Technical Report

CMU/SEI-87-TR-034
ESD-TR-87-197

Inertial Navigation System Simulator Program: Top-Level Design

Mark H. Klein
Stefan F. Landherr
October 1987

Technical Report

CMU/SEI-87-TR-34

ESD-TR-87-197

October 1987

Inertial Navigation System Simulator Program: Top-Level Design

Mark H. Klein

Ada Embedded Systems Testbed Project

Stefan F. Landherr

Resident Affiliate

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

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.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

SEI Joint Program Office

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

Inertial Navigation System Simulator Program: Top-Level Design

Abstract: A real-time Ada application, an Inertial Navigation System (INS) simulator, is being developed by the Ada Embedded Systems Testbed Project as a vehicle to analyze issues regarding the use of Ada in the real-time embedded domain and to provide a context for future experimentation. The technical philosophy behind developing a real-time Ada artifact is to: (1) select a representative (e.g., strict timing demands, multiple concurrent activities, low-level I/O, error handling, interrupts, and periodic activities) real-time application; (2) use Ada tasks as the unit of concurrency for the real-time design; and (3) apply any relevant practical results being produced by the real-time scheduling research community. In particular, the INS simulator must satisfy a set of timing requirements that are similar to an INS with respect to data updating, message transmission, and message reception. This document discusses the top-level design of this application from three points of view: data flow perspective, the concurrency and control perspective, and the Ada module perspective.

1. Introduction

The Inertial Navigation System (INS) simulator application [Meyers 87a] consists of two programs executing on separate computers: the INS simulator program [Meyers 87b, Landherr 87a] and the external computer program [Meyers 87c].

The requirements for this development include:

- **Designing for Ada**, assuming a high-quality compiler and runtime system.
- **Using Ada constructs** whenever it is possible to do so and still meet performance requirements; in critical cases, it may be necessary to make explicit calls to runtime services.
- **Using incremental development** in critical areas. Three important examples are the real-time design to tune task priorities, the communications link interface, and the timing of application functions.
- **Creating common design and code**, which will be reused for the external computer (EC) system.

Since the intention of the Ada Embedded Systems Testbed (AEST) Project is to implement the INS simulator program on more than one computer and runtime system, the top-level design of the INS simulator program is described in a general manner. Aspects specific to a particular implementation will be described in the detailed design.

The top-level design document contains four chapters and one appendix:

1. **Introduction**
2. **Data Flow Analysis**: describes the overall flow of data in terms of data stores, data transforms, and the data flows between stores and transforms.
3. **Control and Concurrency**: describes the real-time design and motivates the concurrent threads of processing and control (i.e., the tasking structure).
4. **Module Structure**: defines the top-level packages and their interdependencies.
5. **Appendix A, Allocation of Task Priorities**: specifies the current allocation of Ada task priorities.

2. Data Flow Analysis

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

- contents of major data stores
- functions of major data transforms
- constituents of data flows between stores and transforms

This chapter is the first step in creating a program design that meets the requirements of [Meyers 87b] and [Landherr 87a]. It thus forms the basis for subsequent chapters.

The overall flow of data through the INS simulator program is depicted in Figure 2-1, and the elements of the data flow diagram are described in the following sections.

2.1. Data Stores

The contents of the major data stores are listed below.

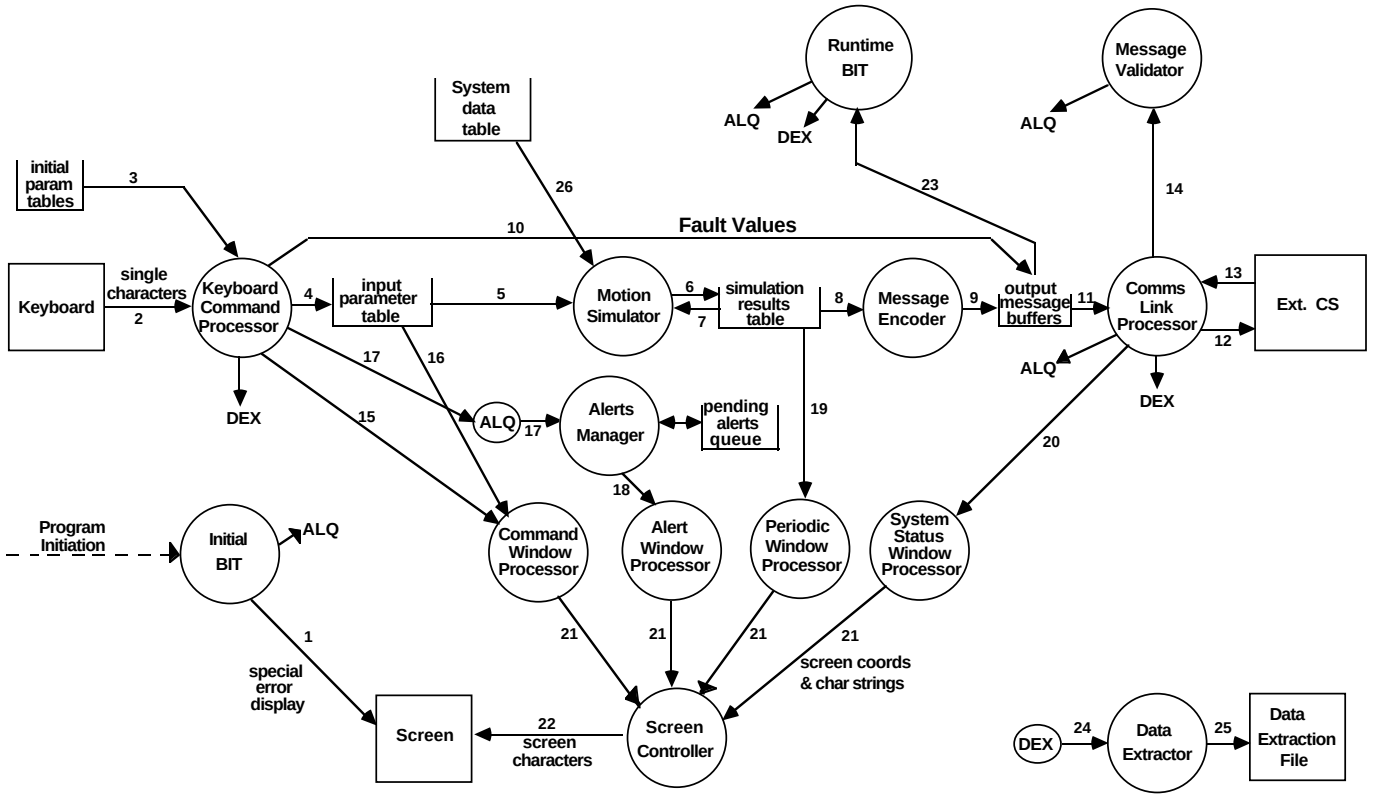
2.1.1. Initial Parameter Tables

This data store consists of two tables:
The sea-state table contains 6 sets of the following parameters:

```
Surge  )
Heave  )       ( Amplitude
Sway   )   X   ( Frequency
Roll   )       ( Phase
Pitch  )
Yaw    )
```

The scenario table contains 9 sets of the following parameters:

```
Lever Arm Constant-   A, B, C
Ship List
Ship Trim
Initial Ship Course
Initial Ship Speed
Initial Latitude
Initial Longitude
Ocean Current-   East, North
```



Legend

- data transformation process
- data store
- device

ALQ - multiple flows to Alerts Manager

DEX - multiple flows to Data Extractor

data flow

List of Data Flows

1. Special Error Display
2. Keyboard Characters
3. Initial Parameters
4. Parameter Settings
5. Input Parameters
6. Simulation Results
7. Intermediate Results
8. Simulation Outputs
9. Periodic Message Values
10. Fault Values
11. Periodic Messages
12. Output Messages
13. Input Messages
14. Input Message Values
15. Echoed Characters
16. Shown Parameters
17. Alert Packets
18. Alert Displays
19. Simulation Values
20. Comms State
21. Screen Packets
22. Screen Characters
23. Periodic Message Fields
24. Data Extraction Packets
25. Data Records
26. Simulated Time

Figure 2-1: INS Simulator: Data Flow Diagram

2.1.2. Input Parameter Table

This table contains the following parameters used as inputs to the motion simulation calculations:

```
Surge  )
Heave  )       ( Amplitude
Sway   )   X   ( Frequency
Roll   )       ( Phase
Pitch  )
Yaw    )
Lever Arm Constant-   A, B, C
Ship List
Ship Trim
Commanded Ship Course  (i.e., at current time)
Commanded Ship Speed
Ocean Current-   East, North
New Commanded Course
Turn Rate
New Commanded Ship Speed
Speed Change Period
EM Log Calibration
Radial Error Estimate
```

2.1.3. Simulation Results Table

The simulation results table contains the following time-dependent variables produced by the motion simulation calculations. The values are used in the attitude and navigation periodic data messages and are also displayed in the periodic display window:

```
Ownship Heading
Ownship Pitch
Ownship Roll
Ownship Heading Rate
Ownship Pitch Rate
Ownship Roll Rate
Ownship Surge
Ownship Heave
Ownship Sway
Ownship Speed
East Component of Ownship Velocity
North Component of Ownship Velocity
Vertical Component of Ownship Velocity
Latitude
Longitude
Integral of Velocity North
Integral of Velocity East
```

2.1.4. Output Message Buffers

The fault buffer contains fault values specified by operator commands.

The output message buffer contains copies of the INS output messages (i.e., test time and status data, attitude periodic data, and navigation periodic data) with any specified faults overwriting the true data.

2.1.5. Pending Alerts Queue

This queue has up to 50 entries in order of priority. Each entry consists of :

```
Alert identification   (with implicit priority)
Time at which alert was issued
```

2.1.6. System Data Table

The system data table contains system-wide constants and state variables, such as:

```
Initial GMT
Simulation time
Time of gyro reset
Time of last start-of-message (SOM) signal
```

2.2. Data Transforms

The functions of the data transforms are listed below.

2.2.1. Initial Built-In Test

The Initial Built-In Test (BIT) transform is started at program initiation and performs the tests specified in Chapter 5 of [Landherr 87a]. If these tests are successful, an INITIAL BIT SUCCESSFUL alert is issued. If there are any failures, the appropriate error messages are sent to the console screen, but not through the normal alerts mechanism.

2.2.2. Keyboard Command Processor

The Keyboard Command Processor transform accepts single characters from the keyboard, echoes them, assembles them into command strings, parses the strings into commands, and performs the actions specified by the commands. The command syntax and command actions are specified in [Landherr 87a]. The majority of commands initialize, change, or display values in the input parameter table. The FAULT command inserts values into the fault buffer. Other commands control the state of the Data Extraction Processor, and the remaining commands control the overall state of the simulation. Certain commands cause an alert to be issued, as do any invalid commands. Also, certain commands cause data extraction records to be written.

