Technical Report
CMU/SEI-87-TR-032
ESD-TR-87-195

VAXELN Experimentation: Programming a Real-Time Periodic Task Dispatcher Using VAXELN Ada 1.1

Mark W. Borger

November 1987

Technical Report
CMU/SEI-87-TR-32
ESD-TR-87-195
November 1987

VAXELN Experimentation: Programming a Real-Time Periodic Task Dispatcher Using VAXELN Ada 1.1

Mark Borger
Ada Embedded Systems Testbed Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
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

Karl H. Shingler SIGNATURE ON FILE
SEI Joint Program Office

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

VAXELN Experimentation: Programming a Real-Time Periodic Task Dispatcher Using VAXELN Ada 1.1

Abstract. The purpose of this paper is to provide the reader with some technical information and observations, Ada source code, and measurement results based on experimentation with respect to developing a real-time periodic task dispatcher in Ada. The results presented here are specific to a μVAX-II/VAXELN 2.3 target system, the VAXELN 1.1 Ada compiler, and a KWV11-C programmable real-time clock. Specifically, these results provide answers to the question: *How can one achieve the effect of scheduling a set of periodic*[1] *Ada tasks when the runtime frequency of some of the individual tasks is less than the clock cycle frequency supported by an Ada runtime implementation?*

Executive Summary

1. Background

The Ada Embedded Systems Testbed Project's investigative approach promotes three typical stages to developing real-time systems: benchmarking; experimentation and prototyping; and designing, coding, and testing an application. To study the performance characteristics of Ada cross-compilers, we are running several existing benchmark test suites to explore the time, space, and capacity constraints associated with individual Ada features. To minimize programming risks such as those inherent in developing low-level device interfaces, we are performing evaluation experiments (i.e., prototyping) to explore programming alternatives available to an application developer, implementation strategies employed by a compiler vendor, and real-time ramifications with respect to using Ada in these high risk areas. We are also designing and implementing an application that is characteristic of real-time embedded systems. This application system provides a context for using the experiment and benchmark results and will be the primary vehicle for investigating the portability of Ada code across several target processors.

The intent of this experimentation was to investigate various programming alternatives available to an application developer for writing a real-time periodic task dispatcher in Ada. The approach was to design and prototype alternative versions of a task dispatcher for the Inertial Navigation System (INS) [INS Specification 87, INSP TLDD 87] simulator being developed by the project to support a detailed schedulability analysis of the INS periodic task set.

[1]In this context, a periodically scheduled task set implies that each task in the set is executed at its own fixed frequency. A periodic task dispatcher is a software component that schedules the individual tasks at their implied runtime frequency.

2. Scope

For this particular target configuration and cross-compiler (VAXELN 2.3/VAXELN Ada 1.1), a total of four different (prototype) periodic task dispatchers were developed. Two different periodic task dispatching approaches were used; for each of these, two different synchronization techniques were used, namely, the Ada rendezvous and the VAXELN semaphore. This paper first discusses the rationale for needing a real-time periodic task dispatcher and then presents the high-level design from which the prototypes were developed. Next, the task dispatcher prototypes are described in some detail, as is the experimentation approach used to test their feasibility. Finally, the empirical results are presented and analyzed, and relevant technical observations are provided.

1. Real-Time Periodic Task Dispatcher

The Ada tasking mechanism provides the real-time application programmer with a facility to do multi-tasking. The decision to use Ada multi-tasking depends mainly on the scheduling requirements of the application. Real-time applications can be classified into three categories by their inherent scheduling requirements [MacLaren 80]: (1) purely periodic scheduling with no aperiodic events, (2) primarily cyclic with some aperiodic events and possible variations in computing loads, and (3) event-driven (totally aperiodic) and no periodic scheduling. Common practice has been to employ a cyclic executive for all three levels, but it has been shown that the benefits of Ada multi-tasking (e.g., supports aperiodic events, monitors intertask dependencies, controls task interaction, and supports cyclic processing at arbitrary frequencies) can be realized with applications having scheduling requirements falling the latter two categories [MacLaren 80]. With Ada multi-tasking, the runtime is responsible for scheduling tasks, whereas with a cyclic executive the application programmer controls the scheduling.

The Inertial Navigation System simulator must not only schedule[2] periodic tasks for execution, but also must handle the scheduling of aperiodic tasks.[3] Its scheduling requirements therefore fall into the second category above. As such, we decided to use Ada tasking wherever possible to meet the application's scheduling requirements. This chapter first motivates the need for a real-time periodic task dispatcher executing on top of the Ada runtime system. It then presents a high-level description of the design of the INS executive subsystem that supports the scheduling of the INS task set via the real-time task dispatcher.

1.1. Motivation and Rationale

One of the most important concerns for developing a real-time application is satisfying timing requirements. The INS simulator has certain real-time requirements that it must meet:

1. scheduling periodic tasks at frequencies of 400, 25, 16, and 1 Hz;
2. providing a task time-out service that must notify waiting tasks after expiry of 10.24 ms; and
3. supporting a time stamp mechanism at a granularity of 2.56 ms.

The ***delay*** statement in Ada was designed to aid in satisfying timing deadlines. However, validated Ada compilers to date have implemented the semantics of this statement by only ensuring that the task that executes it will be suspended from further execution for *at least* the duration specified, rather than supporting a guaranteed upper bound on the duration of time a task's execution will be suspended. To further aggravate this problem, the validated Ada compilers investigated to date have at best supported a 10 ms clock cycle (SYSTEM.TICK). These issues in combination with the INS simulator's requirement for a fine-grained (2.56 ms) notion of time serve as the rationale for using a programmable real-time clock and a real-time task dispatcher on top of the Ada runtime system for supporting periodic task scheduling.

---

[2]We use the term "schedule" loosely in this report to mean that an Ada task has been marked **ready** to be scheduled by the Ada runtime task scheduler.

[3]For example, the INS communication subsystem irregularly requests time-outs through an aperiodic task.

1.2. Top-Level Design

This section provides an overview of the INS simulator's executive subsystem design, which serves as a prototype of the INS simulator's real-time task dispatcher. This subsystem consists of three major components, namely a Real-Time Clock Manager, an Activation Queue Manager, and a Task Manager, each of which is represented by one Ada package as shown in Figure 1-1.

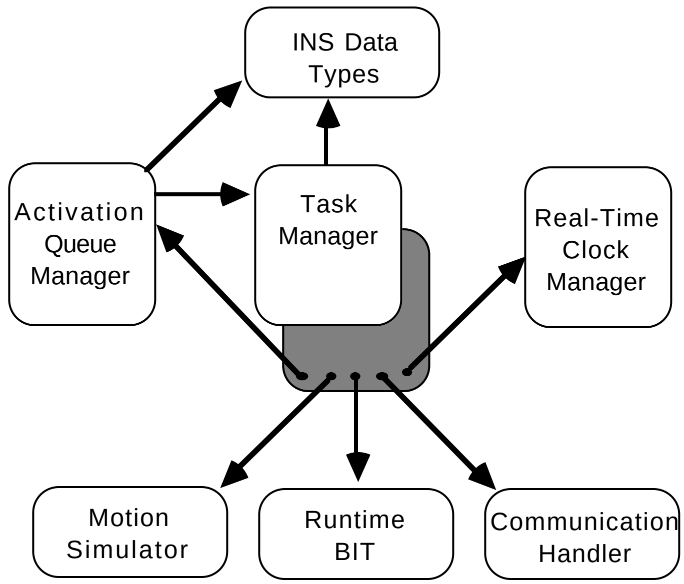

Figure 1-1: INS Executive Subsystem - Package Dependencies

The rounded, unshaded rectangles in the figure represent Ada package specifications, whereas the shaded one represents package bodies; the arrows indicate the dependency relationships (an arrow from A to B implies that A depends on B). The three packages at the bottom of the diagram are a subset of the packages that the executive imports from other INS subsystems to gain visibility of the periodic tasks that are part of the task set. The remaining packages constitute the executive subsystem whose responsibilities include scheduling the periodic task set and servicing time-out requests and cancellations. The following sections briefly describe each of these packages.

1.2.1. INS Data Types

The INS Data Types package (see Appendix A.e) of the INS executive subsystem provides the common data types used by the other packages. Specifically, it defines a data type for representing the executive's notion of time (i.e., the number of ticks since program invocation).

1.2.2. Real-Time Clock Manager

The Real-Time Clock Manager component of the INS executive subsystem provides a set of Ada interfaces to a KWV11-C programmable real-time clock [LSI-11 User's 86]. This component consists of one Ada package (see Appendix A.a — A.d) that provides the necessary data types, procedures, functions, and exceptions for interfacing to multiple KWV11-C real-time clocks via Ada application code [Clock TR 87]. These routines support all four modes of the clock's operation (Single Interval Interrupt, Repeated Interval Interrupts, External Event Timing Zero Base, and External Event Timing Cumulative) in addition to its five different internal clock rates (1 MHz, 100

KHz, 10 KHz, 1 KHz, 100 Hz). In addition to providing a mechanism for establishing a link between clock interrupts and an Interrupt Service Routine (ISR), the Real-Time Clock Manager supports typical programmable clock operations such as setting the clock's operation mode (e.g., repeated interrupts), setting the clock frequency, enabling and disabling clock interrupts, and programming the clock interrupt period.

1.2.3. Activation Queue Manager

The Activation Queue Manager component of the INS executive subsystem implements a single time and priority ordered task activation queue. This component is represented in the design as one package named ***Activation_Queue_Manager***. The package specification (see Appendix A.o, A.p) exports the necessary data types, procedures, and exceptions for accessing the elements of the time-priority ordered task activation queue. Specifically, the package specification defines a data type that represents a task activation record (AR) so that the users of this package can build such data objects. An AR contains the task's name, activation period, activation time, execution priority, and its activation mode (e.g., periodic, aperiodic). The Activation Queue Manager supports typical queue operations such as inserting, fetching, deleting, and re-inserting for activation records via the exported procedural interfaces.

The implementation details of the task activation queue are hidden in the package body. The prototyping described in Chapter 2 presents the details of two different implementations of the activation queue and its corresponding operations.

1.2.4. Task Manager

The Task Manager component of the INS executive subsystem provides a centralized task name service for the entire INS simulator program in addition to supporting the operations of enabling, disabling, and querying the schedulability status (e.g., enabled for activation) of periodic INS tasks. It is represented in the design as one package named ***Task_Manager*** (see Appendix A.q, A.r). The Task Manager also provides a mechanism for registering and canceling time-out requests from the communications subsystem. The package specification exports an enumeration type that contains an enumeration literal for each task in the INS task set. The package exports subprograms to support the aforementioned operations on any of these tasks. Furthermore, the package specification exports a procedure for initializing the INS task activation queue and one for initializing the real-time clock and activating the ***Dispatcher*** task. Initializing the activation queue involves inserting activation records for each of the pre-defined periodic tasks within the INS. The process of programming the real-time clock involves setting up the mode, rate, and Interrupt Service Routine. Finally, the Task Manager implements a real-time periodic task dispatcher on top of the task services provided by the Ada runtime using interrupts generated from a real-time programmable clock.

To implement this task dispatcher, specific knowledge of the mapping between the task ID enumeration literals and the actual Ada task names within the INS simulator program is located in the package body. The ***Dispatcher*** task is a high priority Ada task within the INS simulator program. Its body has a loop that attempts to dispatch a new task at every clock interrupt. Inside the loop it first waits for the signal from the clock ISR indicating that an interrupt just occurred. It then updates its notion of time, namely the current tick number, and then requests, from the ***Activation_Queue_Manager***, an AR of a task that should be scheduled at the current time.

Finally, then, based on the activation mode of the task represented by the returned AR, it takes appropriate action.

1.2.5. Data and Control Flow

A brief description of the data and control flow of the INS executive subsystem follows. This discussion is relative to the data and control diagram appearing in Figure 1-2 and assumes a VAXELN target system.

Step	Description
1	Initialize the activation queue. Initializing the activation queue involves creating new activation records for each of the pre-defined periodic tasks within the INS and inserting those ARs into the activation queue. Depending on the activation queue management approach, either an index for the just-inserted AR is returned or the next tick number at which time a task needs to be scheduled is returned.
2	Program the real-time clock's settings. The process of programming the real-time clock involves setting up the mode, rate, and Interrupt Service Routine. The association between the hardware interrupt and the Ada ISR must be established through a VAXELN service (**CREATE_DEVICE**); this kernel routine returns a device object tag back to the caller; as can be seen in the data/control diagram, this information is passed back to the ***Activate Dispatcher*** subprogram.
3	Activate the task dispatcher and instruct the real-time clock to begin generating interrupts. Prior to starting the real-time clock, the ***Dispatcher*** task is activated via an Ada rendezvous from the ***Activate Dispatcher*** subprogram. The data passed to the ***Dispatcher*** is precisely the device object returned from the **CREATE_DEVICE** kernel service. The ***Dispatcher*** uses this data to properly synchronize with the clock interrupts. Upon activation of the ***Dispatcher***, the real-time clock is started.
	. . .
n	A real-time clock interrupt occurs. The VAXELN kernel transfers control to the ISR associated with the clock interrupt.
n+1	The ISR signals the ***Dispatcher*** using the VAXELN *Signal/Wait* mechanism.
n+2	The ***Dispatcher*** fetches the next AR from the activation queue.
n+3	The ***Dispatcher***, if necessary, activates the appropriate task for execution.

In Figure 1-2, rounded rectangles represent packages, rectangles correspond to individual subprograms in the body of the ***Task Manager***, and parallelograms are Ada tasks. Note: The ***Dispatcher*** task is in the body of the ***Task_Manager*** package.

