DIGITAL 2T-CSRTC PCI Real Time Clock

User's Guide

Part Number: EK-CSRTC-UG. A01

March 1999

Compaq Computer Corporation
Houston, Texas

March 1999

The information in this publication is subject to change without notice.

COMPAQ COMPUTER CORPORATION SHALL NOT BE LIABLE FOR TECHNICAL OR EDITORIAL ERRORS OR OMISSIONS CONTAINED HEREIN, NOR FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL. THIS INFORMATION IS PROVIDED "AS IS" AND COMPAQ COMPUTER CORPORATION DISCLAIMS ANY WARRANTIES, EXPRESS, IMPLIED OR STATUTORY AND EXPRESSLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR PARTICULAR PURPOSE, GOOD TITLE AND AGAINST INFRINGEMENT.

NOTE

The Digital PCI Real Time Clock (RTC) module was tested for EMC compliance without an I/O cable connected to the External User Port because each customer's application/cable/active or passive device may be different. The customer is responsible for verification of continued EMC compliance of the RTC module with any external connection.

S3719

Table of Contents

1 PCI Real Time Clock Module Overview

2 Real Time Clock Registers

3 Hardware Installation

4 PCI Real Time Clock Device Driver

5 Driver Interfaces

A Software Structures for C Language

Figures

Tables

Preface

Overview

The *DIGITAL 2T-CSRTC PCI Real Time Clock User's Guide* describes the DIGITAL PCI Real Time Clock hardware module and the installation procedure for the DIGITAL PCI Real Time Clock device driver for DIGITAL UNIX.

Organization

This guide is organized as follows:

Chapter 1 provides an overview the DIGITAL PCI Real Time Clock module.

Chapter 2 contains the DIGITAL Real Time Clock register address map and a description of the registers.

Chapter 3 describes the DIGITAL Real Time hardware installation.

Chapter 4 describes the DIGITAL Real Time Clock device driver for DIGITAL UNIX installation and testing.

Chapter 5 describes the DIGITAL Real Time Clock driver interfaces.

Appendix A describes the software structures for C language.

Appendix B contains a listing of the rtc_ivp_def.h file.

Appendix C contains a listing of the rtc_ivp.c file.

Conventions

This document uses the following conventions:

Convention	Meaning
Note	A note calls the reader's attention to any item of information that may be of special importance.
Caution	A caution contains information essential to avoid damage to the equipment.
Warning	A warning contains information essential to the safety of personnel.
Italic type	*Italic type* emphasizes important information, indicates variables, and indicates complete titles of manuals.
bold type	**Bold type** indicates text that is highlighted for emphasis.
`Monospaced`	In text, this typeface indicates the exact name of a command, routine, partition, pathname, directory, or file.

Note

The DIGITAL UNIX commands used in this manual are case sensitive and must be entered as shown.

Reader's Comments

Compaq welcomes your comments on this or any other manual. You can send your comments to Compaq in the following ways:

- Internet electronic mail: *reader-comments@digital.com*
- Mail:

  Compaq Computer Corporation

  Information Design

  PKO3-2/21J

  129 Parker Street

  Maynard, MA 01754-2199

For additional information, call 1-800-344-4825.

1
PCI Real Time Clock Module Overview

1.1 Introduction

The DIGITAL PCI Real Time Clock (RTC) module is a single card that occupies one PCI slot and provides the user with several methods of timing events in a PCI based computer system. A wide assortment of user configurable and/or programmable options enable events to be counted or timed. A total of 8 discrete inputs and 8 discrete outputs are also provided. Access to external I/O is provided via a connector on the front panel. The DIGITAL PCI RTC module is designed as a plug-in module to the standard 5V PCI bus and is compliant with the PCI Local Bus, Rev. 2.1 specification.

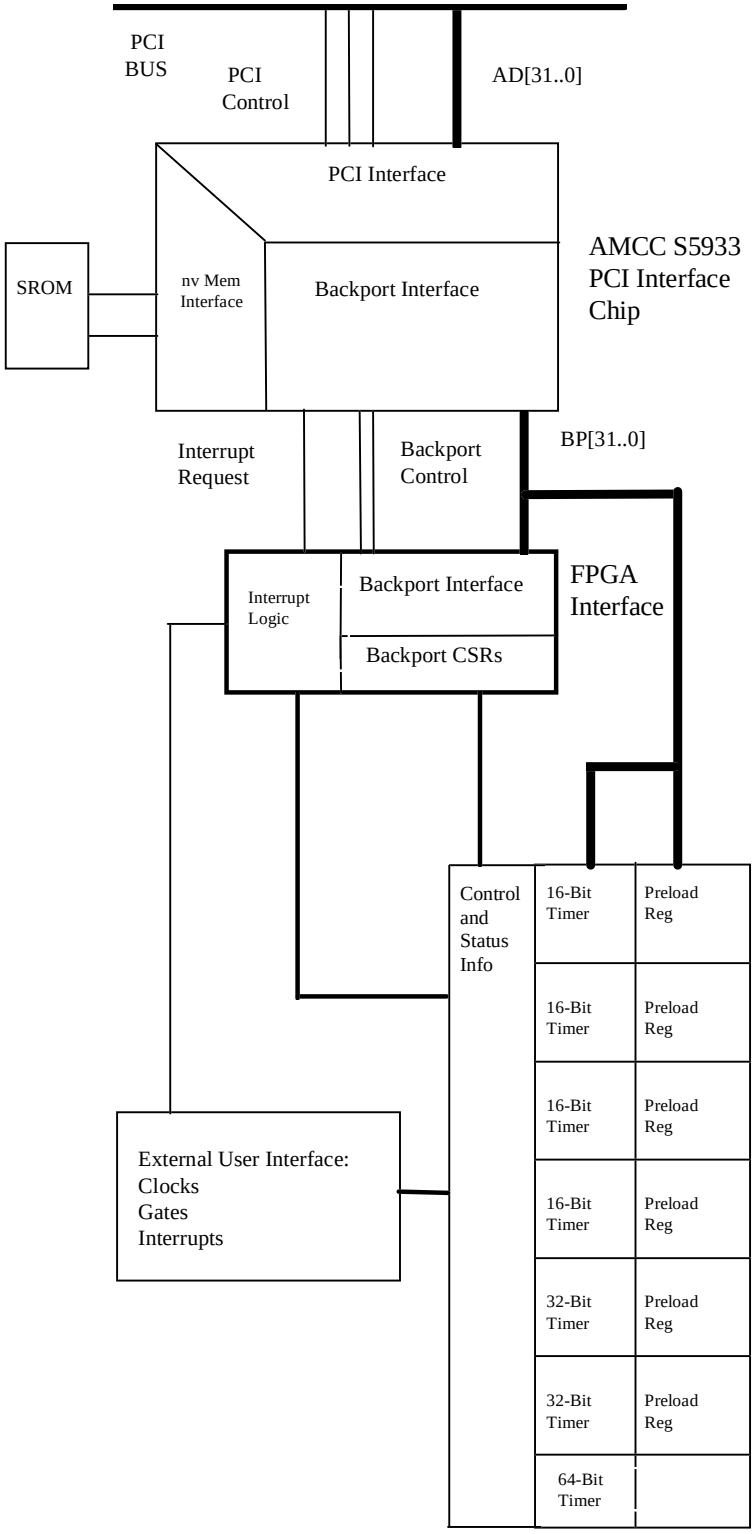
1.2 Major Functional Elements

The DIGITAL PCI RTC module is comprised of the following major functional elements, each of which is described in subsequent chapters of this manual:

- Timers
- Discrete I/O
- External Connections

Figure 1-1 shows a block diagram of the DIGITAL PCI Real Time Clock module.

Figure 1-1 DIGITAL PCI RTC Module Block Diagram



1.3 Timer Descriptions

There are a total of 7 timers on the DIGITAL PCI RTC module, one 64-bit, two 32-bit, and four 16-bit timers. Each timer has several characteristics and modes of operation that are programmed by the user.

- Count up / Count Down
- Count once / Repeat count
- Count enable / Count Disable
- Internal gate / External gate
- Terminal count output enable / Terminal count output disable
- Interrupt enable / Interrupt disable
- Preload count value (except for the 64-bit timer[1])
- One of eight clock frequency selection modes (including SW clocking and external clocking)

1.4 Discrete I/O

A total of 8 discrete inputs and 8 discrete outputs are provided on the DIGITAL PCI RTC module. All discrete points are buffered. A single address is used to access these I/Os. A write access writes the outputs and a read returns the data on the inputs. Note that the discrete outputs can not be read nor can the discrete inputs be written.

1.4.1 Discrete Inputs

The discrete input port is configured as a bank of transparent latches with the latch enable control signal brought out to the user connector. The external control signal LA_DISCRETE_INL is pulled high so that the latches are transparent by default. The user can latch in the data by bringing the line low.

1.4.2 Discrete Outputs

Data written to the discrete output port is registered and appears directly on the user connector.

1.4.3 LED's

There are two external LED's on the DIGITAL PCI RTC module mounting bracket. The green DC OK LED indicates that the board has gone through the initialization phase and is now ready to perform its function. The yellow LED functions as a discrete output indicator. Writing a one to bit zero of the Discrete I/O output register will turn the yellow LED on. Writing a zero will turn the yellow LED off.

---

[1] The 64-bit timer can be loaded with a starting value, however, the timer will be triggered when the timer reaches 0.

1.5 External User Connections

The signals shown in Table 1-1 and Table 1-2 are brought out to the 62-pin D connector on the front panel. All signals are buffered with 74ABT244A devices and are TTL compatible.

Table 1-1 User External Connector Outputs

Outputs	Signal Name	Pin Number
Divide-By Clocks	BUF_DIV2	60
	BUF_DIV4	19
	BUF_DIV8	40
	BUF_DIV16	61
	BUF_DIV32	20
	BUF_DIV64	41
Timer Clocks	BUF_CLK64	18
	BUF_CLK32B	59
	BUF_CLK32A	38
	BUF_CLK16D	17
	BUF_CLK16C	58
	BUF_CLK16B	37
	BUF_CLK16A	16
Timer Outputs	BUF_COUTPUT0	30
	BUF_COUTPUT1	9
	BUF_COUTPUT2	50
	BUF_COUTPUT3	29
	BUF_COUTPUT4	8
	BUF_COUTPUT5	49
	BUF_COUTPUT6	28
Discrete Outputs	BUF_DISCRETE_OUT0	13
	BUF_DISCRETE_OUT1	34
	BUF_DISCRETE_OUT2	55
	BUF_DISCRETE_OUT3	14
	BUF_DISCRETE_OUT4	35
	BUF_DISCRETE_OUT5	56
	BUF_DISCRETE_OUT6	15
	BUF_DISCRETE_OUT7	36

Table 1-2 User External Connector Inputs

Inputs	Signal Name	Pin Number
External Timer Clocks	EXTCLK64	3
	EXTCLK32B	44
	EXTCLK32A	23
	EXTCLK16D	2
	EXTCLK16C	43
	EXTCLK16B	22
	EXTCLK16A	1
External Timer Gates	EXTGATE0	47
	EXTGATE1	26
	EXTGATE2	5
	EXTGATE3	46
	EXTGATE4	25
	EXTGATE5	4
	EXTGATE6	45
External Interrupt	EXT_INTRPT1	27
	EXT_INTRPT2	48
Discrete Inputs	DISCRETE_IN0	10
	DISCRETE_IN1	31
	DISCRETE_IN2	52
	DISCRETE_IN3	11
	DISCRETE_IN4	32
	DISCRETE_IN5	53
	DISCRETE_IN6	12
	DISCRETE_IN7	33
Latch Enable Discrete Port	LA_DISCRETE_INL	21
GROUND	GND	42, 24, 6, 7, 51, 54, 57, 39, 62

1.5.1 Output Descriptions

All outputs driven by the DIGITAL PCI RTC module to the external connector are TTL level signals driven by 74ABT244A devices.

1.5.1.1 Divide-By Clocks

These outputs are copies of the internal divide-by clocks on the DIGITAL PCI RTC module. They are provided for the user to use in their particular application. As an example, the user may wish to synchronize an external piece of hardware to a clock used by the DIGITAL PCI RTC module. These divide-by clocks are free running and are not under software control.

1.5.1.2 Timer Clocks

These outputs are copies of the actual clock used to clock each of the timers. For example, if timer 32B is programmed for SW Clocking (see Section 2.2.5), then the Timer Clock 32B output will assert at the same time as the clock used for the timer.

A potential application for these pins is to allow the user to assert an output under software control by having an unused timer programmed for SW Clocking mode. The application writes the appropriate value to the SW Event Clock Register which will appear on the Timer Clock Output pin.

1.5.1.3 Timer Outputs

These outputs assert if the corresponding timer has it's output enabled in the Output Enable Register and the timer reaches terminal count. These outputs represent the carry out/in function of the respective timer and as such, can be connected to the EXTCLK input of a second timer to permit timer cascading.

Outputs may assert either a level or an edge of either positive or negative polarity. These characteristics are programmable in the Output Mode Register.

1.5.2 Input Descriptions

All signals driven onto the DIGITAL PCI RTC module by external devices must be TTL compatible and at least 50 ns in duration.

Note: The external cable should be properly shielded for signal integrity.

1.5.2.1 External Timer Clocks

The timers are clocked by these inputs when they are programmed in the Clock Divisor Select register for external clocking. The maximum frequency supported on these inputs is 10 MHz.