2.2.3. Motion Simulator

The Motion Simulator transform contains three subordinate transforms that periodically calculate the simulated ship motion as specified in Chapter 4 of [Landherr 87a]:

- Update Ship Attitude
- Update Ship Velocity
- Update Ship Position

The three transforms use values from the system data and input parameter tables, and they calculate values for the simulation results table. In addition, the velocity and position transforms reuse some previously calculated values.

2.2.4. Message Encoder

The Message Encoder transform contains four subordinate transforms that assemble data messages in the format specified in [NAVSEA 82], overwriting message fields with any values currently in the fault buffer:

- Build Attitude Periodic Data Message
- Build Navigation Periodic Data Message
- Build Test Message
- Build Time and Status Message

2.2.5. Communications Link Processor

The Communications Link Processor is a complex transform that does the following:

- Establishes and maintains the communications protocol with the external computer, as specified in Chapter 3 of [Landherr 87a].
- 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.

Certain conditions cause alerts to be issued and/or data extraction records to be written, as specified in [Landherr 87a].

2.2.6. Message Validator

The Message Validator transform checks certain fields of each input message [Meyers 87b]. If an error is found, an alert is issued.

2.2.7. Alerts Manager

The Alerts Manager transform accepts alerts from various other transforms, stores up to 50 alerts in the pending alerts queue, accepts escape characters from the Keyboard Command Processor to display the highest priority pending alert, and sends appropriate alerts to the Alert Window Processor for display.

2.2.8. Command Window Processor

The Command Window Processor transform accepts characters from the Keyboard Command Processor (including editing characters) and arranges for the display of the (edited) command line as specified in [Landherr 87a] by formatting standard packets for the Screen Controller. It also fetches specified values from the input parameter table for appending to the command line.

2.2.9. Alert Window Processor

The Alert Window Processor transform accepts individual alerts from the Alerts Manager and arranges for them to be displayed in the alert window by formatting standard packets for the Screen Controller.

2.2.10. Periodic Window Processor

The Periodic Window Processor transform periodically fetches values from the simulation results table and arranges for them to be displayed in the periodic display window by formatting standard packets for the Screen Controller.

2.2.11. System Status Window Processor

The System Status Window Processor transform accepts changes in the communications state or the data extraction state and arranges for these values to be displayed in the system state window by formatting standard packets for the Screen Controller.

2.2.12. Screen Controller

The Screen Controller transform accepts packets from the four window processors, which contain a string of characters to be displayed at a specified position on the screen, and writes the string onto the screen.

2.2.13. Data Extractor

The Data Extractor transform accepts data extraction records from various other transforms and outputs them to the data extraction file.

2.2.14. Runtime Built-In Test

The Runtime BIT transform periodically checks the contents of the output message buffer for incorrect or out-of-range values. If an error is found, an alert is issued. Each invocation of this transform triggers the writing of a data extraction record.

2.3. Data Flows

The constituents of the data flows to and from the input/output (I/O) devices are specified in [Landherr 87a]. The constituents of the data flow between a data transform and a data store are obvious in most cases, given the foregoing descriptions of data stores and transforms. Nevertheless, for the sake of completeness, each of the data flows in Figure 2-1 is briefly described below. (Note that the subsection numbers correspond to the list of data flows in Figure 2-1).

2.3.1. Special Error Display

(Initial BIT ---> Screen)

This data flow is a stream of ASCII characters containing a once-only text message to indicate the failure of the Initial BIT. It goes directly to the console screen and thus includes certain control characters that may be implementation dependent.

2.3.2. Keyboard Characters

(Keyboard ---> Keyboard Command Processor)

This data flow is simply a stream of ASCII characters typed by the operator on the keyboard, as specified in [Landherr 87a].

2.3.3. Initial Parameters

(Initial Parameter Tables ---> Keyboard Command Processor)
This data flow consists of sets of parameters from either the sea-state table or the scenario table.

2.3.4. Parameter Settings

(Keyboard Command Processor ---> Input Parameter Table)
This data flow consists of individual parameter values supplied by the operator in the SET PARAMETER, CHANGE COURSE, or CHANGE SPEED commands.

2.3.5. Input Parameters

(Input Parameter Table ---> Motion Simulator)
This data flow consists of the various ship simulation parameters required by the three Motion Simulator's subprocesses, as outlined in [Landherr 87a]. (Details of the calculations are given in [Meyers 87b].)

2.3.6. Simulation Results

(Motion Simulator ---> Simulation Results Table)
This data flow consists of the numerical values calculated by the three subprocesses of the Motion Simulator, as outlined in [Landherr 87a]. (Details of the calculations are given in [Meyers 87b].)

2.3.7. Intermediate Results

(Simulation Results Table ---> Motion Simulator)
This data flow consists of the following numerical values:

- Those calculated by the Update Ship Attitude transform and required by the Update Ship Velocity transform.
- Those calculated by the attitude and velocity transforms and required by the Update Ship Position transform.

2.3.8. Simulation Outputs

(Simulation Results Table ---> Message Encoder)
This data flow consists of the numerical values that are required to assemble the attitude and navigation periodic data messages, as specified in [Landherr 87a].

2.3.9. Periodic Message Values

(Message Encoder ---> Output Message Buffers)
This data flow consists of the specially formatted blocks of 16-bit message words that constitute the attitude and navigation periodic data messages, as outlined in [Landherr 87a]. Details are given in [NAVSEA 82].

2.3.10. Fault Values

(Keyboard Command Processor ---> Output Message Buffers)
This data flow consists of individual numerical values supplied by the operator in FAULT commands, as specified in [Landherr 87a].

2.3.11. Periodic Messages

(Output Message Buffers ---> Communications Link Processor)
This data flow consists of complete attitude and navigation periodic data messages, overlaid with any injected fault values, as specified in [Landherr 87a].

2.3.12. Output Messages

(Communications Link Processor ---> External Computer)
This data flow consists of external function (EF) codes and blocks of normal message words, as specified in [Landherr 87a].

2.3.13. Input Messages

(External Computer ---> Communications Link Processor)
This data flow consists of external function codes and blocks of normal message words, as specified in [Landherr 87a].

2.3.14. Input Message Values

(Communications Link Processor ---> Message Validator)
This data flow consists of certain fields of the input messages which must be checked, as specified in [Meyers 87b].

2.3.15. Echoed Characters

(Keyboard Command Processor ---> Command Window Processor)
This data flow consists of a stream of characters. Normally these characters are echoes of the individual characters typed by the operator on the keyboard, but they may also be sequences of characters to edit the command line (e.g., backspace), as specified in [Landherr 87a].

2.3.16. Shown Parameters

(Input Parameter Tables ---> Command Window Processor)
This data flow consists of the numerical values of parameters selected by the operator in SHOW PARAMETER and SHOW commands.

2.3.17. Alert Packets

(Various processes ---> Alerts Manager)
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. A minimal list of alerts is specified in [Landherr 87a].

2.3.18. Alert Displays

(Alerts Manager ---> Alert Window Processor)
This data flow consists of :

- the actual texts of alert messages
- timestamps in readable form
- the number of pending alerts

2.3.19. Simulation Values

(Simulation Results Table ---> Periodic Window Processor)

This data flow consists of numerical values of those simulation results that are displayed on the periodic display window of the console screen, as specified in Figure 2-3 of [Landherr 87a].

2.3.20. Communications State

(Communications Link Controller ---> System Status Window Processor)

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

2.3.21. Screen Packets

(Window Processors ---> Screen Controller)

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

2.3.22. Screen Characters

(Screen Controller ---> Screen)

This data flow consists of a stream of ASCII characters, including control characters to position the cursor, as specified in [Landherr 87a].

2.3.23. Periodic Message Fields

(Output Message Buffers ---> Runtime BIT)

This data flow consists of certain numerical fields of the encoded attitude and navigation periodic data messages, which are to be checked as specified in [Meyers 87b].

2.3.24. Data Extraction Packets

(Various transforms ---> Data Extractor)

This data flow consists of packets of coded data that identify a data extraction event and specify the time of occurrence of the event, as well as any other required data. The details are specified in [Landherr 87a].

2.3.25. Data Records

(Data Extractor ---> Data Extraction File)

This data flow consists of encoded blocks of recorded data, as specified in [Landherr 87a].

2.3.26. Simulation Time

(System Data Table ---> Motion Simulator)

This data flow consists of the current value of simulated time that is to be used in the ship motion calculations. (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.)

3. Concurrency and Control

The purpose of this chapter is to provide a task structure for the INS simulator. The starting point is the data flow structure developed in the preceding chapter (see Figure 2-1). The approach is to derive a set of tasks from the set of data transforms and then analyze the intertask relationships. The first step involves enumerating a rough set of criteria to be used in decomposing data transforms into tasks (i.e., threads of control) and then applying the criteria. The second step involves analyzing the real-time requirements of the INS simulator, discussing the real-time architecture, and walking through several processing chains. The last section of this chapter examines potential areas where modifications may have to be made as we learn more about the behavior of the design through incremental development activities.

3.1. Derivation of Concurrency

There are no hard and fast rules for decomposing a system into a set of concurrent processes. The following set of criteria [Gomaa 84] is being used as a guideline for decomposing and grouping data transforms into a set of tasks. The main criterion is the asynchronous nature of the functions being performed within the data transform. Other criteria include the answers to these questions:

- Is the role of the data transform that of a server, where the clients may proceed independently after synchronization?
- Can the data transform be viewed as an actor [Borger 86] with periodic processing requirements?
- Is the data transform an interrupt service routine (ISR)?
- Is the primary role of the data transform that of a producer or consumer of I/O to or from a device?
- Are the functions of the data transform time critical?
- Can the functions of multiple data transforms be grouped together?

Figure 3-1 illustrates the choice of derived tasks and their relationships to the data transforms. The following sections describe this task set. The first three sections discuss tasks related to the four devices: keyboard, screen, communications link interface, and disk. The remaining data transforms are then discussed, followed by a section that demonstrates the need for a real-time clock and a task dispatcher. Each discussion is followed by a list of the data transforms and the associated set of tasks. (Note that these lists do not present a line-by-line mapping from a specific data transform to a set of tasks.)

Devices are treated as data transforms and interrupt service routines are considered tasks with interrupt entries. Detailed discussion of the control structure within an individual task is, for the most part, left to detailed design.