A sample main program that initiates the executive subsystem is shown below.

```
with Task_Manager;

procedure INS is
begin

  Task_Manager.Initialize_Activation_Queue;
  Task_Manager.Activate_Dispatcher;

end INS;
```

After this initiation sequence, the ***Dispatcher*** runs autonomously, being driven by the real-time clock interrupts (step **n**) and continually performing steps **n+1**, **n+2**, and **n+3**.

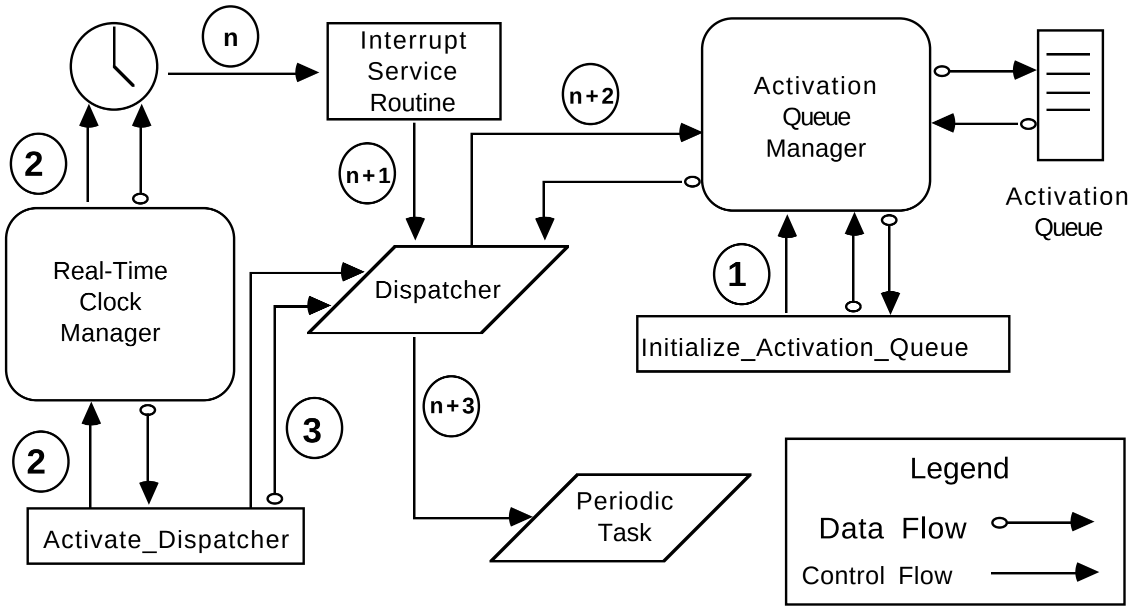

Figure 1-2: INS Executive Subsystem - Data and Control Flow Diagram

2. Real-Time Task Dispatcher Prototyping

To lessen the risks of implementing the INS simulator using Ada tasks, alternative prototype versions of the real-time periodic task dispatcher were developed to assess the schedulability of the INS periodic task set based on estimates of task execution times. This chapter presents the results of this system modeling and analysis.

2.1. Schedulability Analysis

To assess the schedulability of the INS periodic task set, the following four-step approach was taken.

Step 1 - Make real-time measurements

Prior to embarking on the modeling of the INS simulator tasking structure, it was essential to understand the internal operation of the underlying VAXELN [VAXELN Release 86, VAXELN User's 85] runtime executive. Key real-time measurements shown in Table 2-1 were either empirically obtained or taken from the VAXELN performance documentation.

Event	Time
Interrupt latency (VAXELN manual)	33 µsec
Context switch (VAXELN manual)	150 µsec
VAXELN signal/wait (empirical result, no process contention)	285 µsec
Ada rendezvous (empirical result)	1780 µsec
Attitude and Heading calculations (empirical result)	450 µsec

Table 2-1: VAXELN Real-Time Measurements

Step 2 - Estimate CPU utilization for task set

As a second step in the schedulability analysis, runtime estimates for each INS periodic task were made; execution time and CPU utilization estimates for the INS task set appear in Table 2-2. The execution time of the ***Attitude Updater*** was empirically measured to be 0.45 ms, whereas the runtime for the remaining periodic tasks was estimated. The overhead associated with each periodic task represents the context-switching time for entering and leaving the task (2 context switches = 0.30 ms); for the ***Attitude Updater*** the overhead represents the sum of interrupt latency and a context switch to the ***Dispatcher*** (0.03 + 0.15 = 0.18 ms). The synchronization times associated with each periodic task is 1.48 ms, which is the measured Ada rendezvous times less 0.30 ms for context switches; the 0.29 ms of synchronization time for the ***Attitude Updater*** corresponds to the VAXELN *Signal/Wait* time (see Table 2-1). If the analysis is correct, the implication is that only 15% of CPU time is available for the task dispatcher and background processing.

Task ID		Frequency (Hz)	Execution (ms)	Overhead (ms)	Synch (ms)	Execution Utilization (%)	Overhead Utilization (%)	Synch Utilization (%)	Total Utilization (%)
Attitude Updater		400	0.45	0.18	0.29	18.00	7.32	11.40	36.72
Velocity Updater		25	4	0.30	1.48	10.00	0.75	3.70	14.45
Attitude Sender		16	10	0.30	1.48	16.00	0.48	2.37	18.85
Navigation Sender		1	20	0.30	1.48	2.00	0.03	0.15	2.18
Status Display		1	100	0.30	1.48	10.00	0.03	0.15	10.18
Runtime BIT		1	5	0.30	1.48	0.50	0.03	0.15	0.68
Position Updater		0.8	25	0.30	1.48	2.00	0.02	0.12	2.14
Subtotals			164.45	1.98	9.17	58.50	8.66	18.03	85.19

Table 2-2: INS Periodic Task Set - Execution Time and CPU Utilization Estimates

Step 3 - Build INS tasking model

The next step of the analysis was the development of a skeletal INS tasking model. The control logic of each periodic task was virtually the same: an autonomous loop containing first a synchronization point at the top followed by code to perform the task's computation. For the sake of modeling, the computational load of each periodic task was represented using a busy wait mechanism whose variability was between 5 and 10 percent. For instance, the ***Velocity Updater*** task was instrumented with a 4 ms busy wait (see Table 2-2). This busy wait was implemented using an external subprogram call, and its basic unit of time measure was 100 µs; the routine was independently tested to be accurate to within 10%. To achieve the effect of varying the percentage of free CPU time, the duration of all of these busy waits was scalable using a global load factor. For example, a global load factor of 0.75 is equivalent to the duration of each task's busy wait being 75% of its estimated value (0.75 * 4 ms = 3 ms for the ***Velocity Updater***); a load factor of 1.25 increases the duration of the waits to 125% of their estimated values.

Step 4 - Monitor missed deadlines

The final step of the analysis was to vary the global load factor and monitor the model behavior with respect to missed deadlines. For each dispatching technique under investigation, the global load factor was continually increased by 0.05 (its fixed point delta) until a task deadline was missed. This critical load factor value, termed the **schedulability threshold**, was empirically determined for each dispatching alternative implemented. These periodic task dispatching prototypes are described in the next section.

2.2. Periodic Task Dispatching Alternatives

Given the high level design abstraction for the Activation Queue Manager, described in Section 1.2.3, two different queue management approaches were implemented, each associated with its own periodic task dispatcher. For each of these two different task dispatching prototypes, two different synchronization techniques were employed, namely the Ada rendezvous and the VAXELN semaphore. This section describes the two dispatching approaches, hereafter referred to as the general-purpose queue management (GPQM) and the static queue management (SQM) approaches.

2.2.1. General Purpose Queue Management

In the general-purpose queue management approach, the **ordered** activation queue is implemented as an array of indices into a table of existing activation records. Thus, the manipulation (e.g., insertion, deletion) of the ARs in the queue essentially involves the proper maintenance of these indices and the AR table entries. For instance, inserting a new AR into the queue involves creating a new entry in the AR table, locating the proper queue position of this new AR based on its activation time and priority, and finally inserting its AR table index at the proper queue position while at the same time relocating any other queue elements affected by the insertion. Deletion of a specific element is similar in logic to insertion; however, at present, no mechanism is in place for reclaiming space in the AR table when ARs are deleted. Fetching an AR, of course, removes the element from the head of the ordered queue.

In this implementation, the task ***Dispatcher*** calls the Activation Queue Manager (AQM) every clock tick (2.56 ms), passing it the current time (i.e., tick number). The AQM compares this time to the activation time of the AR at the head of the queue (in this implementation, the first array element); if the values are equal, then the first AR is returned; otherwise, a null AR is returned. When a non-null AR is returned (i.e., taken off the queue), its activation mode value is checked; if it represents a periodic task, a new activation time is computed, and the AR gets updated within the table and is re-inserted into the queue. It is possible that more than one AR meets the activation time criteria specified in the ***Get_Activation_Record*** call; in such cases the first AR is always returned since it is guaranteed to have the highest execution priority; the other qualifying ARs have their activation times incremented by 1 tick and are re-inserted into the queue; however, the original schedule for the delayed tasks is maintained.

2.2.2. Static Queue Management

In the static queue management approach, the activation queue is implemented as a statically sized array of activation records. The ARs are never moved from their initial position in the array, and one special array element is reserved for the AR of the ***Communications Controller*** task, which is called when a time-out has expired. In the purist sense, the data structure is not managed as an ordered queue, but rather as an array of elements, of which one is always marked as the next AR to be returned upon a fetch operation. In this scheme, the AQM maintains information regarding the next task to be scheduled and when to schedule it by performing a linear search of the array upon each insert and fetch operation. A benefit to this approach is that the need for special processing to resolve scheduling conflicts is obviated by the linear searching upon each fetch and insert operation, since the search implicitly resolves conflicts.

In this implementation, the task, ***Dispatcher*** calls the Activation Queue Manager only at the times when tasks are scheduled to be activated. Upon each insert (e.g., time-out request) and fetch (e.g., get next AR) operation, the AQM returns the next activation time. When an AR is returned (i.e., taken off the queue) to the ***Dispatcher***, its activation mode value is checked by the AQM; if it represents a periodic task, a new activation time is computed, and the AR gets re-inserted into the queue. To handle scheduling conflicts easily, the ***Dispatcher*** fetches ARs from the AQM when the current time is either equal to (no conflicts) or past (a conflict has occurred) that time specified by the AQM as the next time to schedule.

3. Results

Empirical results produced from the schedulability analysis are presented in this chapter from two different perspectives. First, a comparison of the two queue management approaches and their associated task dispatching prototypes is made by analyzing their effects on total CPU utilization when the synchronization mechanism is held fixed. Second, an analysis of the performance ramifications of the two synchronization techniques, namely the Ada rendezvous and the VAXELN semaphore, is done with respect to total CPU utilization. Finally, relevant technical observations are provided.

3.1. Dispatching Techniques

Tables 3-1 and 3-2 show that the calculations performed by the ***Attitude Updater*** require 18% CPU utilization and that the elapsed cycle time for the general-purpose queue management (GPQM) task dispatcher is 0.10 ms (0.32 - 0.22 = 0.10 ms) slower than the looping time of the static queue management (SQM) task dispatcher. These ***Dispatcher*** cycle times measure the elapsed time (from when the ***Dispatcher*** is signaled by the ISR) of resetting the clock's interrupt flag, updating the ***Dispatcher***'s notion of time, and fetching the next AR. However, this cycle-time measurement does not include the elapsed time for activating the next periodic task to be scheduled since this time has already been accounted for as the synchronization overhead associated with each periodic task. Note: These cycle times were empirically measured using a programmable real-time clock.

Given the minor difference (0.10 ms) between the GPQM and SQM elapsed dispatching loop times, it is not surprising to find that their effective CPU utilization percentages differ by only 4% (12.8 - 8.8 = 4.0 [Tables 3-1 and 3-2]) regardless of the synchronization mechanism employed to schedule the periodic tasks. By adding in the corresponding context switching overhead (6%), the total CPU utilization percentage for each dispatching technique can be obtained. Since only one context switch, namely the one necessary to switch from the ***Dispatcher*** to another process context, is recorded as dispatching overhead for either approach, the relative difference of their total CPU utilization remains 4%. For instance, the difference in total CPU utilization percentage between the GPQM and SQM techniques using VAXELN semaphores for synchronization is 4% (97 - 93 = 4% [Table 3-2]). Comparing the ***Dispatcher*** segments of the two columns labeled "Estimate (100%)" in either Figure 3-1 or Figure 3-2 illustrates this small difference in total CPU utilization percentages attributable to the change in dispatching methods.

The imputation of the synchronization and context switching overhead for the individual periodic tasks depends on the synchronization mechanism in use. In the case of Ada rendezvous, 1.78 ms (2 context switches + synchronization time = 2 * 0.15 + 1.48 = 1.78 ms) of total synchronization overhead is charged to each periodic task; for VAXELN semaphores, only the signaling time of 0.28 ms is associated with the individual tasks since a context switch out the dispatcher has already been counted.

