

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

1.0 OVERVIEW

The AMCC S5933 PCI Controller provides add-in cards with bus mastering capabilities on the PCI bus. The S5933 internal FIFO is used to transfer data between the add-on and the PCI bus as a PCI initiator (bus master). The S5933 allows burst transfers at the full PCI bandwidth.

This application note discusses how the S5933 may be used in a PCI bus mastering (DMA) application. A brief background of DMA as it relates to the ISA bus is provided in Section 2. The remaining sections describe the bus master capabilities of the S5933 including add-on hardware support and required software support. Performance for bus mastering applications on the PCI bus is also discussed. An example C-Language program is provided in Appendix A to set up PCI DMA transfers with the S5933 controller.

2.0 DMA BACKGROUND

In ISA bus-based personal computers, there were two potential bus masters: the host CPU and the system DMA controller. There was no protocol defined for alternate bus masters to gain control of the bus and transfer data to other cards or platform memory. Most ISA add-in cards were designed as ISA bus slaves. The DMA controller or the CPU were used for data transfers to and from ISA cards.

For performance reasons, some ISA add-in cards required bus mastering capabilities. To become an ISA bus master, a free DMA channel was utilized. The ISA card asserted a DMA request to the 8237 DMA controller, the 82C37 asserted the HOLD input to the CPU. When the 82C37 received the HLDA acknowledge back from the CPU, it asserted an acknowledge to the add-in card. The add-in card then had control of the ISA bus to perform data transfers. For this reason, bus mastering is also referred to as DMA.

The PCI bus protocol has specific provisions to allow bus mastering (DMA) by any device connected to the PCI bus. If add-in cards perform their own data transfers, the host CPU is freed to perform other tasks (code execution, etc.). The PCI bus provides a dedicated request (REQ#)/grant(GNT#) pair to each PCI bus device (or card slot). These are all routed to a central bus arbiter that determines which device controls the PCI bus, based on a predefined priority scheme.

3.0 S5933 ARCHITECTURE

The S5933 performs DMA (bus master) transfers on the PCI bus through its FIFO interface. It has two, independent FIFOs. Each FIFO is 8-deep by 32-bits wide. One is used to transfer data from the PCI bus to the add-on, and the other is used to transfer data from the add-on to the PCI bus. The S5933 only performs DMA transfers to and from memory-mapped targets.

Each FIFO has an associated address and transfer count register. These registers may be defined by either the host CPU or add-on logic (configurable at reset). Each FIFO has a programmable priority and management scheme. Each FIFO also has the ability to generate interrupts when the transfer count expires or error conditions occur during a PCI bus transfer.

3.1 Configuring the S5933 for DMA Transfers

A DMA (bus master) transfer may be set up by either the host CPU, or add-on logic. This is defined for the S5933 at reset and cannot change during operation. Initiating a DMA transfer involves setting up the source/destination addresses and transfer counts as well as enabling the transfer. The FIFO relative priorities and FIFO management schemes are always programmed by the host CPU.

If an external, non-volatile boot memory is used with the S5933, the contents of offset 45h define FIFO operation. The following bits functions are defined:

Bit 7 Bus Master Register Access

- 0 Address and transfer count registers only accessible from the add-on interface
- 1 Address and transfer count registers only accessible from the PCI interface (default)

Bit 6 RDFIFO# or RD# Operation

- 0 Synchronous Mode — RDFIFO# and RD# functions as enables
- 1 Asynchronous Mode — RDFIFO# and RD# functions as clocks (default)

Bit 5 WRFIFO# or WR# Operation

- 0 Synchronous Mode — WRFIFO# and WR# functions as enables
- 1 Asynchronous Mode — WRFIFO# and WR# functions as clocks (default)

If no external non-volatile boot memory is used with the S5933, the default configuration for bus mastering is for transfers to be set up by the host CPU. Bits 6 and 5 define how the add-on FIFO interface operates. The default configuration for FIFO read and write strobes is to be asynchronous to the PCI clock (provided to the add-on by the S5933 BPCLK output). For more information on the FIFO add-on bus interface, refer to the S5933 PCI Controller Data Book.

3.1.1 PCI Initiated DMA Transfers

If no external non-volatile boot device is used, or if location 45h, bit 7 is 1, the address and transfer count registers are only accessible from the PCI bus interface. This requires that the PCI host write these register to initiate a DMA transfer. In this configuration, the S5933 can be programmed to generate a PCI interrupt (INTA#) when the DMA transfer count reaches zero or when an error occurs during a DMA transfer (target or master abort conditions).

PCI initiated DMA transfers must be enabled through the Bus Master Control/Status Register (MCSR). The register is at offset 3Ch in the S5933 PCI Operation Registers. S5933 Read DMA transfers and Write DMA transfers have separate control bits and can be individually enabled. The enable bits are not automatically cleared upon completion of a DMA transfer. DMA transfers remain enabled until these bits are cleared by the host CPU.

3.1.2 Add-on Initiated DMA Transfers

If an external, **serial** non-volatile boot device is used and location 45h, bit 7 is 0, the address and transfer count registers are only accessible from the add-on bus interface. This requires that add-on logic write these register to initiate a DMA transfer. In this configuration, the S5933 can be programmed to generate an add-on interrupt (IRQ#) when the DMA transfer count reaches zero or when an error occurs during a DMA transfer. A serial boot device is required because the transfer enable inputs used for add-on initiated DMA are multiplexed with the byte-wide nv-memory interface.

Add-on initiated DMA transfers are enabled using the AMREN and AMWEN inputs. The bus master enable bits in the Bus Master Control/Status Register (MCSR) are ignored. Asserting AMREN enables the S5933 to request control of the PCI bus for a PCI read transfer when the appropriate FIFO conditions are met. Asserting AMWEN enables the S5933 to request control of the PCI bus for a PCI write transfer when the appropriate FIFO conditions are met (see section 3.4 on FIFO management schemes). If an

enable is deasserted during a DMA transfer, the current PCI bus transaction completes and the S5933 deasserts REQ#, giving up control of the PCI bus.