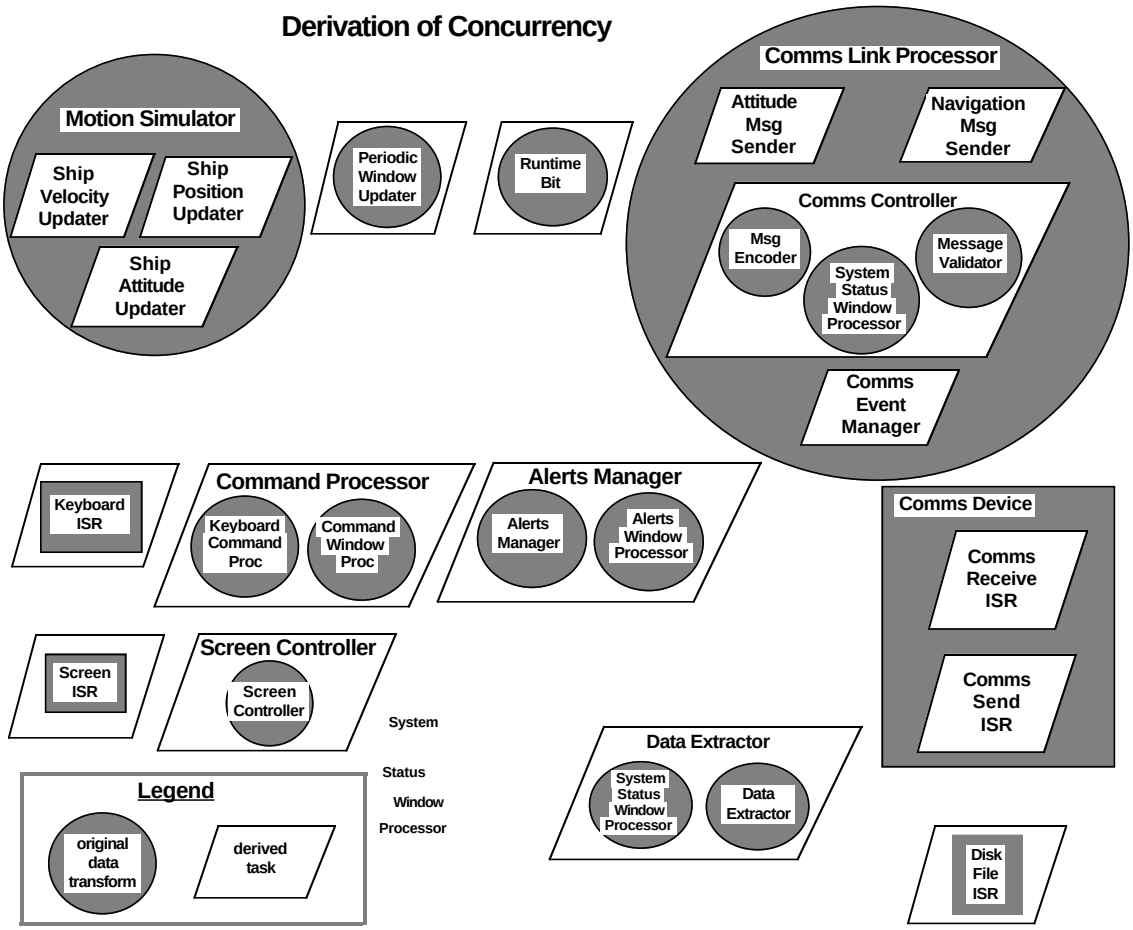


Figure 3-1: Derivation of Tasks

3.1.1. Keyboard

The Keyboard and the Keyboard Command Processor comprise the data transforms that are needed for the keyboard. Additionally, the Keyboard Command Processor uses functions of the Command Window Processor to echo characters on the screen command line. The keyboard maps onto the Keyboard ISR. The Keyboard Command Processor and the Command Window Processor execute in the Command Processor task. The Keyboard ISR is a producer, which places characters received from the keyboard into a buffer. The Command Processor task is the consumer, removing characters from the buffer and subsequently parsing the command string and executing the command.

- Data Transforms:
  - Keyboard
  - Keyboard Command Processor
  - Command Window Processor
- Tasks:
  - Keyboard ISR
  - Command Processor

3.1.2. Screen

Five data transforms comprise the functions needed to handle the screen. The Screen maps onto a Screen ISR. The Command Window Processor, Alert Window Processor, and System Status Window Processor each will map onto one or more subprograms (not tasks) and will consequently execute in the thread of control of the calling task. It may be the case that one subprogram is used by multiple tasks (e.g., the System Status Window Processor). Since Ada requires that all subprograms be reentrant, this poses no problem [LRM 6.1(9)]. The functions of the Screen Controller transform are handled by the Screen Controller, which is a monitor that controls access to the screen. This task has the responsibility of handling requests from multiple clients to write data to the screen. The Periodic Window Processor maps onto a separate task because it is an active entity that is responsible for periodically updating the screen.

- Data Transforms:
  - Screen
  - Screen Controller
  - Periodic Window Processor
- Tasks:
  - Screen ISR
  - Screen Controller
  - Periodic Display Updater

3.1.3. Communications Link Interface

The data transforms whose functions involve communications are the Communications Link Processor, the Message Encoder, and the Message Validator. As stated in the previous chapter, the Message Encoder and Message Validator build and validate data messages respectively. These are synchronous functions. A message only has to be built immediately before it is sent, and it can't be sent until it is built. A message is validated only after it arrives, and its validity can be acknowledged only after validation is complete. Therefore, these data transforms will be implemented by collections of subprograms.

The Communications Link Processor must send two types of periodic messages that differ in frequency. In addition, it must handle incoming messages from the external computer. One's first inclination is to create a task for sending attitude messages (one every 61.44 ms), a task for sending

navigation messages (one every 983.04 ms), and a task to receive messages. Since sending and receiving do not take place simultaneously and only one message can be sent at a time, either a monitor task or a semaphore is needed to arbitrate between those tasks that want to communicate with the external computer. Moreover, if a SOM external function, which indicates the start of a message from the external computer, is received during sending, then sending must be aborted in favor of receiving a message. All of this requires potentially complex intertask communication. Therefore, another alternative will be adopted.

Since only one communications task is logically active at a time, the sending of the two messages and receiving can be collapsed into one task, Comms Controller. Periodic tasks are still needed to initiate the sending of each message. The arrival of an external function from the external computer also must be communicated to this task. In addition, the communications protocol requires a time-out facility, which allows the INS to wait for a small period of time for a response from the EC. We will see in Section 3.1.5 that a facility other than the Ada delay statement will be needed for this. Entry calls to Comms Controller provide a natural mechanism for communicating the occurrence of an event such as the arrival of an external function. However, from a structural point of view, the associated accept statements must be in the body of the Ada task itself and cannot be in a called subprogram. Thus, a large unwieldy task would be necessary in order to implement the communications protocol, which includes specific steps for sending, receiving, and enabling.

An alternative is to break out from the Comms Controller a fourth task called the Comms Event Manager. This is a server task with two types of entries: entries for posting events and entries that allow a client to wait for the occurrence of one of several events. Events can now be posted to the Comms Event Manager instead of the Comms Controller, thus allowing Comms Controller to be decomposed into subprograms that handle the receiving, sending, and enabling aspects of communications—a more modular approach. In addition, there will be two ISRs, one for receiving and one for sending.

To summarize: the Communications Link Processor maps onto four tasks: Attitude Message Sender, Navigation Message Sender, Comms Controller, and Comms Event Manager. The Comms Controller encompasses the functions from three data transforms: Message Encoder, Message Validator, and System Status Window Processor (see Section 3.1.2, which states that the System Status Window Processor maps onto a subprogram). The communications device maps onto the Comms Receive ISR and Comms Send ISR.

- Data Transforms:
  - Comms Link Processor
  - Message Encoder
  - Message Validator
  - External Computer
- Tasks:
  - Attitude Message Sender
  - Navigation Message Sender
  - Comms Controller
  - Event Manager
  - Comms Receive ISR
  - Comms Send ISR

3.1.4. Remaining Data Transforms

The Motion Simulator and Runtime BIT transforms support periodic processing functions. The Motion Simulator maps onto three tasks: Ship Attitude Updater, Ship Velocity Updater, and Ship Position Updater, which are responsible for performing periodic calculations at separate frequencies. Runtime BIT maps onto one task of the same name.

Two transforms execute in the thread of control of the Alerts Manager task: the Alerts Manager transform and the Alerts Window Processor.

The Data Extractor maps onto one server task that supports multiple clients in writing information to disk. The System Status Window Processor functions also to execute in this thread of control. (Refer to Section 3.1.2 for a brief discussion of subprograms that execute in multiple threads of control.) There is also an ISR for the disk controller.

- Data Transforms:
  - Motion Simulator
  - Runtime BIT
  - Data Extractor
  - System Status Window Processor
  - Alerts Manager
  - Alerts Window Processor
- Tasks:
  - Ship Attitude Updater
  - Ship Velocity Updater
  - Ship Position Updater
  - Runtime BIT
  - Data Extractor
  - Alerts Manager

3.1.5. Real-Time Clock

Real-time requirements have not yet been addressed. Appendix A of the behavioral specification for the INS simulator [Landherr87a] lists the the timing requirements. A mechanism for scheduling periodic tasks to accommodate these requirements has been presumed in the discussion thus far.

The Ada delay statement would be used in an ideal Ada design (i.e., presuming an ideal Ada compiler). However, the resolution of the delay statement in commercially available Ada compilers (at the time of this design) is 10 milliseconds or more. Note, however, that the Ship Attitude Updater must execute once every 2.56 milliseconds and that the communications protocol requires 10.24 millisecond time-outs and 5.12 millisecond sleep intervals [Landherr 87a]. This necessitates using a programmable real-time clock to achieve a notion of time with higher resolution. Two additional tasks are needed: one to receive the interrupt from the clock and track time, and another to manage a list of tasks that must be scheduled in order to meet the specified deadlines. These tasks are the Clock ISR and Dispatcher, respectively. The Clock ISR services interrupts generated by the real-time clock every 2.56 milliseconds. The Clock ISR cannot have much responsibility in that it must be ready to service the interrupt when it occurs; therefore, the dispatcher function is placed in its own task. In addition to these two tasks, an interface must be provided to the Comms Controller task to allow it to request time-outs. This function could be part of the Dispatcher; but for modularity, a Time-Out Server task will be introduced. This task will both accept requests for time-outs and issue a time-out to the Comms Event Manager after the specified time-out period has expired.

- Data Transforms:
  - <no data transforms>
- Tasks:
  - Clock ISR
  - Dispatcher
  - Time-Out Server

Figure 3-2 shows all of the identified tasks and their interdependencies, and the next section will discuss the intertask control structure.

