided during HW package access. Each Module algorithm is responsible for ensuring that HW package read/write conflicts do not arise.

3.5 Submodule Packages

All submodule packages declare at least one task, declare any state data needed for its algorithms, and contain an Initialize subprogram. The Initialize subprogram for each submodule is called at startup. Upon initialization, default system-level and Module-level interfaces must be established. Module-level interfaces do not change after initialization.

Submodule packages do not declare visible subprograms other than Initialize. An exception to this rule is made for subprograms that must be visible to debug the submodule; these subprograms may only be called by the debug routine or internally by the submodule. Beyond these characteristics, the internal structure of a submodule package depends on its function (JA, Executor, or other).

3.5.1 Job Assigner (JA) Submodule Packages

When called, the JA.Initialize subprogram calls Message_Handler.Create_Link to create a command link for receiving command messages from an Executor submodule in the default higher level Module. For all C/S channels in its own Module, JA.Initialize then calls Message_Handler.Create_Link for each status link and calls Message_Handler.Connect_Link for each command link. To send and receive messages across these system-level and Module-level interfaces, instantiations of Message_ Handler.Send and Message_Handler.Receive are made within the JA submodule package.

A JA submodule contains hidden subprograms to maintain knowledge of the current state of the Module and to respond appropriately to new algorithm commands, including any needed reconfiguration of system-level interfaces. Figure 8 shows the interfaces to a Job_Assigner (JA) package.

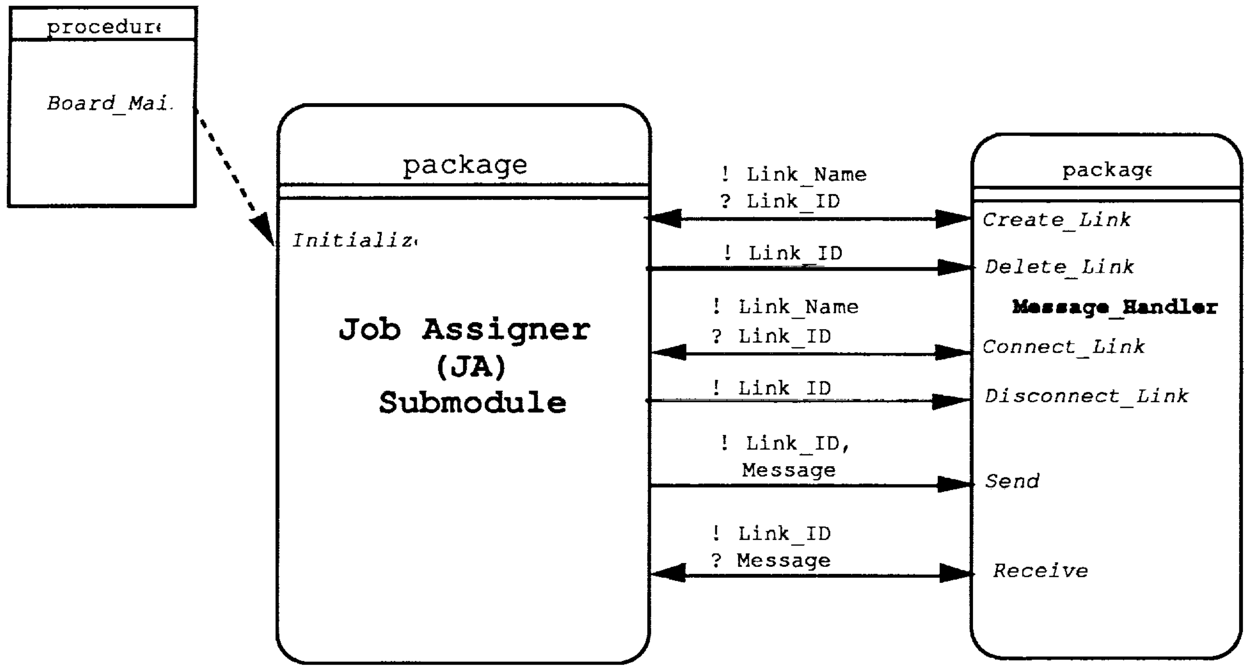


Figure 8. Job Assigner Interface.