3.2 DMA Address Registers

There are two DMA address registers: the Master Write Address Register (MWAR) and the Master Read Address Register (MRAR). These registers are located at offsets 24h and 2Ch, respectively, in the S5933 PCI Operation Registers. These are written with the beginning memory address of the DMA transfer. The S5933 requires that DMA transfers start on double-word boundaries (A1 and A0 = 0).

During a DMA transfer, the address registers are incremented by four after each completed data phase. If a PCI target disconnects and requests a retry from the S5933, the correct address is maintained to allow the transfer to begin from where it was disconnected.

3.3 DMA Transfer Count Registers

There are two DMA transfer count registers: the Master Write Transfer Count Register (MWTC) and the Master Read Transfer Count Register (MRTC). These registers are located at offsets 28h and 30h, respectively, in the S5933 PCI Operation Registers. These are written with the a DMA transfer byte count of up to 64 Mbytes. The transfer count registers are decremented by four after each completed data phase. The transfer count registers do not reset to their initial value after reaching zero, the PCI host must reprogram them.

Although S5933 DMA transfer must begin on double-word boundaries, the transfer count does not have to be a multiple of four bytes. When the transfer count decrements below four, only the byte enable corresponding to the number of bytes left are asserted. For example, if a transfer count of 10 was programmed into one of the transfer count registers, two data transfers would complete with all byte enables asserted (8 bytes). The final data transfer would have only BE0# and BE1# asserted, transferring the final two bytes.

For add-on initiated DMA transfers, the transfer counts are enabled or disabled through the Add-on General Control/Status Register (AGCSTS), bit 28. The transfer counts for read and write transfers cannot be individually enabled. This may be useful in applications where the amount of data to be transferred is not known. If transfer counts are disabled, the S5933 continues to transfer data according to the conditions listed above, but transfer counts are ignored.

3.4 FIFO Management Schemes

The S5933 provides flexibility in how the FIFO is managed for DMA transfers. The FIFO management scheme determines when the S5933 requests the PCI bus (asserts REQ#). The most efficient way to utilize the capabilities of the PCI bus is with burst transfers. Requesting the PCI bus every time the FIFO contains a single double-word is an inefficient use of the bus, and limits the performance of other PCI devices within a system. It is more desirable request the bus when multiple operations are required, allowing the S5933 to perform a burst transfer.

The management scheme is configurable for the PCI to add-on and add-on to PCI FIFOs (and may be different for each). Bus mastering must be enabled for the management scheme to apply (via the MCSR enable bits or AMREN/AMWEN). The FIFO management option is programmed through the Bus Master Control/Status Register (MCSR).

For the PCI to add-on FIFO (DMA reads), there are two options. The FIFO can be programmed to request the bus when any FIFO location is empty or only when four or more locations are empty. After the S5933 is granted control of the PCI bus, the management scheme does not apply. The device continues to read as long as there is an open FIFO location. For DMA read transfers, the S5933 maintains control of the PCI bus until one of the following events:

- The read transfer count (MRTC) reaches zero
- Bus mastering is disabled (with the MSCR enable bit or AMREN)
- Another master requests the bus and the Latency Timer is expired
- The PCI target aborts the transfer
- The PCI to add-on FIFO becomes full

For the add-on to PCI FIFO (DMA writes), there are two management options. The FIFO can be programmed to request the bus when any FIFO location is full or only when four or more locations are full. After the S5933 is granted control of the PCI bus, the management scheme does not apply. The device continues to write as long as there is data in the FIFO. For DMA write transfers, the S5933 maintains control of the PCI bus until one of the following events:

- The write transfer count (MWTC) reaches zero
- Bus mastering is disabled (with the MSCR enable bit or AMWEN)

- Another master requests the bus and the Latency Timer is expired
- The PCI target aborts the transfer
- The add-on to PCI FIFO becomes empty

There are two special cases for the add-on to PCI FIFO management scheme. The first case is when the FIFO is programmed to request the PCI bus only when four or more locations (16 bytes) are full, but the transfer count is less than 16 bytes. In this situation, the FIFO ignores the management scheme and finishes transferring the data. The second case is when the S5933 is configured for add-on initiated bus mastering. In this situation, the FIFO management scheme must be set to request the PCI bus when one or more locations are full.

3.5 S5933 DMA Channel Priority

In many applications, the S5933 performs both DMA read and write transfers. This requires a priority scheme be implemented between the two FIFOs. If the FIFO management condition for initiating a PCI read and a PCI write are both met, a method must exist to determine which transfer is performed first.

Bits D12 and D8 in the Bus Master Control/Status Register (MCSR) control the read and write DMA channel priority, respectively. If these bits are both set or both clear, priority alternates, beginning with read transfers. If the read priority is set and the write priority is clear, read cycles take priority. If the write priority is set and the read priority is clear, write cycles take priority. Priority arbitration is only done when neither FIFO has control of the PCI bus (the PCI to add-on FIFO never interrupts an add-on to PCI FIFO transfer in progress and vice-versa).

3.6 S5933 DMA Interrupts

The S5933 can generate interrupts under the following conditions: the read transfer count reaches zero, the write transfer count reaches zero, or an error occurs on the PCI bus during a DMA transfer.

Which interface (PCI bus or add-on bus) receives the interrupt is determined by which side initiated the DMA transfer. If PCI initiated DMA transfers are used, a PCI INTA# interrupt is generated when an interrupt condition is met. If add-on initiated DMA transfers are used, IRQ# is generated to the add-on interface. Interrupts are optional and may be disabled.

3.6.1 Transfer Count Interrupts

Transfer count interrupts may come from two sources: read transfer count and/or write transfer count. One or both interrupts may be enabled, or both may be disabled. A read transfer count interrupt is generated when the Master Read Transfer Count Register (MRTC) decrements to zero. A write transfer count interrupt is generated when the Master Write Transfer Count Register (MWTC) decrements to zero. These registers decrement when a PCI transfer completes successfully.

If add-on initiated DMA transfers are used with transfer counts disabled, these interrupt sources are disabled.