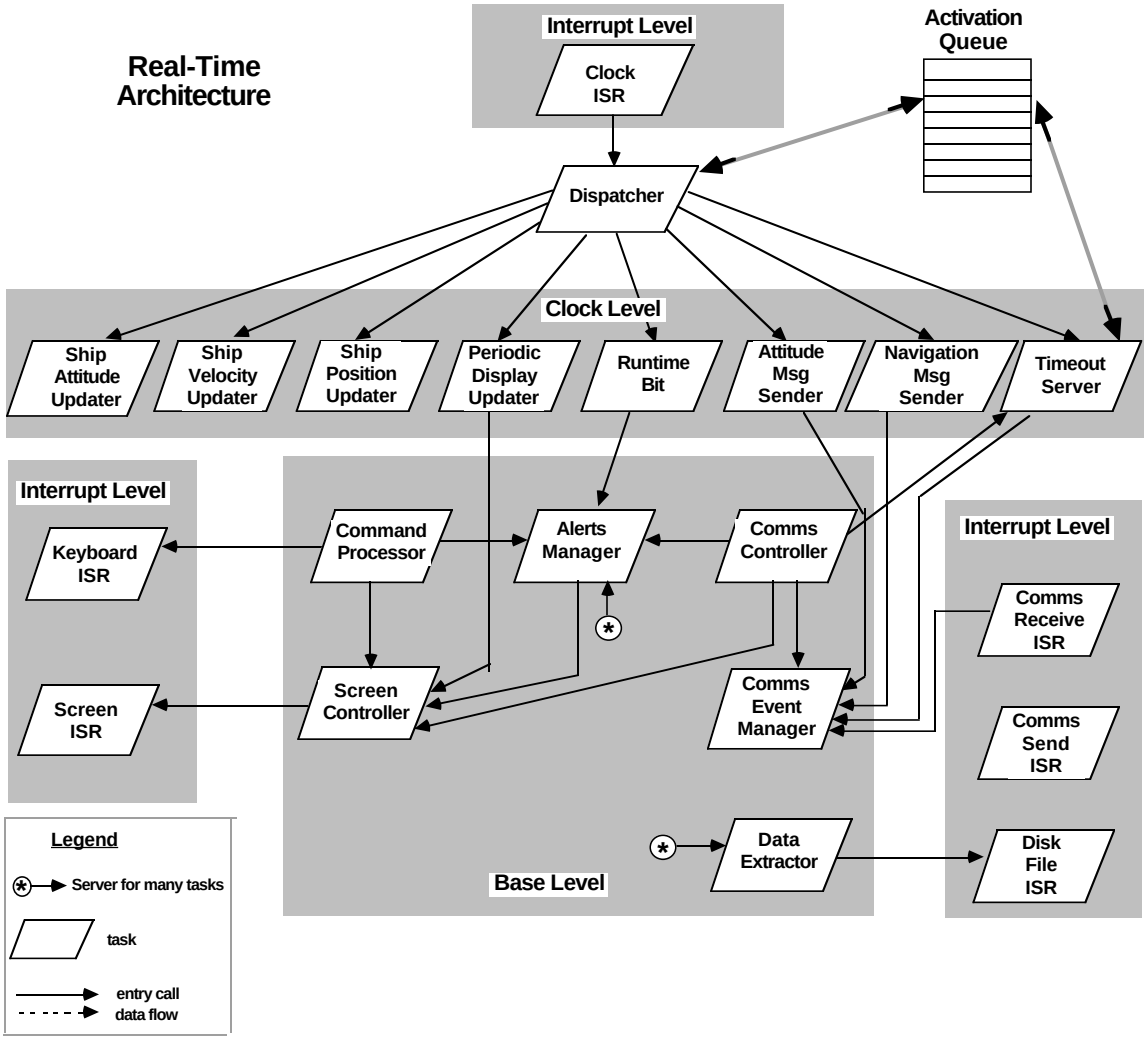


Figure 3-2: Real-Time Architecture

3.2. Intertask Control Structure

The purpose of this section is to describe the tasking structure presented in the previous section in terms of intertask relationships. This includes a discussion of the real-time design (i.e., the interaction between the clock, the dispatcher, and periodic tasks), the interaction between various processing levels, and several example processing chains.

The broad levels of processing used throughout this section are those of [Allworth 81]:

- Interrupt Level: highest priority tasks that serve as interrupt service routines.
- Clock Level: tasks with a cyclic control structure that are activated by an entry call from the Dispatcher.
- Base Level: the remaining tasks that are executed in the background, including servers to clock-level tasks and producers and consumers of I/O that interact with interrupt-level tasks.

3.2.1. Real-Time Clock, Dispatcher, and Activation Queue

As stated in Section 3.1.5, the need for a real-time clock was motivated by the lack of resolution of the Ada delay statement. Using a real-time clock removes the need to rely on the delay statement to achieve periodic processing and provides a mechanism for a task to be suspended for a specified time period.

Figure 3-2 exhibits the real-time architecture, which consists of a dispatcher, an activation queue, a time-out server, and a set of tasks in each of three processing levels. Each of these will be discussed in turn, followed by a discussion of allocation of task priorities. Figure 3.1 shows the mapping between the task hierarchy shown in Figure 3-2 and the data flow diagram shown in Figure 2-1.

3.2.1.1. Dispatcher

The Dispatcher works in concert with the Ada runtime system scheduler. The programmable real-time clock generates periodic 2.56 millisecond interrupts which are handled by a clock interrupt service routine, Clock ISR, which rendezvous in turn with the Dispatcher. (If this latter rendezvous proves to be too costly, then, as mentioned in Section 3.3, an alternative may be to incorporate the Dispatcher into the ISR.) The Dispatcher maintains a time-ordered, prioritized activation queue of clock-level Ada tasks; and it signals (through an entry call) the top task if the time specified for the activation of that task has arrived (and if that task is currently enabled). If the subject task is specified to be periodic, then the Dispatcher automatically re-inserts another element for that task, with the next scheduled time, into the task activation queue.

3.2.1.2. Activation Queue

Each entry in the clock-level activation queue is a record containing:

- task identification
- activation time
- activation priority
- activation mode (single-shot or periodic)

Elements are inserted in the queue in order of activation time and are further ordered by activation priority, if necessary. The activation queue will be initialized with the full set of periodic tasks listed in Appendix A. Since the Dispatcher automatically reschedules periodic tasks, the activation queue will

always contain an element for each periodic task, although in constantly changing order. The other elements in the activation queue correspond to time-out alarms inserted by the Time-Out Server.

Certain periodic tasks, such as those for initiating the sending of messages, may be in a disabled state. When the Dispatcher finds that such a task is due to be activated but is in fact disabled, it simply reschedules the task and examines the next element in the activation queue. (Note: This implies a constant phase for periodic message scheduling with respect to program start; i.e., it is not affected by the time of arrival of the select data messages from the external computer.)

3.2.1.3. Allocation of Ada Priorities

The primary mechanism for achieving deadlines will be the normal Ada tasking priorities. It will be assumed that all potential target implementations allow tasking priorities in the range 0 to 15 at least (but no more than this limited number can be assumed).

All interrupt-level tasks must have higher Ada priority than any other tasks. Within the interrupt level, the relative priorities of the interrupt handlers should be ordered according to the expected frequency of occurrence (i.e., in order of priority):

1. programmable real-time clock
2. external computer communications parallel interface
3. console keyboard
4. console screen

Within the clock level, the priorities of periodic tasks should be in the same relative order as the respective task frequencies, according to the rate monotonic algorithm [Liu 73]. The strict ordering may be tempered by the expected duration of the task; i.e. it may be wasteful to allow preemption of a very short, low-priority task.

Within the base level, the relative task priorities should be such that tasks involved in "completing" high-priority periodic activities are not preempted by tasks "completing" lower priority activities. This necessitates some overlap in the priorities of clock-level and base-level tasks. An example of this is the Comms Event Manager, which is a base-level task of higher priority than all clock-level tasks.

Appendix A lists the current allocation of task priorities.

3.2.1.4. Processing Chain Initiated by Clock

This processing chain is initiated by an interrupt (every 2.56 milliseconds) from the real-time clock. The interrupt is handled by the Clock ISR. Clock ISR rendezvous with the Dispatcher, which looks at the activation queue to determine if there are any clock-level tasks whose periodic processing must be initiated this tick.

In this example, suppose that the Attitude Message Sender is due to be activated. Once activated, it will execute unless there is a higher priority task in progress. When execution begins, Attitude Message Sender immediately makes an entry call to the Comms Event Manager. Now recall that when communications is enabled and a message is neither being sent nor received (i.e., comms is idle), the Comms Controller has to be ready to either handle a request from one of the periodic message senders to send a message or respond to the arrival of a start-of-message (SOM) external function

from the external computer. The Comms Controller makes an entry call to the Comms Event Manager when it is in a idle state and waits for one of the two previously mentioned triggering events to occur. In this case, the Comms Event Manager communicates to the Comms Controller that a request to commence sending the attitude periodic message has occurred. Comms Controller then calls the subprogram responsible for handling the sending aspects of the communications protocol. This subprogram uses the Comms Event Manager for notification that a time-out has occurred and that an external function has arrived from the external computer. It also calls other subprograms to build messages in the appropriate format and still other subprograms for sending EFs and data to the external computer. See Section 3.2.3.4 for the processing chains that involve the communication ISRs.

3.2.2. Time-Out Server

The time-outs required by the communications protocols are implemented by a Time-Out Server, because the Ada delay statement is inadequate. Upon request of Comms Controller, the server causes an appropriate entry to be inserted in the activation queue. If the time-out period expires, the Dispatcher signals the Time-Out Server, which in turn notifies Comms Controller. If, before the expiration of the time-out period, the client task changes state (e.g., receives some awaited input) and no longer requires the time-out, then the client requests that the Time-Out Server cancel the previously issued time-out.

3.2.3. Communications Link Interface

Intertask relationships will be discussed by task, followed by an example processing chain that is initiated by the communications parallel interface.

3.2.3.1. Comms Controller

The Comms Controller is the driver task for the communications subsystem. It rendezvous with the Time-Out Server to request a time-out notification after a specified time-out period. It rendezvous with the Screen Controller to post the status of the communications link on the screen. Finally, it rendezvous with the Comms Event Manager to wait for events to occur and to post the completion of sending a message.

3.2.3.2. Comms Event Manager

The Comms Event Manager is a server which clients use to post events or to wait for the occurrence of one of several events of interest. When a client posts an event, control returns immediately back to the client. When a client waits on an event, control returns to the client only after one of the events for which it is waiting is posted. The triggering event is communicated to the client. Below is a list of the events that occur (and the posting task) and a list of tasks that wait on event combinations.

