Technical Report

CMU/SEI-87-TR-029
ESD-TR-87-188

VAXELN Experimentation: Programming a Real-Time Clock and Interrupt Handling Using VAXELN Ada 1.1

Mark W. Borger

October 1987

Technical Report

CMU/SEI-87-TR-29
ESD/TR-87-188
October 1987

VAXELN Experimentation: Programming a Real-Time Clock and Interrupt Handling Using VAXELN Ada 1.1

Mark W. Borger

Ada Embedded Systems Testbed Project

Approved for public release.
Distribution unlimited.

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 Clock and Interrupt Handling Using VAXELN Ada 1.1

Abstract: This report describes the results of implementing an interrupt handler totally in Ada for a MicroVAX II/VAXELN 2.3 target system, the VAXELN 1.1 Ada compiler, and a KWV11-C programmable real-time clock. It provides an overview of VAXELN interrupt handlers and the operation of the real-time clock; discusses and demonstrates the use of VAXELN kernel services to establish a link between the clock's interrupt and the starting address of an interrupt service routine; presents an Ada package of interfaces to the KWV11-C device; provides Ada source code examples demonstrating the use of this package; and presents relevant observations, recommendations, and measurement results.

1. Introduction

This paper provides the reader with technical information and observations, Ada source code, and the results of our work in developing a real-time clock interface in Ada. The results are specific to a MicroVAX II/VAXELN 2.3 target system, the VAXELN 1.1 Ada compiler, and a KWV11-C programmable real-time clock; and they provide answers for such questions as:

- How does one write an interrupt service routine (ISR) in Ada?
- How is an Ada ISR associated with the occurrence of a hardware device interrupt?
- How can one control the operation of a KWV11-C programmable real-time clock using an Ada interface?

1.1. Background

We originally intended to investigate programming alternatives available to a real-time application developer for writing an interrupt handler, along with other appropriate Ada routines for a programmable real-time clock. Our approach was to code a simple Ada application which included:

- A main program that directs the real-time clock to generate interrupts at a frequency of 500 Hz, either through an existing interface or a newly developed one.
- A simple application task scheduler that logs a message to an external text file when it is called by the interrupt service routine.
- An interrupt service routine that handles time interrupts by invoking the application task scheduler.

Within this framework, the main program is also responsible for opening and closing the log file, enabling and disabling the timer interrupts, establishing the connection between the clock's interrupt vector and the service routine's starting address, and programming the clock rate. We originally intended to analyze both the run-time costs and software engineering tradeoffs (e.g., time and space performance, maintainability) associated with the implementation alternatives; specifically, we planned to measure the interrupt handler's execution speed, object code size, and the associated interrupt latency. However, we found only one alternative for implementing an interrupt handler totally in Ada for our target configuration and cross-compiler (VAXELN 2.3/VAXELN Ada 1.1), to use VAXELN kernel services to establish a link between the clock's interrupt and the starting address of an interrupt service routine. Thus, instead of following our original plan to examine various programming alternatives, we conducted a detailed study of this single VAXELN Ada interrupt-handling technique.

2. VAXELN Kernel

In contrast to the general purpose, time-sharing VAX/VMS operating system, VAXELN [DEC 85, DEC 86a] is a compact, more specialized run-time executive which supports the execution of application programs on "bare" VAX (i.e., no operating system support present) target machines. In particular, VAXELN Ada applications running on "bare" VAX target machines are supported entirely by the VAXELN run-time executive (i.e., kernel), by VAXELN services (e.g., file server), and by the VAXELN Ada run-time library. For an application system running under the VAXELN execution environment, these modules must be linked with the application's object code to produce a system load module (see Figure 2-1).

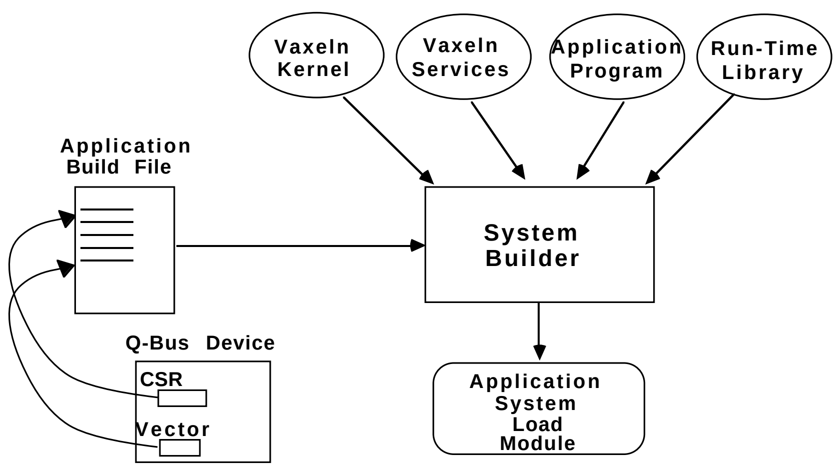

Figure 2-1: VAXELN Build Process

The VAXELN kernel is a layer of software between the VAX processor and application code. It provides mechanisms to communicate between processes; to control system resource usage; to create, suspend, resume, and delete jobs and processes; to schedule jobs and processes; and to maintain information about the user programs defined for a particular system. In a sense, the kernel is object-based since it exports most of its services through a set of procedures and functions (i.e., operations) which manipulate kernel objects (i.e., data structures). The predefined kernel objects include: AREA, DEVICE, EVENT, MESSAGE, NAME, PORT, PROCESS, and SEMAPHORE. The operations defined for these objects include creation, deletion, assignment, and comparison.

2.1. Interrupt Handling

The VAXELN kernel supports the notion of interrupt service routines (ISRs) for handling device interrupts in software. Since an ISR is invoked directly by the VAXELN kernel each time the device generates an interrupt, the ISR has the responsibility of taking appropriate action to service those interrupts. A VAXELN kernel service, namely CREATE_DEVICE (see Figure 2-2) establishes such a connection between a hardware interrupt and an ISR.