3.6.2 PCI Bus Error Condition Interrupts

In some situations, the PCI bus may signal an error condition during an S5933 DMA transfer. These error conditions include a target abort or master abort on the PCI bus. If one of these conditions exists, it is important to notify the device which initiated the transfer that it cannot complete successfully. Interrupts on PCI error conditions are only enabled if one or both of the transfer count interrupts are enabled. There is no individual enable bit for PCI error interrupts.

A PCI target abort indicates an error condition where no number of retries to the target will result in a successful data transfer. A PCI master abort occurs when a PCI bus master (the S5933, in this case) attempts to access a PCI target which is either non-existent or disabled. In either of these situations, the device which set up the DMA transfer is notified with an interrupt (either INTA# or IRQ#).

3.6.3 Controlling S5933 DMA Interrupts

For PCI initiated DMA transfers, interrupts are enabled through the S5933 Interrupt Control Status Register (INTCSR). This register is located at offset 38h in the S5933 PCI Operation Registers. INTCSR is also accessed during interrupt service routines to determine the interrupt source and clear the interrupt. The following INTCSR bits relate to DMA Operations:

Bit 14 Enable Interrupt on Write Transfer Complete. If set, INTA# is generated when MWTC decrements to zero during a DMA transfer.

Bit 15 Enable Interrupt on Read Transfer Complete. If set, INTA# is generated when MRTC decrements to zero during a DMA transfer.

Bit 18 Write Transfer Complete Interrupt. When set, this bit indicates MWTC has decremented to zero and INTA# has been asserted. Writing a one to this bit clears the interrupt source and deasserts INTA#. Writing a zero to this bit has no effect.

Bit 19 Read Transfer Complete Interrupt. When set, this bit indicates MRTC has decremented to zero and INTA# has been asserted. Writing a one to this bit clears the interrupt source and deasserts INTA#. Writing a zero to this bit has no effect.

Bit 20 Master Abort Interrupt. When set, this bit indicates that the S5933 had to perform a master abort and INTA# has been asserted. Writing a one to this bit clears the interrupt source and deasserts INTA#. Writing a zero to this bit has no effect.

Bit 21 Target Abort Interrupt. When set, this bit indicates that the S5933 received a target abort and INTA# has been asserted. Writing a one to this bit clears the interrupt source and deasserts INTA#. Writing a zero to this bit has no effect.

For add-on initiated DMA transfers, interrupts are enabled through the S5933 Add-on Interrupt Control/Status Register (AINT). This register is located at offset 38h in the S5933 Add-on Operation Registers. AINT is also accessed during interrupt service routines to determine the interrupt source and clear the interrupt. The following AINT bits relate to DMA Operations:

Bit 14 Enable Interrupt on Write Transfer Complete. If set, IRQ# is generated when MWTC decrements to zero during a DMA transfer.

Bit 15 Enable Interrupt on Read Transfer Complete. If set, IRQ# is generated when MRTC decrements to zero during a DMA transfer.

Bit 18 Write Transfer Complete Interrupt. When set, this bit indicates MWTC has decremented to zero and IRQ# has been asserted. Writing a one to this bit clears the interrupt source and deasserts IRQ#. Writing a zero to this bit has no effect.

Bit 19 Read Transfer Complete Interrupt. When set, this bit indicates MRTC has decremented to zero and IRQ# has been asserted. Writing a one to this bit clears the interrupt source and deasserts IRQ#. Writing a zero to this bit has no effect.

Bit 21 Bus Master Error Interrupt. When set, this bit indicates that the S5933 had to perform a master abort or received a target abort and IRQ# has been asserted. Writing a one to this bit clears the interrupt source and deasserts IRQ#. Writing a zero to this bit has no effect.

Multiple PCI devices may be assigned to a single hardware interrupt by the host. PCI device drivers are, therefore, required to determine if the current interrupt was generated by the device it services. If not, it must pass control, or “chain” the previous interrupt handler to service the interrupt.

Each application’s code must install its own interrupt vector. To do this, the Interrupt Line Configuration Register is read. A value between 0 and 15 is returned, corresponding to a PC hardware IRQ number. This must be translated into a software interrupt number. The following table shows the conversions for a PC:

3.6.4 PCI Interrupt Considerations

If the S5933 is configured to generate PCI bus interrupts (INTA#) for DMA transfer counts and PCI bus error conditions, considerations must be made for the interrupt handler. The interrupt vector must be obtained, and because hardware interrupts may be shared, interrupt handlers may have to be “chained.”

3.6.4.1 Enabling PCI Interrupts

The Interrupt Pin (INTFIN) PCI Configuration Register may be loaded out of non-volatile memory offset 7Dh by the S5933 at reset. The value loaded into this register identifies which PCI interrupt: INTA#, INTB#, INTC#, or INTD# is used. The default value is 01h, identifying INTA#. For the S5933, the INTA# output should be connected to the PCI bus INTA# pin. If PCI interrupts are not required, this register may be initialized to 00h (interrupts disabled) or ignored.

The four PCI interrupts are mapped within the system chipset to standard PC IRQ numbers (0-15). The 82C59A compatible interrupt controllers (two cascaded controllers) each have an interrupt mask register. The master mask register, located at system I/O location 21h controls the masking of IRQ0-7, and the slave mask register, located at system I/O location 1Ah controls the masking of IRQ8-15. Application software must make sure the S5933 interrupt line (indicated by the PCI Interrupt Line Register described in Section 3.6.4.2) is unmasked. If the interrupt is masked, the handler never executes.

3.6.4.2 PCI Host Interrupt Handlers

The S5933 Interrupt Line (INTLN) PCI Configuration Register is loaded out of non-volatile memory offset 7Ch by the S5933 at reset. The value loaded into the register is a hardware interrupt number for the host interrupt controller (IRQ0 to 15). This value may be used by the host, or the BIOS may overwrite it with its own value.