Posted Events:

- arrival of an external function - Comms Receive ISR
- expiration of time-out period - Time-Out Server
- completion of sending a message - Comms Controller
- request to send a message - Attitude Message Sender
- request to send a message - Navigation Message Sender

Events Waited Upon:

- arrival of an external function or time-out expiration - Comms Controller
- arrival of an external function or request to send a message - Comms Controller
- completion of sending a message - Navigation or Attitude Message Sender

3.2.3.3. Communications Interface ISRs

Data flowing between base-level and the interrupt-level communications ISRs is mediated by two communications buffers: the comms input buffers and the comms output buffer. When data arrives from the communications interface, the Comms Receive ISR handles the processing interrupt. This ISR examines the data, determines if it is EF or non-EF data (referred to below simply as data), and places it into one of two buffers. In addition, the arrival of an EF causes a rendezvous with the Comms Event Manager. Subprograms that execute in the Comms Controller thread of control remove elements from the two buffers.

Sending is similar to receiving. The outgoing data is placed in a single buffer that does not distinguish between EFs and data. The Comms Send ISR receives control upon an interrupt that indicates the successful transmission of an element.

3.2.3.4. Processing Chain Initiated by the Communications Parallel Interface

Assume that the Comms Controller is sending a message (as it was in the example discussed in Section 3.2.1.4). A start-of-message (SOM) EF arrives: the Comms Receive ISR receives control upon interrupt, places the EF in the comms input buffer and posts its arrival to the Comms Event Manager. The next time the Comms Controller needs to wait for an EF, the Comms Event Manager will allow it to proceed immediately and will read the next EF from the Comms EF Input Buffer. If the next EF is the SOM, the Comms Controller will flush the comms output buffer and the comms data input buffers and call a subprogram that handles the portion of the protocol for receiving a message from the external computer. Flushing the buffers will require a critical section.

3.2.4. Keyboard

There is one salient intertask relationship involving the keyboard: that between the Keyboard ISR and the Command Processor. The data flow between these two tasks is mediated by a ring buffer, the keyboard input buffer.

3.2.5. Screen

The interaction between the Screen Controller and the Screen ISR is similar to that between the Comms Controller and the Comms Send ISR. A buffer is used to mediate differences in production and consumption rates.

3.2.6. Remaining Tasks

The Alerts Manager and Data Extractor are monitor tasks. The Alerts Manager arbitrates multiple clients' requests to write an alert to the screen, and the Data Extractor arbitrates multiple clients' requests to record significant events on disk.

3.2.7. Process Interleaving

The previous sections highlighted the important task interactions. This section briefly presents task interaction from the viewpoint of sharing the processor.

In each 2.56 millisecond interval, a portion of time is committed to executing the Clock ISR (which includes the calculation of ship attitude and heading), leaving only a certain amount of free time in which to execute all the other tasks.

Figure 3-3 depicts rough estimates for committed and free time allocations for one of the intended target systems, VAXELN. Also shown is a typical example of process interleaving:

1. A base-level task is executing in the free time, (i.e., is regularly preempted by the Clock ISR).
2. At one of these clock interrupts, the Dispatcher finds that a clock-level task is due to be initiated and sends it a signal.
3. When the Clock ISR is exited, the Ada scheduler finds that this clock-level task is now the highest priority eligible task and transfers CPU control to it.
4. The clock-level task then executes in the free time, possibly over more than one basic clock cycle.
5. When the clock-level task has completed its function, it voluntarily suspends itself by waiting for another signal.
6. The Ada scheduler then transfers control to the base-level task, which resumes executing in the free time.

Note that in this implementation the Dispatcher is not a separate task but is part of the Clock ISR and that clock-level tasks are not activated by an Ada rendezvous but by a faster runtime system signal/wait mechanism. Also note that entering or exiting an interrupt service routine should not require a context switch in the usual Ada sense, but that two context switches are required for each activation of a clock-level task.

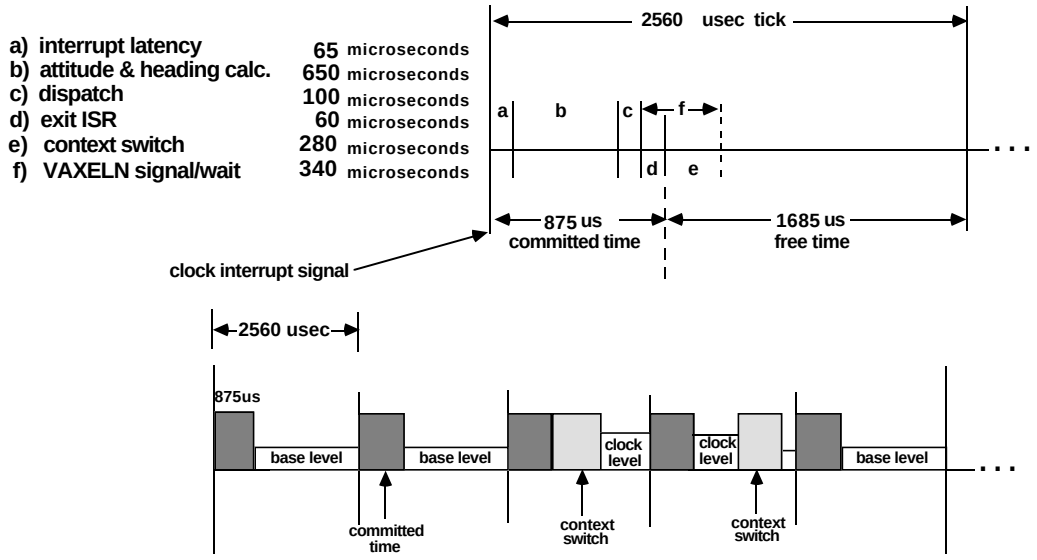


Figure 3-3: Process Interleaving

3.3. Potential Modifications and Tradeoffs

This section enumerates areas that are potential targets for future modification or enhancement depending upon the results of our experimental activities in incremental development.

Areas related to performance of the Ada rendezvous:

1. On the available target runtime systems, an Ada task rendezvous takes 1 millisecond or more, which is an unacceptable overhead for some critical functions. In such cases, a faster signal/wait mechanism must be used. This mechanism may be provided by the runtime system services, or it may have to be coded in assembler. Such a mechanism most likely will be incompatible with existing Ada constructs (e.g., select statement) thus the general design should remain within this potential restriction.
2. The Dispatcher currently rendezvous with the two separate tasks to initiate the sending of attitude and navigation messages. A better solution may be to have the Dispatcher rendezvous directly with the Comms Event Manager to post a request to send a periodic message.
3. Since the attitude calculation (performed in the Ship Attitude Updater) is to be performed at the same frequency as clock-generated interrupts, it should be moved into the Clock ISR.

Areas related to implementation support for ISRs:

1. The ability to treat a hardware interrupt as an entry in an ordinary task may not be provided in Ada implementation (one example is DEC VAXELN). A signal/wait mechanism must be used instead. The general design should be capable of being modified to handle such cases.
2. An explicitly coded ISR may not be needed for the screen and keyboard. The use of Ada TEXT_IO will be explored.

Areas related to contention for resources:

1. An alternative to be considered is using simple lock/unlock mechanisms to control access to data stores shared by independent tasks (rather than implementing a full-fledged monitor task).
2. Input tables may require a monitor if precision is such that assignment is not an atomic action. This needs to be further investigated.
3. To ensure that calculations use and messages send time-consistent data, a critical section may be necessary when extracting data from the results table for constructing messages and when extracting data that is subsequently reused in other calculations.

4. Module Structure

This chapter describes the top-level structure of the INS simulator program. It brings together the data flow, concurrency, and control elements discussed in the previous chapters and assembles them into a hierarchical module structure. The purpose of the chapter is to define the top-level modules, their external interfaces, and their mutual dependencies.

Although the eventual physical modules will be Ada packages, the top-most partitioning of the INS simulator program is into subsystems, which are groups of logically related packages. Figure 4-1 depicts these subsystems with their constituent packages and indicates the major dependencies between the subsystems.

The individual subsystems are described in the sections that follow the figure. The dependencies between packages within the subsystem and between subsystems are described, with emphasis on the latter. The description of each subsystem is based on a structure diagram, which uses the pictorial conventions defined in Figure 4-2. The locations of previously identified tasks are shown, but individual data types, objects, and subprograms are not detailed here. (These will appear in the detailed design document and its package specifications.)