1.5.2.2 External Timer Gates

The timers are gated by these inputs when they are programmed in the Gate Mode Control register for external clocking.

1.5.2.3 External Interrupts

These inputs allow the user to have an external event post an interrupt over the PCI bus. The external interrupt must be a low-high-low pulse of at least 100 ns.

1.6 Software Considerations

The following sections describe the software considerations.

1.6.1 Terminal Count

The timers reach terminal count when programmed with a value equal to one less than the desired value. For example, if a count value of AAAA is desired in one of the 16-bit timers, then it must be pre-programmed with a value of AAA9.

1.6.2 Interrupt Considerations

The hardware supports interrupt generation from the seven timers plus the two external interrupt inputs. It is possible to have the timer board in a perpetual interrupt request state if the interrupt service routine can not keep pace with the interrupt generating sources. For example, if the application programs a timer to roll over at 200 ns intervals with interrupts enabled then by the time the first interrupt is serviced (which clears posted interrupts), another one is already posted. This may cause undesired results such as apparent system lockup or severe system performance degradation, depending upon the particular system the DIGITAL PCI RTC module is used in.

1.7 User Implementation Responsibilities and Cautions

- The user should account for operating system latency when programming the timers.
- Note that the Real Time Clock does not queue multiple interrupts.
- Please take note of Interrupt Considerations (Section 1.6.2).

1.8 Environmental and Electrical Specifications

The following are the environmental and electrical specifications for the DIGITAL PCI RTC module.

Operating Temperature:	5°C (41°F) to 50°C (122°F)
Operating Humidity:	10% to 95% with maximum wet bulb temperature of 32°C (90°F) and minimum dewpoint of 2°C (36°F)
Non-operating Temperature:	-40°C (-40°F) to 66°C (151°F)
Non-operating Humidity:	Up to 90% relative humidity
Electrical Requirements:	+5 Vdc, 2.5 A

2
Real Time Clock Registers

2.1 Introduction

PCI Interface Chip Registers are located at Base Address Register zero (BAR0). The PCI Interface Chip Register offsets start at the address assigned to the Real Time Clock module by the host system firmware or BIOS. These registers control the AMCC S5933 chip that is used to interface the RTC hardware to the PCI bus. For example, on an AlphaStation 4100 system the Interrupt Control and Status Register can be accessed by adding its offset (0x38) to the value 0x7ffe0000 read from the PCI configuration BAR0. A layout of these registers can be found in Table 2-1. The BAR0 register will contain an address that is assigned to PCI MEMORY space. The prefetch bit in the register is clear. The area pointed to by BAR0 is 40 (hex) bytes in size.

Table 2-1 PCI Interface Chip Registers

Address Offset (hex)	Abbreviation	Register Name
00	OMB1	Outgoing Mailbox Register 1
04	OMB2	Outgoing Mailbox Register 2
08	OMB3	Outgoing Mailbox Register 3
0C	OMB4	Outgoing Mailbox Register 4
10	IMB1	Incoming Mailbox Register 1
14	IMB2	Incoming Mailbox Register 2
18	IMB3	Incoming Mailbox Register 3
1C	IMB4	Incoming Mailbox Register 4
20	FIFO	FIFO Register Port (bidir)
24	MWAR	Master Write Address Register
28	MWTC	Master Write Transfer Count Register
2C	MRAR	Master Read Address Register
30	MRTC	Master Read Transfer Count Register
34	MBEF	Mailbox Empty/Full Status
38	INTCSR	Interrupt Control/Status Register
3C	MCSR	Bus Master Control/Status Register

The Real time Clock Register address is located at Base Address Register one (BAR1). The Real Time Clock Register offsets start at the address assigned to the Real Time Clock by the host system firmware or BIOS. These registers control the clock functionality of the hardware. For example, on an AlphaStation 4100 system the 32B (timer) register can be accessed by adding its offset 0x08 to the value 0x7ffe0000 read from the PCI configuration BAR1. A layout of these registers can be found in Table 2-2. The BAR1 register will contain an address that is assigned to PCI MEMORY space. The prefetch bit in the register is clear. The area pointed to by BAR1 is 40 (hex) bytes in size.

Table 2-2 DIGITAL RTC Register Address Map

Address Offset	Register	Access	
		Read	**Write**
00	64LO	Read 64LO	Load 64LO
04	64HI	Read 64HI	Load 64HI
08	32B	Read 32B	Preload 32B
0C	32A	Read 32A	Preload 32A
10	16DC	Read 16DC	Preload 16DC
14	16BA	Read 64BA	Preload 16BA
18	Discrete I/O	8 Input Points	8 Output Points
1C	N/A		
20	Clock Divisor Select	X	X
24	Gate Mode Control	X	X
28	Gate On/Off	X	X
2C	Output Mode Control	X	X
30	Interrupt Enable Mask Register	X	X
34	Interrupt Reason/Interrupt Overrun Register	X	
38	Count Mode Control Register	X	X
3C	Software Event Clock Register		X

2.2 Register Descriptions

The following sections describe the DIGITAL Real Time Clock registers.

2.2.1 64-Bit Timer - 64LO and 64HI

The 64-bit timer is conceptually divided into two 32-bit timers that are accessible via reads and writes to separate address locations. Data written to either of these addresses will immediately be loaded into that portion of the 64-bit timer. There is no preload register associated with the 64-bit timer.

A read of the 64-bit timer is handled in a special way to account for the fact that two reads of this timer are required to return the full count value. All reads of the lower timer register (64LO) cause the upper bits of the 64-bit timer to be written to a snapshot holding register. The application can then read the upper 32 bits that are stored in the snapshot register (64HI) to make the full 64-bit value. Direct access to the upper 32 bits of the 64-bit timer is not allowed.

64LO	64HI
Read	Read
[31..0] Lower 32 bits of the 64-bit timer	Timer bits [63..32] stored in snapshot register

64LO	64HI
Write	Write
Loads data into bits [31..0] of the 64-bit timer	Loads data into bits [63..32] of the 64-bit timer

2.2.2 Timers 32B and 32A

32B and 32A are two 32-bit timers. Data written to either of these two locations is stored in a 32-bit preload register. Data read from these addresses returns the current value in the 32-bit timer. The timer cannot be directly written to, nor can the preload register be directly read. Data is written to the timer when the preload register is written, or when in repeat mode and termination has occurred.

32B	32A
Read	Read
Returns current 32-bit timer value	Returns current 32-bit timer value

32B	32A
Write	Write
Stores data in timer 32B’s preload register	Stores data in timer 32A’s preload register

2.2.3 Timers 16DC and 16BA

16DC and 16BA are four 16-bit timers that are accessed in pairs. 32 bits of data written to either of these two locations is stored in two of four 16-bit preload registers; 16D and 16C or 16B and 16A. Data read from these addresses returns the current values of a pair, either 16D and 16C, or 16B and 16A, of 16-bit timers. The timers can not be directly written to, nor can the preload registers be directly read. Data is written to the timer when the preload register is written, or when in repeat mode and termination has occurred.

Attempted word accesses to the individual 16-bit counters will produce erroneous results.

There are limitations on the use of the four 16-bit timers (16A, 16B, 16C, and 16D) such that only one 16-bit timer per registered pair may be utilized with the other timers (32A, 32B, and 64).

For example: 16A and (16C or 16D), or 16B and (16C or 16D) may be used, but not both 16A and 16B, or 16C and 16D.

16DC		**16DC**	
Read		Write	
[31..16] 16-bit timer 16D	[15..0] 16-bit timer 16C	[31..16] Preload register 16D	[15..0] Preload register 16C

16BA		**16BA**	
Read		Write	
[31..16] 16-bit timer 16B	[15..0] 16-bit timer 16A	[31..16] Preload register 16B	[15..0] Preload register 16A

2.2.4 Discrete I/O

This port provides 8 discrete inputs and 8 discrete outputs. The bit definition of this register is shown in the table below. The output port can not be read and the input port can not be written.

The input port is configured as a transparent latch and the latch enable is provided on an external pin. User hardware is free to use the latch enable signal to control writes to the latches. In case the user hardware cannot control the latch enable signal, the user can, from an external source, tie the control signal to high which forces the latches transparent and any data present on the input port is available to application software.

Discrete I/O Port	
Read [7:0]	Write [15:8]

2.2.5 Clock Divisor Select Register Description

The clock divisor select register is a read/write register used by software to determine the source of the clock for each of the timers. There are a total of eight possible clock sources for each timer. Six of them are simple “Divide-By” frequencies from a base of 20 MHz (see Figure 2-1).

The seventh clock source allows the user, via an external connection, to provide a clock source. The eighth clock source is via software control. When the Clock Divisor Select Register is programmed to select the SW Clock source and the user writes to the SW Event Clock Register, the timer is clocked. This mode is used as an event counter under complete control of the software.

2.2.5.1 Clock Divisor Select Register

Bits	[31..21]	[20..18]	[17..15]	[14..12]	[11..9]	[8..6]	[5..3]	[2..0]
Timer	NA	64	32B	32A	16D	16C	16B	16A
Default	X	000	000	000	000	000	000	000

Timers 16A, 16B, 16C, 16D, and 32A, 32B, and 64 have the following bit definitions:

Bit Value	Clock Select	Frequency
000	SW Clocking	10 MHz Maximum
001	External Clocking	10 MHz Maximum
010	Divide by 2	10 MHz
011	Divide by 4	5 MHz
100	Divide by 8	2.5 MHz
101	Divide by 16	1.25 MHz
110	Divide by 32	625 kHz
111	Divide by 64	312.5 kHz

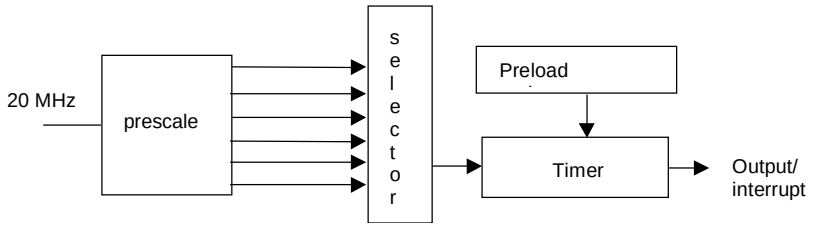
2.2.5.2 Example Clock Rates

The table below shows the time required to roll a full count value for the timers provided on the DIGITAL PCI RTC at the different clock divisor frequencies.

Frequency	Clock Period	16-Bit Timers	32-Bit Timers	64-Bit Timer
10 MHz	100 ns	6.55 ms	7 min 9 sec	58,654 years
5 MHz	200 ns	13.11 ms	14 min 19 sec	117,309 years
2.5 MHz	400 ns	26.21 ms	28 min 38 sec	234,619 years
1.25 MHz	800 ns	52.43 ms	57 min 16 sec	469,240 years
625 kHz	1.6 us	104.86 ms	114 min 32 sec	938,479 years
312.5 kHz	3.2 us	209.72 ms	229 min 4 sec	1,876,958 years

Figure 2-1 shows the frequency prescaler.

Figure 2-1 Frequency Prescaler



Each of the prescaler frequencies is available to each of the counters on the DIGITAL PCI RTC module. You can choose any of the available frequencies and load the timer with any number you like to get the timer value you desire. For example, if you want an interrupt on a 1 usec interval, you can choose the 10 MHz input and, since an n-1 preload value is required, load the timer with 10 - 1 (1001b). If you want a 1 sec interval, you could choose the .3125 MHz input and load the counter with 312500 - 1 (100 1100 0100 1011 0011b).

2.2.6 Gate Mode Control Register

The read/write Gate Mode Control Register is used to select what type of gate is to be used for each of the timers. A timer gate is used on each timer that specifies the conditions when the timer is to start and/or stop timing. This permits a hands-off synchronization method of starting or stopping a timer based on an external event.

The three programmable gate characteristics are:

- Internal or external gate selection
- Positive or negative gate selection
- Level or edge triggered gate

Note: The "*!*" symbol indicates active low signal.

Timer	[22..16]	[14..8]	[6..0]
	Internal / *!External*	Positive / *!Negative*	Level / *!Edge*
64	[16]	[8]	[0]
32B	[17]	[9]	[1]
32A	[18]	[10]	[2]
16D	[19]	[11]	[3]
16C	[20]	[12]	[4]
16B	[21]	[13]	[5]
16A	[22]	[14]	[6]

2.2.6.1 Internal Gate

Internal gate control is essentially a software gate. This is useful when an event counter is desired. For example, a timer that is programmed with an internal gate might be used in conjunction with the SW Clocking. When software desires, it can then update the SW Event Clock Register causing the timer (counter in this case) to count that event. (The notion of positive/negative and level/edge triggers does not apply when in the Internal Gate Mode.)

2.2.6.2 External Gate

Each timer has an external pin that can be used to control the gate of the timer. There are four modes of external gating:

- Positive Level
- Negative Level
- Positive Edge
- Negative Edge.

2.2.6.3 External Level Gate

When programmed for level gating, the timer is permitted to count while the level is true. For example, when programmed for Positive Level and the external gate input is driven high (+5 Vdc), the timer counts. This counting continues until the external gate is driven low (GND). Similarly, when in the negative level gate mode, the timer counts when the external gate input is driven low (GND) and continues counting until the input is driven high.

2.2.6.4 External Edge Gate

When programmed for edge gating, the timer starts counting when the edge is detected. For example, the timer starts counting when a rising edge is detected if programmed for a positive edge gate. Counting continues even if a second edge is presented at the external gate input. Subsequent edges have no effect on the gate; it remains on until the Gate On/Off bit is reset by software.

