Technical Report

CMU/SEI-89-TR-038
ESD-TR-89-049

Inertial Navigation System Simulator Program: Top-Level Design

Kenneth J. Fowler

January 1990

Technical Report

CMU/SEI-89-TR-038

ESD-TR-89-049

January 1990

Inertial Navigation System Simulator Program: Top-Level Design

Kenneth J. Fowler

Software Systems Program,

Real-Time Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This document is available through SAIC/ASSET: 1350 Earl L. Core Road; PO Box 3305; Morgantown, West Virginia 26505 / Phone: (304) 284-9000 / FAX: (304) 284-9001 / World Wide Web: http://www.asset.com/sei.html / e-mail: webmaster@www.asset.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / Attn: BRR / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8274 or toll-free in the U.S. — 1-800 225-3842).

Table of Contents

List of Figures

List of Tables

Inertial Navigation System (INS) Simulator Program: Top-Level Design

Preface

Ada design work on the Inertial Navigation System (INS) Simulator by the Software Engineering Institute (SEI) Real-Time Embedded Systems Testbed (REST) Project was undertaken to investigate the application of explicit concurrency to hard real-time requirements. Through a strategy of Ada task partitioning based on behavioral modes (e.g., active or passive, depending upon the implemented operations), design rules were sought to enable effective program structures within stringent real-time environments. The INS Behavioral Specification [Landherr 87a] and the INS Program Top-Level Design [Klein 87] reflected the status of this work prior to changes brought about by the pace of investigation.

Recent advances in schedulability analysis have shown that the dynamic performance of an Ada task implementation can be reliably predicted, and that improvement is achievable through analytical means. With the goal of giving maximum visibility to these techniques, the presentation of the Top-Level Design has been updated to give proper emphasis and visibility to this approach for real-time design. In addition, greater weight has been placed on portability issues; the objective is to reduce implementation-dependent decisions by abstracting out those concerns from the top-level description. Reflecting these concerns, the current INS Behavioral Specification [Landherr 89] documents the immediate goals of the effort. Runtime BIT and data extraction have consequently been deleted from the requirements and design of the INS to focus on the central problems involved in constructing real-time software.

1. Introduction

Abstract: Hard real-time systems have consistently proven to be some of the most difficult for successful software implementation. Attributes often associated with the intractable nature of real-time are *concurrency*, *severe timing constraints*, the *complexity of real-world devices*, and *limited resources*. In this experiment, an actual embedded hard real-time application (Inertial Navigation Set, AN/WSN-5) is simulated and ported to a variety of target processors. The effort is specifically directed at investigating the capability of Ada for providing program development solutions in the hard real-time regime. Special emphasis is focused on applying the built-in concurrency capabilities of Ada. The effort contends with typical cross-targeting issues such as board-level execution and memory configuration, device communications, and runtime debugging of the application. This report presents the top-level design of the application and addresses the solution in terms of a concurrency abstraction. Beginning with a classical data flow analysis of the requirements, Ada tasks are derived from *analyzable* categories, specifically *periodics, aperiodics*, and *servers*. This classification scheme is predicated on work actively being conducted on a scheduling technique that quantifies the effect of task preemption and blocking, behavior fundamental to the concept of parallelism in Ada. In a corollary report [Borger 89], a schedulability analysis of the INS is described within the framework of the task set developed in this top-level design.

The Inertial Navigation System (INS) Simulator application [Meyers 88a] consists of two programs executing on separate computers: the INS simulator program [Meyers 88b] and the external computer system (EC) program [Meyers 88c]. Since the principal function of the EC is to act as a test driver for the operation of the INS simulator, the role of the EC will not be discussed any further in this document. A detailed description of the requirements for the simulator application is presented in the Behavioral Specification for the INS [Landherr 89].

The requirements for this development include:

- **Adopting Ada** concurrency primitives for the software implementation. In isolated instances, alternative techniques may provide better performance in the face of real-time constraints. Selected alternatives may take the form of operating system primitives or assembly language routines.
- **Using incremental development** to explore the solution space for the most robust implementation. Examples of two principal areas for prototyping are:
    1. *Concurrency architecture*—seek to maximize processor utilization (i.e. efficient use of available horsepower) while retaining task schedulability for process deadlines.
    2. *External interfaces*—seek to maximize the efficiency of device communications in real-time.
- **Reusing development software** for the external computer system (EC) and in areas where a common pattern of functionality is observed.

The goals for this investigation are:

- Select a representative application (e.g., strict timing demands, mix of periodic and event-driven process requirements, limited memory resources, error handling, low-level I/O, and interrupts) as an ongoing Ada artifact for experimentation in this domain. Make explicit use of Ada tasks within the real-time design.
- Apply any relevant practical results being produced by the real-time scheduling research community.
- The INS simulator must satisfy a set of timing requirements that are similar to a real world INS with respect to data updating, message transmission, and message reception.

This document discusses the top-level design of the application from two points of view reflecting different perspectives: data flow and transformation, and the concurrency and control perspective (Ada tasking) with modularization (Ada packaging). The effort at improving INS performance, allied with advances in schedulability research by the SEI Real-Time Scheduling in Ada (RTSIA) Project have exposed some promising alternative strategies in hard real-time software design. The changes which are inevitably incurred by the adoption of a new paradigm are reflected in this update. Since the intention of the REST Project is to implement the INS Simulator on more than one computer and runtime system, the top-level design description presents the INS Simulator program from an abstract perspective and in a general manner. Design aspects which are specific to a particular implementation will be described in the detailed design.

The top-level design document contains three chapters:

1. **Data flow analysis**: describes the overall flow of data in terms of data stores, data transforms, and the data flows between stores and transforms.
2. **Concurrency and control**: describes the real-time design and motivates the concurrent threads of processing and control (i.e., the tasking structure).
3. **Packaging**: defines the top-level packages grouped within functional "subsystems," their interdependencies, and task containment.

The remainder of the introduction is devoted to a brief tutorial on inertial navigation aids. It covers the purpose of inertial navigation aids, an operational description, and a list of INS elements which have a correspondence in the simulation.

1.0.1. INS Overview

Since the recent generation of naval vessels depends upon accurate, split-second guidance, modern inertial navigation systems have become a mission-critical component of shipboard operations. When we focus on the INS problem of continuously locating and tracking vessels over vast distances, of the ocean, the solution requires a considerable amount of distributed equipment, including one or more gyroscopes, various motion sensors, and computers for performing calculation and communications. Data acquisition of detected changes in vessel movement and sea state must be periodically sampled in real-time. Additional inputs are typically commanded changes in course and speed. Reducing the data stream to a useful form requires rapid calculation of the ship's smoothed attitude composed of ship's motion (see Figure 1-1, 1-2, and 1-3) with six degrees of freedom plus velocity/acceleration components. Once attitude has been resolved to a fixed stable platform, navigation computations follow to determine the ship's current position in latitude and longitude, and surface velocity.

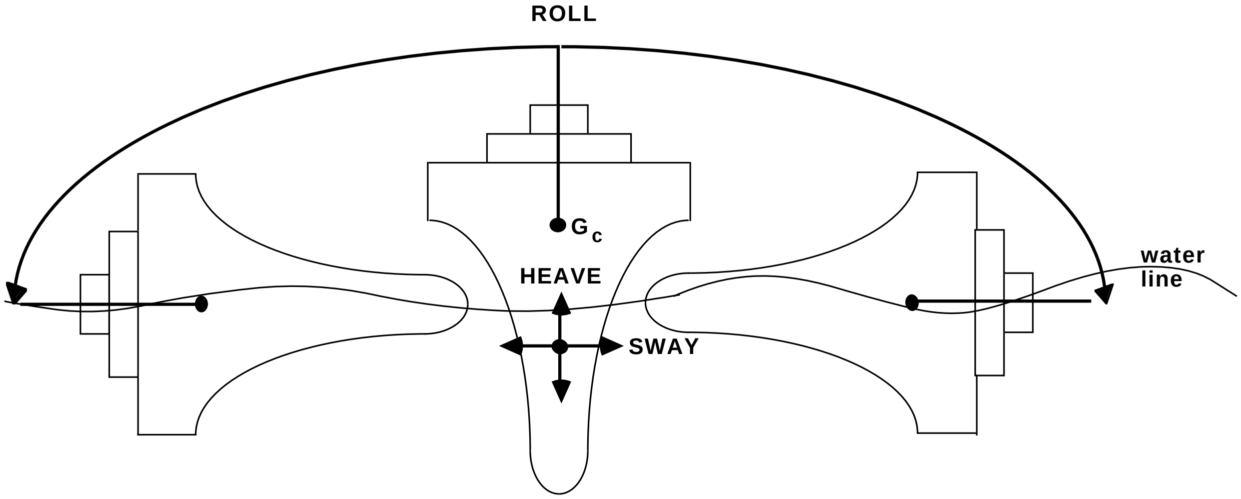


Figure 1-1: INS Simulator: Ship's Motion: Roll, Heave, Sway

In summing up, the INS simulator described here models the behavior of each of the ship's defined independent motions (depicted and enumerated below) through the use of uncoupled-sinusoids:

1. **pitch** - an angular rotation of the ship identified in Figure 1-2
2. **roll** - an angular rotation of the ship identified in Figure 1-1
3. **yaw** - an angular rotation of the ship identified in Figure 1-3
4. **surge** - a linear displacement of the ship identified in Figure 1-2
5. **heave** - a linear displacement of the ship identified in Figure 1-1
6. **sway** - a linear displacement of the ship identified in Figure 1-1

The reader should note that all rotational and translational motions are measured with respect to the ship's center of gravity — G_c. The linear distance between G_c and a ship's motion sensor must be factored into the 3-vector calculations as a *lever arm constant*. Each motion parameter then is composed of the following elements:

- *amplitude* - in *degrees* of angular rotation or *feet* of displacement

- *frequency* - in *radians/sec*
- *phase* - in *degrees* of angular offset

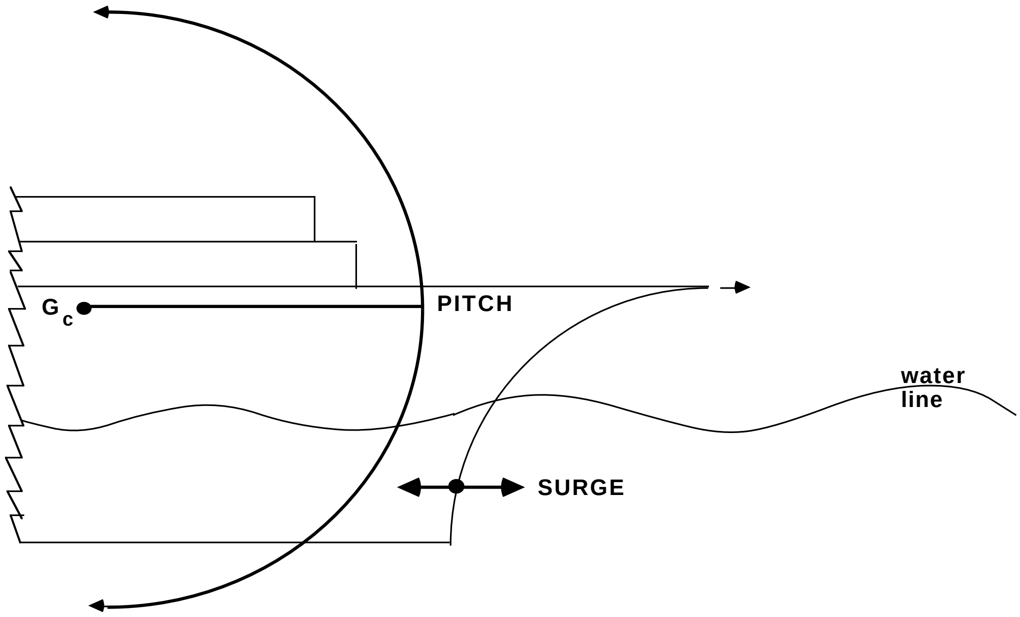


Figure 1-2: INS Simulator: Ship's Motion: Pitch, Surge

Other principal values computed by the INS are as follows:

- **Heading** - the ship's current direction of forward movement, taking into account Course (commanded direction of travel) and Yaw (see Figure 1-3) measured in *degrees*
- **List**, the ship's *deviation from vertical, when at rest, about the roll axis, due to center of gravity displacement* measured in *degrees*
- **Trim**, the ship's *fixed deviation from the horizontal about the pitch axis, due to center of gravity displacement* measured in *degrees*
- **Ocean Velocity**, *North, East* components measured in *knots* (e.g., *nautical miles per hour*
- **Ship Speed**, measured in *knots* (nautical miles per hour)
- **Ship Velocity**, *North, East, Vertical* components measured in *knots*
- **Velocity Integrals**, the ship's cumulative distance covered, measured in *feet*
- **Latitude and Longitude**, the ship's global coordinate location measured in degrees, minutes, and seconds

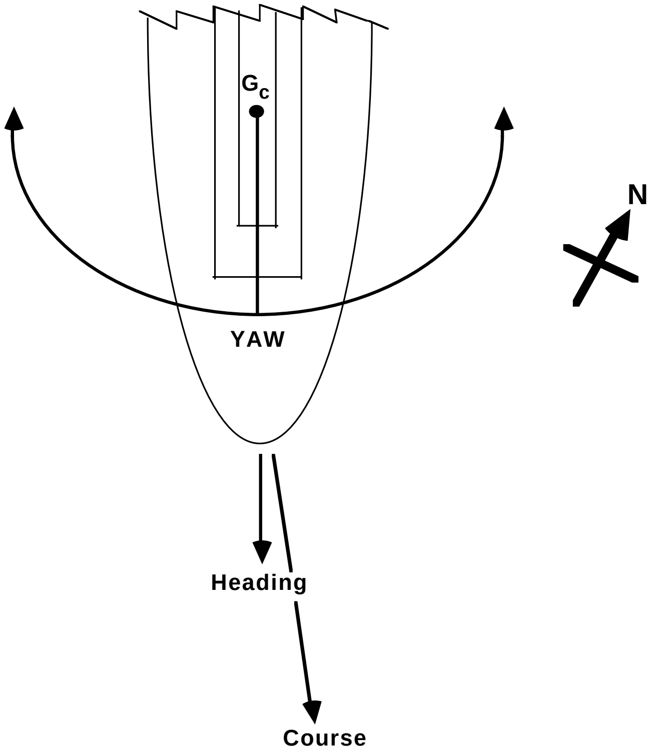


Figure 1-3: INS Simulator: Ship’s Motion: Yaw with Heading and Course Indicated

2. Data Flow Analysis

This chapter describes the overall flow of data through the INS simulator program. Details include:

- data-flow diagrams
- purpose and content of the principal data stores
- function and constituent data flows of the principal data transforms

This chapter is the first step in creating a program design that meets the requirements of [Meyers 88b] and [Landherr 89]. It thus forms the basis for subsequent chapters.