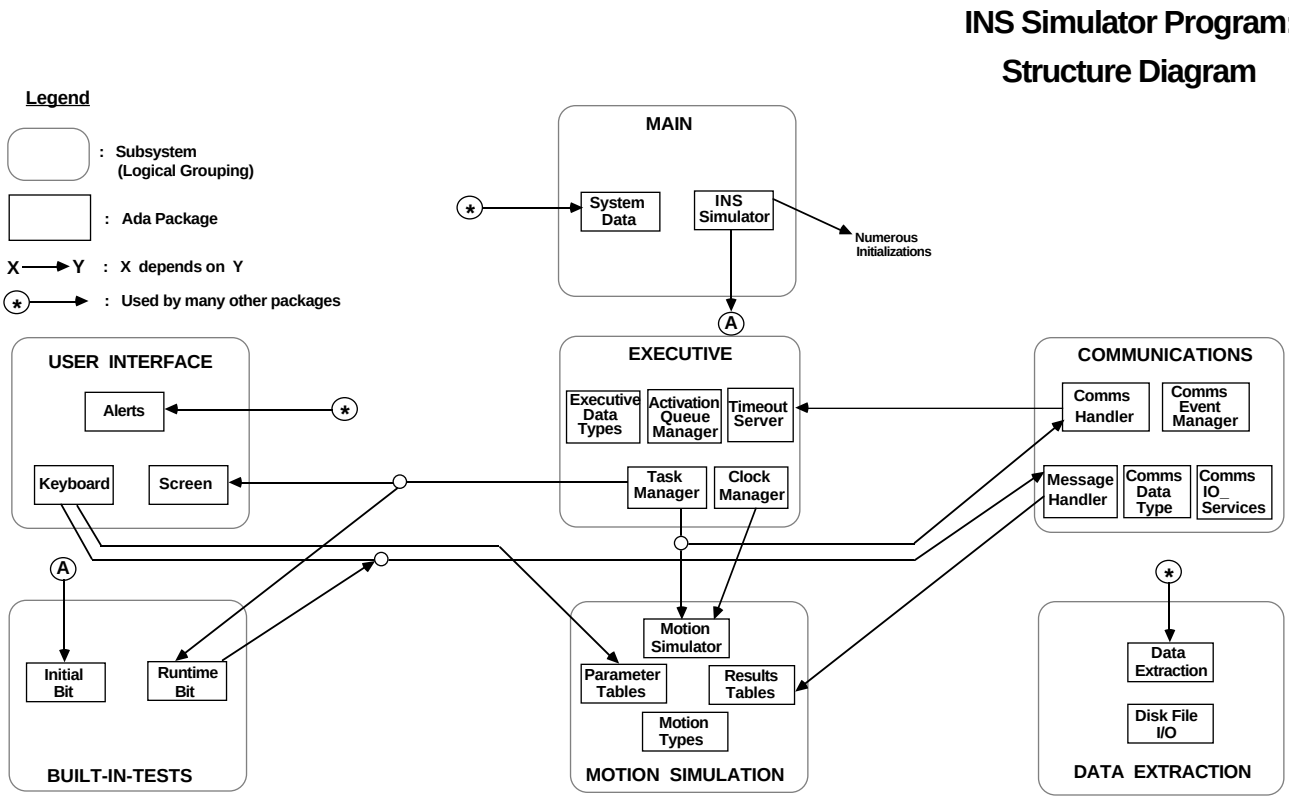


Figure 4-1: INS Simulator: Top-Level Structure Diagram

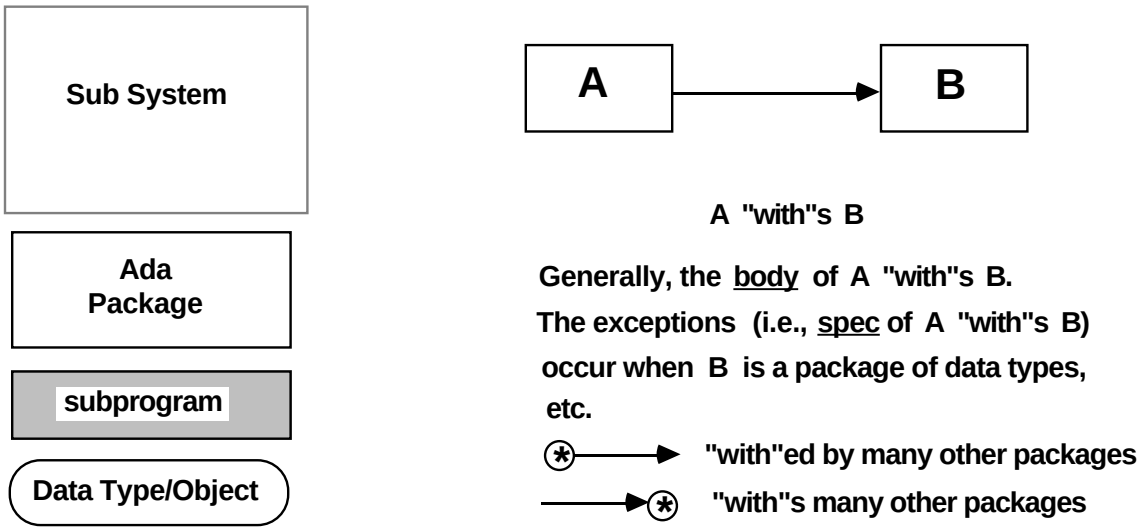


Figure 4-2: Legend for Subsystem Structure Diagrams

4.1. Main Subsystem

As shown in Figure 4-3, the main subsystem consists of two modules:

- System_Data package, a (small) collection of global data
- INS_Simulator, the main program itself

The rationale for this subsystem grouping is that none of the other subsystems is an appropriate place for these two modules.

4.1.1. System_Data Package

The System_Data package corresponds to the system data table described in Section 2.1.6 and contains at least the data items defined there. Additional data types and objects may well be added during detailed design.

4.1.2. INS_Simulator Main Program

Although it is an implicit thread of control, the main program is not shown as a task because of its special nature. It is initiated immediately after the elaboration of library packages, the elaboration of its own declarative part, and the initiation of all library tasks. It first calls the Initial_BIT procedure, followed by calls to various initialization procedures in the other subsystems. The details of these initializations are properly left to the detailed design stage (some may even be left to the coding stage because they are so implementation dependent).

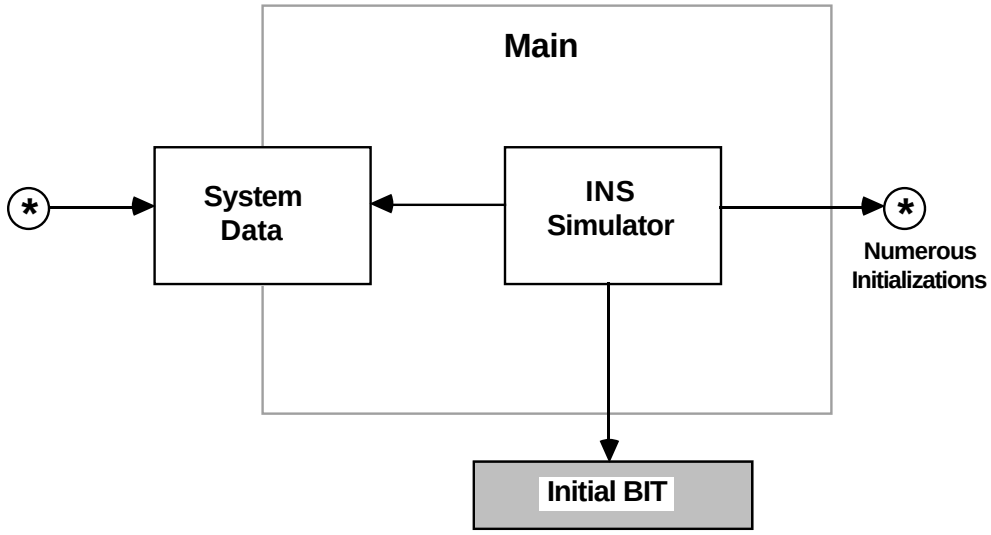


Figure 4-3: Main Subsystem: Structure Diagram

4.2. Executive Subsystem

As shown in Figure 4-4, the executive subsystem consists of five packages:

Clock_Manager
Task_Manager
Activation_Queue_Manager
Time_Out_Server
Executive_Data_Types

4.2.1. Clock_Manager Package

This package serves to isolate the real-time clock device dependencies and to export a set of sub-programs that can be used to interact with the clock, including initializing the clock. The Clock ISR resides in the Clock_Manager package. Upon a clock interrupt, this ISR is called; subsequently, it calls the Dispatcher which resides in the Task_Manager package.

4.2.2. Task_Manager Package

The Dispatcher resides in the Task_Manager. The Task_Manager exports an enumeration type that contains a set of task ids that are used when referring to a task. It exports procedures that allow periodic tasks to be enabled and disabled and a procedure that activates the Dispatcher task.

4.2.3. Activation_Queue_Manager Package

This package serves as an abstract data type for the activation queue. It provides a set of operations that act upon the activation queue, facilitating the insertion, deletion, and reading of activation records. These operations are needed by the Dispatcher and the Time_Out_Server.

4.2.4. Time_Out_Server Package

This package provides procedures to register requests to start and cancel a time-out clock.

4.2.5. Executive_Data_Types Package

This package contains data types that are common to other packages in the executive subsystem.

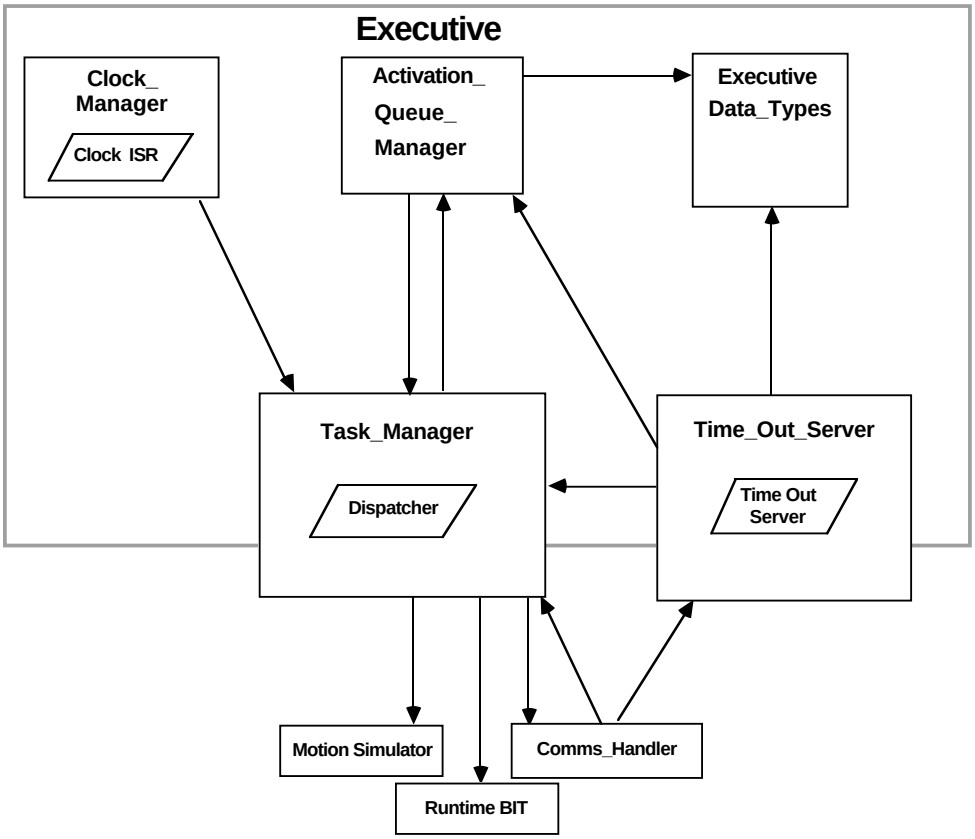


Figure 4-4: Executive Subsystem: Structure Diagram

4.3. User Interface Subsystem

As shown in Figure 4-5, the user interface subsystem consists of five packages:

Alerts_Manager
Keyboard_Manager
Keyboard_IO
Screen_Window_Manager
Screen_IO

4.3.1. Alerts_Manager Package

The Alerts_Manager package exports three procedural interfaces:

- Initialization (called by the main program)
- Register a new alert (called from several other subsystems and also from Keyboard_Manager)
- Show the next pending alert (called from the Keyboard_Manager)

4.3.2. Keyboard_Manager Package

The Keyboard_Manager package exports one procedural interface:

- Initialization (called from the main program)

4.3.3. Keyboard_IO Package

The Keyboard_IO package, which serves to isolate the keyboard device dependencies, exports two procedural interfaces:

- Initialization (called from Keyboard_Manager)
- Get a character (called from Keyboard_Manager)

4.3.4. Screen_Window_Manager Package

The Screen_Window_Manager package exports one task entry and several procedural interfaces:

- Initialization (procedure called from main program)
- Activate the Periodic_Display_Processor (task entry called from Task_Manager)
- Erase the command line
  Echo a character
  Backspace over a character
  Display an integer
  Display a real number (all called from Keyboard_Manager)
- Display an alert
  Display the number of pending alerts (both called from Alerts_Manager)
- Display communications status (called from Comms_Handler)
- Display data extraction status (called from Data_Extraction)

The initialization procedure is directly accessible; the others are accessible through the four window subpackages.