NOTE: To inhibit further counting once triggered, the application software ***must*** turn off the gate in the CSR Gate On/Off register.

2.2.7 Gate On/Off

This read/write register is used as the main gate on/gate off control for each of the timers. If a timer has it's gate programmed in the off state, that timer can not count even if a valid gate trigger occurs (see Section 2.2.6.2). The on/off bits can also be used to override a valid gate event and cause the timer to stop counting.

The Gate On/Off register can also be thought of as a GATE RETRIGGER register when a given timer is programmed for the count once mode in the Count Mode Register. After the timer hits terminal count in the count once mode, it's corresponding bit in the Gate On/Off register must be written back to zero before the timer can be used again.

Timer	Bits [6..0]
	On / *!Off*
64	[0]
32B	[1]
32A	[2]
16D	[3]
16C	[4]
16B	[5]
16A	[6]

2.2.8 Output Mode Control Register

The output mode control register is used to control the output mode of each timer. An output of a timer asserts, if enabled, when the timer reaches terminal count. Terminal count for a timer programmed to count up is reached when it hits the maximum count value. When programmed to count down, terminal count is reached when the timer hits zero. The timers reach terminal count when programmed with a value equal to one less than the desired value. For example, if a count value of AAAA is desired in one of the 16-bit timers, then it must be pre-programmed with a value of AAA9.

Three fields are provided for each timer's output mode control:

- On / !Off
- Positive / !Negative
- Level / !Edge

There are four programmable modes for each timer output:

- Positive Level
- Negative Level
- Positive Edge
- Negative Edge

When selected for the level output mode, the output will assert when the timer reaches terminal count. Assertion level is determined by the state of the positive/negative bit. The output will remain asserted for one terminal count, de-asserted for the following terminal count, and so forth.

When selected for the edge output mode, the output will assert a 50 ns pulse whose polarity is determined by the state of the positive/negative bit when the timer reaches terminal count. Each subsequent terminal count causes the output to pulse. This pulse can be used as a clock input to a second timer programmed for external clocking.

Timer	**[22..16]**	**[14..8]**	**[6..0]**
	On / *!Off*	Positive / *!Negative*	Level / *!Edge*
64	[16]	[8]	[0]
32B	[17]	[9]	[1]
32A	[18]	[10]	[2]
16D	[19]	[11]	[3]
16C	[20]	[12]	[4]
16B	[21]	[13]	[5]
16A	[22]	[14]	[6]

2.2.9 Interrupt Enable Mask Register

Interrupt Source	Bits [8..0]
	Enable / *!Disable*
External Interrupt 2	[8]
External Interrupt 1	[7]
Timer 16A	[6]
Timer 16B	[5]
Timer 16C	[4]
Timer 16D	[3]
Timer 32A	[2]
Timer 32B	[1]
Timer 64	[0]

2.2.10 Interrupt Reason/Interrupt Overrun Register

The Interrupt Reason/Overrun Register is a 32-bit read only register with the bits defined in the following table. Undefined bits are read as zeros.

Interrupts, when enabled, come from two types of sources; timers and external inputs. The Interrupt Reason/Overrun Register is provided to record these interrupts. The overrun portion of this register records a second interrupt from a given source if the first interrupt did not get serviced by the time the second one asserts.

Although it is possible for this register to record several interrupts, only the first one that asserts is reported to the AMCC S5933 device. Subsequent interrupting events are stored in the Interrupt Reason/Overrun Register for application software.

The interrupt service routine (ISR) must perform a read of the INTCSR PCI operation register in the AMCC S5933 device to determine that the RTC is the interrupting device. The ISR then writes back to the INTCSR to clear the interrupt from the PCI bus. After these two events have happened, the ISR must read the Interrupt Reason/Overrun Register of the RTC to determine what caused the interrupt. Reading the reason register clears all posted interrupts and overrun bits.

Interrupt Source	Bits [24..16]	Bits [8..0]
	Overrun/!No Overrun	Interrupt/!No Interrupt
External Interrupt 2	[24]	[8]
External Interrupt 1	[23]	[7]
Timer 16A	[22]	[6]
Timer 16B	[21]	[5]
Timer 16C	[20]	[4]
Timer 16D	[19]	[3]
Timer 32A	[18]	[2]
Timer 32B	[17]	[1]
Timer 64	[16]	[0]

2.2.11 Count Mode Control Register

The Count Mode Control Register is used to select count up or count down and repeat count or count just once. Default values are count down and count once.

When programmed for the repeat count mode, all timers except the 64-bit timer are reloaded with the count value stored in the preload register. The 64-bit timer, because it does not have this preload register, will simply wrap it's count value and continue counting.

Timer	Bits [14..8]	Bits [6..0]
	Down/*!Up*	Repeat/*!Count Once*
Timer 64	[8]	[0]
Timer 32B	[9]	[1]
Timer 32A	[10]	[2]
Timer 16D	[11]	[3]
Timer 16C	[12]	[4]
Timer 16B	[13]	[5]
Timer 16A	[14]	[6]

2.2.12 Software Event Clock Register

When the Clock Divisor Select Register is programmed for SW Clocking mode, a write to this write only register will cause the corresponding timer to count once. This provides a software event count mechanism.

Timer	Bits [6..0]
	Clock/*!No Clock*
Timer 16A	[6]
Timer 16B	[5]
Timer 16C	[4]
Timer 16D	[3]
Timer 32A	[2]
Timer 32B	[1]
Timer 64	[0]

3
Hardware Installation

3.1 Installation Procedures

To install the DIGITAL PCI RTC module, perform the following steps:

Caution

Static electricity can damage sensitive electronic components. When handling the DIGITAL PCI RTC module, use an antistatic wriststrap that is connected to a grounded surface on your computer system.

1. Perform a normal power down of your computer system and disconnect the power cable.
2. Remove the cover from your computer (refer to your system documentation).
3. Put on an antistatic wriststrap.

Caution

An antistatic wriststrap must be worn when handling any module to prevent damage to the module.

4. Select an empty PCI expansion slot and remove the screw that secures the slot bulkhead cover plate.
5. Grasp the top edge of the module and carefully insert it into the slot, then firmly seat it.
6. Secure the module to the bulkhead with the screw removed in step 4.
7. Replace the computer cover.
8. If applicable, connect the site specific cable.

Perform a normal power up of your computer system.

4
PCI Real Time Clock Device Driver

4.1 Introduction

The DIGITAL PCI RTC module is a single card that occupies one PCI slot. It requires two blocks of I/O space, one is 256 bytes and the other is 64 bytes. It uses one interrupt line.

Although the DIGITAL PCI Real Time Clock provides users several methods of timing events in a PCI based computer system, this version of the PCI Real Time Clock Device Driver provides access to a limited subset of these methods. Likewise, although there are several clocks available, the driver only uses the 64 bit timer for read access. The driver source, which is shipped with the kit, should provide a useful starting point for developers to add access to the remaining methods and clocks.

This driver has been written to interface with the DIGITAL PCI RTC module

The DIGITAL PCI Real Time Clock Device Driver is currently supported on the AlphaServer 2100A, 4100, 8200, and 8400 running DIGITAL UNIX V4.0.

4.2 Device Driver Installation

The DIGITAL PCI Real Time Clock Device Driver is installed by using the DIGITAL UNIX *setld* command. The DIGITAL PCI Real Time Clock Device Driver subset will be delivered on a 3.5 " floppy disk.

Example 4-1 is a sample installation on a system named LINDEN1. It is assumed that the device driver floppy that contains the kit is in a local directory. User responses in the example are shown in **bold type**.

Example 4-1 Device Driver Installation

```
root@linden1 {1} mount /dev/fd0c /mnt
root@linden1 {2} setld -l /mnt/output

The subsets listed below are optional:

     There may be more optional subsets than can be presented on a single
     screen. If this is the case, you can choose subsets screen by screen
     or all at once on the last screen. All of the choices you make will
     be collected for your confirmation before any subsets are installed.

     1) DIGITAL UNIX V4.0 based PCI Real Time Clock Device Driver


Or you may choose one of the following options:

     2) ALL of the above
     3) CANCEL selections and redisplay menus
     4) EXIT without installing any subsets

Enter your choices or press RETURN to redisplay menus.

Choices (for example, 1 2 4-6): 1

You are installing the following optional subsets:

        DIGITAL UNIX V4.0 based PCI Real Time Clock Device Driver


Is this correct? (y/n): y

Checking file system space required to install selected subsets:

File system space checked OK.

1 subset(s) will be installed.

Loading 1 of 1 subset(s)....


DIGITAL UNIX V4.0 based PCI Real Time Clock Device Driver
   Copying from /mnt/output (disk)
   Verifying

1 of 1 subset(s) installed successfully.

root@linden1 {3}
```

At this point it is necessary for the driver to be either built statically into the kernel or loaded dynamically. The directions for this are contained in Example 4-2.

Example 4-2 /usr/sys/io/RTC/rtc_build file

```
Building and Linking the 'rtc' Driver

-------------------------------------------------------

    Building rtc.mod

        One time setup:

            cd /usr/sys/conf
            echo '/usr/sys/io/RTC:' >> BINARY.list
            ./sourceconfig BINARY

        You compile (build) your driver with these commands:

            cd /usr/sys/BINARY
            make rtc.mod

-------------------------------------------------------

        The following steps require that the driver has been successfully
        built into rtc.mod

-------------------------------------------------------

    Dynamically linking rtc.mod into the kernel

        One time setup:

            cd /var/subsys/
            ln -s /subsys/device.mth rtc.mth
            ln -s /usr/sys/BINARY/rtc.mod rtc.mod

        Everytime you want to dynamically load the driver:

            sysconfig -v -c rtc

        If you wish to unload the dynamically loaded driver:

            sysconfig -u rtc

-------------------------------------------------------

    Statically linking rtc.mod into the kernel

        One time setup:

            cd /usr/sys/conf
            echo '/usr/sys/io/RTC:' >> HOSTNAME.list
            doconfig -c HOSTNAME

        The new kernel resides in /usr/sys/HOSTNAME.  Copy it to the
        root directory and shutdown.  It will be necessary to cycle the
        power off and then back on the first time you boot the rtc
        kernel.
```

Note: If debug printf's to the console are desired, then change the value of the RTC_Developer_Debug variable to 1. Using a value greater than 1 will not effect the output since it is a true or false check that is done for print output.

```
root@linden1 {1} sysconfig -r rtc RTC_Developer_Debug=1
```

4.3 Testing the Module after Installation.

To verify that the hardware and device driver are installed correctly, there is a script provided in the `/usr/examples/RTC` directory. This script evokes tests that check each timer on the Real Time Clock module and verifies the modes that the clock supports.

To start the test:

```
root@linden1 {1} rtc_test

Installation verification test for the RT clock is complete.

Mon Jun 22 09:39:22 EDT 1998
```

If there is an error condition, check that the driver is installed and that the device is recognized by the system.

5
Driver Interfaces

5.1 RTC Driver Interfaces

The following sections describe the RTC driver interfaces.

5.1.1 open()

The **open()** function establishes a connection between the DIGITAL PCI Real Time Clock device named by the path parameter and a file descriptor. The opened file descriptor is used by subsequent I/O functions, such as **read()**, **write()**, and **ioctl()**.

SYNTAX

```
#include "rtcreg.h"
#include <sys/fcntl.h>

int open (path, oflag, mode );
const char      *path;
int             oflag;
mode_t  mode;
```

PARAMETERS

path	"/dev/rtc"
oflag	O_RDWR
mode	This parameter is not used.

EXAMPLE

```
int     fd;
if ((fd = open("/dev/rtc", O_RDWR, 0)) < 0) {
        perror ("open");
        exit(1);
}
```

RETURN VALUES

Upon successful completion, the **open()** function returns the file descriptor, a nonnegative integer. Otherwise, a value of -1 is returned and errno is set to indicate the error. Please see the **open()** man page for a description of the possible values for errno.

5.1.2 close()

This system call is used by an application to release exclusive control of the DIGITAL PCI Real Time Clock.

SYNTAX

```
int close (fd);
int     fd;
```

PARAMETERS

fd A valid open file descriptor as returned by **open()**.

EXAMPLE

```
int     fd;

if (close(fd)) {
        perror (“close”);
        exit(1);
}
```

RETURN VALUES

Upon successful completion, the **close()** function returns a value of 0 (zero). Otherwise, a value of -1 is returned and errno is set to indicate the error. Please see the **close()** man page for a description of the possible values for errno.

5.1.3 ioctl()

The **ioctl()** system call performs device specific actions on or for a device. The action is determined by the *request* parameter. The following sections will describe each of the requests and provide examples of their usage.

Each **ioctl()** system call will have the same general syntax. The differences of the specific requests will be discussed in the following sections.

SYNTAX

```
#include <sys/ioctl.h>
#include "rtcreg.h"

int ioctl (fd, request, arg)
int             fd;
unsigned long   request;
void            *arg;
```

PARAMETERS

fd A valid open file descriptor as returned by **open()**.

request The ioctl command to be performed on the device.

RTC_GETSLOT
RTC_GETPCICFG
RTC_GETBASEADDR
RTC_GETBASEADDR1
RTC_READ_TIMER
RTC_START_TIMER
RTC_STOP_TIMER
RTC_TEST
RTC_REG_DUMP
RTC_CHECK_INT
RTC_INT_COUNT

arg The parameters for this request.