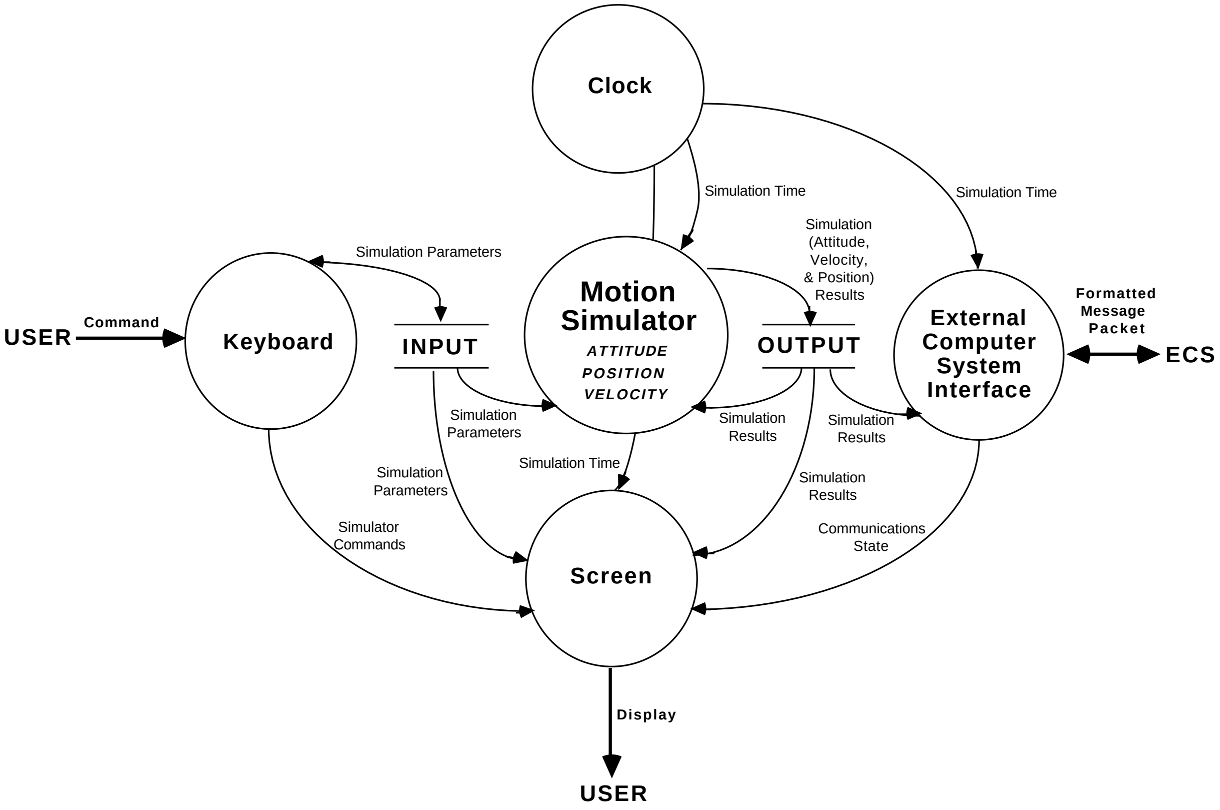


Figure 2-1: INS Simulator: High-Level Data Flow Diagram

The overall flow of data through the INS simulator program is illustrated in progressive detail by the following data-flow diagrams. Figure 2-1 depicts the system at the highest level. From this perspective we can see the overall architecture of INS and begin to identify its asynchronous components. The appearance of five transforms, four of which represent possible devices, indicates some *independence* in the threads of execution. At this point, however, we will not consider concurrent behavior but restrict our discussion to the principal data paths. The major transforms and data flows are as follows:

- **Keyboard:** User man-machine interface (MMI) commands in the form of a character string are entered at the keyboard. Valid commands encompass the setting of simulator scenario, seastate and other motion parameters into the input store. Command string characters are echoed to the display screen. On user request, current parameters may be retrieved from the input store for Screen display.
- **Clock:** Simulated time is generated for use by the motion simulator, the external computer system interface, and the user console display screen.
- **Motion simulator:** Input store parameters are used by the motion simulator to compute new attitude, position, and velocity results, which are then deposited into the output store. The results from the previous calculation of the motion simulator are retrieved from the output store and used for update to the next simulator state.
- **External computer system (EC) interface:** Motion simulator results are retrieved from the output store and message-formatted in anticipation of periodic output to the EC. EC commands are received and validated with a response when indicated by the requirements detailed in [Meyers 88c].
- **Screen:** The simulation parameters and results are received at periodic intervals for display at the user MMI. Simulation alerts are received from the keyboard and EC interface for display to the user. Communications status is received from the EC interface for screen display.

In transitioning from the first abstraction of the solution in Figure 2-1 to the greater level of detail depicted in Figure 2-2, we can identify a series of software design choices made to accommodate the behavioral requirements identified in [Landherr 89]. Each "cloud" represents transforms and stores from the previous Figure 2-1 and encompasses internal data transforms and stores within it. Any implementation-dependent features continue to remain hidden as an aid in target processor portability. While hidden dependencies might entail additional architectural features, at this juncture the intent is to restrain the design to a common structure. A method referred to as *Design Approach for Real-Time Systems* (DARTS) [Gomaa 84], guided the INS data flow analysis. The data flow and data transform elements of the second, and final, data flow diagram are described in the following sections.

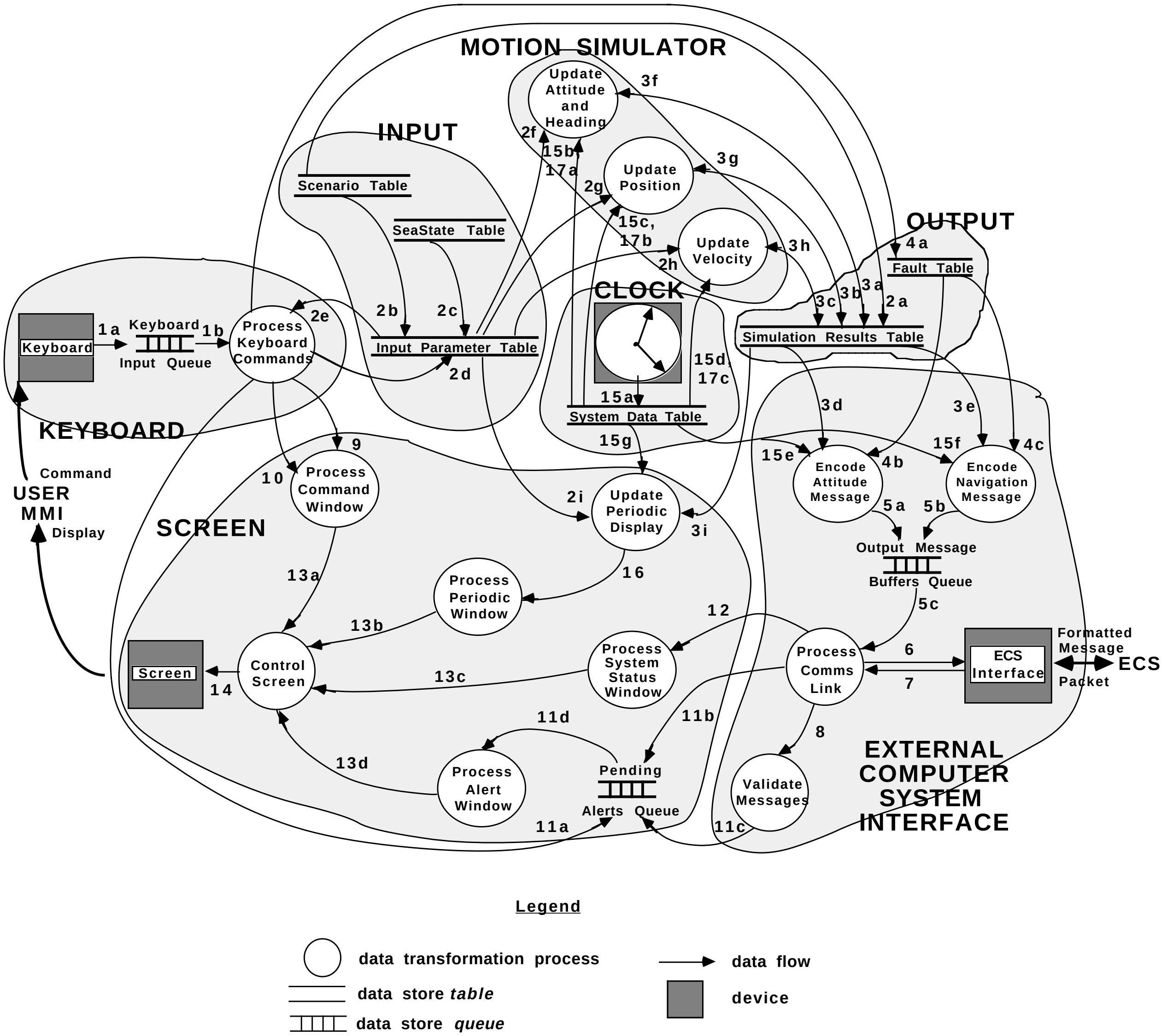
2.1. Data Stores

The nine major data stores of the INS Simulator consists of two data constant tables:

- Scenario_Table — *ship's navigation-preconditions store*
- Sea_State_Table — *ship's attitude-preconditions store*

four shared-resource persistent data tables:

- Input_Parameter_Table — *simulation inputs store*
- Simulation_Results_Table — *simulation outputs store*
- System_Data_Table — *simulation time store*



List of Data Flows

1. Keyboard Characters
2. Simulation Parameters
3. Simulation Results
4. Fault Values
5. Simulation Results Messages
6. Periodic Output Packets
7. Periodic Input Packets
8. Input Message Values
9. Command Line Character Echo
10. Command Line Parameter String
11. Simulation Alerts
12. Communications State
13. Screen Display Packets
14. Screen Display Characters
15. Simulation Time
16. Simulation State
17. Simulation Constants

Figure 2-2: INS Simulator: Low-Level Data Flow Diagram

- Fault_Table — *simulation faults store*

and three data queues:

- Keyboard_Input_Queue — *decouple arrival of user input commands from command processing*
- Pending_Alerts_Queue — *decouple simulation alert arrivals from display processing*
- Output_Message_Buffers_Queue — *decouple simulation periodic output message arrivals from communications processing*

A description of the contents of the INS data stores is provided in the following sections.

2.1.1. Scenario Table

A startup simulation state can be chosen with the Scenario_Table which contains 15 user-selectable sets of the following static parameters (subcomponents shown as above):

Lever Arm Constants
(3-coordinate distance of WSN-5 from ship's center of gravity)
Ship List
Ship Trim
Initial Ship Course
Initial Ship Speed
Initial Latitude
Initial Longitude
Ocean Current
East
North

The constant Scenario_Table provides the following Output Data Flows:

(Scenario_Table --(**2a**)--> Simulation_Results_Table)

(Scenario_Table --(**2b**)--> Input_Parameter_Table)

The first data flow (2a) consists of the initial ship's navigation state selected by default. The data represents an initial motion simulator output prior to the first simulation-generated results. The second data flow (2b) consists of the same initial ship's navigation state selected either by default or by the most recent user command of the SELECT SCENARIO for motion simulator input.