Task ID	Frequency (Hz)	Execution (ms)	Overhead (ms)	Synch (ms)	Execution Utilization (%)	Overhead Utilization (%)	Synch Utilization (%)	Total Utilization (%)
Attitude Updater	400	0.45	0.18	0.29	**18.00**	7.32	11.40	37
Velocity Updater	25	4	0.30	1.48	10.00	0.75	3.70	14
Attitude Sender	16	10	0.30	1.48	16.00	0.48	2.37	19
Navigation Sender	1	20	0.30	1.48	2.00	0.03	0.15	2
Status Display	1	100	0.30	1.48	10.00	0.03	0.15	10
Runtime BIT	1	5	0.30	1.48	0.50	0.03	0.15	1
Position Updater	0.8	25	0.30	1.48	2.00	0.02	0.12	2
Subtotals		164.45	1.98	9.17	58.50	8.66	18.03	85
Dispatcher Mode								
General/Rendezvous	400	**0.32**	0.15	0.00	**12.80**	**6.00**	0.00	19
Totals			2.13	9.17	71.30	14.66	18.03	**104**
Static/Rendezvous	400	**0.22**	0.15	0.00	**8.80**	**6.00**	0.00	15
Totals			2.13	9.17	67.30	14.66	18.03	**100**

Table 3-1: General/Rendezvous and Static/Rendezvous Estimated CPU Utilization[4]

It is clear from inspecting Figure 3-1 that the estimated CPU utilization associated with both the GPQM and SQM dispatching techniques, when using the Ada rendezvous for task synchronization, is equal to or exceeds 100%; obviously in these cases, the INS task set would not be schedulable without incurring missed deadlines. Nevertheless, empirically it is important to determine the critical point at which the task set becomes schedulable for each different dispatching approach. The **schedulability threshold** represents this critical scheduling point and by its very nature is expressed in terms of a percentage of the sum of the periodic tasks' estimated CPU utilizations. For example, a schedulability threshold of 82% for the INS task set implies that the tasks are schedulable (i.e., will not miss deadlines) for only up to, but not including, a periodic task set CPU utilization level that is 82% of the original estimate (see Tables 3-1 and 3-2).

Task ID	Frequency (Hz)	Execution (ms)	Overhead (ms)	Synch (ms)	Execution Utilization (%)	Overhead Utilization (%)	Synch Utilization (%)	Total Utilization (%)
Attitude Updater	400	0.45	0.18	0.29	**18.00**	7.32	11.40	37
Velocity Updater	25	4	0.00	0.28	10.00	0.00	0.70	11
Attitude Sender	16	10	0.00	0.28	16.00	0.00	0.45	16
Navigation Sender	1	20	0.00	0.28	2.00	0.00	0.03	2
Status Display	1	100	0.00	0.28	10.00	0.00	0.03	10
Runtime BIT	1	5	0.00	0.28	0.50	0.00	0.03	1
Position Updater	0.8	25	0.00	0.28	2.00	0.00	0.02	2
Subtotals			0.18	1.97	58.50	7.32	12.65	78
Dispatcher Mode								
General/Semaphore	400	**0.32**	0.15	0.00	**12.80**	**6.00**	0.00	19
Totals			0.33	1.97	71.30	13.32	12.65	**97**
Static/Semaphore	400	**0.22**	0.15	0.00	**8.80**	**6.00**	0.00	15
Totals			0.33	1.97	67.30	13.32	12.65	**93**

Table 3-2: General/Semaphore and Static/Semaphore Estimated CPU Utilization

[4]Since tasks under VAXELN Ada are implemented as separate processes, the process switching times in the table coincide with Ada task switches.

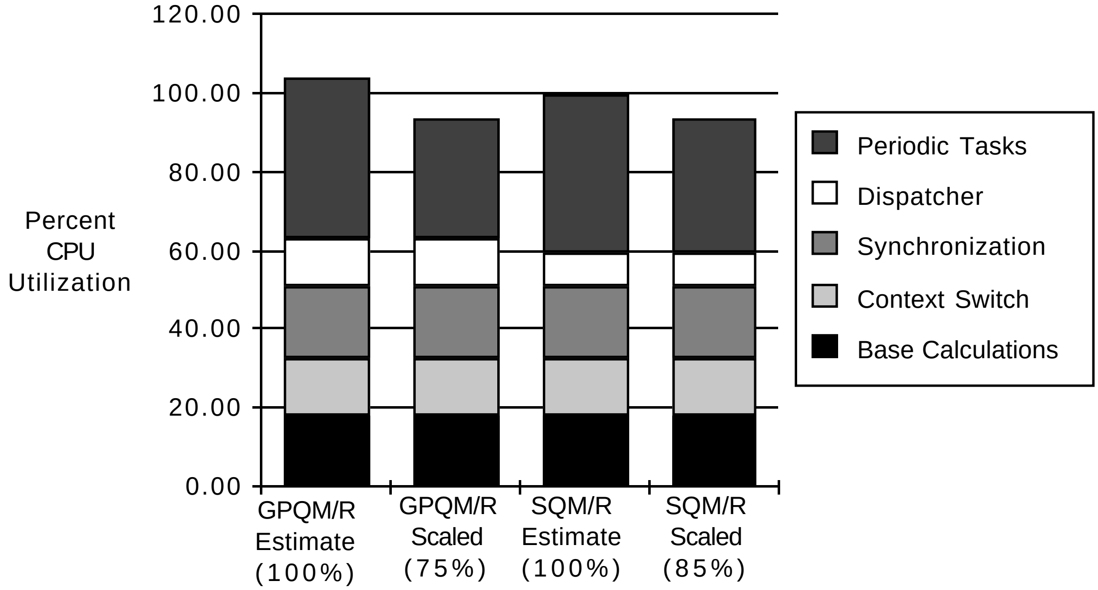

Figure 3-1: General/Rendezvous and Static/Rendezvous Scaled CPU Utilization

Since the amount of CPU utilization consumed by the periodic tasks varies directly with the value of the global load factor, the corresponding "Periodic Tasks" segments of the "Estimate" columns in Figures 3-1 and 3-2 must be adjusted so that the entire CPU utilization is below 100%, thus making the task set theoretically schedulable.

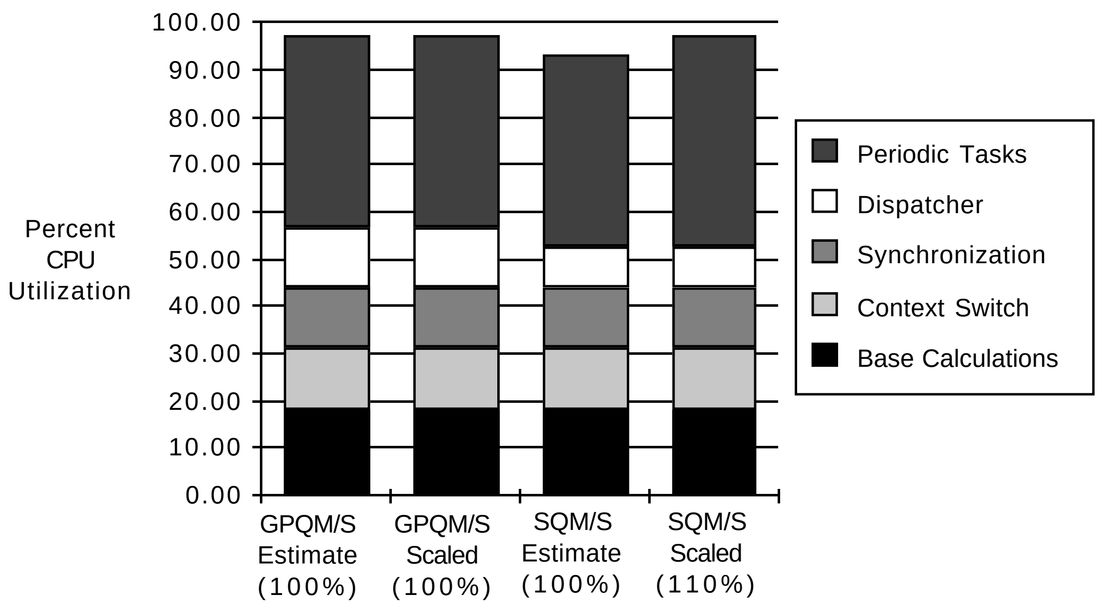

Figure 3-2: General/Semaphore and Static/Semaphore Scaled CPU Utilization

For example, the schedulability threshold for the GPQM ***Dispatcher*** using Ada rendezvous for task synchronization is 75%. One can observe from the first two columns of the bar chart in Figure 3-1 that the "Periodic Task" segment shrinks to 75% of its original size to reach a total CPU utilization level under 100%. The schedulability thresholds can be read from Figures 3-1 and 3-2 and are summarized in Table 3-3.

		Base Calculations (%)	Context Switch (%)	Synch (%)	Dispatcher Execution (%)	Periodic Tasks (%)	Total Utilization (%)	Schedulability Threshold (%)
GPQM/R Estimate		18.00	14.66	18.03	12.80	40.50	104	**75**
GPQM/S Estimate		18.00	13.16	12.65	12.80	40.50	97	**100**
SQM/R Estimate		18.00	14.66	18.03	8.80	40.50	100	**85**
SQM/S Estimate		18.00	13.16	12.65	8.80	40.50	93	**110**

Table 3-3: Estimated CPU Utilizations and Schedulability Thresholds

Interpretation of the schedulability threshold data in Table 3-3 indicates that, assuming the same synchronization mechanism, changing from the GPQM ***Dispatcher*** to the SQM ***Dispatcher*** yields a 10% (85 - 75 = 110 - 100 = 10%) increase in the schedulability threshold.

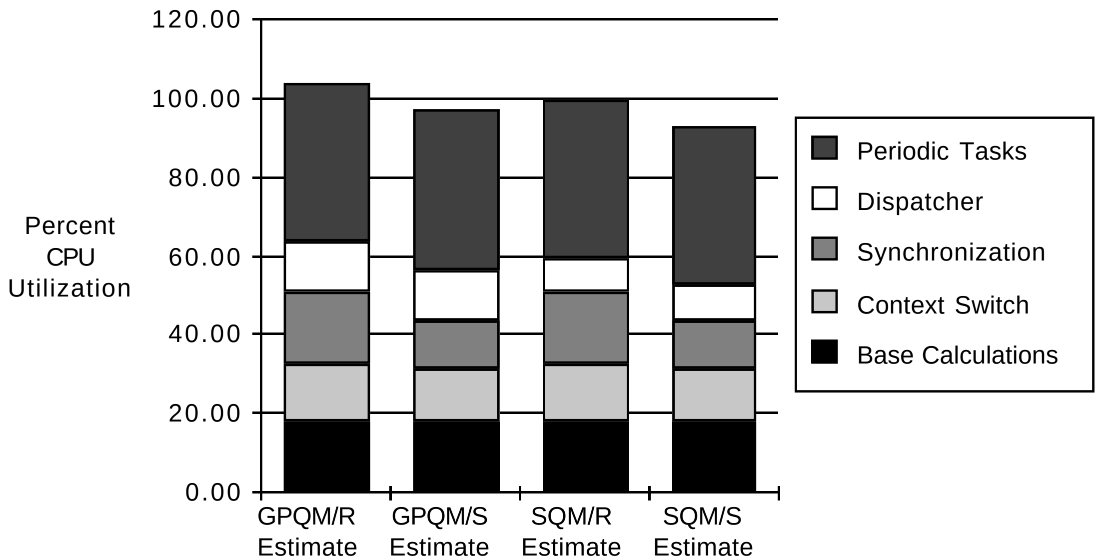

Figure 3-3: Rendezvous Versus Semaphore Comparison

3.2. Synchronization Mechanisms

The difference in total CPU utilization (computed from the data in Tables 3-1 and 3-2) when varying the synchronization mechanism used by the ***Dispatcher*** is 7%. Specifically, for the GPQM ***Dispatcher***, a change in its synchronization mechanism from the Ada rendezvous to a VAXELN semaphore results in a 7% (104 - 97 = 7%) savings in CPU utilization; for the SQM ***Dispatcher***, this savings is equal to 7% (100 - 93 = 7%). This implies that using VAXELN semaphores for task synchronization uses roughly 7% less CPU time than Ada rendezvous for this real-time periodic task dispatcher application.

Since the (estimated) execution times of both the INS simulator's base calculations and periodic tasks are constant, Table 3-3 can be used to illustrate the implications of the synchronization mechanism employed for scheduling the periodic tasks on total CPU. The bar chart (generated from this data) in Figure 3-3 clearly illustrates the pervasive effect of the Ada rendezvous on the percent of context switch, synchronization, and dispatching CPU utilization.

Finally, interpretation of the schedulability threshold data in Table 3-3 indicates that, assuming the same dispatching approach is being used, a 25% (100 - 75 = 110 - 85 = 25%) increase in the schedulability threshold results if the synchronization mechanism is changed from the Ada rendezvous to a VAXELN semaphore. Furthermore, a 35% improvement in the schedulability threshold is obtained when changing from the GPQM ***Dispatcher*** and the Ada rendezvous for synchronization to the SQM and VAXELN semaphores.

3.3. Technical Observations

The total estimated CPU utilization for the Interrupt Service Routine and the periodic task, without including the empirical results for the ***Dispatcher***'s utilization, is quite high. In the case of using Ada rendezvous for synchronization, it is 85%, and similarly for VAXELN semaphores, it totals 78%. It is clear from the tables in Tables 3-1 and 3-2 that a savings of 11% CPU utilization would be gained if the synchronization between the ISR and the ***Dispatcher*** could be eliminated. Quite simply this could be done by moving ***Dispatcher*** responsibilities into the ISR. In practice, however, this was not possible since numerous VAXELN Ada ISR restrictions limited the number of ***Dispatcher*** implementation alternatives. These ISR restrictions disallow tasking operations, input/output operations, and accessing variables not in the immediate scope of the ISR, and strongly recommend against making subprogram calls external to the ISR.