EXAMPLE

```
if (ioctl (fd, RTC_CHECK_INT, 0)) {
        perror ("RTC_CHECK_INT");
        exit(1);
}
```

RETURN VALUES

Upon successful completion, the **ioctl()** function returns a value of 0 (zero). Otherwise, a value of -1 is returned and errno is set to indicate the error. Please see the **ioctl()** man page for a description of the possible values for errno.

5.1.3.1 RTC_GETSLOT

Using RTC_GETSLOT as the request field of the ioctl function will return the PCI slot number that the device resides in.

```
int     slot_num;

if (ioctl (fd, RTC_GETSLOT, &slot_num)) {
        perror ("RTC_GETSLOT");
        exit (1);
}
printf ("the slot number is %d\n", slot_num);
```

5.1.3.2 RTC_GETPCICFG

Using RTC_GETPCICFG as the request field of the ioctl function will return a copy of the contents of the PCI configuration header of the Real Time Clock device.

```
#include<io/dec/pci/pci.h>

struct pci_config_hdr       hdr;

if (ioctl (fd, RTC_GETPCICFG, &hdr)) {
        perror ("RTC_GETPCICFG");
        exit (1);
}
printf ("rtc bar0 = 0x%x,\tbar1 = 0x%x\n", hdr.bar0, hdr.bar1);
```

5.1.3.3 RTC_GETBASEADDR

Using RTC_GETBASEADDR as the request field of the ioctl function will return a copy of the contents of the base address of the AMCC PCI interface and it's registers. This address provides access for enabling and disabling the PCI interrupt on the device. The driver enables interrupts and provides an interrupt handler when loaded so this request is for driver development and not meant for general use.

```
io_handle_t amcc_base;

if (ioctl (fd, RTC_GETBASEADDR, &amcc_base)) {
        perror ("RTC_GETBASEADDR");
        exit (1);
}
```

5.1.3.4 RTC_GETBASEADDR1

Using RTC_GETBASEADDR1 as the request field of the ioctl function will return a copy of the contents of the base address of the Real Time Clock registers. This address is the base offset to access the registers in the Real Time Clock Registers Address Map (see Chapter 2).

```
io_handle_t rtc_base;

if (ioctl (fd, RTC_GETBASEADDR1, &rtc_base)) {
        perror ("RTC_GETBASEADDR1");
        exit (1);
}
```

5.1.3.5 RTC_START_TIMER

Using the RTC_START_TIMER as the request field of the ioctl function will initialize the registers used to enable the 64-bit timer. The timer is enabled using the count up mode and will repeat on roll over. The frequency of the count is 1.25 MHz, or each clock period is equal to 800 ns. This timer will not interrupt on completion.

```
if (ioctl (fd, RTC_START_TIMER)) {
        perror ("RTC_START_TIMER");
        exit (1);
}
```

5.1.3.6 RTC_STOP_TIMER

Using the RTC_STOP_TIMER as the request field of the ioctl function will stop the 64-bit timer by clearing the gate enable bit for this register.

```
if (ioctl (fd, RTC_STOP_TIMER)) {
        perror ("RTC_STOP_TIMER");
        exit (1);
}
```

5.1.3.7 RTC_READ_TIMER

Using RTC_READ_TIMER as the request field of the ioctl will read the 64-bit timer. By passing a pointer of type rtc_reg_data_t to this ioctl, the upper and lower 32 bits of the 64-bit timer can be read.

```
unsigned long timer_lo, timer_hi;

rtc_reg_data_t regptr;

if (ioctl (fd, RTC_TEST, 0)) {
        perror ("RTC_TEST");
        exit (1);
}

time_lo = regptr.data[0];  /* store the lower 32 bits of the 64-bit timer */

time_hi = regptr.data[1];  /* store the upper 32 bits of the 64-bit timer */
```

5.1.3.8 RTC_REG_DUMP

Using RTC_REG_DUMP as the request field of the ioctl will cause the values of the DIGITAL PCI Real Time Clock's CSRs to be printed to the system console.

```
rtc_reg_data_t     regs;

if (ioctl (fd, RTC_REG_DUMP, 0)) {
        perror ("RTC_REG_DUMP");
        exit (1);
}
```

5.1.3.9 RTC_CHECK_INT

Using the RTC_CHECK_INT as the request field of the ioctl will return the value of the interrupt status register.

```
unsigned int interrupt_status;

if (ioctl (fd, RTC_CHECK_INT, &interrupt_status)) {
        perror ("RTC_CHECK_INT");
        exit (1);
}
```

5.1.3.10 RTC_INT_COUNT

Using the RTC_INT_COUNT as the request field of the ioctl will return the number of interrupts that have occurred. This call will also clear the current interrupt count after the current count has been returned to the user.

```
unsigned int interrupt_status;

if (ioctl (fd, RTC_CHECK_INT, &interrupt_status)) {
        perror (“RTC_CHECK_INT”);
        exit (1);
}
```

5.1.3.11 RTC_WRITE_LW

Using the RTC_WRITE_LW as the request field of the ioctl will write 32 bits of data to the specified address offset. This call must be used in conjunction with RTC_GETBASEADDR1. The base offset of the Real Time Clock Registers must be added to the offset of the timer to form the correct address for this call.

```
rtc_reg_data_t outport;

unsigned long clock_data, timer_reg_addr;

  outport.reg[0] = timer_reg_addr;  /* address offset of the timer plus the base offset */

  outport.data[0] = clock_data;  /* data to be written to the timer */

  if ((err = ioctl(rtc_fd, RTC_WRITE_LW, &outport)) < 0)

          perror("RTC_WRITE_LW"),exit(1);
```

5.1.3.12 RTC_READ_LW

Using the RTC_READ_LW as the request field of the ioctl will read 32 bits of data to the specified address offset. This call must be used in conjunction with RTC_GETBASEADDR1. The base offset of the Real Time Clock Registers must be added to the offset of the timer to form the correct address for this call.

```
rtc_reg_data_t inport;

unsigned long clock_data, timer_reg_addr;

   inport.reg[0] = timer_reg_addr;  /* address offset of the timer plus the base offset */

   if ((err = ioctl(rtc_fd, RTC_READ_LW, &inport)) < 0)

          perror("RTC_READ_LW"),exit(1);

   clock_data = inport.data[0];
```

5.1.4 mmap()

Note: Since the RTC PCI interface does not support PCI burst transactions, care should be taken to use memory barriers (MB) after each read or write.

The rtcmmap routine is invoked by the kernel as a result of an application calling the mmap(2) system call. The rtcmmap routine makes sure that the specified offset into the memory mapped device's memory is valid.

The routine returns the page frame number corresponding to the page at the specified offset.

SYNTAX

void *mmap (void *addr, size_t len, int prot, int flags, int filedes, off_t off);

PARAMETERS

addr Specifies the starting address of the new region (truncated to a page boundary).
l*en* Specifies the length in bytes of the new region (rounded up to a page boundary).
p*rot* Specifies access permissions as either PROT_NONE or the result of a logical OR operation on any combination of PROT_READ, PROT_WRITE, and PROT_EXEC.
flags Specifies attributes of the mapped region as the results of a bitwise-inclusive OR operation on any combination of MAP_FILE, MAP_ANONYMOUS, MAP_VARIABLE, MAP_FIXED, MAP_SHARED, MAP_PRIVATE, MAP_INHERIT, or MAP_UNALIGNED.
filedes Specifies the file to be mapped to the new mapped file region returned by **open()**.
off Specifies the offset into the file that gets mapped at address addr.

EXAMPLE

```
volatile caddr_t mem;

if ((mem = mmap(0,PAGESIZE,PROT_WRITE,MAP_SHARED,rtc_fd,0)) == (void*) -1)
        perror("mmap failed"), exit(1);
```

RETURN VALUES

Upon successful completion, the **mmap()** function returns the page frame number corresponding to the page at the specified offset. Otherwise, a value of -1 is returned and errno is set to indicate the error. Please see the **mmap()** man page for a description of the possible values for errno.

5.1.4.1 Using the Page Frame Number to Access Data

The PCI memory mapping on the Alpha architecture is performed through regions called Dense memory space or Sparse memory space. The RTC module is designed to use Sparse memory space because it cannot decode PCI burst cycles. The side effect of using Sparse memory is that some extra manipulation of the data is needed to retrieve the expected data at the expected offsets. To help with this manipulation, the Real Time Clock example code provides macros for integer (32 bit) reads and writes. The value stored in **base** is the page frame number and **offset** would be the offset for one of the RT clock registers.

```
/* Store Integer Macro */

#define sti(base, offset, data) { \
   unsigned int *va; \
   unsigned long write_data = \
      (unsigned long) data << (((u_long)offset & 3) << 3); \
   va = (unsigned int *) ((((u_long)offset) << 5) + \
```

```
        ((u_long)base ) | 0x18); \
    *(int *)va = write_data; \
    (void)asm("mb"); }

/* Load Integer Macro */
#define ldi(base, offset, dest) { \
    unsigned int *va; \
    unsigned long data=0; \
    va = (unsigned int *) ((((u_long)offset) << 5) + \
        ((u_long)base ) | 0x18); \
    data = (unsigned long) *(int *)va; \
    read_shift(((u_long)offset & 3), data, 4); \
    dest = (unsigned int) data; }

/* Read Shift macro required for ldi */
#define read_shift(address,data,width) \
    { data = ( (data >> ((address & 3) << 3)) & \
            ( (1 << ((8*width)-1)) + ((1L << ((8*width)-1L))-1L) ) ); }
```

A
Software Structures for C Language

The following sections describe the DIGITAL Real Time Clock register layout using software structures for the C programming language.

struct rtc_reg_data_t

This definition may be found in rtcreg.h. It is a parameter to the RTC_REG_DUMP ioctl request.

```
typedef struct {
    int reg[16];
    int data[16];
} rtc_reg_data_t;
```

struct RTC_IO

This structure is used by the RTC_START_TIMER ioctl request to pass the necessary information to the driver to start the clock.

```
struct RTC_IO {
   struct RTC   rtc_ptr;
   unsigned int timer_status;
};
```

struct RTC

This structure represents the RTC register layout. This definition may be found in rtcreg.h. Addressing may be system dependent. Current layout is for alpha machines. The hardware description of this register layout may be found in Section 2.1. The layout and definition of each of the registers may be found in Section 2.2.

```
struct RTC {

   Timer timer;              /* Timer                          */
   unsigned int reserved_0;  /* reserved field                  */
   unsigned int reserved_1;  /* reserved field                 */
   Clk_sel clk_sel;          /* Clock divisor select           */
   Gate_m gate_m;            /* Gate mode                      */
   Gate_enable gate_enable;  /* Gate control (on/off)          */
   Output_m output_m;        /* Output mode                    */
   unsigned int int_enable;  /* Interrupt enable mask reg      */
   unsigned int int_reason;  /* Interrupt reason/overrun reg   */
   Count_m count_m;          /* Count mode                     */
   unsigned int sw_clock;    /* SW Event Clock register        */
   };
```

struct Timer

This definition may also be found in rtcreg.h. It is a reflection of the first six entries of the DIGITAL PCI Real Time Clock registers as defined in Section 2.2. These six entries hold seven timers. The 64-bit timer takes up two 32-bit locations, and there are two 16-bit timers in the last two entries.

```
typedef struct {

   unsigned int out_64lo;        /* */
   unsigned int out_64hi;        /* */
   unsigned int out_32b;         /* */
   unsigned int out_32a;         /* */
   unsigned short out_16b;       /* */
   unsigned short out_16a;       /* */
   unsigned short out_16d;       /* */
   unsigned short out_16c;       /* */
   } Timer;
```

struct Clk_sel

This definition may also be found in rtcreg.h. It represents the Clock Divisor Register. Refer to Section 2.2.5 for the field definitions.

```
typedef struct {          /* Clock Divisor Select Register layout */
   unsigned int cs_16a :3,
               cs_16b :3,
               cs_16c :3,
               cs_16d :3,
               cs_32a :3,
               cs_32b :3,
               cs_64  :3;
   } Clk_sel;
```

struct Gate_m

This definition may also be found in rtcreg.h. It represents the Gate Mode Control register. A definition of the bit fields may be found in Section 2.2.6.

```
typedef struct {          /* layout of the Gate Mode Register */
   unsigned int     trig    :7,  /* level/edge */
             unused1 :1,
             pulse   :7,  /* positive/negative */
             unused2 :1,
             enable  :7,  /* internal/external */
             unused3 :9;
   } Gate_m;
```

struct Gate_enable

This definition may also be found in rtcreg.h. It represents the layout of the Gate On/Off Register. A discussion of it's fields may be found in Section 2.2.7.

```
typedef struct {          /* layout for the Gate On/Off register */
   unsigned int ge_16a :1,
                ge_16b :1,
                ge_16c :1,
                ge_16d :1,
                ge_32a :1,
                ge_32b :1,
                ge_64  :1,
                unused_3  :25;
   } Gate_enable;
```

struct Output_m

This definition may also be found in rtcreg.h. It represents the layout of Output Mode Control Register. A description of the fields of this register may be found in Section 2.2.8.

```
typedef struct {          /* layout of the Output Mode Register */
   unsigned int     trig    :7,  /* level/edge */
                unused1 :1,
                pulse   :7,  /* positive/negative */
                unused2 :1,
                enable  :7,  /* on/off */
                unused3 :9;
   } Output_m;
```

struct Count_m

This definition may also be found in rtcreg.h. It represents the Count Mode Control Register. A discussion of it's fields may be found in Section 2.2.11.

```
typedef struct {             /* layout of the Count Mode Control Register */
   unsigned int       down_up     :7,       /* down/up */
                unused1     :1,
                repeat_once :7,       /* repeat/count once */
                unused2     :17;
   } Count_m;
```