2.1.2. SeaState Table

The Sea_State_Table contains 7 user-selectable sets of initial ship motion parameters. Each of the motion parameters is a composite variable in three parts (subcomponents are shown in bold script):

- **Surge**
  - *amplitude*
  - *frequency*
  - *phase*
- **Heave**
  - *amplitude*
  - *frequency*
  - *phase*
- **Sway**
  - *amplitude*
  - *frequency*
  - *phase*
- **Roll**
  - *amplitude*
  - *frequency*
  - *phase*
- **Pitch**
  - *amplitude*
  - *frequency*
  - *phase*
- **Yaw**
  - *amplitude*
  - *frequency*
  - *phase*

The constant Sea_State_Table provides the following Output Data Flow:

(Sea_State_Table --(**2c**)--> Input_Parameter_Table)

This data consists of a single set of motion parameter constants to be used by the attitude, position, and velocity processes of the motion simulator. The parameters reflect the initial default set or the most recent user command of SELECT SEASTATE.

2.1.3. Input Parameter Table

The Input_Parameter_Table contains starting simulation values to be employed as parameter inputs to the motion simulation calculations. Note that the set of input parameters builds upon the initial sea-state and scenario and incorporates other variables which record state transitions resulting from user commanded changes in navigation. The input parameters are also displayed at the INS screen periodic display window:

- **Surge**
  - *amplitude*
  - *frequency*
  - *phase*
- **Heave**
  - *amplitude*
  - *frequency*
  - *phase*
- **Sway**
  - *amplitude*
  - *frequency*
  - *phase*
- **Roll**
  - *amplitude*
  - *frequency*
  - *phase*
- **Pitch**
  - *amplitude*
  - *frequency*
  - *phase*
- **Yaw**
  - *amplitude*
  - *frequency*
  - *phase*
- **Lever Arm Constants**
  - *Forward*
  - *Right*
  - *Down*
- **Ship List**
- **Ship Trim**
- **Ocean Current**
  - *East*
  - *North*
- **Ship Course**
  - *New Ship Course*
  - *Turn Rate*
  - *New Ship Direction*
- **Ship Speed**
  - *New Ship Speed*
  - *Speed Change Period*

2.1.4. Simulation Results Table

The Simulation_Results_Table contains time-dependent variables produced by the motion simulation calculations. The results will be used in building the attitude and navigation periodic data messages. The results are also used to recalculate simulation results in the next update period, and are additionally displayed at the INS screen periodic display window. Note that the ship motion parameters *Pitch, Roll,* and *Yaw* are now calculated (discrete) values:

- **Attitude**
  - *Heading*
  - *Pitch*
  - *Roll*
  - *Yaw*
  - *Heading Rate*
  - *Pitch Rate*
  - *Roll Rate*
  - *Yaw Rate*
- **Displacement**
  - *Surge*
  - *Heave*
  - *Sway*
- **Velocity**
  - *North*
  - *East*
  - *Vertical*
  - *Velocity Integral North*
  - *Velocity Integral East*
- **Position**
  - *Latitude*
  - *Longitude*
- **Speed**
- **Speed Time**
- **Speed State**
  - **{***Static* **or** *Changing* **or** *Requesting_Change***}**
- **Course**
- **Course Time**
- **Course State**
  - **{***Static* **or** *Changing* **or** *Requesting_Change***}**

2.1.5. System Data Table

The System_Data_Table contains system-wide constants, state variables, and operations returning data (*15,17*) such as:

Initial Time (Start GMT)
Time of Gyro Reset (TGR)
Simulation Time (Current GMT)
Time of Last Start-Of-Message (SOM GMT) signal

The System_Data_Table receives the following Input Data Flow:

(Real_Time_Clock --(*15a*)--> System_Data_Table)

This data consists of CLOCK "ticks" to be used as Simulated_Time by other processes within the INS Simulator.

2.1.6. Fault Table

The Fault_Table contains data (**4**) in the form of a multi-element list with each element structured as follows:

```
Fault
   boolean
   value
```

The 19 elemental faults include:

```
Heading, Pitch, Roll,
Heading_Rate, Pitch_Rate, Roll_Rate,
Vertical_Velocity, East_Velocity, North_Velocity,
Speed, Latitude, Longitude, East_Current, North_Current,
TGR, Current_GMT, SOM_GMT, Distance_North, Distance_East
```

2.1.7. Keyboard Input Queue

The Keyboard_Input_Queue contains up to 128 characters from the operator input stream (**1a**) destined to be echoed at the terminal screen command window.

2.1.8. Output Message Buffers Queue

The Output_Message_Buffers_Queue contains INS output periodic messages for transmission to the external computer system, with any specified faults also reflected in the data.

2.1.9. Pending Alerts Queue

The Pending_Alerts_Queue has up to 50 entries in order of priority. Each entry consists of:

```
Alert identification   (importance ordered)
Time at which alert was issued
```

2.2. Data Transformations

Each of the data flows and data transforms depicted in Figure 2-2 are briefly described below. Detailed requirements can be obtained from [Landherr 89]. Note that the data flow numbers correspond to those given in Figure 2-2.

2.2.1. Process Keyboard Commands

2.2.1.1. Data Flow Inputs

(Keyboard_Input_Queue --(**1b**)--> Process_Keyboard_Commands)

This data flow is simply a queued stream of ASCII characters typed by the operator on the keyboard.

(Input_Parameter_Table --(**2e**)--> Process_Keyboard_Commands)

This data flow consists of the feedback of the numerical values of input parameters selected by the operator in a SHOW PARAMETER command.

2.2.1.2. Data Transform

The Process_Keyboard_Commands data transform gathers incoming character strings from the keyboard input queue, echoes them to the screen, assembles them into command strings, parses the strings into commands, interprets the command syntax, and performs the actions specified by the commands.

2.2.1.3. Data Flow Outputs

(Process_Keyboard_Commands --(**2d**)--> Input_Parameter_Table)

This data flow consists of individual parameter values supplied by the operator as in the SET PARAMETER, INCREASE SPEED TO, SELECT SCENARIO, or SELECT SEASTATE commands.

(Process_Keyboard_Commands --(**4a**)--> Fault_Table)

This data flow consists of individual numerical values supplied by the operator in FAULT commands.

(Process_Keyboard_Commands --(**11a**)--> Pending_Alerts_Queue)

This data flow consists of individual numerical values representing simulation alerts.

(Process_Keyboard_Commands --(**9**)--> Process_Command_Window)

This data flow consists of isolated characters echoing the keystrokes of the user when typing commands at the keyboard. The data may also be a string of control characters to edit the command line (e.g., backspace).

(Process_Keyboard_Commands --(**10**)--> Process_Command_Window)

This data flow consists of a string image of the numerical value of a SHOW PARAMETER result (see **2e** above) intended for display at the screen command window. The parameter string will be concatenated after the last echoed command character (**9** above).

2.2.2. Update Attitude and Heading

2.2.2.1. Data Flow Inputs

(Input_Parameter_Table --(**2f**)--> Update_Attitude_and_Heading)

This data flow consists of the various ship simulation parameters, such as SEASTATE and SCENARIO, required by the motion simulator's update attitude and heading process.

(System_Data_Table --(**15b,17a**)--> Update_Attitude_and_Heading)

This data flow consists of the current value of simulation time (elapsed interval) required by the motion simulator's update attitude and heading process plus any required simulation constants (note that the mechanism for updating system time is target dependent and will involve interfacing to the system clock or a separate real-time clock.)

(Simulation_Results_Table --(**3f**)--> Update_Attitude_and_Heading)

This data flow consists of numerical values for simulation results previously calculated by the motion simulator data transforms and required as an intermediate value by the transform for a succeeding update.

2.2.2.2. Data Transform

The Update_Attitude_and_Heading data transform periodically calculates the simulated ship rotational and translational motions. The data transform uses values from the system data and input parameter tables to calculate new pitch, roll, yaw, and heading values for entry into the simulation results table. The data transform also uses values previously calculated by the Update_Velocity data transform (ship's course), from the results table, as part of the update process.

2.2.2.3. Data Flow Outputs

(Update_Attitude_and_Heading --(**3a**)--> Simulation_Results_Table)

This data flow consists of the numerical values for simulation results (*ship's heading, pitch, roll,* and *yaw* with associated change *rates*) calculated by the Update Attitude_and_Heading process of the motion simulator.

2.2.3. Update Position

2.2.3.1. Data Flow Inputs

(Input_Parameter_Table --(**2g**)--> Update_Position)

This data flow consists of the various ship simulation parameters, such as SEASTATE and SCENARIO, required by the motion simulator’s update position process.

(System_Data_Table --(**15c,17b**)--> Update_Position)

This data flow consists of the current value of simulation time (elapsed interval) required by the motion simulator’s update position process plus any required simulation constants.

(Simulation Results Table --(**3g**)--> Update Position)

This data flow consists of numerical values for simulation results previously calculated by the motion simulator processes Update_Attitude_and_Heading and Update_Position, and required as an intermediate value by the data transform for a succeeding update.

2.2.3.2. Data Transform

The Update_Position data transform periodically calculates the simulated ship motion. The transform uses values from the System_Data_Table and Input_Parameter_Table to calculate new values of *latitude* and *longitude* for entry into the Simulation_Results_Table. In addition, the data transform uses previously calculated values, by Update_Attitude_and_Heading, Update_Position, and Update_Velocity, from the Simulation_Results_Table as part of the update process.

2.2.3.3. Data Flow Outputs

(Update_Position --(**3b**)--> Simulation_Results_Table)

This data flow consists of the numerical values for simulation results calculated by the update position process of the motion simulator.

2.2.4. Update Velocity

2.2.4.1. Data Flow Inputs

(Input_Parameter_Table --(**2h**)--> Update_Velocity)

This data flow consists of the various ship simulation parameters, such as SEASTATE and SCENARIO, required by the motion simulator’s update velocity process.

(System_Data_Table --(**15d,17c**)--> Update_Velocity)

This data flow consists of the current value of simulation time (elapsed interval) required by the motion simulator's velocity subprocess plus any required simulation constants.

(Simulation Results Table --(**3h**)--> Update Velocity)

This data flow consists of numerical values previously calculated by the motion simulator processes and required as an intermediate value by the transform for a succeeding update.

2.2.4.2. Data Transform

The Update_Velocity data transform periodically calculates the simulated ship motion. The data transform uses values from the System_Data_Table and the Input_Parameter_Table to calculate new values for entry into the Simulation_Results_Table. In addition, the data transform uses previously calculated values, by Update_Attitude_and_Heading, and Update_Velocity data transforms, from the Simulation_Results_Table as part of the update process.

2.2.4.3. Data Flow Outputs

(Update_Velocity --(**3c**)--> Simulation_Results_Table)

This data flow consists of the numerical values calculated by the update velocity process of the motion simulator.

2.2.5. Encode Attitude Messages

2.2.5.1. Data Flow Inputs

(Fault_Table --(**4b**)--> Encode_Attitude_Messages)

This data flow consists of fault values to be inserted into motion simulator attitude periodic output data messages.

(System_Data_Table --(**15e**)--> Encode_Attitude_Messages)

This data flow consists of the current value of GMT and TGR required by the Encode_Attitude_Messages data transform.

(Simulation_Results_Table --(**3d**)--> Encode_Attitude_Messages)

This data flow consists of the numerical values that are required to assemble the attitude periodic data messages.

2.2.5.2. Data Transform

The Encode_Attitude_Messages data transform assembles data messages into the format specified in [NAVSEA 82], overwriting message fields with any values currently in the fault buffer.

2.2.5.3. Data Flow Outputs

(Encode_Attitude_Messages --(**5a**)--> Output_Message_Buffers_Queue)

This data flow consists of Attitude periodic messages.

2.2.6. Encode Navigation Messages

2.2.6.1. Data Flow Inputs

(Fault_Table --(**4c**)--> Encode_Navigation_Messages)

This data flow consists of fault values to be inserted into motion simulator navigation periodic output data messages.

(System_Data_Table --(**15f**)--> Encode_Navigation_Messages)

This data flow consists of the current value of GMT and TGR required by the encode navigation messages data transform.

(Simulation_Results_Table --(**3e**)--> Encode_Attitude_Messages)

This data flow consists of the numerical values that are required to assemble the navigation periodic data messages.

2.2.6.2. Data Transform

The encode navigation messages transform assembles data messages into the format specified in [NAVSEA 82], overwriting message fields with any values currently in the fault buffer.

2.2.6.3. Data Flow Outputs

(Encode_Navigation_Messages --(**5b**)--> Output_Message_Buffers_Queue)

This data flow consists of navigation periodic messages.

2.2.7. Process Comms Link

2.2.7.1. Data Flow Inputs

(Output_Message_Buffers_Queue --(**5c**)--> Process_Comms_Link)

This data flow consists of complete, formatted, attitude and navigation periodic data messages, possibly reflecting injected fault values.