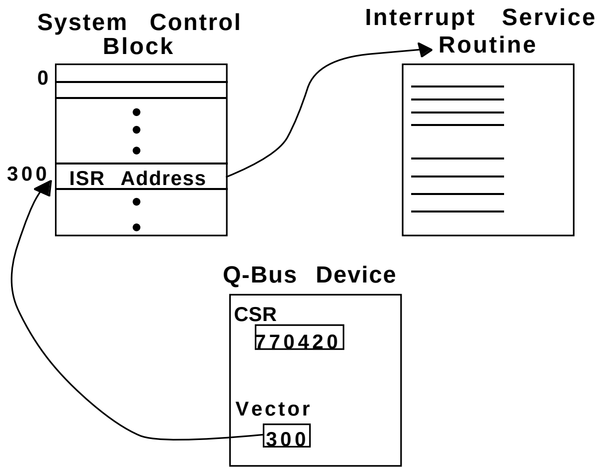

Figure 2-2: Associating a Device Interrupt with an ISR Via System Control Block Entry

This kernel service places the starting address of the ISR into the processor's system control block (SCB) [DEC 84] in order to link a device's interrupts to the ISR. At invocation, the CREATE_DEVICE procedure requires a device name, an interrupt vector number, and starting address of the ISR (which must match the information specified during the system build process; see Figure 2-1). In return, the out parameters are the device's base address (i.e., the address of its first control/status register), the address of a communication region that can be shared by an application and an ISR, and a VAXELN device object. The application code subsequently uses the device object to synchronize with the device's corresponding ISR.

2.2. Synchronizing the Application with Intercepts

The VAXELN kernel employs an object-based, signal/wait model (see Figure 2-3) for synchronizing application code with the hardware interrupts. Specifically, the kernel treats the device object returned from a CREATE_DEVICE call as a binary semaphore. When an interrupt occurs, the kernel invokes the appropriate ISR, which must signal the occurrence of the interrupt through the corresponding device object. This signaling is performed by a call to the non-blocking SIGNAL_DEVICE kernel service which sets the value of the device object (i.e., binary semaphore). The application code synchronizes with an ISR and, therefore, with the occurrence of a particular interrupt by waiting for this device signal, using either the WAIT_ANY or WAIT_ALL kernel service (see [DEC 86b] for further details). Calls to these services suspend until the specified conditions (in this case, a device object value of at least one) are satisfied or, optionally, a timeout occurs; if a wait on a device signal is satisfied, the device object's value is reset to zero.

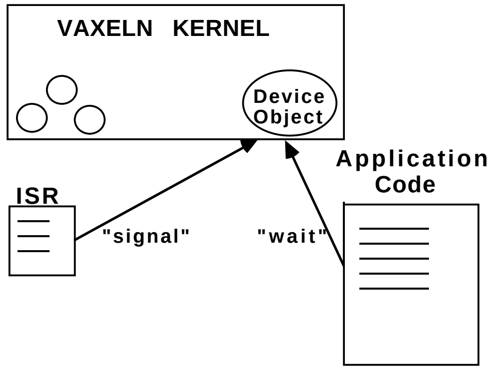

Figure 2-3: VAXELN Signal/Wait Synchronization Model

2.3. Data Sharing

The ISR and application code share data through an interrupt communication region. For example, the ISR passes data to the application code by reading the device data registers and placing the values into the communication region for later use by the application. The ISR receives the starting address of the communication region as a parameter from the VAXELN kernel; subsequently, the application code uses the data region address returned by the CREATE_DEVICE to access this shared region.

3. KWV11-C Programmable Real-time Clock

The KWV11-C printed circuit board is a programmable real-time clock that is Q-bus compatible.

3.1. Functional Description

The KWV11-C supports five clock rates (1 MHz, 100 KHz, 10 KHz, 1 KHz=z, 100 Hz), which are derived internally from a 10 MHz crystal oscillator. The device has a 16-bit counter that can generate processor interrupts, four programmable operation modes, and two Schmitt triggers, each with slope and level controls that can start the clock or generate interrupts. Refer to [DEC 86c] for further details about the Schmitt triggers.

The KWV11-C can generate two distinct interrupts, clock overflow and Schmitt trigger, and therefore requires two interrupt vectors. It has two read/write device registers that can be addressed by the processor; a control/status register (CSR) and buffer/preset register (BPR). The CSR allows you to control the operation of the device (e.g., enable interrupts, select clock rate, start the internal counter) and to query regarding its current operating status. The BPR supports two different functions depending on the clock's current mode of operation; both functions deal with interfacing with the clock's counter. In one case (Modes 0 and 1) it provides a mechanism for the loading the counter, and in the other (Modes 2 and 3) it provides indirect reading of the counter's current value.

3.2. Modes of Operation

The KWV11-C can operate in any one of four modes:

- Mode 0 Single Interval Interrupt: The GO command (i.e., setting the GO bit of the clock's CSR) is used to load the counter with the 2's complement of the number of ticks to wait before generating an interrupt. The counter increments at the selected clock rate until an overflow occurs and an interrupt is generated (assuming the INTOV flag of the CSR is set). It then waits for another GO command.
- Mode 1 Repeated Interval Interrupts: Same as Mode 0 except that the counter is re-loaded and continues counting after it overflows. This mode supports repeated interrupts whose period is the value in the BPR.
- Mode 2 External Event Timing: The counter increments at the selected clock rate and upon input (i.e., high logic signal) at Schmitt trigger #2, its contents are loaded into the BPR, where the value can be read. This firing of Schmitt trigger #2 can be simulated under program control by setting the MAIN_ST2 bit of the clock's CSR. In this mode, the counter continues without interruption.
- Mode 3 External Event Timing Zero Base: Same as Mode 2 except that the counter is reset to zero after its contents are loaded into the BPR.

3.3. Program Control

This section presents typical programming scenarios that can be used to control the KWV11-C for each of its operational modes. We assume that, where necessary, interrupt service routines are already associated with the clock's interrupts. Appendix B contains sample Ada code corresponding to each scenario.

Single Interval Interrupt (Mode 0)

1. Load the BPR with the 2's complement of the number of clock ticks to wait before generating an counter overflow interrupt.
2. Load the CSR with the appropriate settings: mode 0, desired clock rate, and the interrupt enable flag (INTOV) set to TRUE.
3. Set the CSR's GO bit to load the counter from the BPR. The counter increments at the selected clock rate until it overflows. A counter overflow interrupt is generated, and the CSR overflow flag (OVFLO) flag is set.
4. To repeat this process, clear the overflow flag (OVFLO) and set the GO bit.