The empirical results illustrate the pervasive effect of the Ada rendezvous on the schedulability of the INS task set. Using the Ada rendezvous for synchronizing between the ***Dispatcher*** and the periodic tasks rather than VAXELN semaphores, regardless of the dispatching technique employed, results in an increase in total CPU utilization of 7%. Furthermore, for both dispatching methods implemented, given the original execution time estimates for the INS periodic tasks, using the Ada rendezvous as the synchronization mechanism results in missed task deadlines. Only when these estimates are scaled by 75% and 85% for the GPQM and SQM dispatching approaches, respectively, does the task set become schedulable assuming Ada rendezvous for task synchronization.

Interpretation of the schedulability threshold data in Table 3-3 further demonstrates the impact of the Ada rendezvous on the task set schedulability. The empirical results show that, assuming the same dispatching approach is being used, a 25% increase in the schedulability threshold results if the synchronization mechanism is changed from the Ada rendezvous to a VAXELN semaphore; moreover, a 35% improvement in the schedulability threshold is obtained when changing from the GPQM ***Dispatcher*** and the Ada rendezvous for synchronization to the SQM and VAXELN semaphores.

Based on real-time scheduling theory, the optimal rate-monotonic scheduling algorithm [Lui 73] guarantees schedulability of the INS task set for a processor utilization below 70% since the individual periodic tasks priorities are assigned in direct proportion to their execution frequencies.

However, since the INS task set CPU utilization is greater than 70%, another schedulability test based on the rate-monotonic algorithm, namely task-lumping [Sha 87], was necessary to calculate the theoretically expected schedulability thresholds. The schedulability thresholds determined empirically were consistent with those computed theoretically. For example, given the original execution time estimates for the INS periodic tasks, the SQM dispatching approach using VAXELN semaphore for task synchronization yielded a total CPU utilization level of 93%. Furthermore, it was found empirically that the task set was schedulable until the original time estimates of the periodic tasks were scaled by 1.1 or until the total CPU utilization level reached 97% (ISR + Scaled Periodic Tasks + Dispatcher = 37 + 1.1 * 41 + 15 = 97.1%). Similarly, solving for the schedulability threshold using the task-lumping method results in an expected threshold value of 1.12.

References

[Clock TR 87] Borger, M.W.
VAXELN Experimentation: Programming a Real-Time Clock and Interrupt Handling Using VAXELN Ada 1.1.
Technical Report CMU/SEI-87-TR-29, Software Engineering Institute, October, 1987.

[INS Specification 87]
Landherr, S.F., and Klein, M.H.
INS Behavioral Specification.
Technical Report CMU/SEI-87-TR-33, Software Engineering Institute, June, 1987.

[INSP TLDD 87] Klein, M.H., Landherr, S.F.
INS Simulator Program: Top-Level Design.
Technical Report CMU/SEI-87-TR-34, Software Engineering Institute, July, 1987.

[LSI-11 User's 86] *LSI-11 Analog System Users' Guide*
Digital Equipment Corporation, Maynard, MA, 1986.

[Lui 73] Liu, C.L., Layland, J.W.
Scheduling Algorithms for Multi-programming in a Hard-Real-Time.
JACM 20(1):46-61, January, 1973.

[MacLaren 80] MacLaren, Lee.
Evolving Toward Ada in Real-Time Systems.
In *Proceedings of the ACM-Sigplan Symposium on the Ada Programmng Language*. November, 1980.

[Sha 87] Sha, L., Lehoczky, J.P., and Rajkumar, R.
A Schedulability Test for Rate-Monotonic Priority Assignment.
Computer Science Department ART Project, Carnegie Mellon University, July, 1987

[VAXELN Release 86]
VAXELN Ada 1.1 Release Notes
Digital Equipment Corporation, Maynard, MA, 1986.

[VAXELN User's 85]
VAXELN User's Guide.
Digital Equipment Corporation, Maynard, MA, 1985.

Appendix A: INS Executive: Ada Source Code for SQM/Rendezvous Dispatcher

A.a. KWV_Register_Definitions Package Specification

```
--                                                                    --
-------------- SEI Ada Embedded Systems Project Prologue ---------------
--                                                                    --
-- Unit name    : KWV_Register_Definitions package specification
-- Experiment # : PA01
-- Version      : 1.0
-- Author       : Mark W. Borger
--              :
-- Date created : 20 Feb 1987
-- Last update  : 12 Mar 1987
--              :
-- Host Machine : VAX/VMS 4.5
-- Target Machine: VAXELN 2.3
--                                                                    --
------------------------------------------------------------------------
--                                                                    --
-- Abstract     :  This package specification provides the necessary
----------------:  data types to access the Control Status and Buffer
----------------:  Registers of a KWV11-C real-time programmable clock.
----------------:
----------------:
--                                                                    --
-------------------------- Revision History ----------------------------
--                                                                    --
-- Date      Version   Author              History
-- 12 Mar 87   1.0     Mark W. Borger      Added prologue
--
--                                                                    --
--------------------------- End of Prologue ----------------------------
--                                                                    --

with SYSTEM;            use SYSTEM;
with VAXELN_SERVICES;

package KWV_Register_Definitions is

  ----------
  --  KWV11-C Control Status Register layout
  ----------
  type KWV_CSR_RECORD is record
    go            : BOOLEAN;     -- start the counter
    mode          : UNSIGNED_2;  -- mode of operation
    rate          : UNSIGNED_3;  -- clock rate
    int_ovf       : BOOLEAN;     -- interrupt on overflow
    ovf_flag      : BOOLEAN;     -- counter overflow occurred
    maint_st1     : BOOLEAN;     -- simulate firing of st1
    maint_st2     : BOOLEAN;     -- simulate firing of st2
    maint_osc     : BOOLEAN;     -- simulate one cy. of osc
    dio           : BOOLEAN;     -- disable internal oscillator
    flag_overrun  : BOOLEAN;     -- true if ovf occurs with ovf_flag still set
    st2_go_enable : BOOLEAN;     -- assertion of st2_flag sets go bit
    st2_int_enable : BOOLEAN;    -- assertion of st2_flag causes an interrupt
    st2_flag      : BOOLEAN;     -- start interrupt request for st2
  end record;

  for KWV_CSR_RECORD use record at mod 2;
    go            at 0 range 0..0;
    mode          at 0 range 1..2;
    rate          at 0 range 3..5;
    int_ovf       at 0 range 6..6;
    ovf_flag      at 0 range 7..7;
    maint_st1     at 0 range 8..8;
    maint_st2     at 0 range 9..9;
    maint_osc     at 0 range 10..10;
    dio           at 0 range 11..11;
    flag_overrun  at 0 range 12..12;
```

```
    st2_go_enable  at 0 range 13..13;
    st2_int_enable at 0 range 14..14;
    st2_flag       at 0 range 15..15;
  end record;

  for KWV_CSR_RECORD'SIZE use 16;


  ----------
  --  KWV11-C Buffer/Preset Register layout
  ----------
  subtype KWV_BPR_TYPE is VAXELN_SERVICES.KWV_COUNTER_TYPE;


  ----------
  --  Record type containing the KWV11-C's CSR and Buffer/Preset Register
  ----------
  type KWV_REGISTERS is record
    CSR : KWV_CSR_RECORD;   -- control/status register
    BPR : KWV_BPR_TYPE;     -- buffer/preset register
  end record;
  pragma PACK(KWV_REGISTERS);



  procedure Put_CSR (CSR : in KWV_CSR_Record;
        Register_Address : in ADDRESS );


  function Get_CSR (Register_Address : in ADDRESS) return KWV_CSR_Record;


end KWV_Register_Definitions;
```

A.b. KWV_Register_Definitions Package Body

```
--                                                                       --
------------- SEI Ada Embedded Systems Project Prologue ---------------
--                                                                       --
-- Unit name    : KWV_Register_Definitions package body
-- Experiment # : PA01
-- Version      : 1.0
-- Author       : Mark W. Borger
--              :
-- Date created : 23 Mar 1987
-- Last update  :
--              :
-- Host Machine : VAX/VMS 4.5
-- Target Machine: VAXELN 2.3
--                                                                       --
-----------------------------------------------------------------------
--                                                                       --
-- Abstract     :  This package body provides the necessary interface
----------------:  for reading and writing the KWV11-C's CSR.
----------------:
--                                                                       --
-------------------------- Revision History --------------------------
--                                                                       --
-- Date      Version   Author               History
--
--                                                                       --
--------------------------- End of Prologue --------------------------
--                                                                       --

with UNCHECKED_CONVERSION;

package body KWV_Register_Definitions is

  function Convert_It is new UNCHECKED_CONVERSION(KWV_CSR_Record, UNSIGNED_WORD);
  function Convert_It is new UNCHECKED_CONVERSION(UNSIGNED_WORD, KWV_CSR_Record);

  procedure Put_CSR (CSR : in KWV_CSR_Record;
```

```
        Register_Address : in ADDRESS ) is

    Current_CSR   : UNSIGNED_WORD;
    CSR_Unsigned : UNSIGNED_WORD;
      for CSR_Unsigned use at Register_Address;

  begin
    Current_CSR := Convert_It(CSR);
    WRITE_REGISTER(Current_CSR, CSR_Unsigned);
  end Put_CSR;


  function Get_CSR (Register_Address : in ADDRESS)
    return KWV_CSR_Record is

    CSR          : KWV_CSR_Record;
    Current_CSR  : UNSIGNED_WORD;
    CSR_Unsigned : UNSIGNED_WORD;
      for CSR_Unsigned use at Register_Address;

  begin
    Current_CSR := READ_REGISTER(CSR_Unsigned);
    CSR := Convert_It(Current_CSR);
    return CSR;
  end Get_CSR;



end KWV_Register_Definitions;
```

A.c. Real-Time Clock Manager Package Specification

```
--                                                                    --
-------------- SEI Ada Embedded Systems Project Prologue --------------
--                                                                    --
-- Unit name    : KWV11_Clock_Manager
-- Experiment # : PA01
-- Version      : 1.0
-- Author       : Mark W. Borger
--              :
-- Date created : 17 Mar 1987
-- Last update  : 18 Mar 1987
--              :
-- Host Machine : VAX/VMS 4.5
-- Target Machine: VAXELN 2.3
--                                                                    --
------------------------------------------------------------------------
--                                                                    --
-- Abstract     :  This package specification provides the necessary
----------------:  data types, procedures, functions, and exceptions
----------------:  for interfacing to multiple KWV11-C real-time clocks
----------------:  (Q-bus device) via Ada application code.  All four modes
----------------:  of the clock's operation are supported in addition to
----------------:  its five different internal clock rates.  To use these
----------------:  routines one must first invoke the Initialize procedure
----------------:  to create a clock device object and get a clock identifier.
----------------:  This device object can be used by the application to wait
----------------:  on a device signal from an Interrupt Service Routine; the
----------------:  clock id is used as a key for the remainder of the package's
----------------:  interfaces.  The Initialization exception is raised if
----------------:  the VAXELN kernel device object cannot be created for
----------------:  whatever reason.  The Clock_Not_Initialized exception is
----------------:  if a specified clock id is invalid.
----------------:  These routines only support counter overflow interrupts
----------------:  and not Schmitt trigger interrupts.  The counter routines
----------------:  (Start_Counting, Read_Counter, Stop_Counting) should only
----------------:  be used in modes Mode_Two or Mode_Three; when used in any
----------------:  mode, the Invalid_Clock_Mode exception will be raised.
----------------:
----------------:
--                                                                    --
```

```
--------------------------- Revision History ---------------------------
--                                                                     --
-- Date      Version   Author               History
-- 18 Mar 87   1.0     Mark W. Borger       Added Display_CSR procedure.
-- 22 Mar 87   1.0     Mark W. Borger       Added Invalid_Clock_Mode exception.
--                                                                     --
--                                                                     --
---------------------------- End of Prologue ---------------------------
--                                                                     --

with VAXELN_SERVICES;
with CONDITION_HANDLING;
with SYSTEM;

package KWV11_Clock_Manager is

  ----------
  -- Data types imported from SYSTEM package
  ----------
    subtype ADDRESS is SYSTEM.ADDRESS;

  ----------
  -- Data types imported from CONDITION_HANDLING package
  ----------
    subtype COND_VALUE_TYPE is CONDITION_HANDLING.COND_VALUE_TYPE;

  ----------
  -- Data types imported from VAXELN_SERVICES package
  ----------
    subtype DEVICE_TYPE        is VAXELN_SERVICES.DEVICE_TYPE;
    subtype KWV_COUNTER_TYPE   is VAXELN_SERVICES.KWV_COUNTER_TYPE;
    subtype VECTOR_NUMBER_TYPE is VAXELN_SERVICES.VECTOR_NUMBER_TYPE;

  ----------
  -- Local Data types
  ----------
    type Clock_ID is private;

    type Clock_Mode is (Mode_Zero, Mode_One, Mode_Two, Mode_Three);

        for Clock_Mode use (Mode_Zero => 0, Mode_One   => 1,
                            Mode_Two  => 2, Mode_Three => 3);

    type Clock_Rate is (Stop,       Rate_1MHZ, Rate_100KHZ,
                        Rate_10KHZ, Rate_1KHZ, Rate_100HZ);

        for Clock_Rate use (Stop        => 0, Rate_1MHZ  => 1,
                            Rate_100KHZ => 2, Rate_10KHZ => 3,
                            Rate_1KHZ   => 4, Rate_100HZ => 5);


    procedure Initialize (Clock_Name : in  STRING;
                    Clock_Identifier : out Clock_ID;
                                Mode : in  Clock_Mode;
                                Rate : in  Clock_Rate;
                       Vector_Number : in  VECTOR_NUMBER_TYPE;
                     Service_Routine : in  ADDRESS;
                         CSR_Address : out ADDRESS;
                       Device_Object : out DEVICE_TYPE );

  procedure Re_Initialize (Clock_Identifier : in  Clock_ID;
                                       Mode : in  Clock_Mode;
                                       Rate : in  Clock_Rate );


  procedure Display_CSR          (Clock_Identifier : in  Clock_ID);
  procedure Enable_Interrupts    (Clock_Identifier : in  Clock_ID);
  procedure Disable_Interrupts   (Clock_Identifier : in  Clock_ID);
  procedure Generate_Interrupts  (Clock_Identifier : in  Clock_ID);
  procedure Reset_Interrupt_Flag (Clock_Identifier : in  Clock_ID);
  procedure Reset_Overrun_Flag   (Clock_Identifier : in  Clock_ID);
  procedure Set_Interrupt_Period (Clock_Identifier : in  Clock_ID;
                                            Period : in  KWV_COUNTER_Type );

  procedure Start_Counting    (Clock_Identifier : in  Clock_ID);
  procedure Read_Counter      (Clock_Identifier : in  Clock_ID;
```

```
                              Number_Of_Ticks : out KWV_COUNTER_Type);
  procedure Stop_Counting     (Clock_Identifier : in  Clock_ID;
                              Number_Of_Ticks : out KWV_COUNTER_Type);


  function Interrupts_Enabled (Clock_Identifier : in  Clock_ID) return BOOLEAN;
  function Current_Mode       (Clock_Identifier : in  Clock_ID) return Clock_Mode;
  function Current_Rate       (Clock_Identifier : in  Clock_ID) return Clock_Rate;
  function Interrupt_Period   (Clock_Identifier : in  Clock_ID) return KWV_COUNTER_Type;
  function Interrupt_Flag_On  (Clock_Identifier : in  Clock_ID) return BOOLEAN;
  function Overrun_Flag_On    (Clock_Identifier : in  Clock_ID) return BOOLEAN;

  Invalid_Clock_Mode    : EXCEPTION;
  Initialization_Error  : EXCEPTION;
  Clock_Not_Initialized : EXCEPTION;

  private

    subtype Clock_ID_Range is NATURAL range 0..31;
    type Clock_ID is new Clock_ID_Range;

end KWV11_Clock_Manager;
```