(EC Interface --(**7**)--> Process_Comms_Link)

This data flow consists of external function codes and blocks of message words.

2.2.7.2. Data Transform

The Process Comms Link transform performs the following functions:

- communicates with the external computer system (EC) via a defined [Meyers 88b] message transfer protocol over the [NAVSEA 82] NTDS Parallel Interface.
- receives input messages from the external computer and passes them to the message validator
- fetches messages from the output message buffer and sends them to the external computer
- generates alerts to be issued to the screen when certain interface conditions are detected

2.2.7.3. Data Flow Outputs

(Process_Comms_Link --(**6**)--> EC_Interface)

This data flow consists of external function *(EF)* codes and blocks of message words. *External functions are communications protocol signals; their sole purpose is to govern the orderly flow and progression of data I/O.*

(Process_Comms_Link --(**8**)--> Validate_Messages)

This data flow consists of certain *fields* of the input messages which must be *validated*. *A field is an isolated unit of data, encompassing one or more bits, with a specified location within the prescribed format of a message. Validation of a field implies a test for legality based on some discrimination technique prescribed by the communications protocol.*

(Process_Comms_Link --(**11b**)--> Pending_Alerts_Queue)

This data flow consists of packets of coded data identifying an alert and specifying the time of occurrence of the condition that triggered the alert.

(Process_Comms_Link --(**12**)--> Process_System_Status_Window)

This data flow consists of a single coded value which specifies the newly changed state of the communications interface (i.e., up or down).

2.2.8. Validate Messages

2.2.8.1. Data Flow Inputs

(Process_Comms_Link --(**8**)--> Validate_Messages)

See Process_Comms_Link subsection.

2.2.8.2. Data Transform

The validate messages transform checks certain fields of each input message. If an error is found, an alert is issued.

2.2.8.3. Data Flow Outputs

(Validate_Messages --(**11c**)--> Pending_Alerts_Queue)

See Process_Comms_Link subsection.

2.2.9. Update Periodic Display

2.2.9.1. Data Flow Inputs

(Input_Parameter_Table --(**2i**)--> Update_Periodic_Display)

This data flow consists of simulation values reflecting the motion simulator parameters to be used in update processing.

(Simulation_Results_Table --(**3i**)--> Update_Periodic_Display)

This data flow consists of simulation values reflecting the motion simulator results of update processing.

(System_Data_Table --(**15g**)--> Update_Periodic_Display)

This data flow consists of the current value of Greenwich Mean Time (GMT) and Time of Gyro Reset (TGR) required by the Update_Periodic_Display data transform.

2.2.9.2. Data Transform

The Update_Periodic_Display data transform is a periodic process which gathers data from both the parameter and results table for eventual display of the simulation state on the periodic window of the operator screen.

2.2.9.3. Data Flow Outputs

(Update_Periodic_Display --(**16**)--> Process_Periodic_Window)

This data flow consists of simulation state values from the Input_Parameter_Table, the Simulation_Results_Table, and the System_Data_Table.

2.2.10. Process Command Window

2.2.10.1. Data Flow Inputs

(Process_Keyboard_Commands --(**9**)--> Process_Command_Window)

See Process_Keyboard_Commands subsection.

(Process_Keyboard_Commands --(**10**)--> Process_Command_Window)

See Process_Keyboard_Commands subsection.

2.2.10.2. Data Transform

The Process_Command_Window transform accepts characters from the Process_Keyboard_Commands data transform (including line characters) and arranges for the display of the (edited) command line by formatting standard packets for the Control_Screen data transform to be appended to the command line for window display.

2.2.10.3. Data Flow Outputs

(Process_Command_Window --(**13a**)--> Control_Screen)

This data flow consists of packets of coded screen coordinates and text strings that define data to be displayed on the console screen.

2.2.11. Process Periodic Window

2.2.11.1. Data Flow Inputs

(Update_Periodic_Display --(**16**)--> Process_Periodic_Window)

See Process_Periodic_Display subsection.

2.2.11.2. Data Transform

The Process_Periodic_Window transform formats standard packets, from values obtained from the Update_Periodic_Display, for output in the periodic display window.

2.2.11.3. Data Flow Outputs

(Process_Periodic_Window --(**13b**)--> Control_Screen)

See Process_Command_Window subsection.

2.2.12. Process Alert Window

2.2.12.1. Data Flow Inputs

(Pending_Alerts_Queue --(**11d**)--> Process_Alert_Window)

This data flow consists of alert display strings.

2.2.12.2. Data Transform

The Process_Alert_Window transform accepts individual alerts from the pending alerts queue and arranges for them to be displayed in the alert window by formatting standard packets for the Control_Screen data transform.

2.2.12.3. Data Flow Outputs

(Process_Alert_Window --(**13d**)-> Control_Screen)

See Process_Command_Window subsection.

2.2.13. Process System Status Window

2.2.13.1. Data Flow Inputs

(Process_Comms_Link --(**12**)--> Process_System_Status_Window)

See Process_Comms_Link subsection.

2.2.13.2. Data Transform

The Process_System_Status_Window data transform accepts data from the Process_Comms_Link data transform which represents changes in the communications state and arranges for these values to be displayed in the system state window by formatting standard packets for the Control_Screen data transform.

2.2.13.3. Data Flow Outputs

(Process_System_Status_Window --(**13c**)--> Control_Screen)

See Process_Command_Window subsection.

2.2.14. Control Screen

2.2.14.1. Data Flow Inputs

(Process_Command_Window --(**13a**)--> Control_Screen)

See Process_Command_Window subsection.

(Process_Periodic_Window --(**13b**)--> Control_Screen)

See Process Command Window subsection.

(Process_System_Status_Window --(**13c**)--> Control_Screen)

See Process_System_Status_Window subsection.

(Process_Alert_Window --(**13d**)--> Control_Screen)

See Process_Alert_Window subsection.

2.2.14.2. Data Transform

The Control_Screen data transform accepts packets from the previous four window processors, which contain a string of display characters plus an embedded escape sequence that specifies screen location, and writes the packet within the designated window.

2.2.14.3. Data Flow Outputs

(Control_Screen --(**14**)--> Screen)

This data flow consists of a stream of ASCII characters, including control characters to position the cursor.

3. Concurrency and Control

The purpose of this chapter is to provide the reader with an appropriate context for the development of a common (i.e., target independent) real-time software design for the INS. A starting point will be the data flow structure developed in the preceding chapter (see Figure 2-2). The approach will expose concurrent behavior as early in the problem analysis as possible. The next section employs a strategy for devising the real-time architecture of the INS from the data flow analysis. A typical first step involves creating a set of criteria for identifying a task set within the Low-Level Data Flow Diagram (subsequently referred to as the DFD). The succeeding steps apply the criteria to the INS analysis (Figure 2-2) to render a design which satisfies the real-time requirements of the system.

3.1. Criteria for Converting Data Flows to Tasks

When considering possible concurrency, [Gomaa 84] and [Nielson 88] propose a similar rationale for decomposing and grouping data transforms and stores. A heuristic approach is adopted employing rules-of-thumb to decide which data transforms and which data stores can properly be considered for division into separate threads of execution. With a data transform, the requirement for independent (i.e., asynchronous) operation, whether it be periodic or sporadic in nature, is the motivating factor for creation of a task. However, not all tasks can be considered purely independent; when real-time synchronization is required, such as with case sharing resources, then a mediating agent must be created to provide this service. Therefore, the sharing of data stores between client tasks must occur through the synchronized actions of a third-party task which monitors access to the data store. A clarifying point should now be stated: whereas analytical techniques (e.g., closed-form solutions are achievable) for scheduling a task set with analyzable properties, analysis does not speak to the initial derivation — its composition or numbers. These must be determined with heuristic methods. Therefore, in the context of recent experimentation at the SEI on analyzable task sets [Borger 89] the following conversion rules shall be applied:

3.1.1. Data Transforms

- Random data flows create aperiodics
- Regular data flows create periodics

Data flows between an interrupting device and a data transform can be modeled as asynchronous requests upon an aperiodic task. Data transforms which execute time-critical functions at random intervals, often in response to stimulus due to data flowing from alternative data transforms, can also be modeled as aperiodic tasks. Conversely, the repeated execution of data transforms at regular intervals is a criterion for a periodic task.

3.1.2. Data Stores

In addition to the task criteria for data transforms, real-time systems often make use of *shared variables* through data stores that provide concurrent access:

- shared resource will be converted to a server task to model concurrent synchronization.

Applying the rules-of-thumb must also allow for a rational handling of data transforms and stores which fail the test:

- sequential execution of data transforms
- temporal ordering of data transforms
- sequential access to data stores

If a data transform does not satisfy the criteria for a task then it will execute in the thread of control of another task. This can occur either of two ways: first, via a *data flow* to a called subprogram (i.e., "sequential cohesion"); second, through the combining of data transforms for time-ordered execution (referred to by Gomaa and Nielson as "temporal cohesion"). Structured design rules treat temporal behavior as inappropriate (low in cohesiveness) for module construction, but the behavior of tasks is dynamic in time rather than static; therefore temporal bindings are valid considerations. A temporal binding is useful when events occur in a predefined order, thus allowing encapsulation within a single thread of execution which is not truly data coupled. If a data store does not satisfy the criteria for a task then it will be treated as a resource characterized by sequential access.

3.2. Creating An Analyzable Task Set For The INS

The work cited in [Borger 89] offers a detailed explanation of an *analytical* approach to hard real-time scheduling founded on a partition of, and interaction between, tasks which can be modeled as periodics, aperiodics, and servers. The authors apply the principles of this method to a description and assessment of the INS task set. Rather than repeat this material, *Inertial Navigation System Simulator Program: Top-Level Design* (hereafter referred to as *Top-Level Design* will restrict itself to a discussion of how INS tasks are created by application of the criteria mentioned in Section 3.1, (based on an analysis of the INS requirements represented graphically in Figure 2-2,) and to a summary explanation of the theory and supporting principles by which a predictable analysis is possible.

Since no well-defined rules exist for mapping the task criteria to the INS analysis, techniques must be carefully crafted to make the candidate selection process understandable. First, to reduce the complexity at each design step, individual "clouds" (shaded areas in Figure 2-2) are isolated and treated as closely bound ensembles or "subsystems." This approach, while convenient from the developer's perspective (and fairly common in practice), is founded on (a) the functional cohesiveness of the constituent data transforms, and (b) a straightforward producer-consumer data flow between data transforms and stores. This is an important abstraction for identifying parallelism because the separation of concerns allows derivation of tasks in any particular subsystem to be considered for their analyzable properties alone and apart from the order in which other subsystems are treated. Yet, concurrent behavior may overlap the functional structure of subsystems. Tasks can merge functional groupings within a single thread of execution if the task criteria fails and sequential or temporal cohesion are the primary considerations. For instance, both the keyboard and screen subsystems overlap at the window processing interface. In this case, a data coupling implies sequential cohesion as the criterion. Within the EC Subsystem, Process_Comms_Link performs both sending and receiving of messages with the EC over the NTDS Parallel Interface. Since these functions cannot occur simultaneously they will be executed within a single thread and the ordering points to temporal cohesion as the primary criterion. To aid this presentation in the *Top-Level Design*, two methods have been adopted. First, an itemization and explanation of the chosen task criteria is shown for each of the DFD elements. Equivalence of tasks to data transforms and stores is identified with an "**<=>**" construction. The flows between tasks are treated as channels in a producer-consumer context for data movement. Data entering a subsystem is followed through each task, undergoing one or more transforms, until it exits the subsystem. Second, a diagram of the subsystem elements is illustrated as an expanded, or "telescoped," segment of Figure 2-2 (image reduced in the upper-left hand side of each figure) with a visual mapping of tasks to data transforms and stores. The tasks are illustrated as shaded parallelograms, a widely adopted modeling technique within the Ada literature. Each task, shown with a "clouded" center, is a paradigm of dynamic behavior. Ada tasks encompass one or more DFD elements while simultaneously connoting separate threads of concurrent execution. Ada tasks, while subject to the concurrency primitives of the language, have a predictable behavior when formed according to a restricted set of analyzable rules (discussed more fully in Section 3.3 and in detail in [Borger 89]). Employing the Ada task icon as a "black-box" with an open window into the interior is an information-hiding mechanism for depicting some internal details and delaying others for a later design decision. The DFD elements within the black-box are illustrated as being "lassoed" from within it, or looped by it, to show its capture within the thread and separation from other data transforms and stores.