Repeated Interval Interrupts (Mode 1)

1. Load the BPR with the 2's complement of the number of clock ticks representing the period at which counter overflow interrupts are to be generated.
2. Load the CSR with the appropriate settings: mode 1, desired clock rate, and the interrupt enable flag (INTOV) set to TRUE.
3. Set the CSR's GO bit to load the counter from the BPR. The counter increments at the selected clock rate until it overflows. The count value is then re-loaded from the BPR, an counter overflow interrupt is generated, and the CSR overflow flag (OVFLO) flag is set.
4. To allow subsequent interrupts, the overflow flag (OVFLO) must be cleared. If a second counter overflow occurs before the flag is reset, the flag overrun (FOR) bit is set.
5. To stop the clock from generating interrupts, clear the CSR's GO bit.

External Event Timing (Mode 2)

In this mode, interrupts can be generated while monitoring external events; external events can be counted; and the elapsed time of external events can be recorded. The scenario below addresses only the latter application.

1. Load the CSR with the appropriate settings: mode 2, and desired clock rate.
2. Set the CSR's GO bit to start the counter at the beginning of the external event that is to be timed. At this point, the counter is cleared and begins incrementing at the selected clock rate.
3. Upon completion of the timed event, simulate an external event by setting the maintenance flag of the the second Schmitt trigger (MAIN ST2). This input at ST2 causes the contents of the counter to be loaded into the BPR and sets the ST2 interrupt flag (INT2). Note: the counter continues ticking.
4. Accessing the value stored in the BPR gives the number of counter ticks that elapsed since the CSR's GO bit was set and the ST2 input occurred.
5. To stop the clock's counter, clear the CSR's GO bit.

External Event Timing Zero Base (Mode 3)

The programming scenario for Mode 3 is identical to that of Mode 2 except the counter is automatically cleared after every ST2 input.

4. Ada Interface to KWV11-C Clock

This section presents information specific to implementing an VAXELN Ada interface for a KWV11-C device operating within a MicroVAX II/VAXELN 2.3 target environment. We also discuss the process of handling device interrupts with VAXELN Ada code.

4.1. Access to Device Registers

The KWV11-C has two 16-bit read/write device registers, the control/status register (CSR) and the buffer/preset register (BPR). The CSR allows you to control the operation of the device (e.g., enable interrupts, select clock rate, start the internal counter) and to query regarding its current operating status. The BPR supports reading from and writing to the clock's counter. At the lowest level of the KWV11-C interface, data types must be defined and laid out using Ada representation specifications to allow full access to, and control of, the contents of the device registers. The following Ada package serves this purpose.

```
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;     -- enable 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;     -- interrupt overrun
    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;
```

This package also provides two primitive operations for reading and writing the contents of the clock's control/status register, namely Put_CSR and Get_CSR.

4.2. Ada Interrupt Service Routine

When writing an interrupt service routine (or any Ada subprogram) that will be invoked by the VAXELN kernel, the following requirements must be satisfied to ensure proper run-time behavior [DEC 86d].

- Each subprogram must either be a stand-alone program library unit or must be declared at the outer-most level of a library package (i.e., its specification or body).
- The subprogram's name must be exported via the appropriate VAXELN Ada pragma (e.g., EXPORT_PROCEDURE) in order to resolve any external references during linking.
- The subprogram must be compiled with a pragma SUPPRESS_ALL to disable stack overflow and underflow checks that would otherwise fail when invoked on the kernel stack.
- The subprogram must avoid using Ada tasking operations and input/output operations, and should minimize the calls to external subprograms.

The following is a minimal ISR coded in Ada.

```
with SYSTEM;
with VAXELN_SERVICES;           use VAXELN_SERVICES;
with CONDITION_HANDLING;
with KWV_Register_Definitions;  use KWV_Register_Definitions;

procedure Timer_Interrupt_Routine( Device_Registers : in out KWV_Registers;
                                   Interrupt_Region : in SYSTEM.ADDRESS;
                                    ISR_Context     : in ISR_CONTEXT_TYPE ) is
  Return_Code  : CONDITION_HANDLING.COND_VALUE_TYPE;
begin
```

```
  VAXELN_SERVICES.Signal_Device(Status        => Return_Code,
                                Device_Number => 0,
                                ISR_Context   => ISR_Context );
end Timer_Interrupt_Routine;
pragma EXPORT_PROCEDURE(Timer_Interrupt_Routine);
pragma SUPPRESS_ALL;
```

To use this ISR you would "with" the subprogram and then use Timer_Interrupt_Routine'ADDRESS as the address of the service routine in a CREATE_DEVICE call. For example, the following code associates the clock's counter overflow interrupt (first interrupt vector) with the Timer_Interrupt_Routine ISR.

```
with SYSTEM;
with VAXELN_SERVICES;
with CONDITION_HANDLING;
with Timer_Interrupt_Routine;

procedure ISR_Example is
  Device_Name  : constant STRING := "KWV11";
  Registers    : SYSTEM.ADDRESS;
  Return_Code  : CONDITION_HANDLING.COND_VALUE_TYPE;
  Timer_Device : VAXELN_SERVICES.DEVICE_ARRAY_TYPE(0..0) := (others => 0);
begin
    Create_Device (Status          => Return_Code,
                   Device_Name     => Device_Name,
                   Vector_Number   => 1,
                   Service_Routine => Timer_Interrupt_Routine'ADDRESS,
                   Registers       => Registers,
                   Device_Array    => Timer_Device,
                   Device_Count    => 1);
end ISR_Example;
```

Note that the string name of the device being created must match the name of a device specified in this program's VAXELN build file (see Figure 2-1). For instance, a typical build file for the main program might look like this:

```
program ISR_Example /debug /mode=kernel
device KWV11 /register=%O770420 /vector=%O440 /noautoload
terminal CONSOLE /hardcopy
```

4.3. Device Interface

The package specification listed below provides the necessary data types, procedures, functions, and exceptions for interfacing to multiple KWV11-C real-time clocks using Ada application code. These routines support all four modes of the clock's operation in addition to its five internal clock rates; however, only counter overflow interrupts are supported and not Schmitt trigger interrupts. The VAXELN Ada kernel services (KWV_INITIALIZE, KWV_READ, KWV_WRITE) provide the necessary interfaces for supporting the handling of the clock's Schmitt trigger interrupts. This Ada package specification is listed again in Appendix A along with its corresponding body.

```
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,      Rate1MHZ, Rate100KHZ,
                        Rate10KHZ, Rate1KHZ, Rate100HZ);

        for Clock_Rate use (Stop       => 0, Rate1MHZ  => 1,
                            Rate100KHZ => 2, Rate10KHZ => 3,
                            Rate1KHZ   => 4, Rate100HZ => 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;
```