3.5.2 Executor (Ex) Submodule Packages

When called, the Executor.Initialize subprogram calls Message_Handler.Create_Link to create a command link from its JA submodule, to create status links from any default lower level Module JA submodules, and to create any default setpoint data links. Executor.Initialize also calls Message_Handler.Connect_Link to connect a status link to its JA submodule. To send and receive messages across these system-level and Module-level interfaces, instantiations of Message_Handler.Send and Message_Handler.Receive are made within the Ex submodule package. Figure 9 shows the possible interfaces to an Executor (Ex) package.

An Executor submodule contains hidden subprograms to maintain knowledge of its current state and to respond appropriately to new commands from its JA, including any needed reconfiguration of system-level interface message links. These hidden subprograms may be visible for debug.

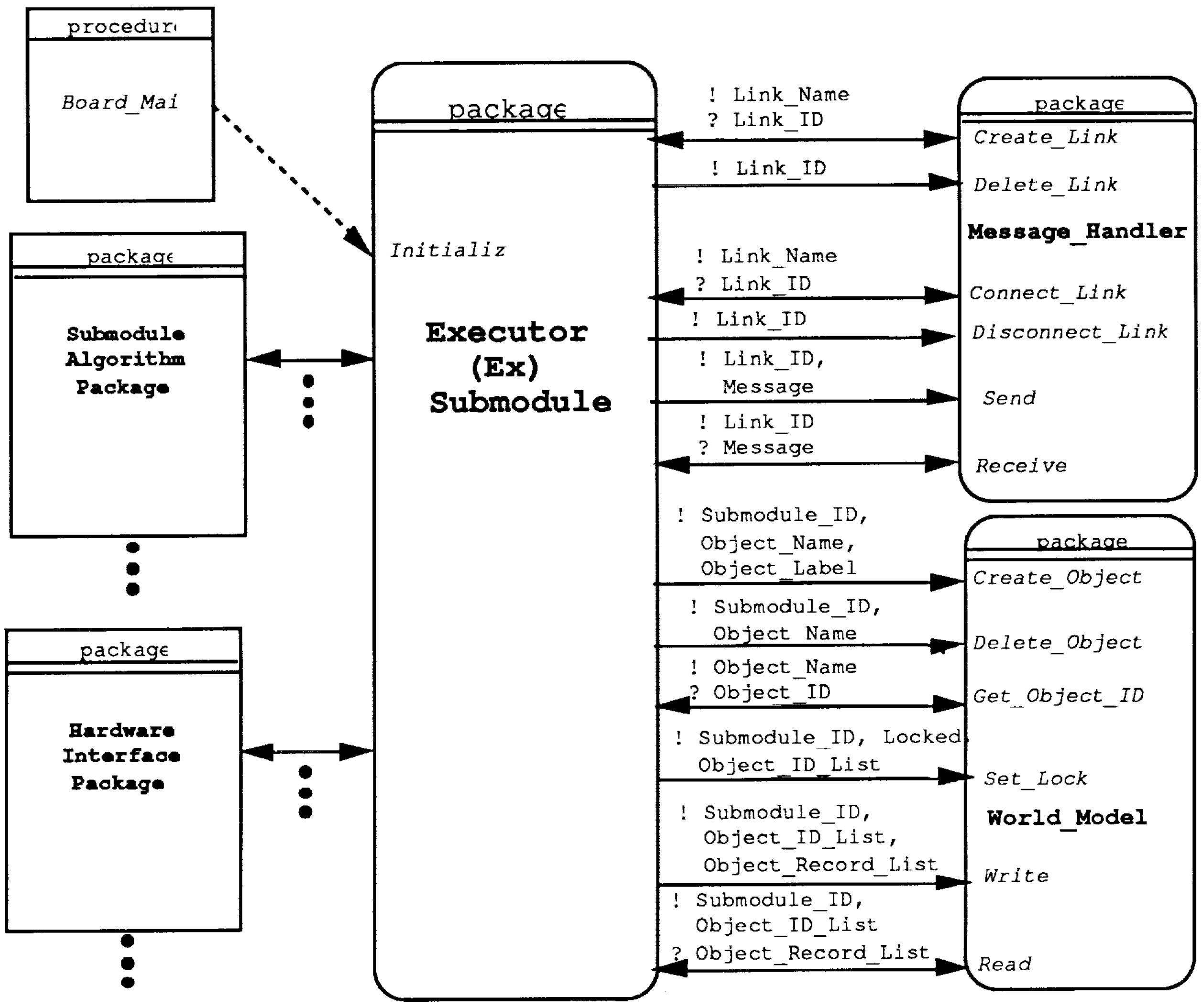


Figure 9. Executor Interface.