In applying the criteria to the INS requirements data flow analysis, the following concurrency derivation is obtained:

3.2.1. Keyboard Tasks

As illustrated in Figure 3-1, three tasks replace related data transforms and stores:

- aperiodic task **Keyboard_ISR** <=> data transform **Keyboard** device
- server task **Input_Buffer_Monitor** <=> data store **Keyboard_Input_Queue**
- aperiodic task **Command_Processor** <=> data transform **Process_Keyboard_Commands** and data transform **Process_Command_Window**

The Keyboard_ISR is a producer which places characters **(1a)** received from the keyboard into the queue of the Input_Buffer_Monitor. Note that the Process_Keyboard_Commands data transform and the Process_Command_Window data transform execute within the thread of the Command_Processor task. The Command_Processor task is a consumer of queued characters **(1b)** from the Input_Buffer_Monitor. The Command_Processor also produces operator entered simulation parameters **(2d)** for consumption by the Input_Parameter_Table and consumes simulation parameters **(2e)** produced by the Input_Parameter_Table as a result of the SHOW command for example. The Command_Processor produces printable characters **(9)** for a transform into single character display packets **(13a)** for consumption by the Control_Screen data transform at the command window. The Command_Processor also produces simulation parameter character strings **(10)** as display packets **(13a)** for consumption by the Control_Screen at the command window.

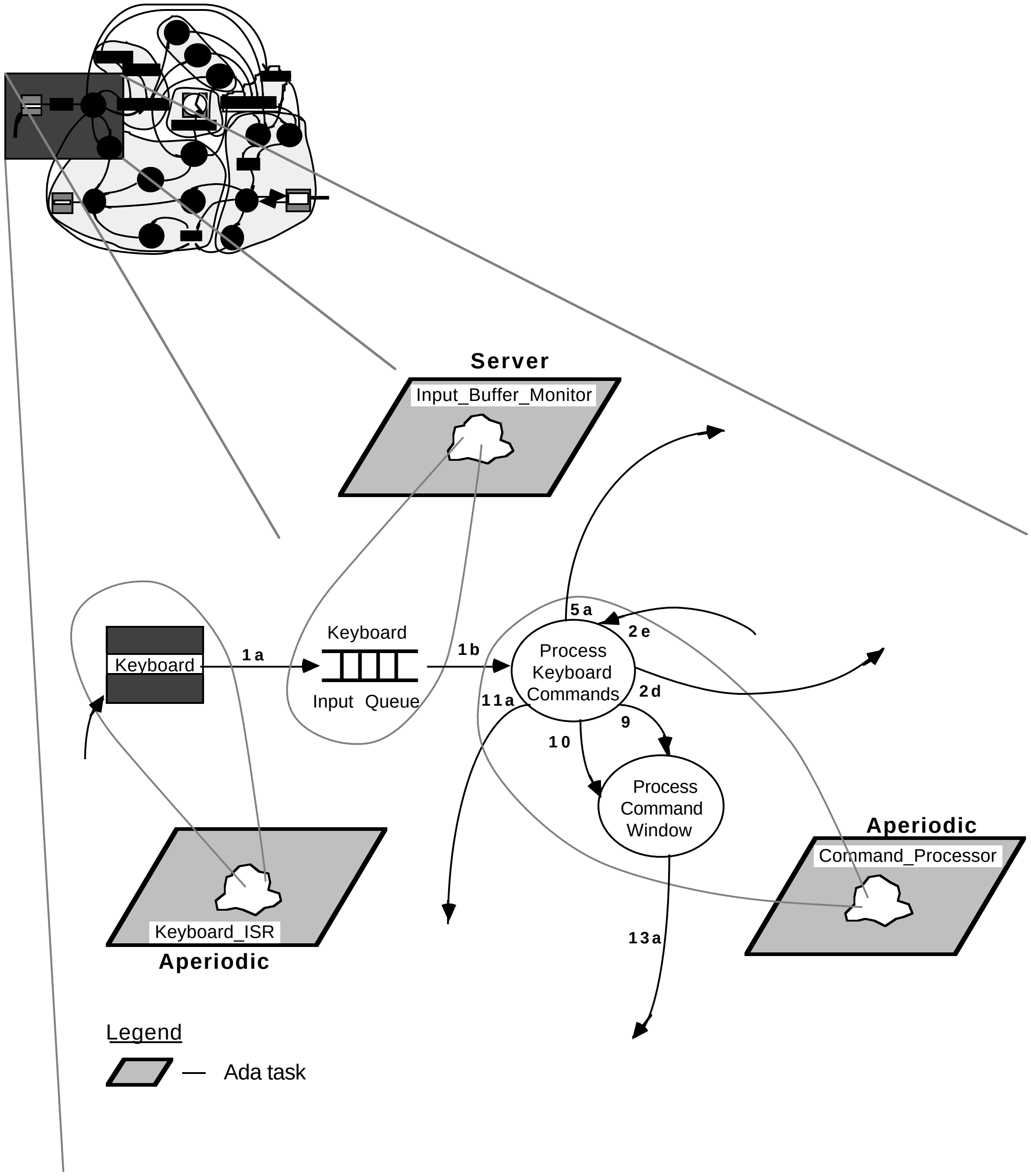


Figure 3-1: Keyboard Related Tasks

3.2.2. Screen Tasks

As illustrated in Figure 3-2, four tasks replace screen related data transforms and data stores:

- aperiodic task **Screen_ISR** <=> data transform **Screen** device
- server task **Alerts_Monitor** <=> data store **Pending_Alerts_Queue** and data transform **Process_Alert_Window**
- aperiodic task **Screen_Controller** <=> data transform **Control_Screen**
- periodic task **Periodic_Display_Updater** <=> data transform **Update_Periodic_Display** and data transform **Process_Periodic_Window**

Note that the transforms Update_Periodic_Display and Process_Periodic_Window execute sequentially every 1024 milliseconds within the thread of the Periodic_Display_Updater Task. The Periodic_Display_Updater consumes data in the form of simulation parameters (2i) from the Input_Parameter_Table, simulation results **(3i)** from the Simulation_Results_Table, and GMT plus TGR **(15g)**from the System_Data_Table. The Periodic_Display_Updater produces a simulation state **(16)** which it transforms into a display packet **(13b)** for the periodic window. The Alerts_Monitor synchronizes concurrent access to the Pending_Alerts_Queue, acting as a shared resource for multiple clients. Alerts **(11a, 11b, 11c)** are queued on arrival and dequeued **(11d)** for transform to display packets **(13d)** which are consumed by the Screen_Controller for display in the alert window. The Screen_Controller consumes display packets **(13a, 13b, 13c, 13d)** from a number of sources for output of screen display characters **(14)** at designated windows.

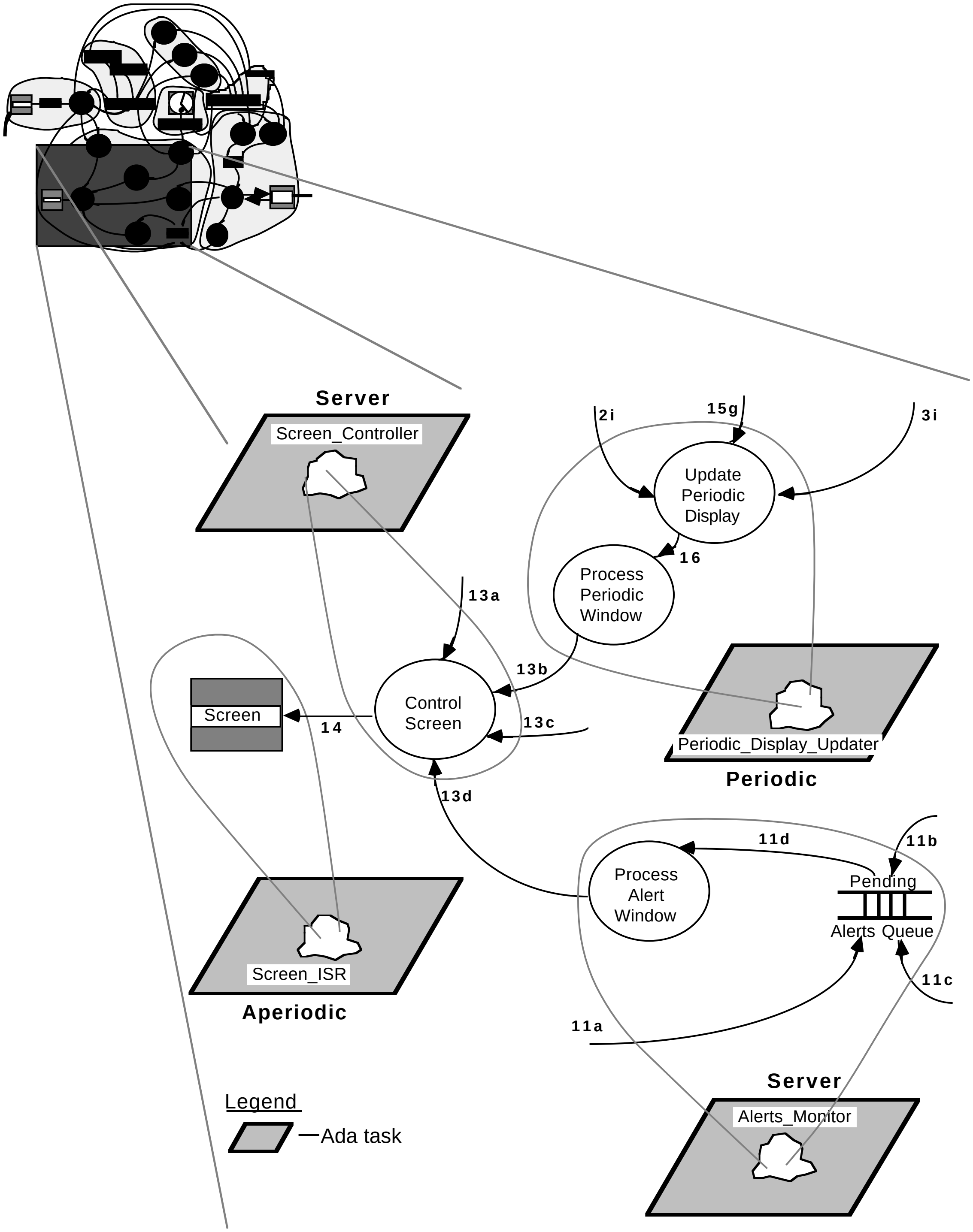


Figure 3-2: Screen Related Tasks

3.2.3. Parallel Interface Tasks

As illustrated in Figure 3-3, four tasks replace data transforms and data stores involved in communications with the external computer system over the [NAVSEA 82] NTDS Parallel Interface:

- aperiodic task **Comms_ISR** <=> data transform **EC_Interface** device
- server task **Comms_Controller** <=> data transform **Process_Comms_Link** and data transform **Process_System_Status_Window** and data transform **Validate_Messages** and data store **Output_Message_Buffers_Queue**
- periodic task **Attitude_Message_Sender** <=> data transform **Attitude_Message_Sender**
- periodic task **Navigation_Message_Sender** <=> data transform **Attitude_Message_Sender**

The Attitude_Message_Sender task executes periodically at a 16 Hz rate consuming simulation time **(15e)** from the System_Data_Table, simulation results **(3d)** from the Simulation_Results_Table, and fault values **(5b)** from the Fault_Table. The Attitude_Message_Sender transforms these inputs into encoded ship’s attitude state information which it produces as a simulation results message **(4a)** for consumption by the Comms_Controller. The Navigation_Message_Sender executes periodically at a 1 Hz rate consuming simulation time **(15f)** from the System_Data_Table, simulation results **(3e)** from the Simulation_Results_Table, and fault values **(5c)** from the Fault_Table. The Navigation_Message_Sender transforms these inputs into encoded ship’s navigation state information which it produces as a simulation results message **(4b)** for consumption by the Comms_Controller. The Comms_Controller consumes simulation results messages **(5a, 5b)**, queuing them internally in the Output_Message_Buffers_Queue, which it retrieves **(5c)** and produces as periodic output messages **(6)** for consumption by the EC according to a synchronization protocol (handshake) described in [NAVSEA 82] and event signaled by the Comms_ISR (not acting as a data conduit). The Comms_Controller consumes periodic input messages **(7)** from the EC according to the same protocol with a waiting for event performed by the Comms_ISR. In addition, the Comms_Controller produces a communications state **(12)** which is converted into a screen display packet **(13)** for consumption by the Screen_Controller with eventual output to a window on the screen. The Comms_Controller also produces simulation alerts **(11)** (based on detected event changes during message communications) and input message values **(8)**, which are validated, thus