4.4. Using the Device Interface

4.4.1. Initializing

The Initialize procedure creates a VAXELN device object for the clock and gives you a private clock identifier. The VAXELN device object can be used by the application to "wait" via a VAXELN kernel call on a device signal originating from an interrupt service routine. The clock identifier is a key for invoking all other subprograms in the package. The Initialization_Error exception is raised if the VAXELN kernel device object cannot be created. The clock's rate and mode are set by the Initialize procedure and can be reset using the Re_Initialize procedure; however, the address of the ISR associated with the clock's counter interrupt can only be specified through the Initialize interface. The Current_Rate and Current_Mode functions respectively return the clock's current rate and mode as set by either the Initialize or Re_Initialize procedure. The Display_CSR subprogram displays the current contents of the clock's control/status register to standard output.

4.4.2. Controlling Operation

The following routines can be used to control the operation of the clock and to query regarding its current operating status: Enable_Interrupts, Disable_Interrupts, Set_Interrupt_Period, Generate_Interrupts, Reset_Interrupt_Flag, Reset_Overrun_Flag, Interrupts_Enabled, Interrupt_Period, Interrupt_Flag_On, Overrun_Flag_On. Given a valid Clock_ID, these routines set, reset, and query current values for the various bit fields of the CSR and BPR associated with the clock device represented by the clock identifier. A brief functional description of each of these subprograms follows:

Enable_Interrupts	Set the **int_ovf** bit of the clock's CSR to enable interrupts when the internal counter overflows.
Disable_Interrupts	Reset the **int_ovf** bit of the clock's CSR to disable interrupts when the internal counter overflows.

Set_Interrupt_Period	Load the 2's complement representation of the specified number of ticks into the clock's BPR. This number of ticks represents the period for interrupt generation.
Generate_Interrupts	Set the GO bit of the clock's CSR to start the internal counter. This subroutine is used in conjunction with Enable_Interrupts.
Reset_Interrupt_Flag	Reset the **ovf_flag** bit of the clock's CSR to allow subsequent interrupts.
Reset_Overrun_Flag	Reset the **flag_overrun** bit of the clock's CSR. This bit is set when a counter overflow occurs and the **ovf_flag** has not been reset after the last interrupt. This indicates that the hardware is generating interrupts faster than the software can service them.
Interrupts_Enabled	Returns a Boolean value indicating whether or not the **int_ovf** bit of the clock's CSR is set.
Interrupt_Period	Returns the current interrupt period value in the clock's BPR.
Interrupt_Flag_On	Returns a Boolean value indicating whether the **ovf_flag** bit of the clock's CSR is set.
Overrun_Flag_On	Returns a Boolean value indicating whether the **flag_overrun** bit of the clock's CSR is set.

4.4.3. Time Measurements for External Events

The Start_Counting, Read_Counter, and Stop_Counting procedures provide support for timing external events. They should be used only in Modes 2 or 3. In any other mode, the Invalid_Clock_Mode exception will be raised. The distinction between Read_Counter and Stop_Counting is that the counter continues to tick when the clock is read by the Read_Counter subprogram and stops counting otherwise. These routines can be used in two ways:

1. Continuous timing

```
Start_Counter(My_Clock_ID);
loop
    . . .
  Read_Counter(My_Clock_ID, Number_Of_Ticks);
    . . .
end loop;
Stop_Counter(My_Clock_ID);
```

2. Single timing

```
Start_Counter(My_Clock_ID);
  < sequence of events to be timed>
Stop_Counter(My_Clock_ID);
```

4.4.4. Miscellaneous

Following are hints for using these routines:

- Be sure to follow the restrictions for implementing an ISR in Ada. See Section 4.2 for details.
- The three most likely causes of the Initialization_Error exception are:

    1. The device name specified in the Initialize call cannot be found in the list of devices created by the System Builder from the main program's build file.
    2. The Initialize procedure was called from a program that was not running in kernel mode.
    3. The device named in the Initialize call is already connected to a VAXELN device object.

  The counter routines should be invoked only when the clock is operating in Mode 2 or Mode 3.

5. Results

This section presents results specific to developing a real-time clock interface in Ada on a MicroVAX II/VAXELN 2.3 target system using the VAXELN 1.1 Ada compiler and a KWV11-C programmable real-time clock. These results take the form of technical observations relevant to an application developer, recommendations to the compiler implementor, and performance measurement results.

5.1. Technical Observations

We made the following observations while experimenting with VAXELN Ada and the real-time clock interfaces.

1. To redirect standard output to a file on a remote DECnet node, the File Access Listener option must be turned on at VAXELN system build time. Furthermore, the file that will receive the output must exist with WORLD read and write access enabled. There are two alternatives for redirecting output: redefine the system logical SYS$OUTPUT at build time, or use Ada TEXT_IO routines (OPEN, PUT, PUT_LINE, CLOSE) with the remote file name.
2. There are guidelines and restrictions for writing an ISR in Ada. See Section 4.2 for details.
3. Using the /map and /full qualifiers on the EBUILD command yields a complete map of everything in a program's executable load module. This information is useful for examining the CSR and vector addresses of the known devices. It is also handy for learning which device drivers are being loaded along with your main program.
4. When building a VAXELN application that calls the CREATE_DEVICE service, device-specific information must be provided in the program's VAXELN build file – minimally, the device name (a string matching that used in the application's CREATE_DEVICE call), the CSR address, the interrupt vector address, and an indication as to whether to load the standard device driver. Additionally, the application must be able to execute in kernel mode.
5. The VAXELN service, KWV_INITIALIZE, results in an access violation when it is used for re-initialization; the program terminates, which is incorrect behavior.
6. An Ada block with local variables whose memory locations are specified with address clauses provides an effective way of accessing data stored in particular locations of memory. For instance, the KWV_READ kernel service returns the starting address of the data it fetches. The following Ada code segment illustrates this technique for accessing data starting at a specific memory address:

```
KWV_READ (Identifier     => Clock_ID,
          Value_Count    => 1,
          Data_Array_Ptr => Clock_Data_Address,
          ST2_Go_Enable  => FALSE,
          Status         => Return_Code );

declare
  Ticks     : INTEGER := 0;
  Clock_Data : array (0..0) of UNSIGNED_WORD;
    for Clock_Data use at Clock_Data_Address;
begin
  Ticks := INTEGER(Clock_Data(0));
  Put_Line(INTEGER'IMAGE(Ticks));
end;
```