To access the WM, Executor subprograms call World_Model.Read or World_Model.Write. Accesses to the WM require identification of a specific object (via Object_ID). The Executor submodule may contain these identifiers locally, or obtain them as part of a command or from a HW package (e.g., user terminal). A command may also instruct the Executor to create a new object through a call to World_Model.Create, using an identifier obtained as part of the command or from a HW package.

Subprograms in the Executor submodule will call SMA and HW subprograms as required by the currently commanded algorithm. Parameters returned by the subprogram call can be written to the WM, stored as submodule state data, sent as setpoint data to another submodule, or used in another SMA or HW subprogram call.

3.5.3 Other Submodule Packages

Other submodule packages are structured similar to the Executor submodule package, except that they do not have any system-level interface command links or status links.

3.6 SMA Packages

The Ada package specification for SMAs can declare constants, state data types (but not actual state data), visible subprograms (including any needed for initialization), and exceptions (which can be handled either locally or propagated up to the calling submodule). SMA state data types must be declared as limited private types. Data can only flow into or out of SMAs via parameter passing. SMAs do not connect to C/S channels or setpoint links, access the WM, or perform Text_IO or Port_IO. SMAs do not declare, contain, or rendezvous with tasks.

3.7 Timing Considerations

A number of Module algorithms can be defined using the same basic SMAs as building blocks. For example, a Module algorithm could be optimized for single-processor timing efficiency – at the expense of code complexity. In such a case, if intermediate terms are shared via state data between a Forward_Kinematics SMA and a Jacobian SMA, the total number of floating point calculations can be reduced, but inter-procedure dependencies and the resultant code complexity will be increased. Alternatively, an algorithm could be optimized for minimum cycle time. In this approach, the Forward_Kinematics and Jacobian SMAs would perform some redundant calculations, eliminating the sharing of intermediate terms but allowing them to run on different processors. CPU usage between the two processors would be greater in total, but as the two sets of computations now would be performed in parallel, total cycle time would be reduced to be the longer of the two[12].

An algorithm is defined not only by the sequence of math operations performed, but also by task scheduling priorities and inter-task data latencies, which are also affected by the distribution of submodules to processors. In general, the effects of task timing and data latency, and the degree to which an algorithm can be parallelized, are characteristics not restricted by the ETB software methodology. Rather, they are specific to the detailed software design of individual algorithms. The effects of timing and latency on an algorithm's performance are, in fact, one of the research issues to be investigated by the ETB.

[12]NASREM states that all 'atomic box processes' and interfaces between them should be the same for all algorithms implemented in that architecture [7]. The ETB architecture allows multiple implementations of 'atomic box structures' to coexist.

4 GLOSSARY

Algorithm. A sequence of software operations that, when performed, results in a known output given a set of inputs. See also System Algorithm, Module algorithm, and Submodule Algorithm.

Asynchronous. Property of a process with respect to another process wherein the two processes are executed independent of each other.

Commands. A message consisting of a command label and a set of parameters (possibly empty). Commands may change the current algorithm or request the current status, but they are not restricted to this use. Commands may not contain copies of world model objects; they must contain references to it. Commands may not contain setpoint data.

C/S Channel. A paired set of links, a command link and a status link, between two submodules. See also Commands, Status, and Message Link.

Cyclically Executing. Software that performs a receive-compute-send cycle, and only at its starting point can it wait for an external event message. The WM can be read or written to at any time during a cycle.

External Event. A condition that occurs outside of a submodule that causes a command, status, or setpoint message to be sent to the submodule. External event messages are queued so that all will be processed by the submodule.

Executor Submodule. A submodule that communicates via a C/S channel with the next lower level Module's Job Assigner submodule, or directly with hardware via a HW interface package. An Executor performs the most time-critical operations of each Module.