4.3.5. Screen_IO Package

The Screen_IO package, which serves to isolate the screen device dependencies, exports three procedural interfaces:

- Initialization (called from Screen_Window_Manager)
- Set the cursor (called from Screen_Window_Manager)
- Write a string (called from Screen_Window_Manager)

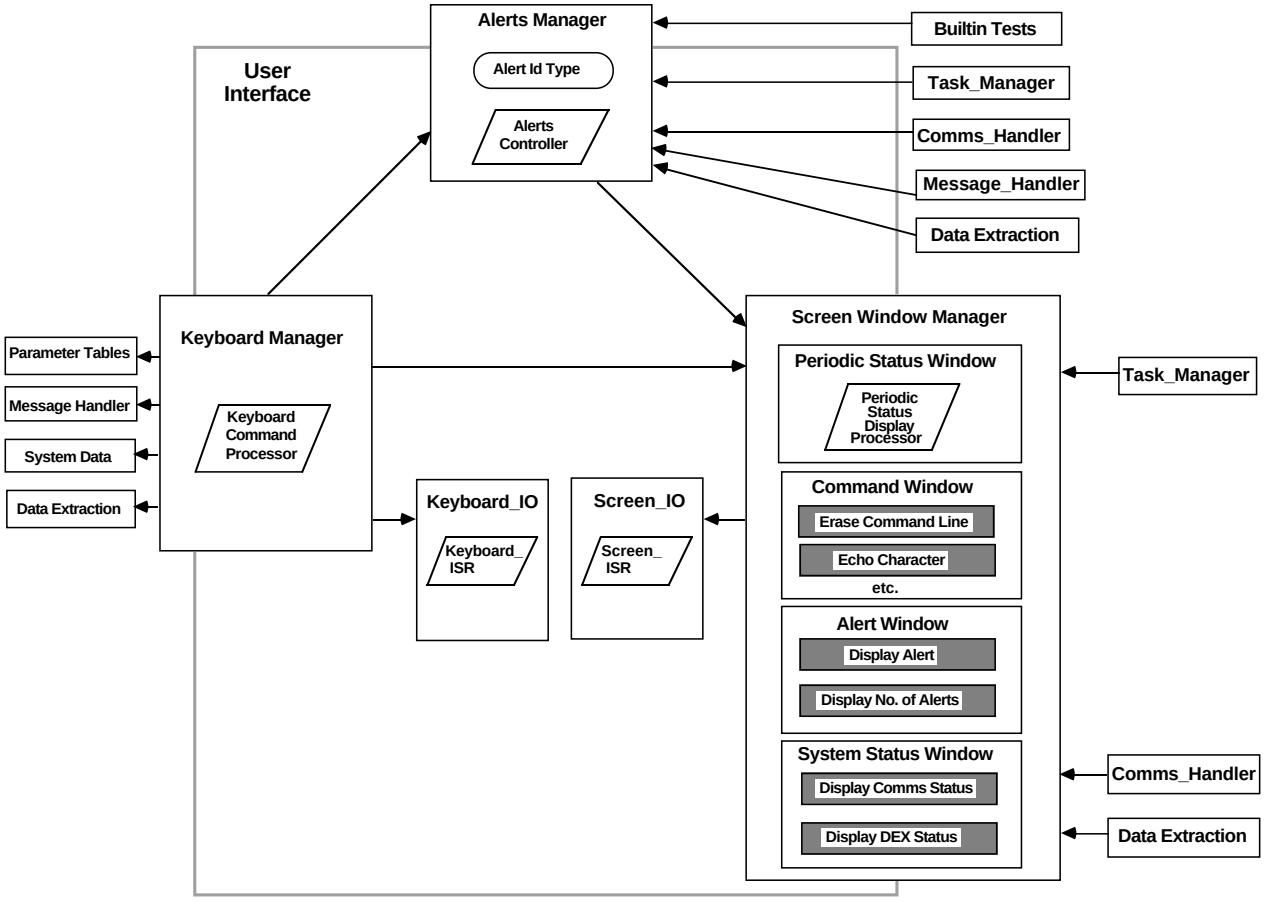


Figure 4-5: User Interface Subsystem: Structure Diagram

4.4. Motion Simulation Subsystem

As shown in Figure 4-6, the motion simulation subsystem consists of four packages:

Motion_Types
Parameter_Tables
Results_Tables
Motion_Simulator

4.4.1. Motion_Types Package

The Motion_Types package defines numerous data types and constants required by the other packages in the motion simulation subsystem. Detailed design may reveal that other subsystems also depend on the Motion_Types package.

4.4.2. Parameter_Tables Package

The Parameter_Tables package contains the sea-state, scenario, and current parameter tables described in Chapter 2. The current parameters table is accessed from Motion_Simulator, and all three tables are accessed from Keyboard_Manager.

4.4.3. Results_Tables Package

The Results_Tables package contains the data items described in Chapter 2 and is accessed by Message_Handler as well as Motion_Simulator.

4.4.4. Motion_Simulator Package

The Motion_Simulator package exports two procedural interfaces and two task entries:

- Initialization (procedure called from main program)
- Update_Ship_Attitude (procedure called from Clock_Manager)
- Activate the Ship_Velocity_Updater (task entry called from Task_Manager)
- Activate the Ship_Position_Updater (task entry called from Task_Manager)

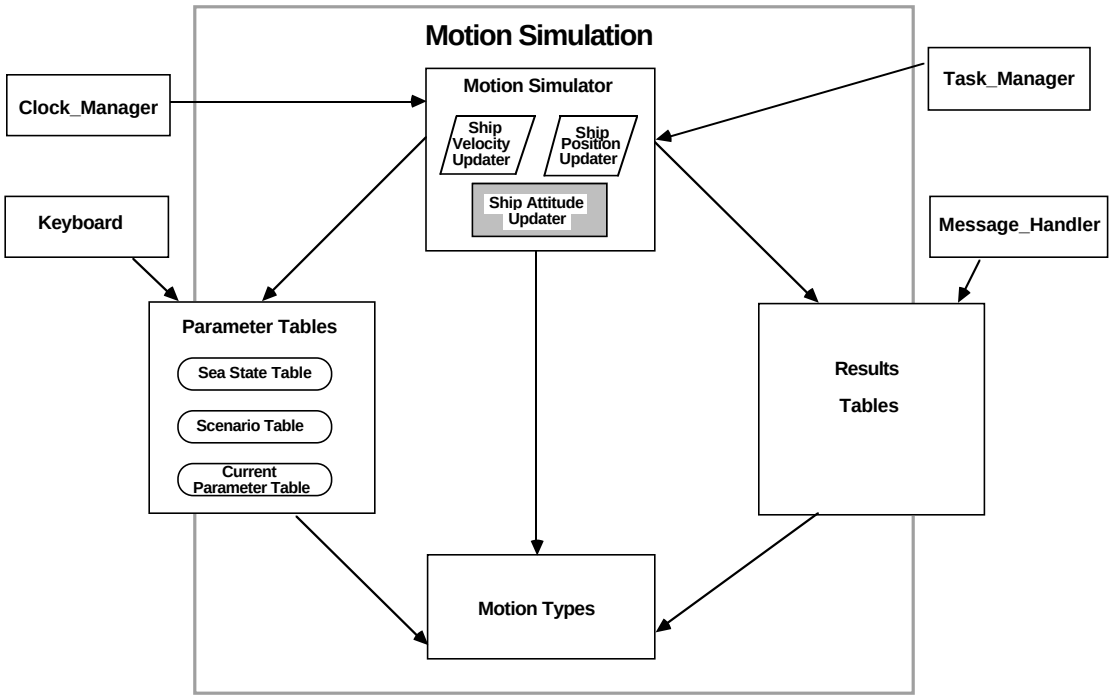


Figure 4-6: Motion Simulation Subsystem: Structure Diagram

4.5. Communications Subsystem

As shown in Figure 4-7, the communications subsystem consists of five packages:

Comms_Handler
Message_Handler
Comms_Event_Manager
Comms_IO_Services
Comms_Data_Types

4.5.1. Comms_Handler Package

This package contains the two periodic tasks which are called by the Dispatcher to initiate the sending of the attitude and navigation periodic messages. It also contains the Comms_Controller, the driver task of this subsystem that implements the communications protocol. The implementation of the protocol relies on services provided by Comms_IO_ Services, Comms_Event_Manager, and the Message_Handler.

4.5.2. Message_Handler Package

This package exports procedures that operate on the message input and output buffers and the faults buffer. It provides two views of the message buffers: an array of 16-bit words or a sequence of bits where specific bit ranges hold encoded data values. Specifically, it exports procedures to do the following: build and validate messages, set a fault value for a selected parameter, examine values of encoded fields in the message output buffer, read or write a word of the array.

4.5.3. Comms_Event_Manager Package

This package basically exports a set of procedures that are procedural interfaces to the entries of the Comms_Event_Manager task. There is an entry for each event that can be posted and a guarded entry for each event combination.

4.5.4. Comms_IO_Services Package

This package handles I/O to and from the communications interface. It provides the capability to send external functions and data words to the external computer, provides operations to act on the communications input buffers (EF buffer and a data buffer), and houses an ISR for sending data and one for receiving data.

4.5.5. Comms_Data_Types Package

Data types that are common to several packages in the communications subsystem reside in this package.