7. There appear to be at least two alternatives for writing to and reading from device registers in memory: directly assigning locations then using the technique described above to access them as Ada variables, or using predefined WRITE_REGISTER and READ_REGISTER subprograms. However, in practice , the first alternative cannot guarantee correct operational behavior–the generated code is likely to contain variable length bit field instructions, which are not permitted by the architecture, for accessing device registers. On the other hand, the WRITE_REGISTER and READ_REGISTER subprograms indicate to the compiler that only permissible instructions will be generated; therefore, the second alternative can guarantee proper run-time behavior. The following example further illustrates this point:

Direct Assignment

```
DA.05   procedure Enable_Interrupts (Clock_Identifier : in  Clock_ID) is
DA.06
DA.07     Current_CSR        : KWV_CSR_Record;
DA.08       for Current_CSR use at Clock_Array(Clock_Identifier);
DA.09
DA.10   begin
...
DA.17      if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
DA.18         Current_CSR.int_ovf := TRUE;
>>>       movzbl  #1,r2
>>>       insv   r2,#6,#1,(r3)
>>>       ret
DA.19       else
DA.20         raise Clock_Not_Initialized;
DA.21       end if;
DA.22
DA.23   end Enable_Interrupts;
```

Read/Write Register Calls

```
RW.06   procedure Enable_Interrupts (Clock_Identifier : in  Clock_ID) is
RW.07
RW.08   Current_CSR  : KWV_CSR_Record;
RW.09   Temp         : UNSIGNED_WORD;
RW.10   CSR_Unsigned : UNSIGNED_WORD;
RW.11      for CSR_Unsigned use at Clock_Array(Clock_Identifier);
RW.12
RW.13   function Convert_It is new UNCHECKED_CONVERSION(UNSIGNED_WORD,
                                                       KWV_CSR_Record);
RW.14   function Convert_It is new UNCHECKED_CONVERSION(KWV_CSR_Record,
                                                         UNSIGNED_WORD);
RW.15
RW.16   begin
...
RW.23   if Clock_Array(Clock_Identifier) /= ADDRESS_ZERO then
RW.24   Temp := READ_REGISTER(CSR_Unsigned);
>>>       movw  (r1),r0
>>>       movw  r0,r2
RW.25         Current_CSR := Convert_It(Temp);
>>>       movw  r2,-20(fp)
>>>       movab  -16(fp),r3
>>>       movw  -20(fp),(r3)+
RW.26         Current_CSR.int_ovf := TRUE;
>>>       movzbl  #1,r3
>>>       insv   r3,#6,#1,-16(fp)
RW.27         Temp := Convert_It(Current_CSR);
>>>       movab   -14(fp),r3
>>>       movw   -16(fp),(r3)+
>>>       movw   -14(fp),r2
```

```
RW.28         WRITE_REGISTER(Temp, CSR_Unsigned);

>>>        cvtwl  r2,r3
>>>        movw   r3,(r1)
>>>        ret
RW.29       else
RW.30         raise Clock_Not_Initialized;
RW.31       end if;
RW.32
RW.33   end Enable_Interrupts;
```

The generated assembler code in these examples (indicated by **>>>** at the start of the line) shows that the **insv** (insert variable length bit field) instruction is used for doing the Boolean assignment **current_CSR.int_ovf := TRUE;** in both code segments (Line DA.18, RW.26). In the first case, the base operand of the instruction is a device register and will yield unpredictable results. In the second case, the base operand of the **insv** instruction is a temporary variable that later performs the necessary type conversion (Line RW.27) prior to the WRITE_REGISTER call. Notice that the WRITE_REGISTER call generates a move word (movw) instruction (Line RW.28) for writing to the device register.

5.2. Recommendations

1. Vendors should supply better documentation and more detailed examples of Ada ISRs. For example, the present documentation does not explicitly state the number and type of ISR parameters. A trouble-shooting checklist for commonly occurring problems would also be useful.
2. In order to support more functionality, the restrictions should be loosened on the ISR code to avoid Ada tasking operations and to minimize the calls to external subprograms. Task entry calls should be permitted from within an ISR in order to provide an interrupt handling capability similar to the one suggested in the Ada Language Reference Manual [DoD 83].

5.3. Performance Measurements

For each example of Ada code presented in Appendix B, we recorded:

- number of lines of code (i.e., number of carriage returns)
- number of statements (i.e., number of semi-colons)
- object code size
- system load module size

We also used the KWV11-C real-time clock to measure the elapsed time from when the hardware generates an interrupt until the application code resumes execution. The interrupt latency time can be more accurately measured using hardware techniques (e.g., logic analayer) and still must be done.

VAXELN Ada Code Sizes

Program Name	LOC	# Stmts	Object Code Size (bytes)	Bytes Per LOC	Load Module(bytes)
Mode0_Test	110	44	5632	51.2	305_152
Mode1_Test	118	45	6144	52.1	305_152
Mode2_Test	70	30	5120	73.1	304_128
Mode3_Test	76	34	5120	67.4	304_128

We used two software measurement techniques to measure the elapsed time.