Hardware Interrupt	Software Interrupt
IRQ0	08h
IRQ1	09h
IRQ2	0Ah
IRQ3	0Bh
IRQ4	0Ch
IRQ5	0Dh
IRQ6	0Eh
IRQ7	0Fh
IRQ8	70h
IRQ9	71h
IRQ10	72h
IRQ11	73h
IRQ12	74h
IRQ13	75h
IRQ14	76h
IRQ15	77h

Once the software interrupt number is calculated, the previous interrupt handler’s vector can be read and stored. The new interrupt vector is then installed. In this manner, numerous devices within a system can share a single hardware interrupt.

Each application's interrupt handler must first check the source of the interrupt. For S5933 applications, this is done by reading the S5933 Interrupt Control/Status Register. The status of all possible S5933 PCI interrupt sources is indicated by this register. If the status bits indicate that no interrupts were generated by the S5933, the handler must call the previous interrupt handler whose vector it replaced. The previous interrupt handler then performs a similar task for the device it services, and this process continues until the device which generated the interrupt is found.

If the interrupt handler determines that the source of the interrupt was the S5933, the interrupt source must be cleared through the Interrupt Control/Status Register (INTCSR). The handler then performs whatever tasks are necessary to service the interrupt (such as rewriting address or transfer count registers). Finally, the handler must clear the 83C59A interrupt controller 'in-service' bit. This bit is set when the host processor acknowledges the interrupt and jumps to the interrupt service routine. A specific end-of-interrupt (EOI) is used to clear this bit. If a PCI device is mapped to hardware interrupts IRQ8 to IRQ15, two EOI commands must be issued. One EOI must be issued for the slave 82C59A which supports IRQ8-15. A second EOI is required for the master 82C59A. The second EOI is a specific EOI for IRQ2 because the slave interrupt controller in a PC is cascaded into the IRQ2 input of the master interrupt controller. Without the end-of-interrupt sequence, the interrupt controllers will not recognize further interrupts from that source.

4.0 PCI DMA PERFORMANCE FACTORS

There are a number of factors which determine DMA performance on the PCI bus. The clock speed and data bus width are important in determining maximum bus bandwidth. The most important factor in DMA performance is traffic on the PCI bus. As the number of PCI devices which require access to the bus increases, the bandwidth available to each individual device decreases.

4.1 PCI Bus Arbitration

Each device on the PCI bus has a dedicated REQ#/GNT# pair which are connected to the system bus arbiter. When a device asserts REQ#, it indicates a data transfer is required. The transfer may be a single data phase or a burst. The bus arbiter asserts GNT# to the device to indicate that it may now perform the transfer.

The bus arbiter may remove GNT# from a PCI bus master on any PCI clock. The current transaction completes, and the PCI master gives up control of the bus. GNT# may already be asserted to the next master, but it is not allowed to drive the PCI bus until IRDY# and FRAME# are deasserted, indicating the bus is idle. If the original bus master has more data to be transferred, it may keep REQ# asserted, but must wait for GNT# again.

The priority scheme used to determine which PCI device controls the bus at a given time is determined by the system. The PCI specification requires a few extra Configuration Registers within PCI bus master devices. During system initialization, these registers are read to determine each device's requirements (if any). Based on these, a priority scheme can be defined which is unique to that system. Ideally, the arbitration scheme gives priority to devices with higher bandwidth requirements, but does not prevent other PCI bus masters from gaining control of the bus.

4.2 PCI Bus Access Latency

There are three components to latency on the PCI bus. The total latency is measured from the time a bus master requests the bus (asserts REQ#) to when the target of the transfer completes the transfer (asserts TRDY#). Each of these components is described in the following sections.

4.2.1 Arbitration Latency

Once a PCI device asserts REQ#, it must wait for the bus arbiter to assert GNT#. This delay is called arbitration latency. This is determined by the priority scheme used by the bus arbiter, the relative priority of the device requesting the bus, and the amount of activity on the PCI bus.

4.2.2 Bus Acquisition Latency

Once a PCI device receives GNT# from the arbiter, it must wait for the current bus master to complete its transaction (indicated by FRAME# and IRDY# deasserted) before driving the PCI bus. Bus acquisition latency is the time from when GNT# is received to when FRAME# is asserted by a particular master. This is determined by the length of the current bus master's transfer.

4.2.3 Target Latency

After a PCI device begins a bus cycle, it must wait for the target of the transfer to complete the cycle (by asserting TRDY#). Target latency is the time from when FRAME# is asserted to when TRDY# is asserted. This is unique for each target and depends

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

on the function of the target (main memory, VGA controller, network interface, etc.).

4.3 PCI Configuration Registers

A PCI device with bus mastering capabilities may implement up to three additional configuration registers. These registers indicate the requirements of a particular PCI device. These are read during system initialization and may be used to define a bus master priority scheme. The S5933 implements all of these registers. They can be boot loaded from an external, non-volatile memory device at reset.

4.3.1 Latency Timer Register

If a bus master is capable of PCI bursts of more than two data phases, this register is required. The Latency Timer Register defines the number of PCI clocks that the S5933 is guaranteed control of the bus. The Latency Timer Register is shown in Figure 1. The value programmed in this register is decremented once every 8 PCI clocks after the S5933 asserts FRAME#.

The default value for this register is 00h, indicating that the S5933 has no minimum transfer length requirement. The system can overwrite the Latency Timer Register with any value. This prevents an individual PCI master from controlling the bus for an extremely long period.

If no other PCI master has requested the bus, the Latency Timer value does not matter (even if it has expired). The S5933 transfers data until the transaction is complete. If another bus master requests the bus and receives GNT# while the S5933 is transferring data, three situations can occur:

- 1) If the Latency Timer has not expired, and the S5933 completes its transaction before it expires, the transaction completes normally.
- 2) If the Latency Timer has not expired, the S5933 transfers data until the latency timer expires where it completes the current data phase, terminates the transaction and gives up control of the bus
- 3) If the Latency Timer has already expired, the S5933 completes the current data phase, terminates the transaction, and gives up control of the bus.

4.3.2 Minimum Grant Register

This register defines how long of a burst period the device typically requires (in units of 250 ns). This read-only register is for information only. It may be used by the system, in conjunction with the Maximum Latency register, to define a PCI bus arbitration scheme (if the bus arbiter is programmable).