A.d. Real-Time Clock Manager Package Body

```
--                                                                    --
-------------- SEI Ada Embedded Systems Project Prologue ---------------
--                                                                    --
-- Unit name    : KWV11_Clock_Manager package body
-- Experiment # : PA01
-- Version      : 1.0
-- Author       : Mark W. Borger
--              :
-- Date created : 17 Mar 1987
-- Last update  :
--              :
-- Host Machine : VAX/VMS 4.5
-- Target Machine: VAXELN 2.3
--                                                                    --
------------------------------------------------------------------------
--                                                                    --
-- Abstract     :  This package body implements the subprograms of its
----------------:  specification.  It maintains a Clock_ID array containing
----------------:  the corresponding clock's CSR address to allow for the
----------------:  control of multiple clocks.
--                                                                    --
-------------------------- Revision History ----------------------------
--                                                                    --
-- Date      Version   Author                      History
-- 22 Mar 87   1.0     Mark W. Borger    Added data structure to contain
--                                       Mode and Rate for each Clock_ID.
--
--                                                                    --
--------------------------- End of Prologue ----------------------------
--                                                                    --

package body KWV11_Clock_Manager is

  ----------
  -- Local Data types
  ----------
    type Clock_Information_Record is record
      Rate : Clock_Rate;
      Mode : Clock_Mode;
    end record;
    type Clock_Info_Array_Type is array(Clock_ID) of Clock_Information_Record;
    Clock_Info  : Clock_Info_Array_Type := (others => (Stop, Mode_Zero));

    type Clock_Array_Type is array(Clock_ID) of ADDRESS;
    Clock_Array : Clock_Array_Type := (others => ADDRESS_ZERO);

    Current_Clock_Number : Clock_ID := Clock_ID'FIRST;
```

```
--                                                                      --
------------------------------------------------------------------------
--                                                                      --
  procedure Initialize (Clock_Name : in  STRING;
                  Clock_Identifier : out Clock_ID;
                              Mode : in  Clock_Mode;
                              Rate : in  Clock_Rate;
                     Vector_Number : in  VECTOR_NUMBER_TYPE;
                   Service_Routine : in  ADDRESS;
                       CSR_Address : out ADDRESS;
                     Device_Object : out DEVICE_TYPE ) is separate;


  procedure Re_Initialize (Clock_Identifier : in  Clock_ID;
                                       Mode : in  Clock_Mode;
                                       Rate : in  Clock_Rate ) is separate;


  procedure Display_CSR          (Clock_Identifier : in  Clock_ID) is separate;

  procedure Enable_Interrupts    (Clock_Identifier : in  Clock_ID) is separate;

  procedure Disable_Interrupts   (Clock_Identifier : in  Clock_ID) is separate;

  procedure Set_Interrupt_Period (Clock_Identifier : in  Clock_ID;
                                            Period : in  KWV_COUNTER_TYPE ) is separate;

  procedure Generate_Interrupts  (Clock_Identifier : in  Clock_ID) is separate;

  procedure Reset_Interrupt_Flag (Clock_Identifier : in  Clock_ID) is separate;

  procedure Reset_Overrun_Flag   (Clock_Identifier : in  Clock_ID) is separate;


  procedure Start_Counting    (Clock_Identifier : in  Clock_ID) is separate;

  procedure Read_Counter      (Clock_Identifier : in  Clock_ID;
                                 Number_Of_Ticks : out KWV_COUNTER_TYPE) is separate;

  procedure Stop_Counting     (Clock_Identifier : in  Clock_ID;
                                 Number_Of_Ticks : out KWV_COUNTER_TYPE) is separate;


  function Interrupts_Enabled (Clock_Identifier : in  Clock_ID)
    return BOOLEAN is separate;

  function Current_Mode       (Clock_Identifier : in  Clock_ID)
    return Clock_Mode is separate;

  function Current_Rate       (Clock_Identifier : in  Clock_ID)
    return Clock_Rate is separate;

  function Interrupt_Period   (Clock_Identifier : in  Clock_ID)
    return KWV_COUNTER_TYPE is separate;

  function Interrupt_Flag_On  (Clock_Identifier : in  Clock_ID)
    return BOOLEAN is separate;

  function Overrun_Flag_On    (Clock_Identifier : in  Clock_ID)
    return BOOLEAN is separate;

end KWV11_Clock_Manager;
```

Initialize procedure

```
with UNCHECKED_CONVERSION;
with VAXELN_SERVICES;           use VAXELN_SERVICES;
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

procedure Initialize (Clock_Name : in  STRING;
                Clock_Identifier : out Clock_ID;
                            Mode : in  Clock_Mode;
```

```
                         Rate : in  Clock_Rate;
                Vector_Number : in  VECTOR_NUMBER_TYPE;
              Service_Routine : in  ADDRESS;
                  CSR_Address : out ADDRESS;
                Device_Object : out DEVICE_TYPE ) is

    Return_Code       : COND_VALUE_TYPE;
    KWV11_CSR_Address : ADDRESS;
    Current_CSR       : KWV_CSR_Record;
    Timer_Device      : DEVICE_ARRAY_TYPE(0..0) := (others => 0);

    function Convert_It is new UNCHECKED_CONVERSION(Clock_Mode, UNSIGNED_2);
    function Convert_It is new UNCHECKED_CONVERSION(Clock_Rate, UNSIGNED_3);

  begin
    ----------
    --  Create the KWV11-C device object and associate with its interrupts the
    --  Interrupt Service Routine.
    ----------
      Create_Device (Status          => Return_Code,
                     Device_Name     => Clock_Name,
                     Vector_Number   => Vector_Number,
                     Service_Routine => Service_Routine,
                     Registers       => KWV11_CSR_Address,
                     Device_Array    => Timer_Device,
                     Device_Count    => 1);

     if CONDITION_HANDLING.Success(Return_Code) then
        Device_Object       := Timer_Device(0);
        Clock_Identifier    := Current_Clock_Number;
        CSR_Address         := KWV11_CSR_Address;
        Clock_Array(Current_Clock_Number) := KWV11_CSR_Address;
        Clock_Info(Current_Clock_Number)  := Clock_Information_Record'(Rate, Mode);
        Current_Clock_Number := Current_Clock_Number + Clock_ID(1);

      ----------
      -- Initialize clock via CSR settings
      ----------
          Current_CSR := KWV_CSR_Record'(
                         go       => FALSE,
                         mode     => Convert_It(Mode),
                         rate     => Convert_It(Rate),
                         others   => FALSE );
        Put_CSR(Current_CSR, KWV11_CSR_Address);
      else
        raise Initialization_Error;
      end if;

  end Initialize;
```

Re_Initialize procedure

```
  with UNCHECKED_CONVERSION;
  with VAXELN_SERVICES;           use VAXELN_SERVICES;
  with KWV_Register_Definitions;  use KWV_Register_Definitions;

  separate (KWV11_Clock_Manager)

  procedure Re_Initialize (Clock_Identifier : in  Clock_ID;
                                       Mode : in  Clock_Mode;
                                       Rate : in  Clock_Rate )  is

    Current_CSR  : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

    function Convert_It is new UNCHECKED_CONVERSION(Clock_Mode, UNSIGNED_2);
    function Convert_It is new UNCHECKED_CONVERSION(Clock_Rate, UNSIGNED_3);

  begin
    ----------
    --  If specified clock's CSR address is non-zero (i.e., the clock exists
    --  and has been initialized) then re-initialize it by clearing the CSR
    --  settings; otherwise raise an exception since the specified clock has
```

```
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then

      Current_CSR := KWV_CSR_Record'(go => FALSE,
                                     mode => Convert_It(Mode),
                                     rate => Convert_It(Rate),
                                   others => FALSE );
      Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));
      Clock_Info(Clock_Identifier) := Clock_Information_Record'(Rate,Mode);
    else
      raise Clock_Not_Initialized;
    end if;

end Re_Initialize;
```

Display_CSR procedure

```
with TEXT_IO;                   use TEXT_IO;
with KWV_Register_Definitions;  use KWV_Register_Definitions;
with UNCHECKED_CONVERSION;

separate (KWV11_Clock_Manager)

procedure Display_CSR (Clock_Identifier : in  Clock_ID) is
  Current_CSR : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

  package Rate_IO    is new ENUMERATION_IO(Clock_Rate);
  package Mode_IO    is new ENUMERATION_IO(Clock_Mode);
  package BOOLEAN_IO is new ENUMERATION_IO(BOOLEAN);

  function Convert_It is new UNCHECKED_CONVERSION(UNSIGNED_2, Clock_Mode);
  function Convert_It is new UNCHECKED_CONVERSION(UNSIGNED_3, Clock_Rate);

  procedure Formatted_String_Put(Str : in STRING) is
  begin
     Put(Str);
     Set_Col(20);
     Put(" => ");
  end Formatted_String_Put;


begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then display contents of CSR;
  --  otherwise raise an exception since the specified clock has
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
     Formatted_String_Put("CSR.go");
     BOOLEAN_IO.Put(Current_CSR.go);  New_Line;

     Formatted_String_Put("CSR.mode");
     Mode_IO.Put(Convert_It(Current_CSR.mode));  New_Line;

     Formatted_String_Put("CSR.rate");
     Rate_IO.Put(Convert_It(Current_CSR.rate));  New_Line;

     Formatted_String_Put("CSR.int_ovf");
     BOOLEAN_IO.Put(Current_CSR.int_ovf);  New_Line;

     Formatted_String_Put("CSR.ovf_flag");
     BOOLEAN_IO.Put(Current_CSR.ovf_flag);  New_Line;

     Formatted_String_Put("CSR.maint_st1");
     BOOLEAN_IO.Put(Current_CSR.maint_st1);  New_Line;

     Formatted_String_Put("CSR.maint_st2");
     BOOLEAN_IO.Put(Current_CSR.maint_st2);  New_Line;

     Formatted_String_Put("CSR.maint_osc");
```

```
      BOOLEAN_IO.Put(Current_CSR.maint_osc);  New_Line;

      Formatted_String_Put("CSR.dio");
      BOOLEAN_IO.Put(Current_CSR.dio);  New_Line;

      Formatted_String_Put("CSR.flag_overrun");
      BOOLEAN_IO.Put(Current_CSR.flag_overrun);  New_Line;

      Formatted_String_Put("CSR.st2_go_enable");
      BOOLEAN_IO.Put(Current_CSR.st2_go_enable);  New_Line;

      Formatted_String_Put("CSR.st2_int_enable");
      BOOLEAN_IO.Put(Current_CSR.st2_int_enable);  New_Line;

      Formatted_String_Put("CSR.st2_flag");
      BOOLEAN_IO.Put(Current_CSR.st2_flag);  New_Line;

     else
        raise Clock_Not_Initialized;
    end if;

  end Display_CSR;
```

Enable_Interrupts procedure

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

procedure Enable_Interrupts (Clock_Identifier : in  Clock_ID) is

  Current_CSR  : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then enable interrupts on counter overflow;
  --  otherwise raise an exception since the specified clock has
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      Current_CSR.int_ovf := TRUE;
      Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));
    else
      raise Clock_Not_Initialized;
    end if;

end Enable_Interrupts;
```

Disable_Interrupts procedure

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

procedure Disable_Interrupts (Clock_Identifier : in  Clock_ID) is

  Current_CSR  : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then disable interrupts on counter overflow;
  --  otherwise raise an exception since the specified clock has
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      Current_CSR.int_ovf := FALSE;
      Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));
```

```
    else
      raise Clock_Not_Initialized;
    end if;

end Disable_Interrupts;
```