Technique #1

The essence of this approach is to start at an interrupt frequency that the software can handle and to increase this frequency until the software can no longer service the interrupts fast enough. This will give a rough measure of the time elapsed from the interrupt occurrence until the application code is re-scheduled and executed. This measurement can be taken by operating the clock in Mode 1 and looping, decrementing the interrupt period by one for each iteration until the clock's overrun flag is set, indicating that software is not keeping up with the interrupt rate. The following pseudo-code represents the logic of this technique (see Appendix C for the Ada code associated with this approach):

```
.  Ticks := 5000;

loop
.  Re-initialize clock
.  Enable clock overflow
.  Ticks := Ticks - 1;
.  Program clock to generate interrupt every Ticks microseconds

.  Start generating the interrupts
.  Wait for a signal device (kernel service) call from the ISR
.  Reset interrupt flag to allow more interrupts to be generated

.  Exit when Overrun_Flag_On(My_Clock_ID);
end loop;

.  Print current value of Ticks
```

Technique #2

This technique is direct and reliable. It can be performed when the clock is operating in either Mode 2 or Mode 3. It combines the counter-reading capability of these modes with the fact that the counter will generate interrupts when it overflows, regardless of the mode of operation. The approach is to enable counter overflow interrupts, start the counter, wait for a signal from the ISR caused by an interrupt, and finally read the current counter value. The following pseudo-code represents the logic of this technique (see Appendix C for the Ada code associated with this approach):

```
. Enable overflow interrupts
. Start Counting
. Wait for a signal device (kernel service) call from the ISR
. Stop Counting (read current counter contents)
. Print number of Ticks
```

VAXELN Ada Software Interrupt Latency (μsec)

(Each average based on 25 data points)

	Technique #1	Mode 2	Technique #2 Mode 3
Maximum time	903.00	331.00	277.00
Minimum time	229.00	270.00	256.00
Average time	357.12	274.20	271.04
Standard Deviation	237.87	11.63	4.70

Bibliography

[DEC 84] Digital Equipment Corporation.
Guide to Writing a Device Driver for VAX/VMS
Maynard, Massachusetts, 1984.

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

[DEC 86a] Digital Equipment Corporation.
VAXELN Release Notes
Maynard, Massachusetts, 1986.

[DEC 86b] Digital Equipment Corporation.
VAXELN Ada User's Manual
Maynard, Massachusetts, 1986.

[DEC 86c] Digital Equipment Corporation.
LSI-11 Analog System Users' Guide
Maynard, Massachusetts, 1986.

[DEC 86d] Digital Equipment Corporation.
VAXELN Ada Version 1.1 Release Notes
Maynard, Massachusetts, 1986.

[DoD 83] U.S. Department of Defense.
Reference Manual for the Ada Programming Language.
ANSI/MIL-STD 1815A, DoD, January, 1983.

Appendix A: KWV11_Clock_Manager Source Code

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 : VAXELN/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 : VAXELN/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;
  pragma INLINE(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;
  pragma INLINE(Get_CSR);


end KWV_Register_Definitions;
```

A.c. KWV11_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 : VAXELN/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,      Rate1MHZ, Rate100KHZ,
                      Rate10KHZ, Rate1KHZ, Rate100HZ);

      for Clock_Rate use (Stop       => 0, Rate1MHZ  => 1,
                          Rate100KHZ => 2, Rate10KHZ => 3,
                          Rate1KHZ   => 4, Rate100HZ => 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. KWV11_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 : VAXELN/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;
pragma INLINE(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;
pragma INLINE(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;
  pragma INLINE(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;
pragma INLINE(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;
pragma INLINE(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;
pragma INLINE(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 which 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;
pragma INLINE(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;
pragma INLINE(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;
pragma INLINE(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;
pragma INLINE(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 in order to
  --  get current value of the clock’ 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 in order to
  --  get current value of the clock’ 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;
pragma INLINE(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;
pragma INLINE(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;
pragma INLINE(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;
pragma INLINE(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;
pragma INLINE(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;
pragma INLINE(Overrun_Flag_On);
```

Appendix B: Examples of KWV11-C Interface

B.a. Mode 0 Operation

```
with SYSTEM;                     use SYSTEM;
with TEXT_IO;                    use TEXT_IO;
with CALENDAR;                   use CALENDAR;
with KWV11_Clock_Manager;        use KWV11_Clock_Manager;
with VAXELN_SERVICES;
with CONDITION_HANDLING;
with UNCHECKED_CONVERSION;
with Timer_Interrupt_Routine;

procedure Mode0_Test is
  Clock_Name      : constant STRING := "KWV11";
  Log_File_Name   : constant STRING := "25::ps:[borger]mode0_test.log";
  Log_File        : FILE_TYPE;
  My_Clock_ID     : Clock_ID;
  My_Clock_Device : DEVICE_TYPE;
  CSR             : ADDRESS;
  Ticks           : KWV_COUNTER_TYPE := KWV_COUNTER_TYPE(100);
  Return_Code     : COND_VALUE_TYPE;
  Result_Code     : INTEGER;

  subtype Date_Time_Type is VAXELN_SERVICES.Date_Time_Type;

  function Future_Time (Time_Interval : Day_Duration) return Date_Time_Type is
     function Time_To_Date_Time is new UNCHECKED_CONVERSION(Time, Date_Time_Type);
  begin
    return Time_To_Date_Time(Clock + Time_Interval);
  end Future_Time;

begin
  ----------
  -- Open external log file on host
  ----------
    Open(Log_File, Out_File, Log_File_Name);
    Set_Output(Log_File);

  ---------
  --  Initialize the clock to operate in mode zero at a 1MHZ rate.
  --  The Interrupt Service Routine is "Timer_Interrupt_Routine".
  ---------
    Initialize(Clock_Name => Clock_Name,
         Clock_Identifier => My_Clock_ID,
                    Mode  => Mode_Zero,
                     Rate => Rate1MHZ,
            Vector_Number => 1,
          Service_Routine => Timer_Interrupt_Routine'ADDRESS,
              CSR_Address => CSR,
            Device_Object => My_Clock_Device );

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

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

```
  -- Set interrupt time period to be 100 ticks (microseconds)
  ----------
    Set_Interrupt_Period(My_Clock_ID, Ticks);

  for Index in 1..100
  loop
    ----------
    --  Start generating the interrupts (in this case, only one interrupt
    --  since the clock is operating in mode 0).
    ----------
      Generate_Interrupts(My_Clock_ID);

    ----------
    --  Wait for a signal device (kernel service) call from the
    --  Timer Interrupt Routine.  Timeout after 5 seconds.
    ----------
      VAXELN_SERVICES.WAIT_ANY (Status => Return_Code,
                                Result => Result_Code,
                                Time   => Future_Time(5.0),
                                Value1 => My_Clock_Device );

    ----------
    --  Determine if signal device (kernel service) call was made, or else
    --  we timed out after 5 seconds.
    ----------
      if CONDITION_HANDLING.Success(Return_Code) and then Result_Code = 1 then
        Put_Line("Device Signal received.");
      else
        Put_Line("WAIT_ANY timed out.");
      end if;

  end loop;