B
rtc_ivp_def.h

The following is a listing of the rtc_ivp_def.h file:

```
/* File:        rtc_ivp_def.h
*
*/

/* LOCAL SYMBOLS and Macro definitions
*/

/* Diag Test option definitions */
#define NUM_QUALS       4       /* Number of Options      */
#define QUP     NUM_QUALS       /* Default timers to count up */
#define QDD     (NUM_QUALS+1)   /* Display test data to console terminal */
#define QVAL    (NUM_QUALS+2)   /* Specify Timer preload value */
#define QTMR    (NUM_QUALS+3)   /* Specify Timer to test */
#define QDLY    (NUM_QUALS+4)   /* Specify delay value for after counter starts to
interrupt rcv'd */
#define QCLK    (NUM_QUALS+5)   /* Specify clock value that the timer will run off of. */
#define MORE_QUALS 7

struct QSTRUCT {
   char id [32];
   int present;
   int val_type;
   union  {                     /* union of values  */
      char *string;
      int integer;
      } value;
   };

struct GLOBALS_STRUCT {
   /* Required Global data for ALL Sable Std IO Diag's   */
   struct QSTRUCT qual[NUM_QUALS + MORE_QUALS];
   unsigned long return_msg;
};

#define msg_success 0

/*
 * Structures
 */
typedef struct {
   volatile unsigned long reg[16];
```

```
    volatile unsigned int data[16];
} rtc_reg_data_t;

/*
 * IOCTL commands
 */
#define RTC_GETSLOT             _IOR('q', 0, int)
#define RTC_GETPCICFG           _IOR('q', 1, pci_config_hdr_t)
#define RTC_START_TIMER         _IOWR('q', 2, struct RTC)
#define RTC_STOP_TIMER          _IOWR('q', 3, struct RTC)
#define RTC_READ_TIMER          _IOWR('q', 4, rtc_reg_data_t)
#define RTC_REG_DUMP            _IOWR('q', 5, rtc_reg_data_t)
#define RTC_CHECK_INT           _IOR('q', 6, int)
#define RTC_GETBASEADDR         _IOR('q', 7, io_handle_t)
#define RTC_GETBASEADDR1        _IOR('q', 8, io_handle_t)
#define RTC_INT_COUNT           _IOR('q', 9, int)
#define RTC_READ_LW             _IOWR('q', 10, rtc_reg_data_t)
#define RTC_READ_W              _IOWR('q', 11, rtc_reg_data_t)
#define RTC_WRITE_LW            _IOWR('q', 12, rtc_reg_data_t)
#define RTC_WRITE_W             _IOWR('q', 13, rtc_reg_data_t)

/* S5933 Register offsets */
#define IMB4                0x1c
#define MBEF                0x34
#define INTCSR              0x38
#define MCSR                0x3c
#define MCSR_NVDATA(MCSR + 2) /* Data in byte 2 */
#define MCSR_NVCMD  (MCSR + 3) /* Command in byte 3 */

#define PCI_ID              0x40
#define PCI_CLCD            0x48
#define PCI_BAR0            0x50
#define PCI_INT             0x7d

/* S5933 Bit definitions */
#define MB_INTSTATUS            0x00820000  /* Int Status - Inbnd mailbox Int */
#define MB_INTENABLE            0x00001f00  /* Enable Int - Inbnd mailbox 4, byte 3 */
#define PCI_CLCD_DATA           0x08030001  /* Class - base system peripheral, rev 01 */
#define PCI_ID_DATA             0x001e1011  /* Vendor and Device ID */
#define PCI_BAR0_DATA           0x10e8ffc0  /* PCI address mask */
#define PCI_INT_DATA            0x01        /* INTA */

/* RT Clock - addon interface registers */
#define   OUT_64LO      0x0
#define  OUT_64HI       0x4
#define  OUT_32B        0x8
#define  OUT_32A        0xc
#define  OUT_16D        0x12
#define  OUT_16C        0x10
#define  OUT_16B        0x16
#define  OUT_16A        0x14
#define  UNUSED_1       0x18
#define  UNUSED_2       0x1c
#define  CLOCK_DIV      0x20
#define  GATE_MODE      0x24
#define  GATE_ON_OFF    0x28
#define  OUTPUT_MODE    0x2c
```

```
#define  INT_ENABLE     0x30
#define  INT_REASON    0x34
#define  COUNT_MODE     0x38
#define  SW_CLK         0x3c


/* RT Clock - addon interface bit definitions */
#define INT_REASON_OVERRUN  0x01ff0000
#define INT_REASON_MASK          0xff


/* Current definition of the timers
 */
typedef struct {
   unsigned int out_64lo;              /* */
   unsigned int out_64hi;              /* */
   unsigned int out_32b;               /* */
   unsigned int out_32a;               /* */
   unsigned short out_16b;             /* */
   unsigned short out_16a;             /* */
   unsigned short out_16d;             /* */
   unsigned short out_16c;             /* */
   } Timer;

typedef struct {
   unsigned int cs_16a :3,
                cs_16b :3,
                cs_16c :3,
                cs_16d :3,
                cs_32a :3,
                cs_32b :3,
                cs_64  :3;
   } Clk_sel;

#define SW_E_CLK 0
#define EXT_CLK 1
#define DIV_BY_2 2
#define DIV_BY_4 3
#define DIV_BY_8 4
#define DIV_BY_16 5
#define DIV_BY_32 6
#define DIV_BY_64 7

typedef struct {
   unsigned char trig;
   unsigned char pulse;
   unsigned char enable;
   unsigned char unused_0;
   } Gate_m;

typedef struct {
   unsigned char trig;
   unsigned char pulse;
   unsigned char enable;
   unsigned char unused_1;
   } Output_m;
```

```
typedef struct {
  unsigned char down_up;
  unsigned char repeat_once;
  unsigned short unused_2;
  } Count_m;

/* The following definitions are used to enable all the control features
 * Used on gate mode, output mode and count mode.
 */
#define t_16A_ON        0x1
#define t_16B_ON        0x2
#define t_16C_ON        0x4
#define t_16D_ON        0x8
#define t_32A_ON        0x10
#define t_32B_ON        0x20
#define t_64_ON         0x40

#define NEG_EDGE_PULSE   0x000
#define NEG_LEVEL_PULSE  0x001
#define POS_EDGE_PULSE   0x010
#define POS_LEVEL_PULSE  0x011

typedef struct {
  unsigned int ge_16a :1,
               ge_16b :1,
               ge_16c :1,
               ge_16d :1,
               ge_32a :1,
               ge_32b :1,
               ge_64  :1,
               unused_3  :25;
  } Gate_enable;

/* This structure represents the RT clock register layout.  Addressing may be system
dependant */
struct RTC {
  Timer timer;                  /* Timer                    */
  unsigned int reserved_0;      /* reserved field           */
  unsigned int reserved_1;      /* reserved field           */
  Clk_sel clk_sel;              /* Clock divisor select     */
  Gate_m gate_m;                /* Gate mode                */
  Gate_enable gate_enable;      /* Gate control (on/off)    */
  Output_m output_m;            /* Output mode              */
  unsigned int int_enable; /* Interrupt enable mask reg     */
  unsigned int int_reason;    /* Interrupt reason/overrun reg */
  Count_m count_m;              /* Count mode               */
  unsigned int sw_clock;      /* SW Event Clock register    */
  };

struct RTC_IO {
  struct RTC rtc_ptr;
  unsigned int timer_status;
  };
```