Set_Interrupt_Period procedure

```
with UNCHECKED_CONVERSION;
with VAXELN_SERVICES;           use VAXELN_SERVICES;
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

procedure Set_Interrupt_Period (Clock_Identifier : in  Clock_ID;
                                       Period : in  KWV_COUNTER_TYPE) is
  Device_Ticks   : KWV_COUNTER_TYPE;
    for Device_Ticks use at (Clock_Array(Clock_Identifier) + 2);
begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then set the current value of the clock
  --  interrupt period using two's complement notation; otherwise raise
  --  an exception since the specified clock has not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
     WRITE_REGISTER((16#FFFF# - Period + 1), Device_Ticks);
   else
      raise Clock_Not_Initialized;
   end if;

end Set_Interrupt_Period;
```

Generate_Interrupts procedure

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

procedure Generate_Interrupts (Clock_Identifier : in  Clock_ID) is
  Current_CSR  : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then start internal counter that causes
  --  interrupts; otherwise raise an exception since the specified clock has
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      Current_CSR.go := TRUE;
      Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));
    else
      raise Clock_Not_Initialized;
    end if;

end Generate_Interrupts;
```

Reset_Interrupt_Flag procedure

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)
```

```
procedure Reset_Interrupt_Flag (Clock_Identifier : in  Clock_ID) is
  Current_CSR : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then clear counter overflow flag to allow
  --  another interrupt to be generated; otherwise raise an exception since
  --  the specified clock has not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      Current_CSR.ovf_flag := FALSE;
      Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));
    else
      raise Clock_Not_Initialized;
    end if;

end Reset_Interrupt_Flag;
```

Reset_Overrun_Flag procedure

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

procedure Reset_Overrun_Flag (Clock_Identifier : in  Clock_ID) is
  Current_CSR : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then clear interrupt overrun flag;
  --  otherwise raise an exception since the specified clock has
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      Current_CSR.flag_overrun:= FALSE;
      Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));
    else
      raise Clock_Not_Initialized;
    end if;

end Reset_Overrun_Flag;
```

Start_Counting procedure

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

procedure Start_Counting (Clock_Identifier : in  Clock_ID) is
  Current_CSR : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then start the clock's internal counter;
  --  otherwise raise an exception since the specified clock has
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
     if (Clock_Info(Clock_Identifier).Mode = Mode_Two or else
         Clock_Info(Clock_Identifier).Mode = Mode_Three)
     then
       Current_CSR.go := TRUE;
       Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));
     else
       raise Invalid_Clock_Mode;
```

```
      end if;
    else
       raise Clock_Not_Initialized;
    end if;

end Start_Counting;
```

Read_Counter procedure

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

procedure Read_Counter (Clock_Identifier : in  Clock_ID;
                         Number_Of_Ticks : out KWV_COUNTER_TYPE) is

  Current_CSR  : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

  Device_Ticks : KWV_COUNTER_TYPE;
    for Device_Ticks use at (Clock_Array(Clock_Identifier) + 2);

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then simulate an external event to
  --  get current value of the clock's internal counter written to the
  --  BUFFER/PRESET register and then read that value and return it while
  --  the clock continues to run; otherwise raise an exception since the
  --  specified clock has not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
     if (Clock_Info(Clock_Identifier).Mode = Mode_Two or else
         Clock_Info(Clock_Identifier).Mode = Mode_Three)
     then
       Current_CSR.st2_int_enable := FALSE;
       Current_CSR.maint_st2 := TRUE;
       Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));

       loop
         Current_CSR := Get_CSR(Clock_Array(Clock_Identifier));
         exit when Current_CSR.st2_flag;
       end loop;

       Number_Of_Ticks := READ_REGISTER(Device_Ticks);
       Current_CSR.st2_flag := FALSE;
       Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));
     else
       raise Invalid_Clock_Mode;
     end if;
   else
     raise Clock_Not_Initialized;
   end if;

end Read_Counter;
```

Stop_Counting procedure

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

procedure Stop_Counting (Clock_Identifier : in  Clock_ID;
                          Number_Of_Ticks : out KWV_COUNTER_TYPE) is

  Current_CSR   : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

  Device_Ticks  : KWV_COUNTER_TYPE;
    for Device_Ticks use at (Clock_Array(Clock_Identifier) + 2);
```

```
begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then simulate an external event to
  --  get current value of the clock's internal counter written to the
  --  BUFFER/PRESET register and then return that value;
  --  otherwise raise an exception since the specified clock has
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
     if (Clock_Info(Clock_Identifier).Mode = Mode_Two or else
         Clock_Info(Clock_Identifier).Mode = Mode_Three)
     then
       Current_CSR.st2_int_enable := FALSE;
       Current_CSR.maint_st2 := TRUE;
       Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));

       loop
         Current_CSR := Get_CSR(Clock_Array(Clock_Identifier));
         exit when Current_CSR.st2_flag;
       end loop;

       Number_Of_Ticks := READ_REGISTER(Device_Ticks);
       Current_CSR.go := FALSE;
       Current_CSR.st2_flag := FALSE;
       Put_CSR(Current_CSR, Clock_Array(Clock_Identifier));
     else
       raise Invalid_Clock_Mode;
     end if;
   else
     raise Clock_Not_Initialized;
   end if;

end Stop_Counting;
```

Interrupts_Enabled function

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

function Interrupts_Enabled (Clock_Identifier : in  Clock_ID) return BOOLEAN is
  Current_CSR  : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then return a BOOLEAN value indicating
  --  whether or not the clock will generate an interrupt when its internal
  --  clock overflows; overflow flag; otherwise raise an exception since
  --  the specified clock has not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      return Current_CSR.int_ovf;
    else
      raise Clock_Not_Initialized;
    end if;

end Interrupts_Enabled;
```

Current_Mode function

```
with UNCHECKED_CONVERSION;
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

function Current_Mode (Clock_Identifier : in  Clock_ID) return Clock_Mode is
  Current_CSR  : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));
```

```
  function Convert_It is new UNCHECKED_CONVERSION(UNSIGNED_2, Clock_Mode);
begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then return current clock mode;
  --  otherwise raise an exception since the specified clock has
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      return Convert_It(Current_CSR.mode);
    else
      raise Clock_Not_Initialized;
    end if;

end Current_Mode;
```

Current_Rate function

```
with UNCHECKED_CONVERSION;
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

function Current_Rate (Clock_Identifier : in  Clock_ID) return Clock_Rate is
  Current_CSR  : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

  function Convert_It is new UNCHECKED_CONVERSION(UNSIGNED_3, Clock_Rate);
begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then return current clock rate;
  --  otherwise raise an exception since the specified clock has
  --  not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      return Convert_It(Current_CSR.rate);
    else
      raise Clock_Not_Initialized;
    end if;

end Current_Rate;
```

Interrupt_Period function

```
with UNCHECKED_CONVERSION;
with VAXELN_SERVICES;           use VAXELN_SERVICES;
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

function Interrupt_Period (Clock_Identifier : in Clock_ID) return KWV_COUNTER_TYPE is
  Device_Ticks  : KWV_COUNTER_TYPE;
    for Device_Ticks use at (Clock_Array(Clock_Identifier) + 2);
begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then return current value of the clock
  --  interrupt period; otherwise raise an exception since the specified
  --  clock has not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      return READ_REGISTER(Device_Ticks);
   else
      raise Clock_Not_Initialized;
   end if;

end Interrupt_Period;
```

Interrupt_Flag_On function

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

function Interrupt_Flag_On (Clock_Identifier : in  Clock_ID) return BOOLEAN is
  Current_CSR  : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then return current BOOLEAN value of counter
  --  overflow flag; otherwise raise an exception since the specified clock
  --  has not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      return Current_CSR.ovf_flag;
    else
      raise Clock_Not_Initialized;
    end if;

end Interrupt_Flag_On;
```

Overrun_Flag_On function

```
with KWV_Register_Definitions;  use KWV_Register_Definitions;

separate (KWV11_Clock_Manager)

function Overrun_Flag_On (Clock_Identifier : in  Clock_ID) return BOOLEAN is
  Current_CSR : KWV_CSR_Record := Get_CSR(Clock_Array(Clock_Identifier));

begin
  ----------
  --  If specified clock's CSR address is non-zero (i.e., the clock exists
  --  and has been initialized) then return current BOOLEAN value of overrun
  --  flag; otherwise raise an exception since the specified clock
  --  has not been initialized properly.
  ----------
   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
      return Current_CSR.flag_overrun;
    else
      raise Clock_Not_Initialized;
    end if;

end Overrun_Flag_On;
```

A.e. INS Data Types Package Specification

```
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
--| MODULE NAME:    INS_Data_Types
--|
--| MODULE TYPE:    Package Specification
--|
--| MODULE PURPOSE:
--|     Export Executive global constants and types.
--|
--|--------------------------------------------------------------------------
--| MODULE DESCRIPTION:
--|     This package defines the constants and global data types
--|     used throughout the executive subsystem.
--|
--|--------------------------------------------------------------------------
```

```
--| REVISION HISTORY:   -- see end of listing
--|---------------------------------------------------------------------------
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pragma PAGE;

package INS_Data_Types is

  Maximum_Priority      : constant NATURAL := 15;
  Maximum_Tick_Value    : constant NATURAL := 34_560_000;
  Maximum_Period_Value  : constant NATURAL := 511;
  Microseconds_Per_Tick : constant NATURAL := 2_560;

  subtype Tick_Range     is NATURAL range 0..Maximum_Tick_Value;
  subtype Period_Range   is NATURAL range 0..Maximum_Period_Value;
  subtype Priority_Range is NATURAL range 0..Maximum_Priority;

end INS_Data_Types;

--~~~~~~~~~~~~~~~~~~~~~~~~  REVISION HISTORY  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
```

A.f. Clock Interrupt Service Routine

```
with VAXELN_SERVICES;
with CONDITION_HANDLING;
with INS_Data_Types;

with SYSTEM;  use SYSTEM;

  procedure Timer_Interrupt_Routine
    (Device_Registers : in ADDRESS;
     Comm_Region      : in out INS_Data_Types.Executive_Communication_Region;
     ISR_Context      : in VAXELN_SERVICES.ISR_CONTEXT_TYPE ) is

    Return_Code : CONDITION_HANDLING.COND_VALUE_TYPE;
    Temp_Int    : INTEGER := 0;

  begin

    for Index2 in 1..110
    loop
      Temp_Int := Temp_Int + Index2;
    end loop;

    Comm_Region.Current_Tick_Number := Comm_Region.Current_Tick_Number + 1;

    if  Comm_Region.Current_Tick_Number >= Comm_Region.Next_Schedule_Time then

      VAXELN_SERVICES.SIGNAL_DEVICE(Status        => Return_Code,
                                    Device_Number => 0,
                                    ISR_Context   => ISR_Context );
    end if;

  end Timer_Interrupt_Routine;

  pragma SUPPRESS_ALL;
  pragma EXPORT_PROCEDURE(Timer_Interrupt_Routine);
```

A.g. Runtime BIT Package Specification

```
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
--| MODULE NAME:    Runtime_BIT
--|
--| MODULE TYPE:    Package Specification
--|
--| MODULE PURPOSE:
--|     This package implements the Runtime Built-In Tests
--|     for the AEST INS simulator program.
--|
```

```
--|-------------------------------------------------------------------------
--| MODULE DESCRIPTION:
--|     This package implements the Runtime Built-In Tests
--|     for the AEST INS simulator program.
--|
--|-------------------------------------------------------------------------
--| REVISION HISTORY:   -- see end of listing
--|-------------------------------------------------------------------------
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pragma PAGE;

package Runtime_BIT is

  task Runtime_BIT_Processor is
    entry Activate;    -- called every 1000 msec by the dispatcher
    pragma PRIORITY(2);
  end Runtime_BIT_Processor;

  procedure Runtime_Tests;  -- implements the tests

end Runtime_BIT;


--~~~~~~~~~~~~~~~~~~~~~~~~  REVISION HISTORY  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
```

A.h. Runtime BIT Package Body

```
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
--| MODULE NAME:    Runtime_BIT
--|
--| MODULE TYPE:    Package Body
--|
--| MODULE PURPOSE:
--|     This package implements the Runtime Built-In Tests
--|     for the AEST INS simulator program.
--|
--|-------------------------------------------------------------------------
--| MODULE DESCRIPTION:
--|     This package implements the Runtime Built-In Tests
--|     for the AEST INS simulator program.
--|
--|-------------------------------------------------------------------------
--| REVISION HISTORY:   -- see end of listing
--|-------------------------------------------------------------------------
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pragma PAGE;

with Load_Control;

package body Runtime_BIT is

  task body Runtime_BIT_Processor is
  begin
    loop
      accept Activate;    -- called every 1000 msec by the dispatcher
      Load_Control.Busy_Wait(50);
    end loop;
  end Runtime_BIT_Processor;

  procedure Runtime_Tests is
  begin
    null;  -- implements the tests
  end Runtime_Tests;

end Runtime_BIT;


--~~~~~~~~~~~~~~~~~~~~~~~~  REVISION HISTORY  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
```