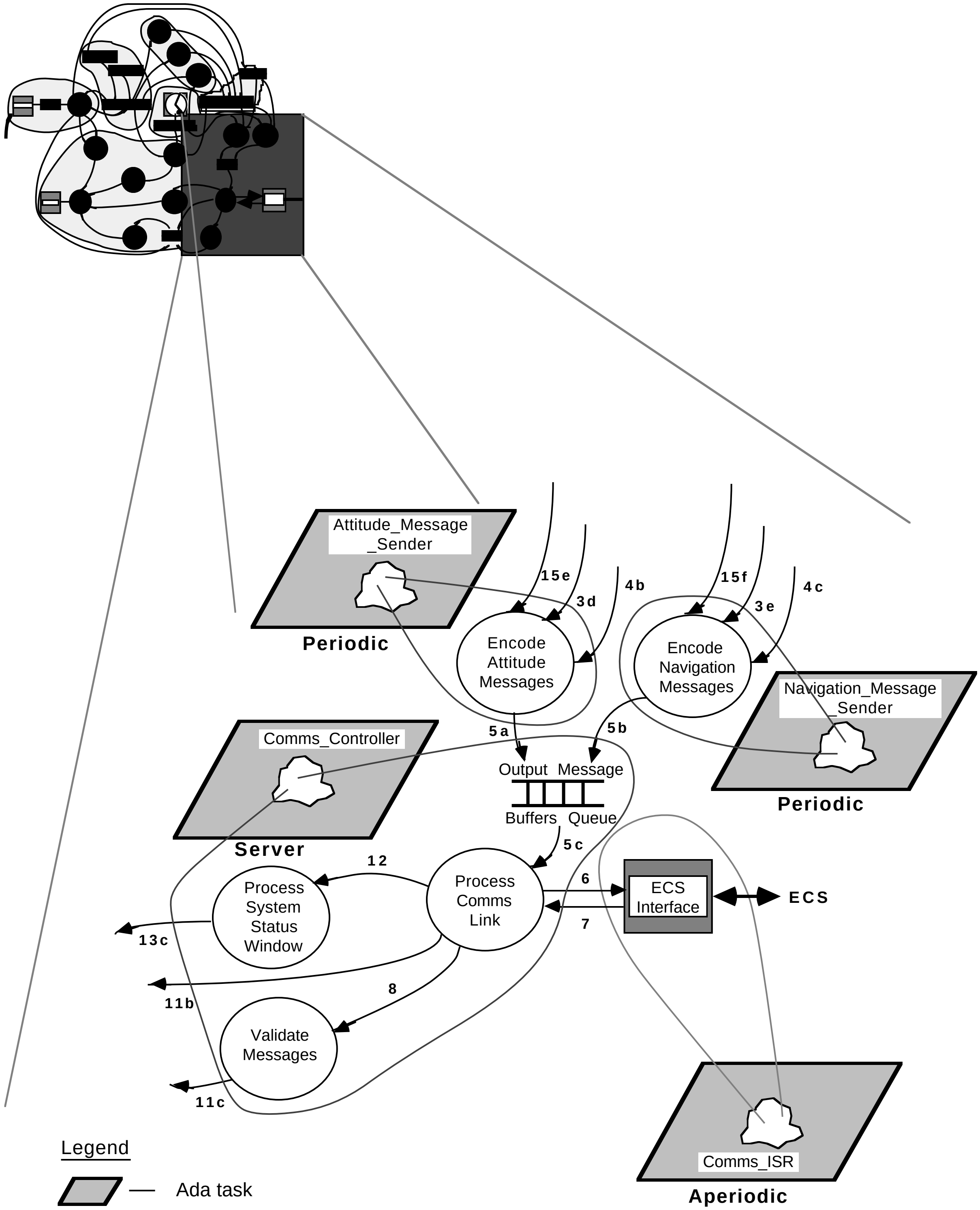


Figure 3-3: Parallel Interface Related Tasks

producing simulation alerts **(11)** when an error is detected. Simulation alerts are consumed by the Alerts_Monitor.

3.2.4. Motion Simulator Tasks

As illustrated in Figure 3-4, three tasks replace data transforms related to simulating the motion of the ship:

- periodic task **Ship_Attitude_Updater** <=> data transform is **Update_Attitude_and_Heading**
- periodic task **Ship_Position_Updater** <=> data transform **Update_Position**
- periodic task **Ship_Velocity_Updater** <=> data transform **Update_Velocity**

The Ship_Attitude_Updater executes at a periodic rate of close to 400 Hz, consuming simulation parameters **(2f)** from the Input_Parameter_Table, calculating new values for ship's attitude, and producing simulation results **(3a)** for consumption by the Simulation_Results_Table.

The Ship_Position_Updater executes at a periodic rate of 0.8 Hz, consuming input parameters **(2g)**, and intermediate results **(3g)**, calculating, and producing new motion simulator values **(3b)** for consumption by the Results_Table_Monitor.

The Ship_Velocity_Updater executes at a periodic rate of 24 Hz, consuming input parameters **(2h)**, and intermediate results **(3h)**, calculating, and producing new motion simulator values **(3c)** for consumption by the Results_Table_Monitor.

3.2.5. Input Parameter Table Task

As illustrated in Figure 3-5, one task replaces the shared simulation input data store:

- server task **Parm_Table_Monitor** <=> data store **Input_Parameter_Table**

The Input_Parameter_Table data store is replaced by a single server task, the Parm_Table_Monitor, acting as a shared resource for multiple clients.

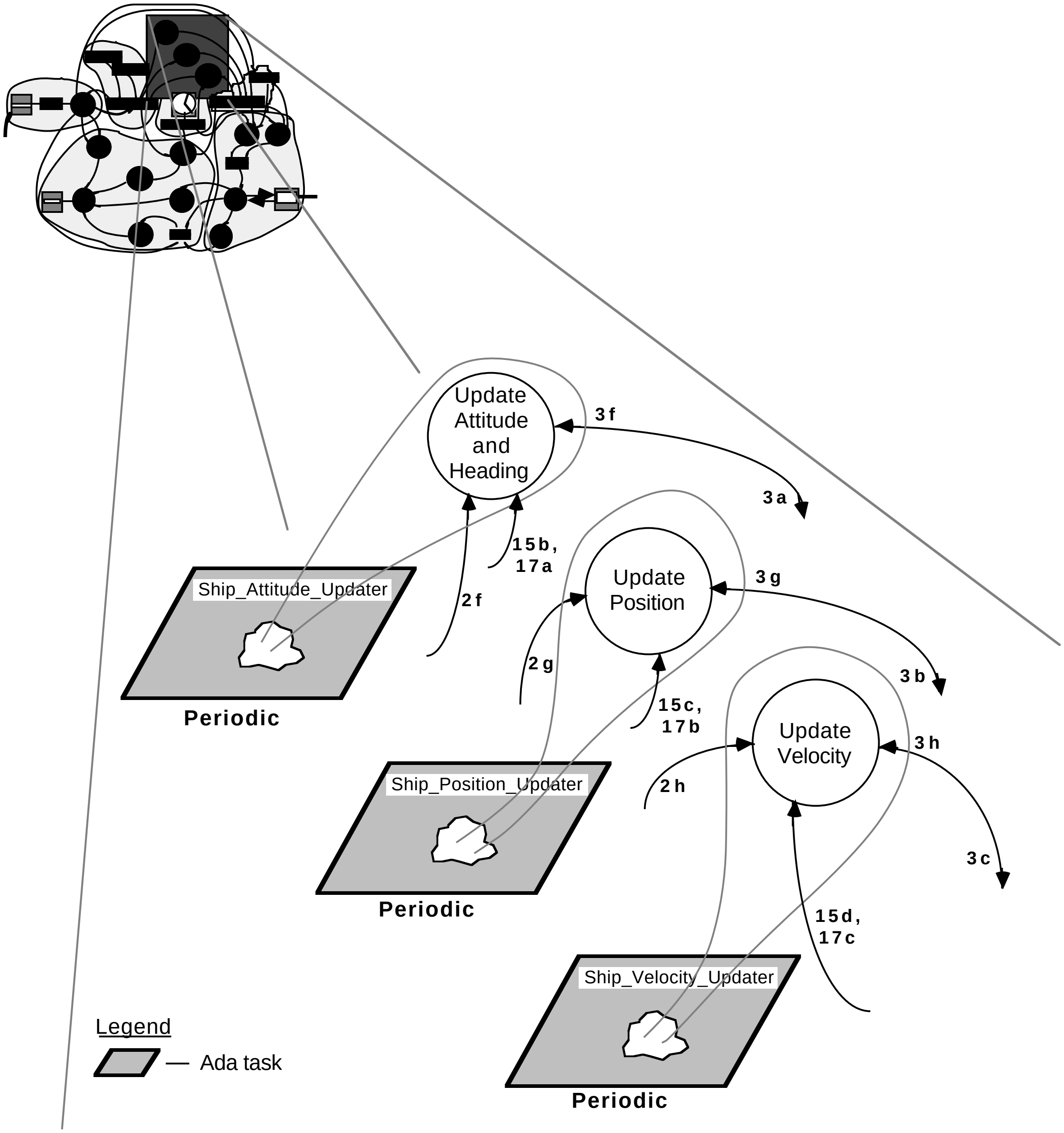


Figure 3-4: Motion Simulator Related Tasks

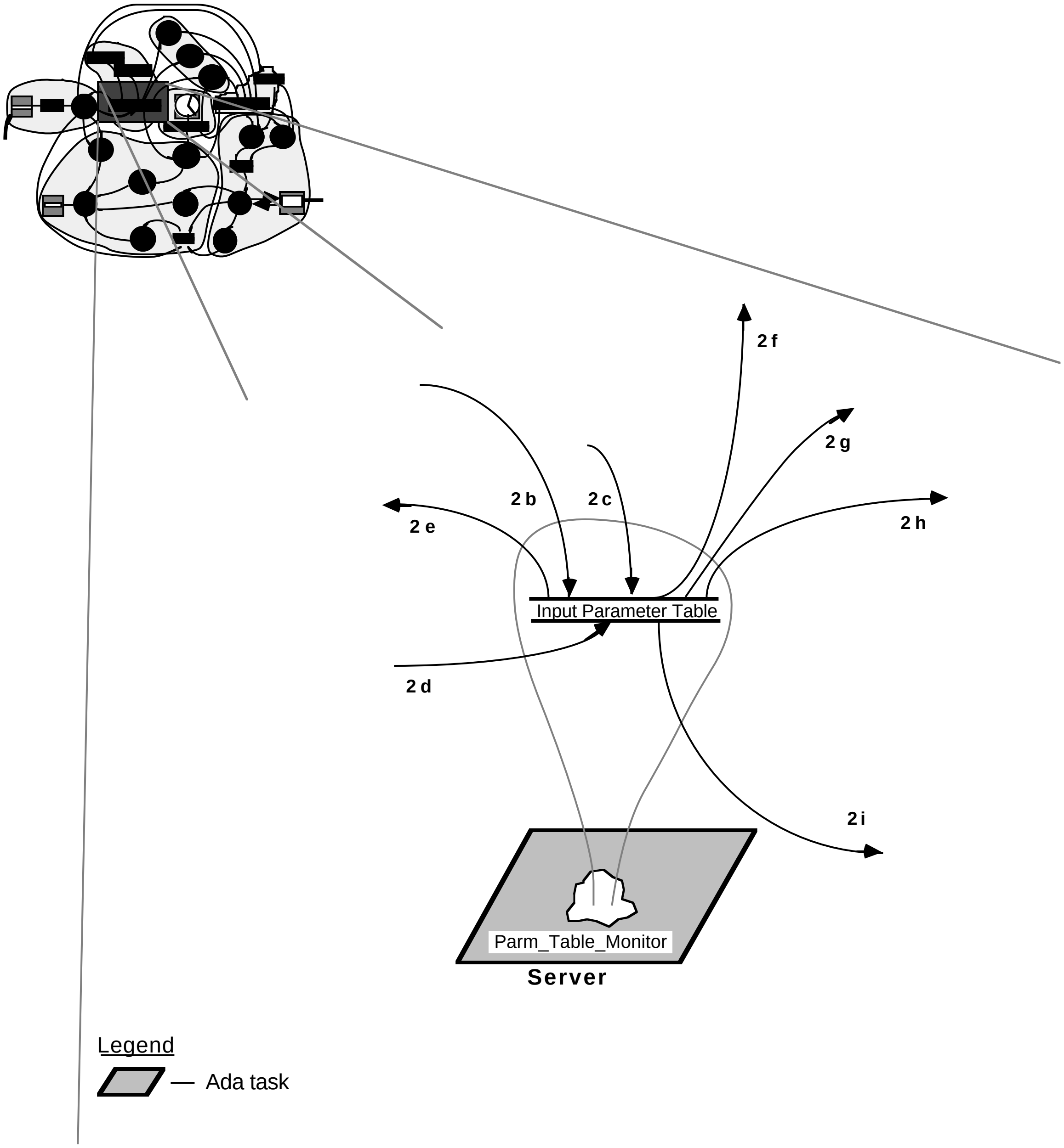


Figure 3-5: Input Parameter Table Task

3.2.6. Simulation Results Table Task

As illustrated in Figure 3-6, one task replaces the shared simulation output data store:

- server task **Results_Table_Monitor** <=> data store **Simulation_Results_Table**

The Simulation_Results_Table data store is replaced by a single **server** task, the results table monitor, acting as a shared resource for multiple clients.

3.2.7. Remaining Data Stores

The Scenario_Table, Sea_State_Table, System_Data_Table, and Fault_Table are protected by access protocols (conditional state dependencies) rather than by a server task. Thus, each of the above tables is referenced and/or updated within the thread of the accessing task.