A value of zero (default for the S5933) indicates there is no strict requirement for burst length. The Minimum Grant Register is shown in Figure 2.

Figure 1. Latency Timer Register Definition

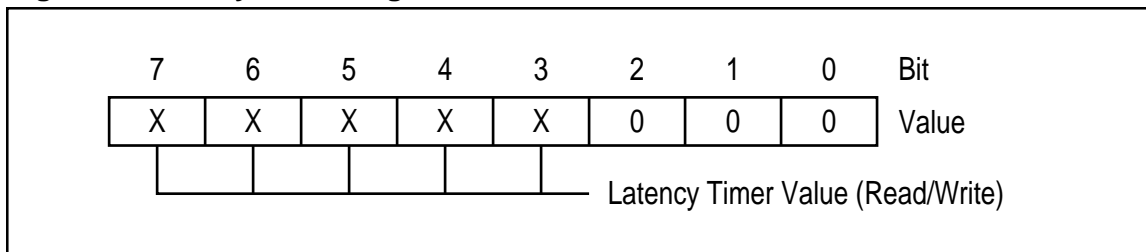


Figure 2. Minimum Grant Register Definition

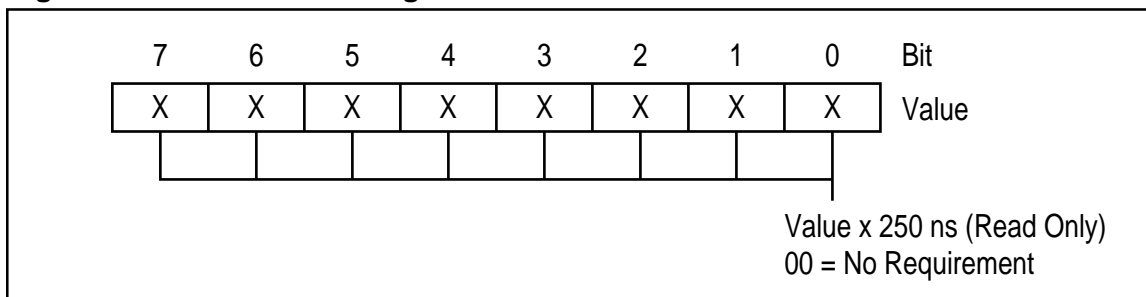
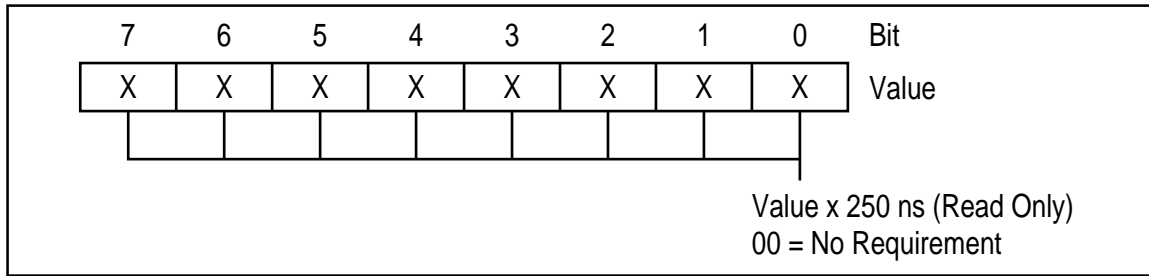


Figure 3. Maximum Latency Register Definition



4.3.3 Maximum Latency Register

This register defines how often the device typically needs PCI bus access (in units of 250 ns). This read-only register is for information only. It may be used by the system, in conjunction with the Minimum Grant register, to define a PCI bus arbitration scheme (if the bus arbiter is programmable).

A value of zero (default for the S5933) indicates there is no strict requirement for access to the PCI bus. The Maximum Latency Register is shown in Figure 3.

4.4 Sample PCI Performance Calculation

The maximum theoretical bandwidth for the PCI bus at 33 MHz is 132 Mbytes per second. The actual bandwidth is less. Achievable bandwidth depends of factors such as bus utilization by other masters, the bus arbitration scheme, and burst length limitations of both the PCI initiator and target. The following examples show bandwidth calculations for two situations: a DMA transfer from the S5933 to main DRAM memory, and a DMA transfer to another S5933 add-in board. These performance calculations are based on the current PCI systems. As chipsets, memory controllers and other PCI devices evolve, performance will increase, accordingly.

4.4.1 S5933 Burst to Main DRAM Memory

Table 1 shows a situation where an add-on device can write data to the S5933 FIFO at a rate of one double-word (32-bits) every 60 ns. The target for the DMA transfer is main DRAM memory. The PCI memory controller allows 4 data phase write bursts. The fifth data phase receives a target requested retry. The PCI Specification requires that a PCI initiator deassert REQ# for two clocks. For this example, a 4 clock latency period is used from the reassertion of REQ# by the S5933 to the reassertion of GNT# by the PCI bus arbiter.

The sequence shown assumes the following initial conditions:

- Master Write Address Register (MWAR) = 100000h
- Master Write Transfer Count Register (MWTC) is disabled
- Bus mastering for the S5933 is already enabled
- The Add-on to PCI FIFO is full (8 dwords)
- The PCI bus arbiter has just asserted GNT# to the S5933

Table 1. Sample S5933 Burst to Main Memory

Time	PCI Bus Activity	Add-on Bus Activity	FIFO Status	REQ#	GNT#
30 ns	Address = 100000h	Idle	8 dwords	0	0
60 ns	Data Transfer 1	Wait State	7 dwords	0	0
90 ns	Data Transfer 2	Write FIFO	7 dwords	0	0
120 ns	Data Transfer 3	Wait State	6 dwords	0	0
150 ns	Data Transfer 4	Write FIFO	6 dwords	0	0
180 ns	Target Disconnect	Wait State	6 dwords	0	0
210 ns	Idle (or other master)	Write FIFO	7 dwords	1	1
240 ns	Idle (or other master)	Wait State	7 dwords	1	1
270 ns	Idle (or other master)	Write FIFO	8 dwords	0	1
300 ns	Idle (or other master)	Idle	8 dwords	0	1
330 ns	Idle (or other master)	Idle	8 dwords	0	1
360 ns	Idle (or other master)	Idle	8 dwords	0	1
390 ns	Idle (or other master)	Idle	8 dwords	0	0
420 ns	Address = 100010h	Idle	8 dwords	0	0
450 ns	Data Transfer 5	Wait State	7 dwords	0	0

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

The sequence would repeat every 390 ns with a 33 MHz PCI bus clock. Four Dwords, or 16 bytes have been transferred to main memory in during this time. This would average out to about 40 MBytes/sec.