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

  ----------
  -- Close external log file on host
  ----------
    Close(Log_File);

  exception
    when Initialization_Error =>
      Put_Line("Error during clock initialization.");
      Close(Log_File);

    when Clock_Not_Initialized =>
      Put_Line("Invalid clock identifier.");
      Close(Log_File);

    when others =>
      Close(Log_File);

end Mode0_Test;
```

B.b. Mode 1 Operation

```
with SYSTEM;                    use SYSTEM;
with TEXT_IO;                   use TEXT_IO;
with CALENDAR;                  use CALENDAR;
with KWV11_Clock_Manager;       use KWV11_Clock_Manager;
with VAXELN_SERVICES;
with CONDITION_HANDLING;
with UNCHECKED_CONVERSION;
with Timer_Interrupt_Routine;

procedure Mode1_Test is
  Clock_Name      : constant STRING := "KWV11";
  Log_File_Name   : constant STRING := "25::ps:[borger]mode1_test.log";
  Log_File        : FILE_TYPE;
  My_Clock_ID     : Clock_ID;
  My_Clock_Device : DEVICE_TYPE;
  CSR             : ADDRESS;
  Ticks           : KWV_COUNTER_TYPE := KWV_COUNTER_TYPE(10_000);
  Return_Code     : COND_VALUE_TYPE;
  Result_Code     : INTEGER;

  subtype Date_Time_Type is VAXELN_SERVICES.Date_Time_Type;

  function Future_Time (Time_Interval : Day_Duration) return Date_Time_Type is
     function Time_To_Date_Time is new UNCHECKED_CONVERSION(Time, Date_Time_Type);
  begin
    return Time_To_Date_Time(Clock + Time_Interval);
  end Future_Time;

begin
  ----------
  -- Open external log file on host
  ----------
    Open(Log_File, Out_File, Log_File_Name);
    Set_Output(Log_File);

  ---------
  --  Initialize the clock to operate in mode zero at a 1MHZ rate.
  --  The Interrupt Service Routine is "Timer_Interrupt_Routine".
  ---------
    Initialize(Clock_Name => Clock_Name,
         Clock_Identifier => My_Clock_ID,
                    Mode  => Mode_One,
                     Rate => Rate1MHZ,
            Vector_Number => 1,
          Service_Routine => Timer_Interrupt_Routine'ADDRESS,
              CSR_Address => CSR,
            Device_Object => My_Clock_Device );

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

  ----------
  -- Set Interrupt time period to be 10_000 ticks (microseconds)
  ----------
    Set_Interrupt_Period(My_Clock_ID, Ticks);
```

```
  ----------
  --  Start generating the interrupts (in this case, repeatedly
  --  since the clock is operating in mode 1).
  ----------
      Generate_Interrupts(My_Clock_ID);

  ----------
  --  Handle 100 interrupts
  ----------
  for Index in 1..100
  loop
    ----------
    --  Wait for a signal device (kernel service) call from the
    --  Timer Interrupt Routine.  Timeout after 2 seconds.
    ----------
      VAXELN_SERVICES.WAIT_ANY (Status => Return_Code,
                                Result => Result_Code,
                                Time   => Future_Time(2.0),
                                Value1 => My_Clock_Device );

    ----------
    --  Reset interrupt flag to allow more interrupts to be generated
    ----------
      Reset_Interrupt_Flag(My_Clock_ID);

    ----------
    --  Determine if signal device (kernel service) call was made, or else
    --  we timed out after 2 seconds.
    ----------
      if CONDITION_HANDLING.Success(Return_Code) and then Result_Code = 1 then
        Put_Line("Device Signal received.");
      else
        Put_Line("WAIT_ANY timed out.");
      end if;

  end loop;

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

  ----------
  -- Close external log file on host
  ----------
    Close(Log_File);

  exception
    when Initialization_Error =>
      Put_Line("Error during clock initialization.");
      Close(Log_File);

    when Clock_Not_Initialized =>
      Put_Line("Invalid clock identifier.");
      Close(Log_File);

    when others =>
      Close(Log_File);

end Mode1_Test;
```

B.c. Mode 2 Operation

```
with SYSTEM;              use SYSTEM;
with TEXT_IO;             use TEXT_IO;
with KWV11_Clock_Manager; use KWV11_Clock_Manager;

procedure Mode2_Read_Test is
  Clock_Name      : constant STRING := "KWV11";
  Log_File_Name   : constant STRING := "25::ps:[borger]mode2_read_test.log";
  Log_File        : FILE_TYPE;
  My_Clock_Id     : Clock_ID;
  My_Clock_Device : DEVICE_TYPE;
  CSR             : ADDRESS;
  Ticks           : KWV_COUNTER_TYPE;

begin
  ----------
  --  Open external log file on host
  ----------
    Open(Log_File, Out_File, Log_File_Name);
    Set_Output(Log_File);

  ---------
  --  Initialize the clock to operate in mode two at a 1MHZ rate
  ---------
    Initialize(Clock_Name => Clock_Name,
         Clock_Identifier => My_Clock_ID,
                    Mode  => Mode_Two,
                     Rate => Rate1MHZ,
            Vector_Number => 1,
          Service_Routine => ADDRESS_ZERO,
              CSR_Address => CSR,
            Device_Object => My_Clock_Device );

  ----------
  --  Repeatedly measure overhead time associated with starting and
  --  stopping the clock’s counting; record this overhead time in the log file
  ----------
    Start_Counting(My_Clock_ID);
    for Index in 1..500
    loop
      Read_Counter(My_Clock_ID, Ticks);
      Put(INTEGER’IMAGE(INTEGER(Ticks)));

      if (Index rem 10) = 0 then
        New_line;
      end if;
    end loop;