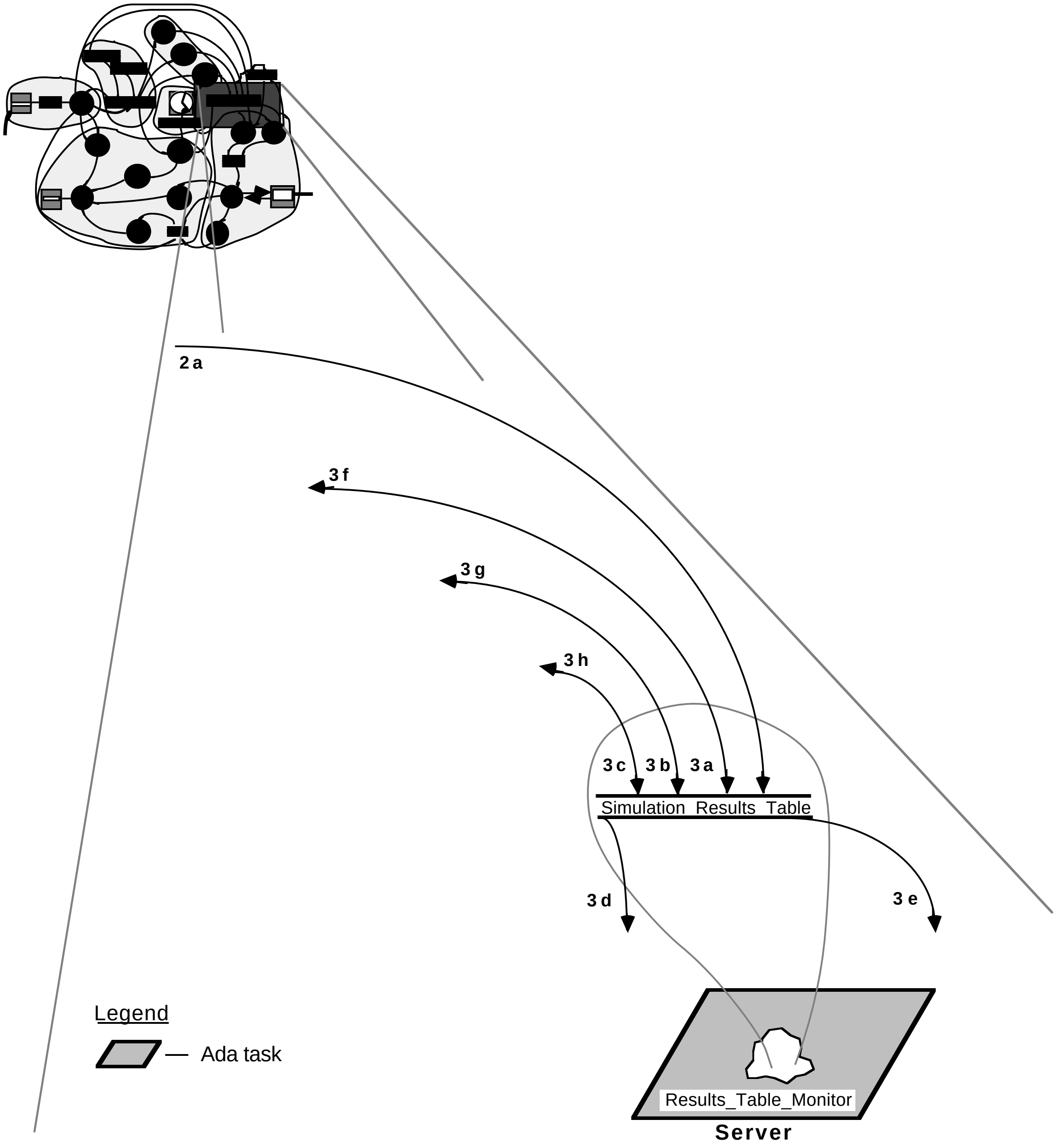


Figure 3-6: Simulation results table task

3.3. A Schedulable Real-Time Architecture for the INS

The tasking structure derived in the previous section is applied within the framework of an analytical concurrency model based on the Rate Monotonic Scheduling (RMS) algorithm. RMS was introduced in a seminal paper by Liu and Layland [Liu 73] in which the authors showed that the scheduling of tasks can be reliably predicted. RMS is essentially a toolkit for building real-time software which has a basis of underlying periodicity. In [Borger 89], several techniques expanding on RMS are applied to facilitate a scheduling of the INS Simulator. To give the reader a better background for this form of analysis the following section summarizes principle aspects of the work.

3.3.1. Software Engineering Aspects

Since our principal concern is with the ability of a real-time INS task set to meet its specified deadlines, then, a scheduling of the tasks which makes this feasible is our primary objective. We employ RMS for a number of reasons, some of which are related to (1) limitations within either the Ada tasking paradigm or implementations (language rules for concurrency, the *rendezvous* model, timing granularity, drift, and jitter), (2) the stringency of the real-time regime in which the INS must perform (hard and soft deadlines, device interactions, performance reliability and maintainability), and (3) because of the advantages provided by RMS compared to the inflexibility of alternative approaches (such as the manual overlay of major and minor execution frames in a cyclical executive).

RMS is advantageous because it offers the analytical means for scheduling a task set and is *optimal* in this regard since it obtains the most efficient use of runtime, as measured in the key resource — CPU utilization, and other successful schedulings of the task set also will be shown possible by RMS while the converse is not provably true. Of course, a feasible schedule may not exist, and RMS will make this condition known. The question remains as to what preconditions are necessary for the use of RMS? In general terms, RMS is applicable to a wide range of real-time domains intended for implementation using a non-deterministic and preemptive tasking model such as exists within Ada. Compared to the use of cyclical executives, sometimes referred to as Time Domain Multiplexing (TDM), in real-time applications, the Ada model is amenable to good software development practices which can benefit from a separation of logical and timing concerns.

3.3.2. Treating Periodics under RMS

Both Ada and RMS require task priorities to be static. For periodic tasks, RMS demands a priority ordering directly proportional to the frequency of execution. Using a brief example (examined in further detail by Borger, Klein, and Veltre), assume a task set composed of the six periodic members of the INS $\{P_1,P_2,P_3,P_4,P_5,P_6\}$ ordered by frequency, highest to lowest (and therefore also by priority, from left to right with ties broken arbitrarily). A simple check can be applied to test whether the total CPU utilization U of the set does not exceed the *worst-case* bound provided by RMS of $n(2^{1/n}–1)$ (where n equals the number of tasks) thus indicating that all deadlines will be met. Specifically, *utilization* is computed by summing up the individual contributions of each task in the periodic set as follows:

$$U = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} + \frac{C_4}{T_4} + \frac{C_5}{T_5} + \frac{C_6}{T_6}$$

where C_i is the execution time and T_i the period (reciprocal of frequency) for each task P_i. We can show that the measured value of C_1 of the first (highest frequency of 400 Hz) periodic task (Update_Ship_Attitude, see Figure 3-7) is equal to 0.5 milliseconds and that T_1 is equal to 2.56 milliseconds giving a utilization factor of 19.53%. The total utilization of the six tasks computes to 64.16%. This value must be less then or equal to $n(2^{1/n}–1)$ for the periodic set $6(2^{1/6}–1) = 0.7348$ or 73.48%, and it is. The periodic members of a task set may be harmonic (i.e., their periods are integer multiples), and this is a special case in RMS. The scheduling of a strictly periodic task set with harmonic frequencies remains feasible even at 100% CPU utilization, assuming that overhead due scheduling action is an insignificant addition to each task’s utilization factor and that all tasks are ready to run simultaneously i.e., no phase shifts. For a purely periodic and non-harmonic task set a figure of 88% is routine, but scheduling can vary from 100% converging down to 69% as the membership approaches infinity. Periodic task sets contend only with *preemption* since this is a rather common event whereby high priority tasks impede the execution of tasks having a lower priority. Furthermore, the above test does not include the effect at runtime of kernel-initiated context switching between tasks, the effects of task interaction (i.e., synchronization), nor the run time devoted to servicing aperiodic requests.

3.3.3. Treating Servers under RMS

Servers can be successfully modeled in RMS using other means which are employed in the analysis. Looking at scheduling feasibility, we can envision task-sets made up of any combination of periodics, aperiodics, and servers; these are the analyzable categories. The problem of periodics aside, the INS must quantify additional factors under RMS to achieve a feasible schedule. The first of these factors is *blocking*, whereby a high priority task is impeded in its execution by a lower priority task. Blocking is associated with shared resource contention and is modeled by a binary semaphore client/server interaction: a client task requests synchronized access from a shared resource server (task) and the critical section is entered upon gaining access. If a higher priority task requests access to the same resource it, becomes blocked by the task with the lower priority until the resource has been released. Placing an upper bound on the amount of blocking which can occur is absolutely essential for reliable scheduling. Uncontrolled blocking has been shown through investigation to be a result of *priority inversion*, a detrimental behavior of First-In-First-Out (FIFO) queuing disciplines. The task entry call in Ada is defined as FIFO, thus engendering priority inversion unless means are found to counter the effect. For obvious reasons, *mutual deadlock*, whereby two or more tasks are stalled waiting on interlocked resources, also needs to be avoided when analyzing for schedulability. The elimination of conditions leading to both deadlock and priority inversion can be realized through the application of a principle known as a *priority ceiling*. The Priority Ceiling Protocol (PCP) is a method for real-time synchronization that relies on a series of concepts:

- priority inheritance for client tasks
- priority ceiling for server tasks

Priority inheritance allows a blocking task (the server client) to inherit the highest priority of the tasks it has blocked, eliminating possible preemption by a medium priority task with a resulting further delay in execution of the blocked tasks. Also, we must restrict synchronization requests to tasks with priorities higher than that inherited by all currently preempted blocking tasks— a *priority ceiling*. Preemption of a blocking (client) task remains possible, but for synchronization a task’s priority must be higher than that of any currently preempted server in the system. This total ordering of priorities guarantees a bound on blocking *since a high priority task can at most be blocked by one lower priority task.* Note that while a blocking task executes at a priority at least equal to that of the highest blocked task, Ada insures that the resource server executes at the higher of the two priorities during rendezvous. The *priority ceiling* of a server task then simply becomes the highest priority of any client task. The identified blocking factors must then be included in the formulation to obtain a total task set utilization. Each periodic task P_i must be tested in isolation with an inequality based on its own utilization, preemption by higher priority tasks (in the case of P_1 there are none), and blocking due to lower priority tasks or other factors (where, except for the last inequality test, it represents the worst-case blocking time). Each term on the left hand side of an inequality must be within the bound on the right hand side to guarantee schedulability of the entire task set. We can summarize the schedulability test with the following general inequality:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} + max\left(\frac{B_i}{T_i}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{1/n}-1)$$

where B_i is the blocking time for each task P_i. Having now dealt with resource contention, other contributors to blocking remain (such as system *interrupts*) and these can also be characterized using analytical techniques based on RMS.