Job Assigner Submodule. A submodule that acts as coordinator for the Module, communicating with other submodules in the Module via C/S channels, as well as with a higher level Module's Executor submodule, if applicable.

High-Level SMAs. A class of SMAs that change their behavior based on the input parameters passed to them; for example, a conditional branch on a Boolean input.

Low-Level SMAs. A class of SMAs that do not have conditional branches dependent on the value of an input parameter.

Message. An information transfer between a sending submodule and a receiving submodule of a message link. Messages can be command, status, or setpoint data messages.

Message Link. A one-way software communication path that transmits messages asynchronously between a single sending submodule and a single receiving submodule.

Module. A logical entity consisting of the software associated with a hardware component (or set of related components) in the robot system. A Module is composed of submodules, including at least a Job Assigner and an Executor submodule.

Module Algorithm. A specific pattern of execution of SMAs by submodules in a Module. The JA submodule coordinates the activation of a selected Module algorithm. See also Job Assigner.

Module-Level Interface. Command links, status links, or setpoint data links between submodules in the same Module.

Object. Data that quantifiably describes or models a physical component of the system that is stored in the WM. An object functionally consists of an identification, a timetag, and an individual data item.

Object Group. An operator-defined set of WM objects.

Setpoint Data. A message consisting of data passed via a setpoint data link from one submodule to another. Setpoint data messages are for data that must be shared between two submodules that is time-critical in nature.

SMA State Data. State data that is stored in a submodule on behalf of an SMA.

State Data. Any data local to a submodule that persists between cycles.

Status. A message consisting of a set of parameters passed via a status link from one submodule to another. Status messages are for (including but not limited to) reporting an error condition or responding to a request for the submodule's current status.

Submodule. A cyclically executing process (Ada task). See also Job Assigner and Executor.

Submodule Algorithm (SMA). An inactive unit of code that is called by a submodule to perform algorithmic operations, typically of a mathematical nature.

Submodule State Data. State data that is local to a submodule; i.e., is never passed in or out to the WM, to SMAs, or to other submodules.

Supervisor Module. One of a hierarchy of operator interface Modules, typically that processes keyboard commands from a human operator and provides status back to the operator display. The highest supervisor Module in the hierarchy is the system supervisor; any lower supervisor Modules are subsystem supervisor Modules.

Subsystem. The subset of Modules and interfaces in the system that are subsidiaries to a subsystem supervisor Module.

Synchronous. Property of a process with respect to another process wherein the execution of the two processes is dependent on a communication protocol between them.

System Algorithm. A specific execution configuration of Module algorithms and Module interfaces (C/S channels and setpoint data links). The system supervisor coordinates activation of a selected system algorithm.

System-Level Interface. Command links, status links, or setpoint data links between submodules in the different Modules.

Wrench. Refers to the combined set of forces (translation in 3 Cartesian axes) and torques (rotation about the 3 axes) being applied to an object such as measured by a force/torque sensor.

World Model. A system-wide distributed database that is accessible to submodules of the system for creating, deleting, reading, and writing objects.

World Model Agent. An instance of a World Model; resident on a specified processor.

5. REFERENCES

[1] Ellison, Karen S., 1989, "Software Architecture Design: The Missing Step," *AIAA Computers In Aerospace VII Conference, Monterrey CA,* American Institute of Aeronautics and Astronautics, Washington, DC.

[2] Albus, J.S., McCain, H.G., 1989, "NASA/NBS Standard Reference Model for Telerobot Control System Architecture (NASREM)," NIST TM 1235, US Dept of Commerce, Gaithersburg, MD.

[3] Bartholomew, M., 1990, "Multi-Algorithm Robot Control System (MARCS) System Design Document (SDD), Design Version," Code 714, GSFC, NASA, Greenbelt MD.

[4] Cherry, G. W., 1990, Software Construction by Object-Oriented Pictures, Thought**Tools, Inc., Canandaigua, NY.

[5] Fiala, John C., 1988, "Manipulator Servo Level Task Decomposition," NIST Technical Note 1255, US Dept. of Commerce, Gaithersburg, MD.

[6] Kelmar, Laura, 1988, "Manipulator Servo Level World Modeling," NIST Technical Note 1258, US Dept. of Commerce, Gaithersburg, MD.