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

  ----------
  --  Close external log file on host
  ----------
    Close(Log_File);

  exception
```

```
    when Initialization_Error =>
      Put_Line("Error during clock initialization.");
      Close(Log_File);

    when Clock_Not_Initialized =>
      Put_Line("Invalid clock identifier.");
      Close(Log_File);

    when others =>
      Close(Log_File);

end Mode2_Read_Test;
```

B.d. Mode 3 Operation

```
with SYSTEM;              use SYSTEM;
with TEXT_IO;             use TEXT_IO;
with KWV11_Clock_Manager; use KWV11_Clock_Manager;

procedure Mode3_Test is
  Clock_Name      : constant STRING := "KWV11";
  Log_File_Name   : constant STRING := "25::ps:[borger]mode3_test.log";
  Log_File        : FILE_TYPE;
  My_Clock_Id     : Clock_ID;
  My_Clock_Device : DEVICE_TYPE;
  CSR             : ADDRESS;
  Ticks           : KWV_COUNTER_TYPE;

begin
  ----------
  --  Open external log file on host
  ----------
    Open(Log_File, Out_File, Log_File_Name);
    Set_Output(Log_File);

  ---------
  --  Initialize the clock to operate in mode three at a 1MHZ rate
  ---------
    Initialize(Clock_Name => Clock_Name,
         Clock_Identifier => My_Clock_ID,
                    Mode  => Mode_Three,
                     Rate => Rate1MHZ,
            Vector_Number => 1,
          Service_Routine => ADDRESS_ZERO,
              CSR_Address => CSR,
            Device_Object => My_Clock_Device );

  ----------
  --  Repeatedly measure overhead time associated with starting and
  --  stopping the clock’s counting; record this overhead time in the log file
  ----------
    for Index in 1..500
    loop
      Start_Counting(My_Clock_ID);
      Stop_Counting(My_Clock_ID, Ticks);
      Put(INTEGER’IMAGE(INTEGER(Ticks)));
    end;

      if (Index rem 10) = 0 then
        New_line;
      end if;
    end loop;

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

  ----------
  --  Close external log file on host
  ----------
    Close(Log_File);
```

```
  exception
    when Initialization_Error =>
      Put_Line("Error during clock initialization.");
      Close(Log_File);

    when Clock_Not_Initialized =>
      Put_Line("Invalid clock identifier.");
      Close(Log_File);

    when others =>
      Close(Log_File);

end Mode3_Test;
```

Appendix C: Software Measurement Techniques Using the KWV11-C Interface

C.a. Technique #1

```
with SYSTEM;                    use SYSTEM;
with TEXT_IO;                   use TEXT_IO;
with KWV11_Clock_Manager;       use KWV11_Clock_Manager;
with VAXELN_SERVICES;
with CONDITION_HANDLING;
with Timer_Interrupt_Routine;

procedure Mode1_Wait_Time is
  Clock_Name      : constant STRING := "KWV11";
  My_Clock_ID     : Clock_ID;
  My_Clock_Device : DEVICE_TYPE;
  CSR             : ADDRESS;
  Ticks           : KWV_COUNTER_TYPE := KWV_COUNTER_TYPE(5000);

begin
  ---------
  --  Initialize the clock to operate in mode zero at a 1MHZ rate.
  --  The Interrupt Service Routine is "Timer_Interrupt_Routine".
  ---------
    Initialize(Clock_Name => Clock_Name,
         Clock_Identifier => My_Clock_ID,
                    Mode  => Mode_Zero,
                     Rate => Rate1MHZ,
            Vector_Number => 1,
          Service_Routine => Timer_Interrupt_Routine'ADDRESS,
              CSR_Address => CSR,
            Device_Object => My_Clock_Device );

  ----------
  --  Loop until the clock's overrun flag is set which will indicate that
  --  software is not keeping up with the interrupt rate.  This will give
  --  a rough measure of the elapsed time from the time of the interrupt
  --  until the application code is re-scheduled and executed.
  ----------
  loop

    ----------
    --  Re-initialize clock
    ----------
      Re_Initialize(My_Clock_ID, Mode_One, Rate1MHZ);

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

    ----------
    --  Decrease number of ticks counted to generate an interrupt
    ----------
      Ticks := KWV_COUNTER_TYPE(INTEGER(Ticks) - 1);

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

```
    -- Program clock to generate interrupt every Ticks microseconds
    ----------
      Set_Interrupt_Period(My_Clock_ID, Ticks);

    ----------
    --  Start generating the interrupts (in this case, repeatedly
    --  since the clock is operating in mode 1).
    ----------
      Generate_Interrupts(My_Clock_ID);

    ----------
    --  Wait for a signal device (kernel service) call from the
    --  Timer Interrupt Routine.
    ----------
      VAXELN_SERVICES.WAIT_ANY (Value1 => My_Clock_Device);

    ----------
    --  Reset interrupt flag to allow more interrupts to be generated
    ----------
      Reset_Interrupt_Flag(My_Clock_ID);

    exit when Overrun_Flag_On(My_Clock_ID);

  end loop;

  Put_Line(INTEGER'IMAGE(INTEGER(Ticks)));

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


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

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

    when others =>
      Put_Line("Unexpected exception raised.");

end Mode1_Wait_Time;
```

C.b. Technique #2

```
with SYSTEM;              use SYSTEM;
with TEXT_IO;             use TEXT_IO;
with KWV11_Clock_Manage; use KWV11_Clock_Manage;
with VAXELN_SERVICES;
with Time_Interrupt_Routine;

procedure Mode2_Wait_Time is
  Clock_Name      : constant STRING := "KWV11";
  My_Clock_Id     : Clock_ID;
  My_Clock_Device : DEVICE_TYPE;
  CSR             : ADDRESS;
  Ticks           : KWV_COUNTER_TYPE;

begin
  ---------
  --  Initialize the clock to operate in mode two at a 1MHZ ate
  ---------
    Initialize(Clock_Name => Clock_Name,
         Clock_Identify   => My_Clock_ID,
                    Mode  => Mode_Two,
                     Rate => Rate1MHZ,
            Vector_Number => 1,
          Service_Routine  => Time_Interrupt_Routine'ADDRESS,
              CSR_Address => CSR,
            Device_Object => My_Clock_Device );

  ----------
  --  Enable counter overflow interrupts
  ----------
    Enable_Interrupts(My_Clock_ID);

  ----------
  --  Measure overhead time associated with handling an interrupt and
  --  continuing application code after a WAIT_ANY
  ----------
    Stat_Counting(My_Clock_ID);
      VAXELN_SERVICES.WAIT_ANY(My_Clock_Device);
    Stop_Counting(My_Clock_ID, Ticks);
    Put(INTEGER'IMAGE(INTEGER(Ticks)));

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

end Mode2_Wait_Time;
```

Table of Contents

List of Figures