The major factor in the calculation is how long it takes for the S5933 to regain control of the PCI bus after the main memory controller disconnects after the fourth data phase. This depends on the number of PCI devices using the bus. Because the transfer is to main memory, the S5933 must compete for bus bandwidth with numerous other devices that are usually on the Primary PCI bus. A typical system may have the host CPU, a VGA controller, an ethernet interface, and a SCSI or IDE drive sharing the Primary PCI bus. If a rotational priority scheme were implemented by the bus arbiter, it could be significantly more than 4 PCI clocks before the S5933 regains control of the bus.

4.4.2 S5933 Burst to Another S5933 PCI Card

Table 2 shows a situation where an add-on device can write data to the S5933 FIFO at a rate of one double-word (32-bits) every 60 ns. The target for the DMA transfer is the pass-thru interface of another S5933 device which can accept data at a rate of one double-word every 30 ns. The S5933 pass-thru interface does not disconnect from a burst write unless the add-on has not read the pass-thru data register within 16 PCI clocks for the first data phase or 8 PCI clocks on successive data phases. In this situation, the initiator S5933 deasserts request because it runs out of data. The target S5933 never disconnects in this situation.

The sequence shown assumes the following initial conditions:

- Master Write Address Register (MWAR) = 100000h
- Master Write Transfer Count Register (MWTC) is disabled
- Bus mastering for the S5933 is already enabled
- The Add-on to PCI FIFO is full (8 dwords)
- The PCI bus arbiter has just asserted GNT# to the S5933

In this example, it is assumed that the AMWEN signal has been deasserted when the FIFO goes empty at 480 ns. This allows the FIFO to refill before another transfer is initiated. This is the most efficient way to utilize the PCI bus. Sending a signal, long burst is better than sending numerous short bursts because less overall time is spent arbitrating for PCI bus control.

The FIFO would be full again at 690 ns. If AMWEN is reasserted when the FIFO is full, and a 4 clock latency is assumed (as with the previous example), the next address phase begins at 810 ns. The process would repeat from there. This results in 15 dwords (60 bytes) being transferred every 810 ns. The results in an average 74 MByte/sec. transfer rate. Again, this is heavily dependent on PCI bus utilization by other devices. Unless the two cards in question share an isolated PCI bus on the secondary side of a PCI to PCI bridge, bandwidth would likely be less than this.

Table 2. Sample S5933 Burst to Another S5933 Device

Time	PCI Bus Activity	Add-on Bus Activity	FIFO Status	REQ#	GNT#
30 ns	Address = 100000h	Idle	8 dwords	0	0
60 ns	Data Transfer 1	Wait State	7 dwords	0	0
90 ns	Data Transfer 2	Write FIFO	7 dwords	0	0
120 ns	Data Transfer 3	Wait State	6 dwords	0	0
150 ns	Data Transfer 4	Write FIFO	6 dwords	0	0
180 ns	Data Transfer 5	Wait State	5 dwords	0	0
210 ns	Data Transfer 6	Write FIFO	5 dwords	0	0
240 ns	Data Transfer 7	Wait State	4 dwords	0	0
270 ns	Data Transfer 8	Write FIFO	4 dwords	0	0
300 ns	Data Transfer 9	Wait State	3 dwords	0	0
330 ns	Data Transfer 10	Write FIFO	3 dwords	0	0
360 ns	Data Transfer 11	Wait State	2 dwords	0	0
390 ns	Data Transfer 12	Write FIFO	2 dwords	0	0
420 ns	Data Transfer 13	Wait State	1 dword	0	0
450 ns	Data Transfer 14	Write FIFO	1 dword	0	0
480 ns	Data Transfer 15	Wait State	0 dwords	1	1
510 ns	Idle (or other Master)	Write FIFO	1 dword	1	1

5.0 DMA SOFTWARE SUPPORT

DMA transfers with the S5933 depend mostly on hardware. Most of the design is the interface on the add-on card which fills and empties the FIFO. There is some software support required for DMA transfers. Address and transfer count registers must be loaded, the FIFOs must be configured, and interrupts (if used) must be enabled and serviced. The following sections provide an overview of what actions are required by software for S5933 DMA operations.

5.1 PCI Initiated DMA Transfers

For PCI initiated DMA transfers, the PCI host CPU (Pentium™, Alpha™, etc.) sets up the S5933 to perform bus master transfers. The following tasks must be completed to setup FIFO bus mastering:

- 1) Define interrupt capabilities.** The PCI to add-on and/or add-on to PCI FIFO can generate a PCI interrupt to the host when the transfer count reaches zero.

INTCSR	Bit 15	Enable Interrupt on read transfer count equal zero
--------	--------	--

INTCSR	Bit 14	Enable Interrupt on write transfer count equal zero
--------	--------	---

- 2) Reset FIFO flags.** This may not be necessary, but if the state of the FIFO flags is not known, they should be initialized.

MCSR	Bit 26	Reset add-on to PCI FIFO flags
------	--------	--------------------------------

MCSR	Bit 25	Reset PCI to add-on FIFO flags
------	--------	--------------------------------

- 3) Define FIFO management scheme.** These bits define what FIFO condition must exist for the PCI bus request (REQ#) to be asserted by the S5933.

MCSR	Bit 13	PCI to add-on FIFO management scheme
------	--------	--------------------------------------

MCSR	Bit 9	Add-on to PCI FIFO management scheme
------	-------	--------------------------------------

- 4) Define FIFO priority scheme.** These bits determine which FIFO has priority if both meet the defined condition to request the PCI bus. If these bits are the same, priority alternates, with read accesses occurring first.