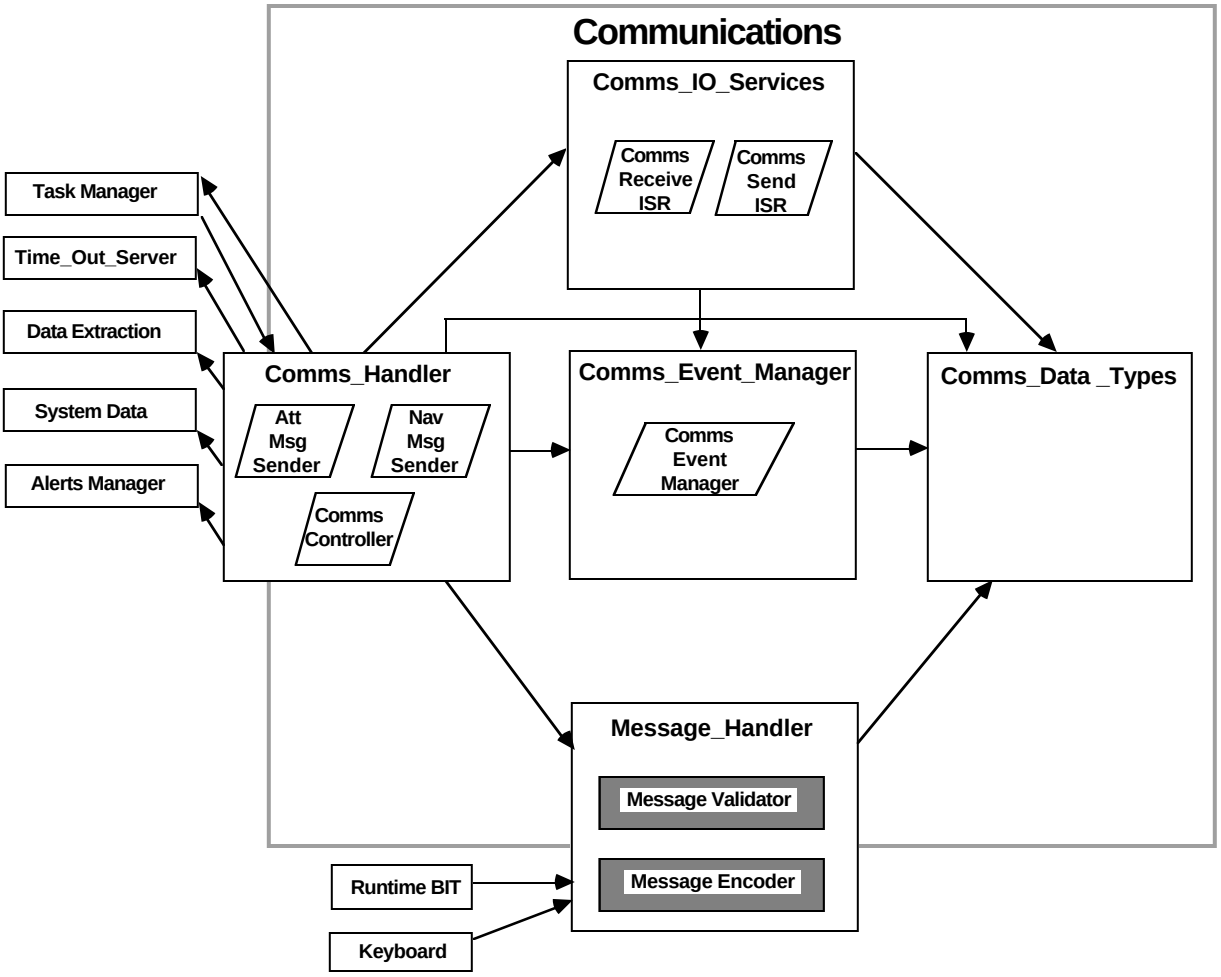


Figure 4-7: Communications Subsystem: Structure Diagram

4.6. Built-In Tests Subsystem

As shown in Figure 4-8, the built-in tests subsystem consists of two modules:

Initial_BIT subprogram
Runtime_BIT package

4.6.1. Initial_BIT Subprogram

The Initial_BIT is a special procedure called from the main program as described in Section 4.1 above.

4.6.2. Runtime_BIT Package

The Runtime_BIT package exports one task entry:

- Activate the Runtime_BIT task (entry called from Task_Manager)

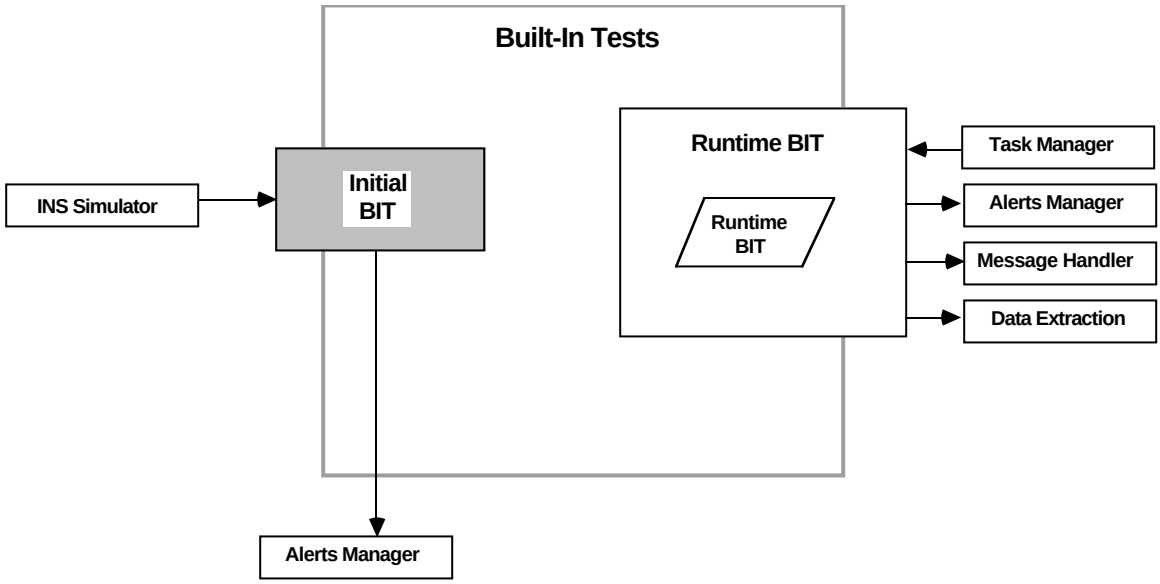


Figure 4-8: Built-In Tests Subsystem: Structure Diagram

4.7. Data Extraction Subsystem

As shown in Figure 4-9, the data extraction subsystem consists of two packages:

Data_Extraction
Disk_File_IO

4.7.1. Data_Extraction Package

The Data_Extraction package exports two procedural interfaces:

- Initialization (called from main program)
- Extract a data record (called from several other subsystems)

4.7.2. Disk_File_IO Package

The Disk_File_IO package serves to isolate the disk device dependencies and exports two procedural interfaces:

- Initialization (called from Data_Extraction)
- Output a data record (called from Data_Extraction)

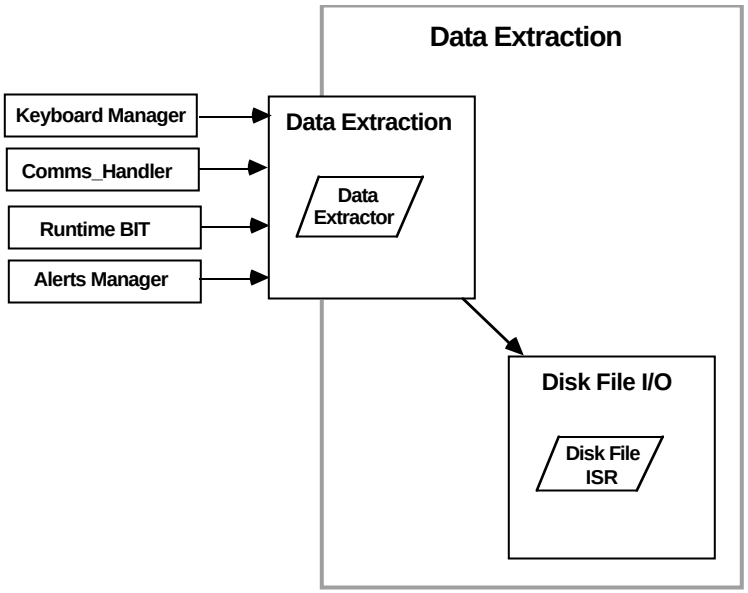


Figure 4-9: Data Extraction Subsystem: Structure Diagram

Glossary

AEST	Ada Embedded Systems Testbed (Project)
BIT	Built-In Test(s)
EC	External Computer (System)
EF	External Function (code—in intercomputer message protocols)
FIFO	First-In First-Out
GMT	Greenwich Mean Time
INS	Inertial Navigation System
ISR	Interrupt Service Routine
RV	(Ada task) rendezvous
SD	Structure Diagram
VAXELN	A real-time operating system for DEC VAX computers

References

[Allworth 81] Allworth, S.T.
Introduction to Real-Time Software Design
MacMillan, London, 1981

[Borger 86] Borger, M. W.
Ada Task Sets: Building Blocks for Concurrent Software Systems
Proceedings of the IEEE Computer Society Second International Conference on Ada Applications and Environments
Miami Beach, FL, April 1986

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

[LRM] United States Department of Defense
Reference Manual for the Ada Programming Language
ANSI/MIL-STD-1815A-1983, February 17, 1983

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

[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 87a] Meyers, B. C.
Systems Specification Document for an Inertial Navigation System Simulator and External Computer
Software Engineering Institute, February 1987. To be published.

[Meyers 87b] Meyers, B. C.
Functional Performance Specification for an Inertial Navigation System Simulator
Software Engineering Institute, February 1987. To be published.

[Meyers 87c] Meyers, B. C.
Functional Performance Specification for an External Computer to Interface to an Inertial Navigation System Simulator
Software Engineering Institute, February 1987. To be published.

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

Appendix A: Allocation of Task Priorities

The principles governing the allocation of task priorities are given in Section 3.2.1.3 of the main document. This appendix lists the current allocation of task priorities.

A.1. Current Allocation

Priority		Period	
Ada	Activation	(msec)	Task
15	-	-	Reserved for debug and monitoring
14	-	2.56	Clock_ISR (includes Ship_Attitude_Updater)
13	-	-	Comms_Receive_ISR
	-	-	Comms_Send_ISR
12	-	-	Keyboard_ISR
11	-	-	Screen_ISR
10	10	5.12	Timeout_Server
"	-	-	Comms_Event_Manager
9	-	-	Comms_Controller
8	8	40.96	Ship_Velocity_Updater
7	7	61.44	Attitude_Periodic_Message_Sender
6	-	-	Screen_Controller
5	-	-	Command_Processor
4	4	983.04	Navigation_Periodic_Message_Sender
3	3	1000.	Periodic_Window_Updater
2	2	1000.	Run_Time_BIT
1	1	1300.	Ship_Position_Updater
0	-	-	Reserved for debug and monitoring

Table A-1: Allocation of Task Priorities

Table of Contents

List of Figures

List of Tables