3.3.4. Treating Aperiodics Under RMS

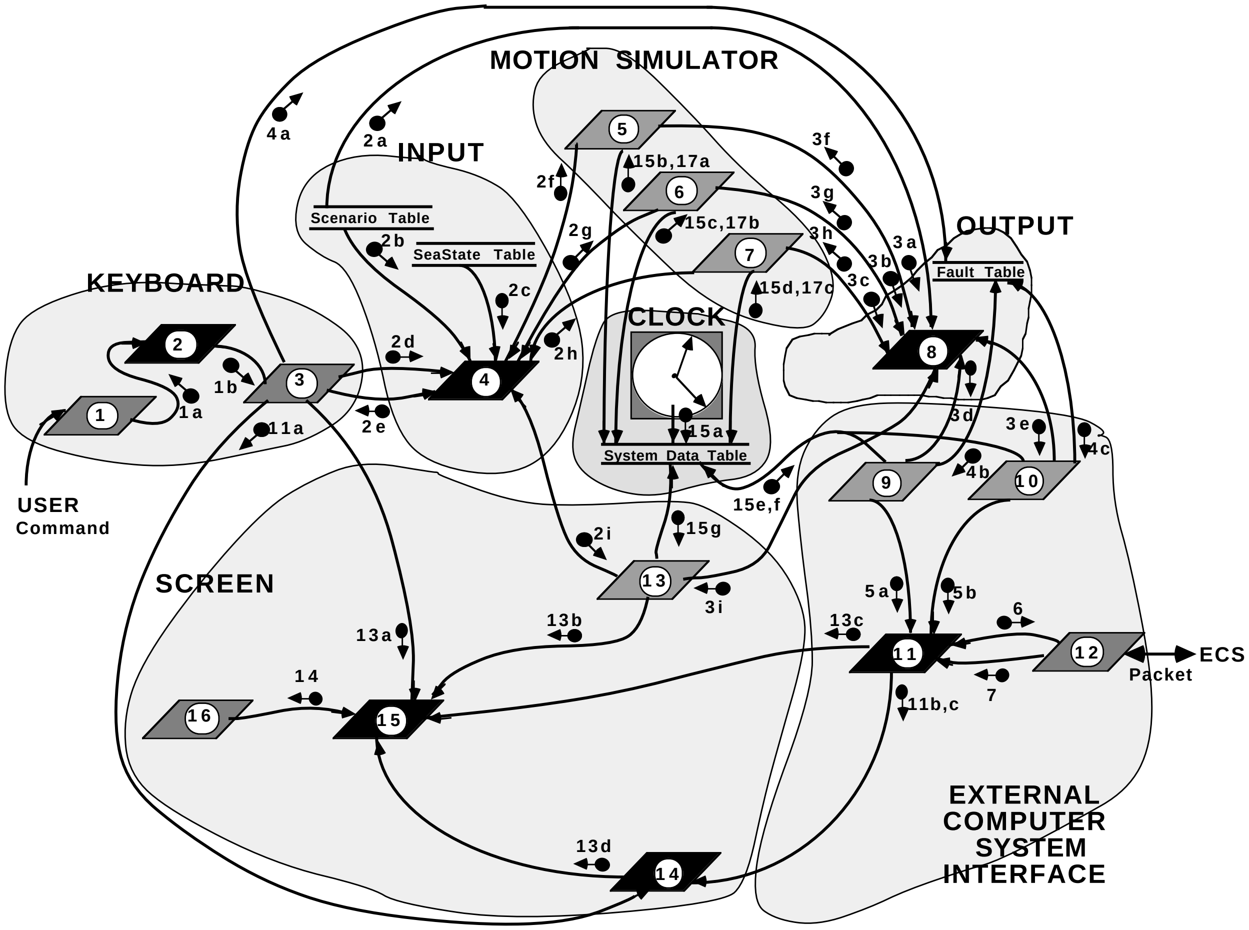
A noteworthy characteristic of real-time systems and the INS in particular is the occurrence of events which tend to arrive at irregular (random) intervals and require immediate service (either relaxed to a certain degree with a soft response requirement i.e., as-soon-as-possible, or with a hard deadline specified). Random arrivals (Poisson distribution) are unbounded in the sense that the number of events which can occur within a fixed time frame has no limit. In such a case, scheduling can not be reasonably predicted. It follows, then, that analyzing the contribution of aperiodics as a utilization component requires placing some bound on the worst-case behavior, in the form a minimum inter-arrival time. Using this knowledge we can satisfy hard deadlines under RMS and, given an average interarrival time, we can predict expected response. There are a number of ways aperiodics can be modeled and this is an active area of investigation. For example, an aperiodic event server can be scheduled to execute within a quota of allocated time intervals proven to be available under RMS in a periodic task schedule. Once transformed, the handling of aperiodic events can be equivalently modeled as periodic tasks. Thus, this mechanism provides the framework for an RMS analysis which meets the requirements for both asynchronous response and task set schedulability. The important principle here is that techniques exist for quantifying all effects on task set utilization. The difficulty remains in discovering the worst-case bounds for each contributing factor of the equation.

3.3.5. INS Task Set Summary

With this knowledge in mind, the following characteristics are reflective of the derived INS task set:

- *Periodic* tasks with hard deadline performance specified
- *Aperiodic* tasks with performance requirements for rapid event response
- *Server* tasks for shared resource synchronization

The INS serves as a natural vehicle for experimentation in priority-based preemptive task scheduling. Since this paradigm is expressed directly in the Ada language, the use of Ada tasks to represent concurrent elements in the model was a straightforward choice. Previous discussion has attempted to characterize the derived task set as an accomplished fact. Such is not the case. Only those processes which have functions derived from the data flow analysis of the previous chapter are exemplified by the current parallel threads and support for specific target processors may dictate a change in dynamics. A detailed design of each implementation will document these dependent architectural features. As an example, the reader should note that task scheduling is an action normally performed by the Ada runtime, but circumstances might require this to be executed at the user level with an application task dispatcher and a real-time clock. By replacing existing data transforms and stores from the DFD (Figure 2-2) with the appropriate units of concurrency, Figure 3-7 summarizes the overall partition of the conforming Ada task structure:



Legend

Task No.	Criteria/ Category	Major DFD Subsystem	Task Name
1-	*Aperiodic*	Keyboard	KEYBOARD_ISR
2-	*Server*	Keyboard	INPUT_BUFFER_MONITOR
3-	*Aperiodic*	Keyboard	COMMAND_PROCESSOR
4-	*Server*	Input	PARM_TABLE_MONITOR
5-	*Periodic*	Motion_Simulator	SHIP_ATTITUDE_UPDATER (400 Hz)
6-	*Periodic*	Motion_Simulator	SHIP_POSITION_UPDATER(0.8 Hz)
7-	*Periodic*	Motion_Simulator	SHIP_VELOCITY_UPDATER (24 Hz)
8-	*Server*	Output	RESULTS_TABLE_MONITOR
9-	*Periodic*	ECSI	ATTITUDE_MESSAGE_SENDER (16 Hz)
10-	*Periodic*	ECSI	NAVIGATION_MESSAGE_SENDER(1 Hz)
11-	*Server*	ECSI	COMMS_CONTROLLER
12-	*Aperiodic*	ECSI	COMMS_ISR
13-	*Periodic*	Screen	PERIODIC_DISPLAY_UPDATER (1 Hz)
14-	*Server*	Screen	ALERTS_MONITOR
15-	*Server*	Screen	SCREEN_CONTROLLER
16-	*Aperiodic*	Screen	SCREEN_ISR

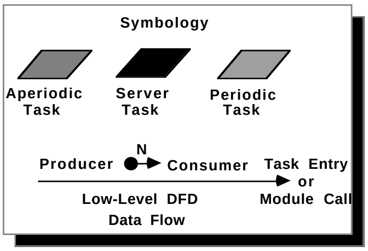


Figure 3-7: INS Simulator: Task Set Overlay of Data Flow Diagram

4. Module Structure

Although the eventual physical structure of the INS will be modular, based fundamentally on Ada *packages* with hierarchical internals and inter-package dependencies, the top-most composition of the simulator program is subsystems, which are groups of logically related packages. Figure 4-1 depicts the partition of these subsystems with their constituent packages and indicates the major dependencies between the subsystems.

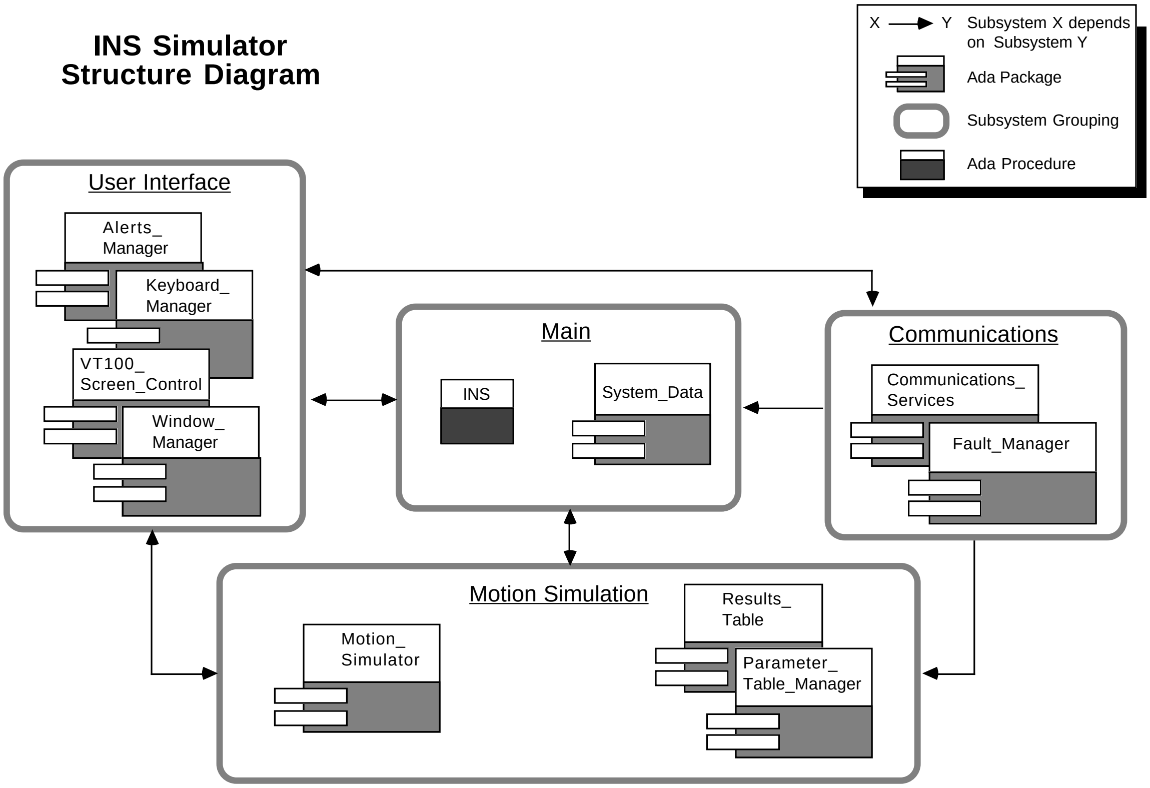


Figure 4-1: INS Simulator: Top-Level Structure Diagram

The completed implementation of the INS Simulator will require a number of implementation-dependent packages to support execution on the intended target processor. These supplementary modules might include an adjunct task dispatcher with supporting operations, a real-time clock interval timer for scheduling and simulator application functions, a mathematics library for motion simulator application, and interface modules (interrupt handlers and device drivers) to meet the special needs of connected hardware. Modifying real-time software for portability often results in changing task sets, thus making performance prediction even more reliant on analytical scheduling techniques.

Table 4-1 shows the location of the implementation-independent tasks already defined within the subsystems based on the static structure given in Figure 4-1 on page 47:

Task	Subsystem
Screen_Controller	**VT100_Screen_Control**
Screen_ISR	**VT100_Screen_Control**
Keyboard_ISR	**Keyboard_Manager**
Command_Processor	**Keyboard_Manager**
Input_Buffer_Monitor	**Keyboard_Manager**
Alerts_Monitor	**Alerts_Manager**
Periodic_Display_Updater	**Window_Manager**
Parm_Table_Monitor	**Parameter_Table_Manager**
Results_Table_Monitor	**Results_Table**
Ship_Attitude_Updater	**Motion Simulation**
Ship_Position_Updater	**Motion Simulation**
Ship_Velocity_Updater	**Motion Simulation**
Comms_Controller	**Communications_Services**
Comms_ISR	**Communications_Services**
Attitude_Message_Sender	**Communications_Services**
Navigation_Message_Sender	**Communications_Services**

Table 4-1: Subsystem Tasks

References

[Borger 89] Borger, M. W., Klein M. H., & Veltre R.
Engineering Real-Time Software in Ada: Observations and Guidelines
Annual Technical Report CMU/SEI-89-TR-22, DTIC: ADA204399, Software Engineering Institute, October 1989

[Gomaa 84] Gomaa, H.
A Software Design Method for Real-Time Systems
CACM, Volume 27, No. 9, September 1984, pp 938-949

[Klein 87] Klein, M.
Inertial Navigation System Simulator Program: Top-Level Design
Technical Report CMU/SEI-87-TR-34, DTIC: ADA200605, Software Engineering Institute, December 1987

[Landherr 87a] Landherr, S.F. & Klein, M.H.
Inertial Navigation System Simulator Program: Behavioral Specification
Technical Report CMU/SEI-87-TR-33, DTIC: ADA200604, Software Engineering Institute, October 1987

[Landherr 89] Landherr, S.F., Klein, M.H., & Fowler K.J.
Inertial Navigation System Simulator Program: Behavioral Specification, Revised
Technical Report CMU/SEI-89-TR-35, Software Engineering Institute, October 1989

[Liu 73] Liu, C.L. & Layland, J.W.
Scheduling Algorithms for Multi-programming in a Hard-Real-Time Environment
JACM, Vol 20, No. 1, January 1973, pp 46-61

[Meyers 88a] Meyers, B.C. & Weiderman, N.H.
Systems Specification Document for an Inertial Navigation System Simulator and External Computer
Technical Report CMU/SEI-88-TR-24, Software Engineering Institute, September 1988

[Meyers 88b] Meyers, B.C. & Weiderman, N.H.
Functional Performance Specification for an Inertial Navigation System Simulator
Technical Report CMU/SEI-88-TR-23, DTIC: ADA204850, Software Engineering Institute, September 1988

[Meyers 88c] Meyers, B.C. & Mumm, H.
Functional Performance Specification for an External Computer to Interface to an Inertial Navigation System Simulator
Technical Report CMU/SEI-88-TR-25, DTIC: ADA200611, Software Engineering Institute, September 1988

[NAVSEA 82] NAVSEA
Interface Design Specification for the Inertial Navigation System AN/WSN-5 to External Computer
NAVSEA T9427-AA-IDS-010/WSN-4, August 1982

[Nielson 88] Nielson, Kjell and Ken Shumate
Designing Large Real-Time Systems with Ada
McGraw-Hill Book Co., NY, 1988