C
rtc_ivp.c

The following is a listing of the rtc_ivp.c file:

```
/*+
 * file:    rtc_ivp.c
 *
 *
 *  MODULE DESCRIPTION:
 *
 *      This module contains the installation verification routines for the
 *          RT Clock device which uses the AMCC S5933 PCI controller as the PCI
 *          bus interface.
 *
 * Authors:
 *
 *      dch
 *
 * Creation Date:
 *
 *      21-Feb-1998
 *
 * Modifications:
 *
 *
-*/
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/file.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <c_asm.h>
#include <io/common/devdriver.h>
#include <signal.h>
#include <sys/timers.h>
#include "rtc_ivp_def.h"
```

```
/* EXTERNAL and LOCAL DEFINES
*/
#define PAGESIZE 8192

/* Since we don't have better granularity to compare our clock interrupt
 * timing, we will use a 1 second delay to wait and check for interrupts.
 */
void rtc_micro_delay(int seconds);
int getclock(int clock_id, struct timespec *tp);

#define krn$_micro_delay(t) rtc_micro_delay(3)

/* These will be filled in by the device driver */
unsigned long      RTC_mem_base[1];
unsigned long      RTC_mem_base1[1];

/*
 * local declaration
 */
static int debug_t6 = 0;
static int debug_flag = 0;
struct GLOBALS_STRUCT *eptr;
char *commandLine;
static int rtc_fd;
static int err;
extern const char *readme[];

/*
/*  Forward Reference/Prototypes Declarations
*/

u_long   S5933_t1();
u_long   S5933_t4();
u_long   S5933_t5();
u_long   S5933_t6();

int timer_isr ();
void int_setup();
void check_and_clear (unsigned int gate, int test_num);
void start_gate (volatile unsigned int gate);
void clear_gate ();
void timer_setup(volatile unsigned int timer, volatile unsigned int timer_v, volatile unsigned
int
timer_v_64hi, volatile unsigned int clock_m, volatile unsigned int gate_m, volatile unsigned
int
output_m, volatile unsigned int count_m);
void outmeml(int a, unsigned long b, unsigned int c);
void outmemw(int a, unsigned long b, unsigned int c);
unsigned int inmeml(int a, unsigned long b);
unsigned int inmemw(int a, unsigned long b);
static void Usage(void);
static void set_present(void);
int rtc_mmap_test();
int read_clock_test();

/*
 * macros for mmap access - bit swizzling is needed for sparse space access
 * on alpha.
```

```
 */
#define read_shift(address,data,width) \
   { data = ( (data >> ((address & 3) << 3)) & \
         ( (1 << ((8*width)-1)) + ((1L << ((8*width)-1L))-1L) ) ); }

#define sti(base, offset, data) { \
   unsigned int *va; \
   unsigned long write_data = \
      (unsigned long) data << (((u_long)offset & 3) << 3); \
   va = (unsigned int *) ((((u_long)offset) << 5) + \
      ((u_long)base ) | 0x18); \
   *(int *)va = write_data; \
   (void)asm("mb"); }

#define ldi(base, offset, dest) { \
   unsigned int *va; \
   unsigned long data=0; \
   va = (unsigned int *) ((((u_long)offset) << 5) + \
      ((u_long)base ) | 0x18); \
   data = (unsigned long) *(int *)va; \
   read_shift(((u_long)offset & 3), data, 4); \
   dest = (unsigned int) data; }

/*+
 *
============================================================================

 * = rtc_ivp - AMCC 5933 RT Clock diagnostic.                          =
 *
============================================================================

 *
 * OVERVIEW:
 *
 *   COMMAND FORM:
 *
 *          rtc_ivp    ([-p <pass_count>]
 *               [-t <test_number>]
 *                    [-g <test_group>]
 *                    [-dd <no param, displays extra info>]
 *                    [-up <no param, set counters to up mode>]
 *                    [-val <data for timer>]
 *                    [-tmr <specify timer 1-7>]
 *                    [-dly <delay in microseconds between gate enable and interrupt>]
 *                    [-clk <clock value 1-7>]
 *
 *          Test 1: CSR setup
 *          Test 2: Mmap access test
 *          Test 3: Read clock test to check timer advance
 *          Test 4: Timer Count and Interrupt test
 *          Test 5: S/W Timer count test
 *          Test 6: Timer daisy chain, external input test
 *
 *
 *   COMMAND OPTION(S):
 *
```

```
 *          -p <pass_count> - Specifies the number of times to run the test.
 *                    If 0, the test runs forever.
 *                    This overrides the value of environment variable, d_passes.
 *                    In the absence of this option, d_passes is used.
 *                    The default for pass_count is 1.
 *
 *          -t <test_number> - Specifies the test number(s) to be run.
 *
 *          -dd <no_arg> - provides extra test info.
 *
 *
 *   COMMAND EXAMPLE:
 *~
 *   >>>rtc_ivp -t 1 -p 100
 *   >>>
 *~
 *   COMMAND REFERENCES:
 *          rtc_ivp
 *
 * FORM OF CALL:
 *          rtc_ivp (argc, *argv[] )
 *
 * RETURN CODES:
 *          msg_success - Normal completion.
 *          msg_failure - Test Failed.
 *
 * ARGUMENTS:
 *          u_long   argc       - number of command line arguments passed by the shell
 *          char *argv[]        - array of pointers to arguments
 *
 * SIDE EFFECTS:
 *          none
-*/
void
main (int argc, char **argv) {

  unsigned int pass_count = 1, tst_num = 0, status;
  io_handle_t temp;

  eptr = (struct GLOBALS_STRUCT  *)malloc(sizeof(struct GLOBALS_STRUCT));

  rtc_fd = open ("/dev/rtc0", O_RDWR);
  if (rtc_fd <0)
   {
    printf("Error opening /dev/rtc0!!\n");
    printf("errno = %d.  0x%x\n", errno, errno);
    perror("rtc open");
    exit(0);
   }

  /* Retrieve the PCI base address for the AMCC registers
   */
  if ((err = ioctl(rtc_fd, RTC_GETBASEADDR, &temp)) < 0) /* set */
              perror("RTC_GETBASEADDR"),exit(1);

  RTC_mem_base[0] = temp;
```

```
/* Retrieve the PCI base address for the RTC registers
 */
if ((err = ioctl(rtc_fd, RTC_GETBASEADDR1, &temp)) < 0) /* set */
            perror("RTC_GETBASEADDR"),exit(1);

RTC_mem_base1[0] = temp;

/* Make sure that the interrupt count is cleared before starting a test.
 */
if ((err = ioctl(rtc_fd, RTC_INT_COUNT, &status)) < 0)
                  perror("RTC_INT_COUNT"), exit(1);

commandLine = argv[0];
while (argc > 1 && argv[1][0] == '-') {
  if (strcmp(argv[1], "-help") == 0) {
    const char **p = readme;
    while (*p)
      printf("%s\n", *p++);
    exit(0);
  } else if (strcmp(argv[1], "-t") == 0 && argc > 2) {
    tst_num = atoi(argv[2]);
  } else if (strcmp(argv[1], "-p") == 0 && argc > 2) {
    pass_count = atoi(argv[2]);
  } else if (strcmp(argv[1], "-tmr") == 0 && argc > 2) {
    eptr->qual[QTMR].value.integer = atoi(argv[2]);
  } else if (strcmp(argv[1], "-val") == 0 && argc > 2) {
    eptr->qual[QVAL].value.integer = atoi(argv[2]);
  } else if (strcmp(argv[1], "-dly") == 0 && argc > 2) {
    eptr->qual[QDLY].value.integer = atoi(argv[2]);
  } else if (strcmp(argv[1], "-clk") == 0 && argc > 2) {
    eptr->qual[QCLK].value.integer = atoi(argv[2]);
  } else if (strcmp(argv[1], "-up") == 0) {
    eptr->qual[QUP].value.integer = 1;
    argc++;
    argv--;
  } else if (strcmp(argv[1], "-dd") == 0) {
    eptr->qual[QDD].value.integer = 1;
    argc++;
    argv--;
  } else
    Usage();
  argc -= 2; argv += 2;
}

set_present();  /* checks input flags and sets the present state */

switch(tst_num) {

   case 1:
             while(pass_count)
          {
               S5933_t1();
               pass_count--;
              }
            break;
   case 2:
       {
```

```
                rtc_mmap_test();
                }
                break;
    case 3:
                while(pass_count)
               {
               read_clock_test();
               pass_count--;
               }
                break;
    case 4:
                while(pass_count)
               {
                    S5933_t4();
                    pass_count--;
                    }
                break;
    case 5:
                while(pass_count)
               {
                    S5933_t5();
                    pass_count--;
                    }
                break;
    case 6:
                while(pass_count)
               {
                    S5933_t6();
                    pass_count--;
                    }
                break;
    default:
                printf("Invalid test selection. Use the -help switch for options\n");
                break;
  }
 /* cleanup */
 free(eptr);
 status = close(rtc_fd);
 if (status < 0)
  {
   printf("Error closing /dev/rtc!!\n");
   printf("errno = %d.  0x%x\n", errno, errno);
   perror("rtc close");
  }
}


/*+

*=====================================================================

 *= start_gate  - Timer gate and interrupt enable function.                    =

*=====================================================================
```

```
 *
-*/
void start_gate (volatile unsigned int gate)
{
  /* enable the interrupt - RMW */

  gate |= inmeml(0, RTC_mem_base1[0] + INT_ENABLE);
  outmeml(0, RTC_mem_base1[0] + INT_ENABLE, gate);

  /* turn on the gate on/off to start timer counting - RMW */

  gate |= inmeml(0, RTC_mem_base1[0] + GATE_ON_OFF);
  outmeml(0, RTC_mem_base1[0] + GATE_ON_OFF, gate);


}

/*+

*=============================================================

 *= clear_gate  -  Timer gate clear function                  =

*=============================================================

 *
-*/
void clear_gate ()
{
  /* Make sure the gate on/off is disabled before starting preloads */
  outmeml(0, RTC_mem_base1[0] + GATE_ON_OFF, 0x00);
  outmeml(0, RTC_mem_base1[0] + INT_ENABLE, 0x000);
}

/*+

*=============================================================

 *= check_and_clear  -  checks int count and clears int and gate enable.    =

*=============================================================

 *
-*/
void check_and_clear (unsigned int gate, int test_num)
{
  volatile unsigned int data = 0, interrupt_count;

 if ((err = ioctl(rtc_fd, RTC_INT_COUNT, &interrupt_count)) < 0)
                perror("RTC_INT_COUNT"), exit(1);

  /* check if the isr was invoked */
  if (interrupt_count != 1)
     printf("T%d Error: Bad Interrupt count - %d, enable mask = %x\n", test_num
,interrupt_count, gate);

  gate ^= gate;  /* we want a mask that will zero the bit passed to us */

  /* disable the interrupt */
  data = inmeml(0, RTC_mem_base1[0] + INT_ENABLE);
  outmeml(0, RTC_mem_base1[0] + INT_ENABLE, data & gate);

  /* turn off the gate on/off! This is critical for count once mode */
```

```
  data = inmeml(0, RTC_mem_base1[0] + GATE_ON_OFF);
  outmeml(0, RTC_mem_base1[0] + GATE_ON_OFF, data & gate);

}

/*+
*=====================================================================

 *= timer_setup  -  Writes all the timer csr registers.                =

 *=====================================================================

 *
-*/
void timer_setup(volatile unsigned int timer, volatile unsigned int timer_v, volatile unsigned
int timer_v_64hi, volatile
                 unsigned int clock_m, volatile unsigned int gate_m, volatile unsigned int
output_m, volatile unsigned int count_m)
{
/*  Write the timer preload with the count value passed in.  The 16bit Timers need word
 *  access so they are not overwritten.
 */
 if (debug_t6)
  outmeml(0, RTC_mem_base1[0] + timer,  timer_v);
 else
  {
   if (timer == 0x14 | timer == 0x12 | timer == 0x10 | timer == 0x16)
     {
      outmemw(0, RTC_mem_base1[0] + timer,  timer_v);
     }
      else
     {
      outmeml(0, RTC_mem_base1[0] + timer,  timer_v);
     }
  }
/*
 * Fill in the Clock, Gate, Output, and Count modes - RMW all these registers
 */
 if (debug_t6) /* enable RMW to the registers */
  {

   clock_m |= inmeml(0, RTC_mem_base1[0] + CLOCK_DIV);
   gate_m |= inmeml(0, RTC_mem_base1[0] + GATE_MODE);
   output_m |= inmeml(0, RTC_mem_base1[0] + OUTPUT_MODE);
   count_m |= inmeml(0, RTC_mem_base1[0] + COUNT_MODE);

  }

 outmeml(0, RTC_mem_base1[0] + CLOCK_DIV, clock_m);
 outmeml(0, RTC_mem_base1[0] + GATE_MODE, gate_m);
 outmeml(0, RTC_mem_base1[0] + OUTPUT_MODE, output_m);
 outmeml(0, RTC_mem_base1[0] + COUNT_MODE, count_m);

 /* Write the upper 32 bits of the 64 bit timer separately for now. */
 /* If the timer offset is 64LO, write 64HI                         */
 if (timer == 0x0)  /* if the timer offset is 64LO, write 64HI */
    outmeml(0, RTC_mem_base1[0] + OUT_64HI,  timer_v_64hi);

}
```

```
/* these routines were originally provided by the firmware console.  It was
 * easier to port the diag tests by creating these routines.
 */
void outmeml(int a, unsigned long b, unsigned int c)
{
  rtc_reg_data_t outport;

  outport.reg[0] = b;
  outport.data[0] = c;

  if ((err = ioctl(rtc_fd, RTC_WRITE_LW, &outport)) < 0) /* set */
            perror("RTC_WRITE_LW"),exit(1);
}

void outmemw(int a, unsigned long b, unsigned int c)
{
  rtc_reg_data_t outport;

  if(b & 0x3)
   {
    b = b & 0xfffffffffffffffcL;
    c = c<<16;
   }

  outport.reg[0] = b;
  outport.data[0] = c;
  if ((err = ioctl(rtc_fd, RTC_WRITE_LW, &outport)) < 0) /* set */
            perror("RTC_WRITE_LW"),exit(1);
}

unsigned int inmeml(int a, unsigned long b)
{
  rtc_reg_data_t inport;

  inport.reg[0] = b;

  if ((err = ioctl(rtc_fd, RTC_READ_LW, &inport)) < 0) /* set */
            perror("RTC_READ_LW"),exit(1);

  return(inport.data[0]);
}

unsigned int inmemw(int a, unsigned long b)
{
  rtc_reg_data_t inport;

  if(b & 0x3)
   {
    b = b & 0xfffffffffffffffcL;
    inport.reg[0] = b;
    if ((err = ioctl(rtc_fd, RTC_READ_LW, &inport)) < 0) /* set */
            perror("RTC_READ_LW"),exit(1);
    return(inport.data[0]>>16);
   }
  else
   {
    inport.reg[0] = b;
    if ((err = ioctl(rtc_fd, RTC_READ_LW, &inport)) < 0) /* set */
```

```
            perror("RTC_READ_LW"),exit(1);
    return(inport.data[0]);
   }
}

/*+
 *
=====================================================================

 * = S5933_t1 - Amcc 5933 test 1.                                    =
 *
=====================================================================

 *
 * Function Desc:
 *          Test #1 for the Amcc 5933 RT Clock device.  Inits all registers to 0.
 *
 * Return Values:
 *          msg_success       If test completes with no errors
 *          msg_failure       If test completes with    errors
 *
-*/
u_long   S5933_t1()
{
   unsigned int cnt = 0;
   unsigned int clock_csr = OUT_64LO;     /* init to csr base */
   volatile unsigned int data_check = 0;

   if(eptr->qual[QDD].present)
     printf("\n S5933_t1  Timer clear - zero out the registers\n");

   if (RTC_mem_base1[0] == 0)
     {
          printf(" No S5933 device present - test exited\n");
          return 0;
     }

   /* After writing zero to the registers, read them back to check
    * and make sure the registers stay init'd.
    */
   for( cnt = 0; cnt < 16; cnt++ )
    {
     /* check to make sure we're not hitting the reserved registers */
     if (clock_csr == 0x18 | clock_csr == 0x1C) {}
     else
      {
       outmeml(0, RTC_mem_base1[0] + clock_csr, 0x00000000);
       if(eptr->qual[QDD].present)
         printf("zeroing address %x\n", RTC_mem_base1[0] + clock_csr);
      }
     clock_csr += 4;
    }

   /* Set the start at the s/w event clock reg so we can
    * read in reverse order.  Needed for the 64HI snapshot
    * register.
    */
```

```