[7] Fiala, John C., 1989, "Note on NASREM Implementation," NIST, US Dept. of Commerce, Gaithersburg, MD.

[8] American National Standards Institute, 1983, Ada Language Reference Manual, ANSI/MIL-STD-1815A-1983, US Department of Defense, Washington, DC.

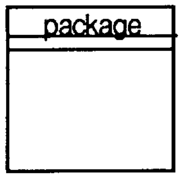
[9] Scoy, Roger Van, 1990, "Summary of the Communications Issues Session," *Proceedings of the Third International Workshop on Real-Time Ada Issues,* Special Issue of AdaLetters.

[10] Sep/Oct, 1991, pp 66-68. *Proceedings of the Fifth International Workshop on Real-Time Ada Issues,* Section 7, Ada Letters.

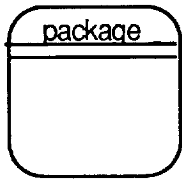
APPENDIX A :PAMELA GRAPH NOTATION

A PAMELA graph is an annotated drawing of an Ada package. PAMELA graphs show the interface and implementation of an Ada package by using standardized symbols.

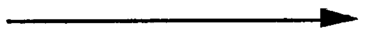
PAMELA graphs are extremely useful for performing inspections of software designs and for communicating designs to team members. Each PAMELA Graph symbol used in this document is shown below with its corresponding Ada interpretation.



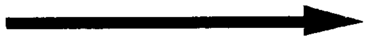
Sequential package, does not contain any active tasks.



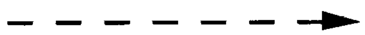
Concurrent package, contains at least one active tasks.



Data flow



Data flow from an external system or device



Parameterless call or Interrupt

!	Icon which denotes an "in" parameter on a data flow
?	Icon which denotes an "out" parameter on a data flow
! ?	Icon which denotes an "in out" parameter on a data flow
<>	Icon which denotes an exception on a data flow

Note 1: These symbols are a subset of the standardized PAMELA symbology.

Note 2: The Icon symbols (!,?,!?) are placed on the side of the caller.

PRECEDING PAGE BLANK NOT FILMED

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE June 1992	3. REPORT TYPE AND DATES COVERED Technical Memorandum

4. TITLE AND SUBTITLE
Software Architecture for a Distributed Real-Time System in Ada, With Application to Telerobotics

5. FUNDING NUMBERS
710

6. AUTHOR(S)
Douglas R. Olsen, Steve Messiora, and Stephen Leake

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA-Goddard Space Flight Center
Greenbelt, Maryland 20771

8. PERFORMING ORGANIZATION REPORT NUMBER
92B00063

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, D.C. 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA TM-4367

11. SUPPLEMENTARY NOTES
Douglas R. Olsen: Hughes STX, Lanham, MD; Steve Messiora: Fairchild Space, Germantown, MD; Stephen Leake: NASA-GSFC, Greenbelt, MD.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The architecture structure and software design methodology presented is described in the context of a telerobotic application in Ada, specifically the Engineering Test Bed (ETB), which was developed to support the Flight Telerobotic Servicer (FTS) program at Goddard Space Flight Center. However, the nature of the architecture is such that it has applications to any multi-processor distributed real-time system. The ETB architecture, which is a derivation of the NASA/NBS Standard Reference Model (NASREM), defines a hierarchy for representing a telerobot system. Within this hierarchy, a Module is a logical entity consisting of the software associated with a set of related hardware components in the robot system. A Module is comprised of submodules, which are cyclically executing processes that each perform a specific set of functions. The submodules in a Module can run on separate processors. The submodules in the system communicate via Command/Status (C/S) interface channels, which are used to send commands down and relay status back up the system hierarchy. Submodules also communicate via setpoint data links, which are used to transfer control data from one submodule to another. A submodule invokes submodule algorithms (SMAs) to perform algorithmic operations. Data that describe or modelsa physical component of the system are stored as objects in the World Model (WM). The WM is a system-wide distributed database that is accessible to submodules in all Modules of the system for creating, reading, and writing objects.

14. SUBJECT TERMS
Distributed System, System Architecture, Robotics, Flight Telerobotic Servicer, Development Test Flight, Software Engineering

15. NUMBER OF PAGES
40

16. PRICE CODE
A03

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