MCSR	Bit 12	Read vs. write priority
------	--------	-------------------------

MCSR	Bit 8	Write vs. read priority
------	-------	-------------------------

- 5) Define transfer source/destination address.** These registers are written with the first address that is to be accessed by the S5933. These address registers are updated after each access to indicate the next address to be accessed. Transfers must start on DWORD boundaries.

MWAR	All	Bus master write address
------	-----	--------------------------

MRAR	All	Bus master read address
------	-----	-------------------------

- 6) Define transfer byte counts.** These registers are written with the number of bytes to be transferred. The transfer count does not have to be a multiple of four bytes. These registers are updated after each transfer to reflect the number of bytes remaining to be transferred.

MWTC	All	Write transfer byte count
------	-----	---------------------------

MRTC	All	Read transfer byte count
------	-----	--------------------------

- 7) Enable Bus Mastering.** Once steps 1-6 are completed, the FIFO may operate as a PCI bus master. Read and write bus master operations may be independently enabled or disabled.

MCSR	Bit 14	Enable PCI to add-on FIFO bus mastering
------	--------	---

MCSR	Bit 10	Enable add-on to PCI FIFO bus mastering
------	--------	---

It is recommended that bus mastering be enabled as the last step. Some applications may choose to leave bus mastering enabled and start transfers by writing a non-zero value to the transfer count registers. This also works, provided the entire 26-bit transfer count is written at once. If transfer count interrupts are enabled, they must be enabled after the transfer count(s) are written. If interrupts are enabled and the transfer count is zero, an interrupt occurs immediately.

5.2 Servicing a PCI Initiated DMA Transfer Interrupt

If interrupts are enabled, a host interrupt service routine is also required. The service routine determines the source of the interrupt and resets the interrupt. The source of the interrupt is indicated in the PCI Interrupt Control/Status Register (INTCSR). Typically, the interrupt service routine is used to set up the next transfer by writing a new address and transfer count value, but some applications may also require other actions. If read transfer or write transfer complete interrupts are enabled, master and target abort interrupts are automatically enabled. Writing a one to these bits clears the corresponding interrupt.

INTCSR	Bit 21	Target abort caused interrupt
INTCSR	Bit 20	Master abort caused interrupt
INTCSR	Bit 19	Read transfer complete caused interrupt
INTCSR	Bit 18	Write transfer complete caused interrupt

5.3 Add-on Initiated DMA Transfers

For add-on initiated DMA transfers, the add-on sets up the S5933 to perform bus master transfers. The following tasks must be completed to setup FIFO bus mastering:

- 1) **Define transfer count abilities.** For add-on initiated bus mastering, transfer counts may be either enabled or disabled. Transfer counts for read and write operations cannot be individually enabled.

AGCSTS	Bit 28	Enable transfer count for read and write bus master transfers
--------	--------	---

- 2) **Define interrupt capabilities.** The PCI to add-on and/or add-on to PCI FIFO can generate an interrupt to the add-on when the transfer count reaches zero (if transfer counts are enabled).

AINT	Bit 15	Enable interrupt on read transfer count equal zero
AINT	Bit 14	Enable interrupt on write transfer count equal zero

- 3) **Reset FIFO flags.** This may not be necessary, but if the state of the FIFO flags is not known, they should be initialized.

AGCSTS	Bit 26	Reset add-on to PCI FIFO flags
--------	--------	--------------------------------

AGCSTS	Bit 25	Reset PCI to add-on FIFO flags
--------	--------	--------------------------------

- 4) **Define FIFO management scheme.** These bits define what FIFO condition must exist for the PCI bus request (REQ#) to be asserted by the S5933. This must be programmed through the PCI interface.

MCSR	Bit 13	PCI to add-on FIFO management scheme
------	--------	--------------------------------------

MCSR	Bit 9	Add-on to PCI FIFO management scheme
------	-------	--------------------------------------

- 5) **Define FIFO priority scheme.** These bits determine which FIFO has priority if both meet the defined condition to request the PCI bus. If these bits are the same, priority alternates, with read accesses occurring first. This must be programmed through the PCI interface.

MCSR	Bit 12	Read vs. write priority
------	--------	-------------------------

MCSR	Bit 8	Write vs. read priority
------	-------	-------------------------

- 6) **Define transfer source/destination address.** These registers are written with the first address that is to be accessed by the S5933. These address registers are updated after each access to indicate the next address to be accessed. Transfers must start on DWORD boundaries.

MWAR	All	Bus master write address
------	-----	--------------------------

MRAR	All	Bus master read address
------	-----	-------------------------

- 7) **Define transfer byte counts.** These registers are written with the number of bytes to be transferred. The transfer count does not have to be a multiple of four bytes. These registers are updated after each transfer to reflect the number of bytes remaining to be transferred. If transfer counts are disabled, these registers do not need to be programmed.

MWTC	All	Write transfer byte count
------	-----	---------------------------

MRTC	All	Read transfer byte count
------	-----	--------------------------

- 8) **Enable Bus Mastering.** Once steps 1-7 are completed, the FIFO may operate as a PCI bus master. Read and write bus master operation may be independently enabled or disabled. The AMREN and AMWEN inputs control bus master enabling for add-on initiated bus mastering. The MCSR bus master enable bits are ignored for add-on initiated bus mastering.

It is recommended that bus mastering be enabled as the last step. Some applications may choose to leave bus mastering enabled (AMREN and AMWEN asserted) and start transfers by writing a non-zero value to the transfer count registers (if they are enabled). This works, provided the entire 26-bit transfer count is written at once. If transfer count interrupts are enabled, they must be enabled after the transfer count(s) are written. If interrupts are enabled and the transfer count is zero, an interrupt occurs immediately.

5.4 Servicing an Add-on Initiated DMA Transfer Interrupt

If interrupts are enabled, an add-on CPU interrupt service routine is also required. The service routine determines the source of the interrupt and resets the interrupt. The source of the interrupt is indicated in the Add-on Interrupt Control Register (AINT). Typically, the interrupt service routine is used to set up the next transfer by writing a new address and transfer count value (if enabled), but some applications may also require other actions. If read transfer or write transfer complete interrupts are enabled, the master/target abort interrupt is automatically enabled. Writing a one to these clears the corresponding interrupt.