    clock_csr = 0x0;

    for( cnt = 0; cnt < 16; cnt++ )
     {
      /* check to make sure we're not reading the reserved registers */
      if (clock_csr == 0x18 | clock_csr == 0x1C) {}
      else
       {
        data_check = inmeml(0, RTC_mem_base1[0] + clock_csr);
        if (data_check != 0)
          printf("T1:  Error - timer data not zeroed at offset %x, data %x\n", clock_csr,
data_check);
        if(eptr->qual[QDD].present)
          printf("reading address %x\n", RTC_mem_base1[0] + clock_csr);
       }

      clock_csr += 4;
     }

  return msg_success;

}

/*+
 *
=========================================================================

 * = S5933_t4 -  Timer count test.                                       =
 *
=========================================================================

 *
 * Function Desc:
 *          This test will check all the timer count modes.
 *
 * Return Values:
 *          msg_success       If test completes with no errors
 *          msg_failure       If test completes with    errors
 *
-*/
u_long   S5933_t4()
{
   int tst_num = 0, i;
   volatile unsigned int load_val = 0, load_val_64hi = 0, delay_val, clk_sel, clk_val, gate_m,
output_m, count_m = 0, timer, enable;
   unsigned char buf[80];

   eptr->return_msg = msg_success;

   if (eptr->qual[QDD].present)
      printf(" S5933_t4 Timer count mode test.\n");

   if (RTC_mem_base1[0] == 0)
      {
           printf(" No S5933 device present - test exited\n");
           return 0;
      }

   int_setup(); /* register vector and enable amcc */
```

```
  if(eptr->qual[QTMR].present)
    tst_num = eptr->qual[QTMR].value.integer;
  else
   {
    printf("\n ================== Test selections ======================  ");
    printf("\n 1 = 16a, 2 = 16b, 3 = 16c, 4 = 16d, 5 = 32a, 6 = 32b, 7 = 64, ");
    printf("\n 8 = gate disable                                          ");
    printf("\n ========================================================  ");
    printf("\n Enter test number: ");
    gets(buf);
    tst_num = atoi(buf);
   }

  if(eptr->qual[QVAL].present)
    load_val = eptr->qual[QVAL].value.integer;
  else
    load_val = 0xfffe;  /* full count - a h/w feature requires (n-1) count */

  if(eptr->qual[QDLY].present)
    delay_val = eptr->qual[QDLY].value.integer;
  else
    delay_val = 209800;  /* 209.8 ms delay */

  if(eptr->qual[QCLK].present)
    clk_val = eptr->qual[QCLK].value.integer;
  else
   {

    clk_val = 0x7;  /* divide by 64 clock */
   }

  if(eptr->qual[QUP].present)
   {
    count_m = 0x0000;
    load_val_64hi = 0xffffffff;
   }
  else
   {
    count_m = 0x4000; /* 16a count down/once, the data will be right-shifted to set the other
timers */
    load_val_64hi = 0x0;
   }

  clear_gate ();

  switch (tst_num) {

    case 1:
        clk_sel = clk_val;
            /* Timer 16a test */
            timer_setup( OUT_16A,   /* offset - 0x14                        */
                         load_val,     /* pre-load value                    */
                         load_val_64hi,  /* 64hi pre-load value             */
                         clk_sel,   /* clock divide by 64                   */
                         0x400000,      /* gate mode = internal/neg/edge    */
                       0x400000,        /* output mode = 64 enabled/neg/edge        */
                         count_m);   /* count mode =  down/repeat           */
```

```
        start_gate (0x40);
    krn$_micro_delay(delay_val); /* + extra for timer preemption */
        check_and_clear(0x40, 4); /* this rountine looks for the right int count and clears
                                     enabled regs and counts for the next test */
        break;

case 2:
        clk_sel = clk_val << 3;
        /* Timer 16b test */
        timer_setup( OUT_16B,   /* offset - 0x12                    */
                     load_val,  /* pre-load value                   */
                     load_val_64hi,  /* 64hi pre-load value         */
                     clk_sel,   /* clock divide by 64               */
                     0x200000,      /* gate mode = internal/neg/edge */
                0x200000,           /* output mode = 64 enabled/neg/edge  */
                     (count_m >> 1));   /* count mode =  once        */

        start_gate(0x20);
    krn$_micro_delay(delay_val); /* + extra for timer preemption */
        check_and_clear(0x20, 4); /* this rountine looks for the right int count and clears
                                     enabled regs and counts for the next test */
        break;

case 3:
        clk_sel = clk_val << 6;
        /* Timer 16c test */
        timer_setup( OUT_16C,   /* offset - 0x10                    */
                     load_val,  /* pre-load value                   */
                     load_val_64hi,  /* 64hi pre-load value         */
                     clk_sel,    /* clock divide by 64              */
                     0x100000,      /* gate mode = internal/neg/edge */
                0x100000,           /* output mode = 64 enabled/neg/edge  */
                     (count_m >> 2));   /* count mode =  once        */

        start_gate(0x10);
    krn$_micro_delay(delay_val); /* + extra for timer preemption */
        check_and_clear(0x10, 4); /* this rountine looks for the right int count and clears
                                     enabled regs and counts for the next test */
        break;

case 4:
        clk_sel = clk_val << 9;
        /* Timer 16d test */
        timer_setup( OUT_16D,   /* offset - 0x0e                    */
                     load_val,  /* pre-load value                   */
                     load_val_64hi,  /* 64hi pre-load value         */
                     clk_sel,    /* clock divide by 64              */
                     0x080000,      /* gate mode = internal/neg/edge */
                0x080000,           /* output mode = 64 enabled/neg/edge  */
                     (count_m >> 3));   /* count mode =  once        */

    start_gate(0x08);
    krn$_micro_delay(delay_val); /* + extra for timer preemption */
        check_and_clear(0x08, 4); /* this rountine looks for the right int count and clears
                                     enabled regs and counts for the next test */
        break;
```

```
case 5:
        clk_sel = clk_val << 12;
        /* Timer 32a test */
        timer_setup( OUT_32A,   /* offset - 0x0C                                */
                    load_val,  /* pre-load value                        */
                    load_val_64hi,  /* 64hi pre-load value              */
                    clk_sel,   /* clock divide                         */
                    0x040000,      /* gate mode = internal/neg/edge     */
                0x040000,          /* output mode = 64 enabled/neg/edge */
                    (count_m >> 4));   /* count mode =  once            */

    start_gate(0x04);
    krn$_micro_delay(delay_val); /* 209.72 mS + extra for timer preemption */
        check_and_clear(0x04, 4); /* this rountine looks for the right int count and clears
                                     enabled regs and counts for the next test */
        break;

case 6:
        clk_sel = clk_val << 15;
        /* Timer 32b test */
        timer_setup( OUT_32B,   /* offset - 0x08                        */
                    load_val,  /* pre-load value                        */
                    load_val_64hi,  /* 64hi pre-load value              */
                    clk_sel,    /* clock divide by 64                   */
                    0x020000,      /* gate mode = internal/neg/edge     */
                0x020000,          /* output mode = 64 enabled/neg/edge */
                    (count_m >> 5));   /* count mode =  once            */

        start_gate(0x02);
    krn$_micro_delay(delay_val); /* + extra for timer preemption */
        check_and_clear(0x02, 4); /* this rountine looks for the right int count and clears
                                     enabled regs and counts for the next test */
        break;

case 7:
        clk_sel = clk_val << 18;
        /* Timer 64LO test */
        timer_setup( OUT_64LO,  /* offset - 0x00                        */
                    load_val,  /* pre-load value                        */
                    load_val_64hi,  /* 64hi pre-load value              */
                    clk_sel,   /* clock divide                         */
                0x010000,          /* gate mode = internal/neg/edge     */
                0x010000,          /* output mode = 64 enabled/neg/edge */
                    (count_m >> 6));   /* count mode =  once            */

        start_gate(0x01);
    krn$_micro_delay(delay_val);  /* 209.72 mS + extra for timer preemption */
        check_and_clear(0x01, 4); /* this rountine looks for the right int count and clears
                                     enabled regs and counts for the next test */
        break;

case 8:
      /* turn off the gate on/off to shut down the timer. */
      outmeml(0, RTC_mem_base1[0] + GATE_ON_OFF, 0x00000000);
      break;
```

```
    default:  printf("\n Not a valid test selection!\n");
                break;
   }

  return msg_success;

}

/*+
 *
=========================================================================

 * = S5933_t5 - S/W Timer count test.                                    =
 *
=========================================================================

 *
 * Function Desc:
 *          This test will check all the timer count modes using the software
 *          clocking mechanism.
 *
 * Return Values:
 *          msg_success       If test completes with no errors
 *          msg_failure       If test completes with    errors
 *
-*/
u_long   S5933_t5()
{
   unsigned int i = 0, tst_num, tmr_offset, bad_input = 0, count_m;
   volatile unsigned int data, load_val = 0, load_val_64hi = 0, sw_clk_val = 0, test_data = 0,
sw_clock_loop = 0;
   unsigned char buf[80];

   eptr->return_msg = msg_success;

   if (eptr->qual[QDD].present)
     printf(" S5933_t5 S/W timer count mode test.\n");

   if (RTC_mem_base1[0] == 0)
     {
          printf(" No S5933 device present - test exited\n");
          return 0;
     }

   int_setup(); /* register vector and enable amcc */

   if(eptr->qual[QTMR].present)
     tst_num = eptr->qual[QTMR].value.integer;
   else
    {
     printf("\n ================== Test selections ======================  ");
     printf("\n 1 = 16a, 2 = 16b, 3 = 16c, 4 = 16d, 5 = 32a, 6 = 32b, 7 = 64, ");
     printf("\n 8 = gate disable                                          ");
     printf("\n ========================================================  ");
     printf("\n Enter test number: ");
     gets(buf);
     tst_num = atoi(buf);
    }
```

```
  /* check for user specified timer preload value
   */
  if(eptr->qual[QVAL].present)
      load_val = test_data = eptr->qual[QVAL].value.integer;
  else
      load_val = test_data = 0xffff;  /* this case needs looking into for QUP */

  if(eptr->qual[QUP].present)
   {
    count_m = 0x0000;
    load_val_64hi = 0xffffffff;
   }
  else
   {
    count_m = 0x4000; /* 16a count down/once, the data will be right-shifted to set the other
timers */
    load_val_64hi = 0x0;
   }

  clear_gate ();

   switch (tst_num) {

    case 1:
         /* Timer 16a test */
         timer_setup( OUT_16A,    /* offset - 0x14                          */
                   load_val,   /* value                                    */
                       load_val_64hi,  /* 64hi pre-load value              */
                       0x00000000, /* clock divide - S/W                   */
                       0x400000,       /* gate mode = internal/neg/edge    */
                   0x400000,           /* output mode = 16a enabled/neg/edge */
                       count_m);   /* count mode =  down/repeat            */
      start_gate(0x40);
         tmr_offset = OUT_16A;
         break;

    case 2:
           /* Timer 16b test */
           timer_setup( OUT_16B,   /* offset - 0x12                        */
                       load_val,  /* pre-load value                       */
                       load_val_64hi,  /* 64hi pre-load value             */
                       0x00000000, /* clock divide - S/W                  */
                       0x200000,       /* gate mode = internal/neg/edge   */
                   0x200000,           /* output mode = 64 enabled/neg/edge */
                       (count_m >> 1));   /* count mode =  once           */
      start_gate(0x20);
      tmr_offset = OUT_16B;
         break;

    case 3:
           /* Timer 16c test */
           timer_setup( OUT_16C,   /* offset - 0x10                        */
                       load_val,  /* pre-load value                       */
                       load_val_64hi,  /* 64hi pre-load value             */
                       0x00000000, /* clock divide - S/W                  */
                       0x100000,       /* gate mode = internal/neg/edge   */
                   0x100000,           /* output mode = 64 enabled/neg/edge */
```

```
                    (count_m >> 2));   /* count mode =  once                    */
  start_gate(0x10);
  tmr_offset = OUT_16C;
      break;

case 4:
        /* Timer 16d test */
        timer_setup( OUT_16D,   /* offset - 0x0e                    */
                    load_val,  /* pre-load value                    */
                    load_val_64hi,  /* 64hi pre-load value          */
                    0x00000000, /* clock divide - S/W                */
                    0x080000,      /* gate mode = internal/neg/edge  */
                0x080000,          /* output mode = 64 enabled/neg/edge  */
                    (count_m >> 3));   /* count mode =  once           */
  start_gate(0x08);
  tmr_offset = OUT_16D;
      break;

case 5:
        /* Timer 32a test */
        timer_setup( OUT_32A,   /* offset - 0x0C                     */
                    load_val,  /* pre-load value                    */
                    load_val_64hi,  /* 64hi pre-load value          */
                    0x00000000,/* clock divide - S/W              */
                    0x040000,      /* gate mode = internal/neg/edge  */
                0x040000,          /* output mode = 64 enabled/neg/edge  */
                    (count_m >> 4));   /* count mode =  once           */
  start_gate(0x04);
      tmr_offset = OUT_32A;
      break;

case 6:
      /* Timer 32B test */
      timer_setup( OUT_32B,   /* offset - 0x08                     */
                    load_val,  /* value                             */
                    load_val_64hi,  /* 64hi pre-load value          */
                    0x00000000,/* clock divide - S/W                */
                    0x020000,      /* gate mode = internal/neg/edge  */
                0x020000,          /* output mode = 64 enabled/neg/edge  */
                    (count_m >> 5));   /* count mode =  once           */
  start_gate(0x02);
  tmr_offset = OUT_32B;
      break;

case 7:
      /* Timer 64LO test */