A.i. Motion Simulator Package Specification

```
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
--| MODULE NAME:    Motion_Simulator
--|
--| MODULE TYPE:    Package Specification
--|
--| MODULE PURPOSE:
--|     This package implements the various motion simulation
--|     calculations that are the core of the AEST INS
--|     simulator program.
--|
--|---------------------------------------------------------------------------
--| MODULE DESCRIPTION:
--|     This package implements the various motion simulation
--|     calculations that are the core of the AEST INS
--|     simulator program.
--|
--|---------------------------------------------------------------------------
--| REVISION HISTORY:   -- see end of listing
--|---------------------------------------------------------------------------
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pragma PAGE;


package Motion_Simulator is

  procedure Update_Attitude_and_Heading; -- called by the clock ISR every 2.56 ms


  task Ship_Velocity_Updater is
    entry Activate;  -- called by the dispatcher every 40.96 msec
    pragma PRIORITY(8);
  end Ship_Velocity_Updater;


  task Ship_Position_Updater is
    entry Activate; -- called by the dispatcher every 1300.0 msec
    pragma PRIORITY(1);
  end Ship_Position_Updater;

end Motion_Simulator;

--~~~~~~~~~~~~~~~~~~~~~~~~  REVISION HISTORY  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
```

A.j. Motion Simulator Package Body

```
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
--| MODULE NAME:    Motion_Simulator
--|
--| MODULE TYPE:    Package Body
--|
--| MODULE PURPOSE:
--|     This package implements the various motion simulation
--|     calculations that are the core of the AEST INS
--|     simulator program.
--|
--|---------------------------------------------------------------------------
--| MODULE DESCRIPTION:
--|     This package implements the various motion simulation
--|     calculations that are the core of the AEST INS
--|     simulator program.
--|
--|---------------------------------------------------------------------------
--| REVISION HISTORY:   -- see end of listing
--|---------------------------------------------------------------------------
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pragma PAGE;
```

```
with Load_Control;
with Task_Manager;

package body Motion_Simulator is

  procedure Update_Attitude_and_Heading is
  begin
    null;
  end Update_Attitude_and_Heading;


  task body Ship_Velocity_Updater is
  begin
    loop
      accept Activate;
      Load_Control.Busy_Wait(40);  -- 4 milliseconds
    end loop;
  end Ship_Velocity_Updater;


  task body Ship_Position_Updater is
  begin
    loop
      accept Activate;
      Load_Control.Busy_Wait(250);  -- 25 milliseconds
    end loop;
  end Ship_Position_Updater;

end Motion_Simulator;

--~~~~~~~~~~~~~~~~~~~~~~~~~  REVISION HISTORY  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
```

A.k. Comms Handler Package Specification

```
package Comms_Handler is

  procedure Time_Out;

  task Attitude_Periodic_Message_Sender is
    entry Activate;
    pragma PRIORITY(7);
  end Attitude_Periodic_Message_Sender;

  task Navigation_Periodic_Message_Sender is
    entry Activate;
    pragma PRIORITY(4);
  end Navigation_Periodic_Message_Sender;

end Comms_Handler;
```

A.l. Comms Handler Package Body

```
with Load_Control;
with Task_Manager;  use Task_Manager;

package body Comms_Handler is

  procedure Time_Out is
  begin
    null;
  end Time_Out;

  task body Attitude_Periodic_Message_Sender is
  begin
    loop
       accept Activate;
       Load_Control.Busy_Wait(100);  -- 10 milliseconds
    end loop;
  end Attitude_Periodic_Message_Sender;
```

```
  task body Navigation_Periodic_Message_Sender is
  begin
    loop
       accept Activate;
       Load_Control.Busy_Wait(200);  -- 20 milliseconds
    end loop;
  end Navigation_Periodic_Message_Sender;

end Comms_Handler;
```

A.m. Screen Area Handler Package Specification

```
package Screen_Area_Handler  is

  task  Periodic_Status_Display_Processor is
    entry  Activate;  -- called every 1000 msec by the dispatcher

    pragma PRIORITY(3);
  end Periodic_Status_Display_Processor;

end Screen_Area_Handler;
```

A.n. Screen Area Handler Package Specification

```
with Load_Control;
package body Screen_Area_Handler  is

  task body Periodic_Status_Display_Processor is
  begin
    loop
      accept Activate;  -- called every 1000 msec by the dispatcher
      Load_Control.Busy_Wait(1_000);  -- 100 milliseconds
    end loop;
  end Periodic_Status_Display_Processor;


end Screen_Area_Handler;
```

A.o. Activation Queue Manager Package Specification

```
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
--| MODULE NAME:    Activation_Queue_Manager (AQM)
--|
--| MODULE TYPE:    Package Specification
--|
--| MODULE PURPOSE:
--|     Implement task activation queue manager.
--|
--|---------------------------------------------------------------------------
--| MODULE DESCRIPTION:
--|     This package provides the necessary data types,
--|     procedures, and exceptions for implementing a time
--|     ordered activation queue.  The package only supports
--|     one such queue whose implementation details are hidden
--|     within the package body.
--|
--|---------------------------------------------------------------------------
--| REVISION HISTORY:   -- see end of listing
--|---------------------------------------------------------------------------
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pragma PAGE;

with Task_Manager;
with INS_Data_Types;

package Activation_Queue_Manager is
```

```
  subtype Priority_Range          is INS_Data_Types.Priority_Range;
  subtype Activation_Time_Range   is INS_Data_Types.Tick_Range;
  subtype Activation_Period_Range is INS_Data_Types.Period_Range;

  subtype Task_ID_Type is Task_Manager.Task_ID_Type;

  type Activation_Mode_Type is (Single_Shot, Periodic, Time_Out, No_Op);

  type Task_Activation_Record is record
    Task_ID                : Task_ID_Type;
    Activation_Period      : Activation_Period_Range;
    Activation_Time        : Activation_Time_Range;
    Activation_Priority    : Priority_Range;
    Execution_Priority     : Priority_Range;
    Activation_Mode        : Activation_Mode_Type;
  end record;

  procedure Insert_Activation_Record (Record_ID : in  Task_Activation_Record;
                           Next_Schedule_Time : out Activation_Time_Range);

  procedure Get_Activation_Record    (Record_ID : out Task_Activation_Record;
                           Next_Schedule_Time : out Activation_Time_Range);

  procedure Delete_Activation_Record   (Task_ID : in  Task_ID_Type);


end Activation_Queue_Manager;

--~~~~~~~~~~~~~~~~~~~~~~~~~  REVISION HISTORY  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
```

A.p. Activation Queue Manager Package Body

```
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
--| MODULE NAME:    Activation_Queue_Manager
--|
--| MODULE TYPE:    Package Body
--|
--| MODULE PURPOSE:
--|     Implement task activation queue manager.
--|
--|---------------------------------------------------------------------------
--| MODULE DESCRIPTION:
--|     This package supports the implementation of a time
--|     ordered task activation queue and its associated
--|     interfaces exported in the package specification.
--|     The activation queue is maintained as a static array of
--|     activation records (ARs) as defined in the package specification.
--|     The ARs are never moved from their initial position in the array and
--|     one special array element is reserved for the AR of the
--|     Communications Controller task, which is called when a time-out has
--|     expired.  The AQM maintains information regarding the next task to be
--|     scheduled and when to schedule it by performing a linear search of
--|     the array upon each insert and fetch operation.  When an AR is
--|     returned (i.e., taken off the queue) to the Dispatcher, its activation
--|     mode value is checked by the AQM; if it represents a periodic task, a
--|     new activation time is computed, and the AR gets re-inserted into the
--|     queue.
--|
--|---------------------------------------------------------------------------
--| REVISION HISTORY:   -- see end of listing
--|---------------------------------------------------------------------------
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pragma PAGE;

with Task_Manager;  use Task_Manager;

package body Activation_Queue_Manager is

  Next_Activation_Time  : Activation_Time_Range := Activation_Time_Range'LAST;
  Next_Task_To_Schedule : Task_ID_Type;
  Activation_Records    : array(Task_ID_Type) of Task_Activation_Record;
```

```
  ----------
  --  Insert the specified ARs information into the AR Table
  ----------
  procedure Insert_Activation_Record (Record_ID : in  Task_Activation_Record;
                             Next_Schedule_Time : out Activation_Time_Range) is
  begin
    Activation_Records(Record_ID.Task_ID) := Record_ID;
      if Record_ID.Activation_Time <  Next_Activation_Time and then
         Record_ID.Activation_Mode /= No_Op then
           Next_Activation_Time  := Record_ID.Activation_Time;
           Next_Task_To_Schedule := Record_ID.Task_ID;
      end if;
      Next_Schedule_Time := Next_Activation_Time;
  end Insert_Activation_Record;

  ----------
  --  Get next AR from the Activation Queue.  Re-schedule any tasks with
  --  same activation time as the one taken off the queue.
  ----------
  procedure Get_Activation_Record    (Record_ID : out Task_Activation_Record;
                             Next_Schedule_Time : out Activation_Time_Range) is
  begin
    Record_ID := Activation_Records(Next_Task_To_Schedule);

    ----------
    --  If current task is periodic, then recompute next activation for
    --  task and then re-insert it into the activation queue.
    ----------
    if Activation_Records(Next_Task_To_Schedule).Activation_Mode = Periodic then
      Activation_Records(Next_Task_To_Schedule).Activation_Time :=
        Activation_Records(Next_Task_To_Schedule).Activation_Time +
        Activation_Time_Range(Activation_Records(Next_Task_To_Schedule).Activation_Period);
    end if;

    ----------
    --  Find next task to be scheduled.
    ----------
    Next_Activation_Time  := Activation_Records(Task_ID_Type’FIRST).Activation_Time;
    Next_Task_To_Schedule := Task_ID_Type’FIRST;

    for Index in Task_ID_Type’SUCC(Task_ID_Type’FIRST)..Task_ID_Type’LAST
    loop
      if Activation_Records(Index).Activation_Time < Next_Activation_Time and then
         Activation_Records(Index).Activation_Mode /= No_Op then
        Next_Activation_Time := Activation_Records(Index).Activation_Time;
        Next_Task_To_Schedule := Index;
      end if;
    end loop;

    Next_Schedule_Time := Next_Activation_Time;

  end Get_Activation_Record;

  ----------
  --  Mark AR associated with Task_ID as not available for scheduling.
  --  Its slot will most likely be used at a later date (e.g., timeouts).
  ----------
  procedure Delete_Activation_Record   (Task_ID : in  Task_ID_Type) is
  begin
    Activation_Records(Task_ID).Activation_Mode := No_Op;
  end Delete_Activation_Record;

end Activation_Queue_Manager;

--~~~~~~~~~~~~~~~~~~~~~~~~~  REVISION HISTORY  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
```

A.q. Task Manager Package Specification

```
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
--| MODULE NAME:    Task_Manager
--|
--| MODULE TYPE:    Package Specification
--|
--| MODULE PURPOSE:
--|     This package provides an interface to initialize the task activation
--|     queue and start the dispatcher of the AEST INS simulator program.
--|---------------------------------------------------------------------------
--| MODULE DESCRIPTION:
--|     This package provides the necessary procedures
--|     to initialize the task activation queue, start the task dispatcher,
--|     enable/disable periodic tasks, and support time-outs for base
--|     level tasks.
--|
--|---------------------------------------------------------------------------
--| REVISION HISTORY:   -- see end of listing
--|---------------------------------------------------------------------------
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pragma PAGE;

with INS_Data_Types;

package Task_Manager is

  ----------
  --  Imported data types
  ----------
    subtype Activation_Time_Range   is INS_Data_Types.Tick_Range;
    subtype Activation_Period_Range is INS_Data_Types.Period_Range;

  type Task_ID_Type is
    (   Ship_Velocity_Updater,
        Attitude_Periodic_Message_Sender,
        Navigation_Periodic_Message_Sender,
        Periodic_Status_Display_Processor,
        Runtime_BIT_Processor,
        Ship_Position_Updater,
        Comms_Controller
    );

  subtype Periodic_Task_ID_Type is Task_ID_Type range
    Ship_Velocity_Updater..Ship_Position_Updater;

  subtype Timeout_Task_ID_Type is Task_ID_Type range
    Comms_Controller..Comms_Controller;


  procedure Initialize_Activation_Queue;

  procedure Activate_Dispatcher;

  function Task_Is_Enabled (Task_ID : in Periodic_Task_ID_Type) return BOOLEAN;

  procedure Enable_Task    (Task_ID : in Periodic_Task_ID_Type);

  procedure Disable_Task   (Task_ID : in Periodic_Task_ID_Type);

  procedure Request_Time_Out (Task_ID : in Timeout_Task_ID_Type;
                          Time_Period : in Activation_Period_Range);

  procedure Cancel_Time_Out  (Task_ID : Timeout_Task_ID_Type);


  Dispatcher_Activation_Error : EXCEPTION;

end Task_Manager;

--~~~~~~~~~~~~~~~~~~~~~~~~~  REVISION HISTORY  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
```

A.r. Task Manager Package Body

```
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
--| MODULE NAME:    Task_Manager
--|
--| MODULE TYPE:    Package Body
--|
--| MODULE PURPOSE:
--|     Implement a periodic task dispatcher.
--|
--|-------------------------------------------------------------------------
--| MODULE DESCRIPTION:
--|     This package body implements a task dispatcher
--|     that gets and re-inserts task activation records
--|     from and onto the activation queue.  The dispatcher
--|     waits for signals from a real-time clock that is
--|     generating interrupts every 2.56 milliseconds.
--|
--|-------------------------------------------------------------------------
--| REVISION HISTORY:   -- see end of listing
--|-------------------------------------------------------------------------
--~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
pragma PAGE;

with Runtime_BIT;
with Comms_Handler;
with Motion_Simulator;
with KWV11_Clock_Manager;
with Screen_Area_Handler;
with Activation_Queue_Manager;

with SYSTEM;  use SYSTEM;

package body Task_Manager is

  package RTB renames Runtime_BIT;
  package COM renames Comms_Handler;
  package MOS renames Motion_Simulator;
  package SAH renames Screen_Area_Handler;
  package AQM renames Activation_Queue_Manager;

  ----------
  --  Imported Data Types
  ----------
    subtype Clock_ID         is KWV11_Clock_Manager.Clock_ID;
    subtype DEVICE_TYPE      is KWV11_Clock_Manager.DEVICE_TYPE;
    subtype KWV_COUNTER_TYPE is KWV11_Clock_Manager.KWV_COUNTER_TYPE;


  type Task_State_Type is (Disabled, Enabled);

  Periodic_Task_State : array (Periodic_Task_ID_Type) of Task_State_Type :=
    (   Ship_Velocity_Updater               => Enabled,
--      Attitude_Periodic_Message_Sender    => Disabled,
--      Navigation_Periodic_Message_Sender  => Disabled,
        Attitude_Periodic_Message_Sender    => Enabled,
        Navigation_Periodic_Message_Sender  => Enabled,
        Periodic_Status_Display_Processor   => Enabled,
        Runtime_BIT_Processor               => Enabled,
        Ship_Position_Updater               => Enabled );

  Clock_IPL               : UNSIGNED_LONGWORD;
  Comm_Region_Address     : ADDRESS;
  Schedule_At_Tick_Number : Activation_Time_Range :=
                              Activation_Time_Range'LAST;


  ----------
  --  Local Subprograms and tasks
  ----------
    procedure Update_Next_Schedule_Time is separate;

    function Current_Tick_Number return Activation_Time_Range is separate;
```

```
  procedure Activate_Task (Task_ID : in Task_ID_Type;
                   Missed_Deadline : out BOOLEAN);

  procedure Time_Out_Task (Task_ID : in Timeout_Task_ID_Type);

  task Dispatcher is
    entry Activate (Clock_Identifier : in Clock_ID;
                    Clock_Device_ID : in DEVICE_TYPE);
    pragma PRIORITY(9);
  end Dispatcher;

  task body Dispatcher is separate;


----------
--  Exported Subprograms
----------
  procedure Initialize_Activation_Queue is separate;

  procedure Activate_Dispatcher is separate;


----------
--  Is the specified task enabled?
----------
  function Task_Is_Enabled (Task_ID : in Periodic_Task_ID_Type)
    return BOOLEAN is
  begin
    return Periodic_Task_State(Task_ID) = Enabled;
  end Task_Is_Enabled;


----------
--  Enable the specified task.
----------
  procedure Enable_Task (Task_ID : in Periodic_Task_ID_Type) is
  begin
    Periodic_Task_State(Task_ID) := Enabled;
  end Enable_Task;


----------
--  Disable the specified task.
----------
  procedure Disable_Task (Task_ID : in Periodic_Task_ID_Type) is
  begin
    Periodic_Task_State(Task_ID) := Disabled;
  end Disable_Task;


----------
--  Activate the specified task.
----------
procedure Activate_Task (Task_ID : in Task_ID_Type;
                 Missed_Deadline : out BOOLEAN) is
  begin
    Missed_Deadline := FALSE;
    if Task_Is_Enabled(Task_ID) then

      case Task_ID is
        when Ship_Velocity_Updater =>
          select
            MOS.Ship_Velocity_Updater.Activate;
          else
            Missed_Deadline := TRUE;
          end select;

        when Attitude_Periodic_Message_Sender =>
          select
            COM.Attitude_Periodic_Message_Sender.Activate;
          else
            Missed_Deadline := TRUE;
          end select;

        when Navigation_Periodic_Message_Sender =>
          select
```

```
            COM.Navigation_Periodic_Message_Sender.Activate;
          else
            Missed_Deadline := TRUE;
          end select;

        when Periodic_Status_Display_Processor =>
          select
            SAH.Periodic_Status_Display_Processor.Activate;
          else
            Missed_Deadline := TRUE;
          end select;

        when Runtime_BIT_Processor=>
          select
            RTB.Runtime_BIT_Processor.Activate;
          else
            Missed_Deadline := TRUE;
          end select;

        when Ship_Position_Updater =>
          select
            MOS.Ship_Position_Updater.Activate;
          else
            Missed_Deadline := TRUE;
          end select;

        when others =>
          null;

      end case;

    else
      null;
    end if;

  end Activate_Task;


----------
--  Time Out the specified task.
----------
  procedure Time_Out_Task (Task_ID : in Timeout_Task_ID_Type) is
  begin
    COM.Time_Out;
  end Time_Out_Task;



procedure Request_Time_Out (Task_ID : in Timeout_Task_ID_Type;
                        Time_Period : in Activation_Period_Range) is
  Next_Time : Activation_Time_Range := NATURAL'FIRST;
begin

  AQM.Insert_Activation_Record(
      (Task_ID, Activation_Time_Range(Time_Period),
       Activation_Time_Range(Time_Period) + Current_Tick_Number,
       10, 10, AQM.Time_Out), Schedule_At_Tick_Number);

end Request_Time_Out;


procedure Cancel_Time_Out  (Task_ID : Timeout_Task_ID_Type) is
begin
  AQM.Delete_Activation_Record(Task_ID);
end Cancel_Time_Out;

end Task_Manager;

--~~~~~~~~~~~~~~~~~~~~~~~~~  REVISION HISTORY  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
--|
```

Load Control Package Specification

```
with KWV11_Clock_Manager;
```

```
package Load_Control is

  subtype Clock_ID is KWV11_Clock_Manager.Clock_ID;

  procedure Initialize   (Clock_Identifier : in Clock_ID);

  procedure Read_Load_Factor;

  procedure Busy_Wait (Time_Period : in POSITIVE);

end Load_Control;
```

Load Control Package Body

```
with Text_IO;

package body Load_Control is

  type Load_Factor_Percentage is delta 0.05 range 0.0..10.0;

  My_Clock_ID : Clock_ID;
  Load_Factor : Load_Factor_Percentage := 1.0;
  Calibration : constant Load_Factor_Percentage := 0.75;
  Factor      : Load_Factor_Percentage;
  Temp        : BOOLEAN;

  package Load_Factor_IO is new Text_IO.Fixed_IO(Load_Factor_Percentage);

  procedure Initialize   (Clock_Identifier : in Clock_ID) is
  begin
    My_Clock_ID := Clock_Identifier;
  end Initialize;

  ----------
  --  Open external Factor file on host; read current value; close file
  ----------
  procedure Read_Load_Factor is
    Factor_File_Name : constant STRING := "25::ps:[borger]load_factor.inp";
    Factor_File      : Text_IO.FILE_TYPE;

  use Text_IO;
  begin
    Open(Factor_File, In_File, Factor_File_Name);
    Load_Factor_IO.Get(Factor_File, Load_Factor);
    Factor := Load_Factor_Percentage(Calibration * Load_Factor);
    Close(Factor_File);
  end Read_Load_Factor;

  procedure Busy_Wait (Time_Period : in POSITIVE) is
  begin
    for Index in 1..INTEGER(Time_Period * Factor)
    loop
        Temp  := KWV11_Clock_Manager.Interrupt_Flag_On(My_Clock_ID);
    end loop;
  end Busy_Wait;

end Load_Control;
```

Activate Dispatcher procedure

```
with TEXT_IO;
with Load_Control;
with Timer_Interrupt_Routine;

separate(Task_Manager)

  procedure Activate_Dispatcher is
    My_Clock_Name   : constant STRING := "KWV11";
    My_Clock_ID     : Clock_ID;
    My_Clock_Device : DEVICE_TYPE;
    CSR_Address     : ADDRESS;
    Period          : KWV_COUNTER_TYPE := KWV_COUNTER_TYPE(2_560);
```

```
use TEXT_IO, INS_Data_Types, KWV11_Clock_Manager;
begin
  ---------
  --  Initialize the clock to operate in mode one at a 1MHZ rate.
  --  The Interrupt Service Routine is "Timer_Interrupt_Routine".
  ---------
    Initialize(Clock_Name       => My_Clock_Name,
               Clock_Identifier => My_Clock_ID,
               Mode             => Mode_One,
               Rate             => Rate_1MHZ,
               Vector_Number    => 1,
               Service_Routine  => Timer_Interrupt_Routine'ADDRESS,
               CSR_Address      => CSR_Address,
         Clock_Priority         => Clock_IPL,
      Communication_Region_Size => Executive_Communication_Region'SIZE,
   Communication_Region_Address => Comm_Region_Address,
              Device_Object     => My_Clock_Device );

  ----------
  --  Update next schedule time in communication region.
  --  Start current tick number at 0 in communication region.
  ----------
  declare
    Comm_Region : INS_Data_Types.Executive_Communication_Region;
      for Comm_Region use at Comm_Region_Address;
  begin
    Comm_Region.Current_Tick_Number := 0;
    Comm_Region.Next_Schedule_Time  := Schedule_At_Tick_Number;
  end;

  ----------
  -- Properly initialize load control
  ----------
    Load_Control.Initialize(My_Clock_ID);
    Load_Control.Read_Load_Factor;

  ----------
  -- Enable clock overflow signals (interrupts)
  ----------
    Enable_Interrupts(My_Clock_ID);

  ----------
  -- Set interrupt time period to be 2_560 ticks (2.56 milliseconds)
  ----------
    Set_Interrupt_Period(My_Clock_ID, Period);

  ----------
  -- Start Dispatcher task
  ----------
    Dispatcher.Activate(My_Clock_ID, My_Clock_Device);

  ----------
  --  Start generating periodic interrupts
  ----------
   Generate_Interrupts(My_Clock_ID);


  exception
    when Initialization_Error =>
      Put_Line("Error during clock initialization.");
      raise Dispatcher_Activation_Error;

    when Clock_Not_Initialized =>
      Put_Line("Invalid clock identifier.");
      raise Dispatcher_Activation_Error;

    when others =>
      Put_Line("Unexpected exception raised back to Dispatcher_Activate_Dispatcher.");
      raise Dispatcher_Activation_Error;

end Activate_Dispatcher;
```

Initialize Activation Queue procedure

```
separate(Task_Manager)

  procedure Initialize_Activation_Queue is
    Next_Time          : Activation_Time_Range := NATURAL'FIRST;
    Activation_Records : constant array(Task_ID_Type)
                           of AQM.Task_Activation_Record :=
   ( (Ship_Velocity_Updater, 16, 16, 8, 8, AQM.Periodic),
     (Attitude_Periodic_Message_Sender, 24, 24, 7, 7, AQM.Periodic),
     (Navigation_Periodic_Message_Sender, 384, 384, 4, 4, AQM.Periodic),
     (Periodic_Status_Display_Processor, 390, 390, 3, 3, AQM.Periodic),
     (Runtime_BIT_Processor, 391, 391, 2, 2, AQM.Periodic),
     (Ship_Position_Updater, 508, 508, 1, 1, AQM.Periodic),
     (Comms_Controller, 0, 0, 10, 10, AQM.No_Op) );

  begin

    for Index in Task_ID_Type
    loop
      AQM.Insert_Activation_Record(Activation_Records(Index),
                               Schedule_At_Tick_Number);
    end loop;

  end Initialize_Activation_Queue;
```

Dispatcher task

```
with KWV11_Clock_Manager;  use KWV11_Clock_Manager;
with VAXELN_SERVICES;
with Text_IO;

separate (Task_Manager)

task body Dispatcher is
  My_Clock_ID          : Clock_ID;
  My_Clock_Device_ID   : DEVICE_TYPE;
  Current_AR           : Task_Activation_Record;
  Task_Missed_Deadline : BOOLEAN;
begin

  ----------
  --  Receive information needed for interfacing with the real-time clock
  ----------
  accept Activate (Clock_ID : in Clock_ID;
           Clock_Device_ID : in DEVICE_TYPE) do
    My_Clock_ID := Clock_ID;
    My_Clock_Device_ID := Clock_Device_ID;
  end Activate;

  ----------
  --  Loop and dispatch a new task for each clock interrupt
  ----------
  loop
    ----------
    --  Wait for a signal device (kernel service) call from the
    --  Timer Interrupt Routine and reset interrupt flag to allow
    --  more interrupts to be generated.
    ----------
      VAXELN_SERVICES.WAIT_ANY (Value1 => My_Clock_Device_ID);
      Reset_Interrupt_Flag(My_Clock_ID);
      Tick_Number := Tick_Number + 1;

      if Tick_Number >= Schedule_At_Tick_Number then
      ----------
      --  Get next activation record (whose task is to be scheduled) from
      --  Activation Queue and take the appropriate action.
      ----------
        Get_Activation_Record(Current_AR, Schedule_At_Tick_Number);

        case Current_AR.Activation_Mode is

          when Periodic | Single_Shot =>
            Activate_Task(Current_AR.Task_ID, Task_Missed_Deadline);
            if Task_Missed_Deadline then
              Text_IO.Put(Task_ID_Type'IMAGE(Current_AR.Task_ID) &
```

```
                                 " Missed deadline.");
                Text_IO.Put_Line(" Tick #: " & INTEGER'IMAGE(Tick_Number));
              end if;

            when Time_Out =>
              Time_Out_Task(Current_AR.Task_ID);

            when others =>
              null;

          end case;
        end if;
    end loop;

    ----------
    -- Stop clock operation
    ----------
      Re_Initialize(My_Clock_ID, Mode_Zero, Stop);

  end Dispatcher;
```

A.s. Main Program

```
with Task_Manager;

procedure INS is
begin
  Task_Manager.Initialize_Activation_Queue;
  Task_Manager.Activate_Dispatcher;
end INS;
```

Table of Contents

List of Figures

List of Tables