      timer_setup( OUT_64LO,  /* offset - 0x00                     */
                    load_val,  /* value                             */
                    load_val_64hi,  /* 64hi pre-load value          */
                    0x00000000,/* clock divide - S/W              */
                0x010000,          /* gate mode = internal/neg/edge  */
                0x010000,          /* output mode = 64 enabled/neg/edge  */
                    (count_m >> 6));   /* count mode =  once           */
  start_gate(0x01);
  tmr_offset = OUT_64LO;
      break;
```

```
  case 8:
        /* turn off the gate on/off to shut down the timer. */
        outmeml(0, RTC_mem_base1[0] + GATE_ON_OFF, 0x00000000);
        break;

  default:  printf("\n Not a valid test selection!\n");
              bad_input = 1;
              break;
  }

if (!bad_input)  /* continue with the test */
  {
   if(eptr->qual[QUP].present)
         {
     if (load_val > 0xffff)
            {
       sw_clock_loop = 0xffffffff - load_val;
            }
          else
       {
            sw_clock_loop = 0xffff - load_val;
           }
        }
   else
     {
     sw_clock_loop = load_val;
     }

    sw_clock_loop += 1;  /* FOD: all timers are now n-1.  */

    for (i = 0; i < sw_clock_loop; i++)
    {
     switch (tmr_offset) {

        case OUT_64LO:
          sw_clk_val = 0x01;
               break;
        case OUT_32B:
          sw_clk_val = 0x02;
               break;
        case OUT_32A:
          sw_clk_val = 0x04;
               break;
        case OUT_16D:
          sw_clk_val = 0x08;
               break;
        case OUT_16C:
          sw_clk_val = 0x10;
               break;
        case OUT_16B:
          sw_clk_val = 0x20;
        break;
        case OUT_16A:
          sw_clk_val = 0x40;
          break;
        }
      outmeml(0, RTC_mem_base1[0] + SW_CLK, sw_clk_val);
```

```
        if (tmr_offset == OUT_64LO | tmr_offset == OUT_32B | tmr_offset == OUT_32A)
          data = inmeml(0, RTC_mem_base1[0] + tmr_offset);  /* read timer */
            else
          data = inmemw(0, RTC_mem_base1[0] + tmr_offset);  /* read timer */

        if(eptr->qual[QUP].present)  /* should the data match the loaded data after the 1st s/w
clock was issued? */
              test_data ++;
        else
              test_data --;

            if (!(i == (sw_clock_loop - 1)))
         {
          if (data != test_data)
              printf("T5 Error: data at timer offset %x is bad: expected = %x, received = %x\n",
tmr_offset, test_data ,data);
             }
       }

      /* Hack alert.  This gets around h/w PASS2 bug */
      outmeml(0, RTC_mem_base1[0] + SW_CLK, sw_clk_val);

      /* PASS1 vs PASS2 difference.  The start gate value will match the
            SW clocking value so we should be able to store both in a variable.
            For now let's use the sw_clk_val.  I think gate needs to go to 0
            for the H/W carry out bit used for INT's to be reset.
           */
      check_and_clear(sw_clk_val, 5); /* this rountine looks for the right int count and clears
                                         enabled regs and counts for the next test */
     }

  return msg_success;
}

/*+
 *
=====================================================================
 * = S5933_t6 - Cascaded timer, external trigger test.                =
 *
=====================================================================
 *
 * Function Desc:
 *          Big test. Everything needs to work on this option. It will require all
 *          the external inputs and outputs to be tied together so that the timers
 *      will drive each other thru the count until the last in the chain
 *          interrupts.
 *
 * Return Values:
 *          msg_success       If test completes with no errors
 *          msg_failure       If test completes with    errors
 *
-*/
u_long   S5933_t6()
{
   int i = 0, data, load_val = 0, load_val_64hi = 0, tst_num, temp = 0, bad_input = 0, clk_val =
```

```
0, clk_sel = 0, delay_val = 0;
  unsigned char buf[80];

  eptr->return_msg = msg_success;

  if (eptr->qual[QDD].present)
    printf(" S5933_t6 daisy chained timer, external trigger test.\n");

  if (RTC_mem_base1[0] == 0)
    {
        printf(" No S5933 device present - test exited\n");
        return 0;
    }

  int_setup(); /* register vector and enable amcc */

  if(eptr->qual[QVAL].present)
    load_val = eptr->qual[QVAL].value.integer;
  else
    load_val = 0x3;  /* test count */

  load_val_64hi = 0;  /* 0 the 64hi pre-load value - counting down */

  if(eptr->qual[QDLY].present)
    delay_val = eptr->qual[QDLY].value.integer;
  else
    delay_val = 900000;  /* make it big for now */

  clear_gate ();

  /* Timer 16a will be gated by an internal clock.  All other timers will be
   * gated by an external input which will be wired from the previous timer.
   */

        debug_t6 = 1;  /* temp used to tell timer_setup not to use word offsets */
        temp = (load_val << 16) | load_val;  /* contains data for 16ab and 16dc */

      clk_sel = 0x24f;
        /* Timer 16a test */
        timer_setup( OUT_16A,   /* offset - 0x14                    */
                temp,  /* pre-load value                           */
                load_val_64hi,  /* 64hi pre-load value             */
                clk_sel,    /* clock divide by 64                  */
                0x600000,      /* gate mode = internal/neg/edge    */
              0x600000,        /* output mode = 64 enabled/neg/edge */
                0x6060);   /* count mode =  down/repeat            */

        clk_sel = 0; /*This is zero because the above clk_sel is a covering the 16bit timer */
        /* Timer 16c test */
        timer_setup( OUT_16C,   /* offset - 0x10                    */
                temp,  /* pre-load value                           */
                load_val_64hi,  /* 64hi pre-load value             */
                clk_sel,    /* clock divide by 64                  */
                0x180000,      /* gate mode = external/neg/edge    */
              0x180000,        /* output mode = 64 enabled/neg/edge */
                0x1818);   /* count mode =  down/repeat            */

      clk_val = 0x1;  /* External clocking */
        clk_sel = clk_val << 15;
        /* Timer 32b test */
        timer_setup( OUT_32B,   /* offset - 0x08                    */
```

```
                    load_val,  /* pre-load value                          */
                    load_val_64hi,  /* 64hi pre-load value               */
                    clk_sel,    /* clock divide by 64                     */
                    0x020000,      /* gate mode = internal/neg/edge       */
                0x020000,          /* output mode = 64 enabled/neg/edge   */
                    0x0202);   /* count mode =  down/repeat              */

            clk_sel = clk_val << 12;
            /* Timer 32a test */
            timer_setup( OUT_32A,   /* offset - 0x0C                      */
                    load_val,  /* pre-load value                          */
                    load_val_64hi,  /* 64hi pre-load value               */
                    clk_sel,   /* clock divide                           */
                    0x040000,      /* gate mode = external/neg/edge       */
                0x040000,          /* output mode = 64 enabled/neg/edge   */
                    0x0404);   /* count mode =  down/repeat              */

            clk_sel = clk_val << 18;
            /* Timer 64LO test */
            timer_setup( OUT_64LO,  /* offset - 0x00                      */
                    load_val,  /* pre-load value                          */
                    load_val_64hi,  /* 64hi pre-load value               */
                    clk_sel,   /* clock divide                           */
                0x010000,          /* gate mode = external/neg/edge       */
                0x010000,          /* output mode = 64 enabled/neg/edge   */
                    0x0100);   /* count mode =  down/once                */

  /* Setup is done.  Now enable the interrupts and gate to start the
   * timers.  Wait some calculated amount of time and check for the
   * 64 bit timer to interrupt which means everything worked.
   */

  /* enable the interrupt */
  outmeml(0, RTC_mem_base1[0] + INT_ENABLE, 0x001);

  /* Enable all timers with the gate on/off */
  outmeml(0, RTC_mem_base1[0] + GATE_ON_OFF, 0x7f);

  krn$_micro_delay(delay_val); /* + extra for timer preemption */

  /* this rountine looks for the right int count and clears
  *  enabled regs and counts for the next test
  */
  check_and_clear(0x01, 6);
  /* disable the rest of the timers not cleared by above. */
  outmeml(0, RTC_mem_base1[0] + GATE_ON_OFF, 0x0);

 debug_t6 = 0;  /* clear this static variable so it won't effect other tests
                */

 return msg_success;
}

/*
  int_setup
*/
void int_setup ()
{
  int return_status;
```

```
  /* AMCC PCI interface int enable */
   outmeml(0, RTC_mem_base[0] + INTCSR, MB_INTENABLE); /* RMW this reg? */
}

/*
  Usage
*/
static void Usage(void)
{
 fprintf(stderr, "\n");
 fprintf(stderr, "Usage: %s\n", commandLine);
 fprintf(stderr, " -p <pass_count> - the number of times to run the test.\n");
 fprintf(stderr, "      The default for pass_count is 1.\n");
 fprintf(stderr, " -t <test_number> - Specifies the test number to be run.\n");
 fprintf(stderr, " -dd <no_arg> - Dumps local test data to console terminal\n");
 fprintf(stderr, "\n");
 fprintf(stderr, " COMMAND EXAMPLE: # rtc_ivp -t 5 -tmr 1 -p 10\n");
 fprintf(stderr, "\n");
   exit(1);
}

static void set_present(void)
{
     if(eptr->qual[QTMR].value.integer)
            eptr->qual[QTMR].present = 1;
     if(eptr->qual[QVAL].value.integer)
            eptr->qual[QVAL].present = 1;
     if(eptr->qual[QDLY].value.integer)
       eptr->qual[QDLY].present = 1;
     if(eptr->qual[QCLK].value.integer)
       eptr->qual[QCLK].present = 1;
     if(eptr->qual[QUP].value.integer)
       eptr->qual[QUP].present = 1;
     if(eptr->qual[QDD].value.integer)
       eptr->qual[QDD].present = 1;
}

void rtc_micro_delay(int seconds)
{
 struct  timespec rtctime;
 time_t s_start_time = 0, n_start_time = 0, elapsed_time = 0;
 int err;

 if ((err = getclock(TIMEOFDAY, &rtctime)) < 0)
            perror("getclock"),exit(1);

 s_start_time = rtctime.tv_sec;
 n_start_time = rtctime.tv_nsec;

    while(seconds > elapsed_time)
      {
       if ((err = getclock(TIMEOFDAY, &rtctime)) < 0)
            perror("getclock"),exit(1);

          elapsed_time = rtctime.tv_sec - s_start_time;
      }
}
```

```
const char *readme[] = {
"",
" RTC Installation Verification Test help information.",
" ----------------------------------------------------",
"",
" This application is used for testing the PCI Real Time Clock.  It's capable",
" of testing all 7 timers on the PCI Real Time Clock option.  The test has",
" flags that can be passed to select the timer to test, the number of passes",
" and the test type.",
"",
"   COMMAND OPTION(S):",
"",
"      -p <pass_count> - Specifies the number of times to run the test.",
"            The default for pass_count is 1.",
"",
"      -t <test_number> - Specifies the test number to be run.",
"",
"       Test 1: CSR initialization test",
"       Test 2: Mmap'd csr read/write test",
"       Test 3: 64 bit Timer read test",
"       Test 4: Timer Count and Interrupt test (counts down at 10Mhz)",
"       Test 5: S/W Timer Count test (counts down using S/W clocking mode)",
"       Test 6: Timer daisy chain test. Requires an external loopback. ",
"",
"      -tmr <timer_number> - Specifies the number that corresponds to the timer",
"",
"       Timer 1: 16 bit (a)",
"       Timer 2: 16 bit (b)",
"       Timer 3: 16 bit (c)",
"       Tmier 4: 16 bit (d)",
"       Timer 5: 32 bit (a)",
"       Timer 6: 32 bit (b)",
"       Timer 7: 64 bit ",
"",
"      -dd <no_arg> - provides extra test info.",
"",
" This is an example of how to test the first 16bit timer using the software",
" timer count test:",
"",
" # rtc_ivp -t 5 -tmr 1",
"",
0
};

int rtc_mmap_test()
{
  volatile unsigned long dummy[20];
  volatile caddr_t mem;
  volatile int *test_addr;
  int n, offset;

  if ((mem = mmap(0,PAGESIZE,PROT_WRITE,MAP_SHARED,rtc_fd,0)) == (void *) -1)
       perror("mmap failed"), exit(1);

  if (mlock(mem, (PAGESIZE)) < 0)
  {
```

```
    perror("mlock");
    exit(1);
  }

  test_addr = (int*)mem;
  printf("RTC mmap: address = %016lx\n", test_addr);

  for (n=0; n<6; n++) {
      offset = n*4;
      sti(test_addr, offset, 0xfeedface);
      ldi(test_addr, offset, dummy[n]);
      printf("RTC mmap init: offset = %016lx, data = %016lx\n",offset, dummy[n]);
      }

  return msg_success;
}

int read_clock_test()
{
  unsigned int old_time, new_time, i;
  float delta_time;
  rtc_reg_data_t regptr;
  struct RTC_IO   rtc_io;

  /* start the 64 bit timer.
   */
  if ((err = ioctl(rtc_fd, RTC_START_TIMER, &rtc_io)) < 0) /* set */
              perror("RTC_START_TIMER"),exit(1);

  /* This test checks for timer values that are out of sync. Because the timer
   * runs at a frequency that is different from the PCI clock, it is important
   * for the hardware to synchronize w/each frequency. If it's not synchronized
   * properly a negative time could occur. This test loops on reading the timer
   * and compares the newly read time with the previous copy.  It basically
   * verifies that time is advancing.
   */
  for (i=0; i<100000; i++)
  {
   /* read the 64 bit timer
   */
   if ((err = ioctl(rtc_fd, RTC_READ_TIMER, &regptr)) < 0) /* set */
              perror("RTC_READ_TIMER"),exit(1);

   old_time = regptr.data[0];

   /* read the 64 bit timer again and compare to previous read.
    */
   if ((err = ioctl(rtc_fd, RTC_READ_TIMER, &regptr)) < 0) /* set */
              perror("RTC_READ_TIMER"),exit(1);

   new_time = regptr.data[0];

   if(new_time < old_time)
     printf(" Error: Negative time read during pass #%d, old = %x new = %x\n",i, old_time,
new_time);
  }
  printf (" Time read verification test complete.\n");

  /* This test is a crude way of checking the clock and verifying a ballpark
   * elapsed time by using the system clock to count off 4 seconds and checking
```

```
   * the 64 bit timers count.
   */

  /* read the 64 bit timer
   */
  if ((err = ioctl(rtc_fd, RTC_READ_TIMER, &regptr)) < 0) /* set */
            perror("RTC_READ_TIMER"),exit(1);
  old_time = regptr.data[0];

  rtc_micro_delay(4);

  /* read the 64 bit timer again and compare to previous read.
    */
  if ((err = ioctl(rtc_fd, RTC_READ_TIMER, &regptr)) < 0) /* set */
            perror("RTC_READ_TIMER"),exit(1);

  new_time = regptr.data[0];
  delta_time = (float) new_time - old_time;
  delta_time = (delta_time*3.2)/1000000; /* convert to microseconds */

  if(delta_time > (float)4.0)
     printf(" Error in 4 second timing check: clock time elapsed = %f microseconds\n",
delta_time);

  printf (" Elapsed time test complete.\n");

  /* Stop the timer.
   */
  if ((err = ioctl(rtc_fd, RTC_STOP_TIMER, &rtc_io)) < 0) /* set */
            perror("RTC_STOP_TIMER"),exit(1);

}
```