AINT	Bit 21	Master/target abort caused interrupt
AINT	Bit 19	Read transfer complete caused interrupt
AINT	Bit 18	Write transfer complete caused interrupt

6.0 SAMPLE S5933 DMA SUPPORT CODE

The following section is a sample program written in C-language to setup the S5933 for DMA operations. The code is written for PCI initiated bus mastering using transfer count interrupts. The code is written for an x86 compatible PCI platform, but could be modified to support other host processors.

The code has been compiled using Borland C/C++ Version 4.5. Because 32-bit registers are used within the code, code must be compiled to generate 386 code (otherwise, 32-bit register mnemonics such as `_EAX` are not recognized).

```

#include <dos.h>
#include <stddef.h>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "amcc.h"

/*****
/*  A2P = Add-on to PCI FIFO
/*  P2A = PCI to Add-on FIFO
*****/

/* Transfer Count Interrupt Enables */
#define EN_READ_TC_INT      0x00008000L
#define EN_WRITE_TC_INT     0x00004000L

/* FIFO Flag Reset */
#define RESET_A2P_FLAGS     0x04000000L
#define RESET_P2A_FLAGS     0x02000000L

/* FIFO Management Scheme */
#define A2P_REQ_AT_4FULL     0x00000200L
#define P2A_REQ_AT_4EMPTY   0x00002000L

/* FIFO Relative Priority */
#define A2P_HI_PRIORITY      0x00000100L
#define P2A_HI_PRIORITY      0x00001000L

/* Enable Transfer Count */
#define EN_TCOUNT          0x10000000L

/* Enable Bus Mastering */
#define EN_A2P_TRANSFERS     0x00000400L
#define EN_P2A_TRANSFERS     0x00004000L

/* Identify S5933 Interrupt Sources */
#define ANY_S5933_INT        0x80
#define READ_TC_INT          0x08
#define WRITE_TC_INT         0x04
#define MASTER_ABORT_INT     0x10
#define TARGET_ABORT_INT     0x20
#define BUS_MASTER_INT       0x20

/* Global Variable Definition */
byte  interrupt_line;
word  op_reg_base_address;
void(interrupt *oldhandler)(void);

```

```

/*****
/*      MAIN Code Segment      */
/*****

void main()
{
    void    setup_pci_dma(void);
    void    setup_int_vect(byte bus_num,byte dev_func);
    word    vendor_id, device_id, index;

    byte    bus_num, dev_func;
    int     bios_present;

/* AMCC Default Vendor/Device ID's */
    vendor_id = 0x10E8;
    device_id = 0x4750;
    index = 0;

/* Disable Host Interrupts */
    disable();

/* Look for a valid PCI BIOS */
    if(pci_bios_present(NULL,NULL,NULL)==SUCCESSFUL){
        bios_present=TRUE;
        printf("PCI BIOS Found\n\n");
    }
    else{
        printf("PCI BIOS not present\n\n");
        bios_present=FALSE;
    }

/* If the BIOS is present, look for the AMCC device */
    if(bios_present){
        if(find_pci_device(device_id,vendor_id,index,&bus_num,
            &dev_func)==SUCCESSFUL){
            printf("AMCC Device Found: Bus=%d Device=%d Function=%d\n\n",
                bus_num, (dev_func>>3), (dev_func&0x7));
        }
        else{
            printf("AMCC Device Not Found\n\n");
        }
    }

/* Find the physical location of the S5933 in I/O space */
    read_configuration_word(bus_num,dev_func,
        PCI_CS_BASE_ADDRESS_0,&op_reg_base_address);

/* Mask Lower 2 bits of BADR0 */
    op_reg_base_address = (op_reg_base_address & 0xFFFC);

/* Call routine to install interrupt vector */
    setup_int_vect(bus_num,dev_func);

/*Enable hardware interrupts */
    enable();

/* Call routine to setup PCI initiated bus mastering */
    setup_pci_dma();
}

```

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

```

/*****
/*      Function:      setup_pci_dma                      */
/*      Purpose:      Configure the S5933 for PCI initiated DMA      */
/*      Inputs:       None                                          */
/*      Outputs:      None                                          */
/*                                                              */
/* The following function assumes the S5933 is configured          */
/* for PCI initiated DMA transfers and sets up the DMA channels    */
*****/

void setup_pci_dma(void)
{
    char src[20], dest[20], rtc[20], wtc[20];
    dword source, destination, temp = 0;
    dword readtc, writetc;

    /* Read in source,destination and transfer counts */
    printf("Input address values in hex format: 0x...\n\n");
    printf("Input transfer count values in decimal\n\n");
    printf("Read from address: ");
    gets(src);
    source=strtoul(src,NULL,16);
    printf("%lX\n",source);

    printf("Write to address: ");
    gets(dest);
    destination=strtoul(dest,NULL,16);
    printf("%lX\n",destination);

    printf("Read byte count: ");
    gets(rtc);
    readtc=strtoul(rtc,NULL,10);
    printf("%d\n",readtc);

    printf("Write byte count: ");
    gets(wtc);
    writetc=strtoul(wtc,NULL,10);
    printf("%d\n",writetc);

    outpd(op_reg_base_address+AMCC_OP_REG_MWAR, destination);
    outpd(op_reg_base_address+AMCC_OP_REG_MRAR, source);
    outpd(op_reg_base_address+AMCC_OP_REG_MWTC, writetc);
    outpd(op_reg_base_address+AMCC_OP_REG_MRTC, readtc);

    /* Enable Transfer Count interrupts */
    temp = inpd(op_reg_base_address+AMCC_OP_REG_INTCSR);
    outpd(op_reg_base_address+AMCC_OP_REG_INTCSR,temp|
        EN_READ_TC_INT|
        EN_WRITE_TC_INT);

    /* Enable Bus Mastering */
    temp = inpd(op_reg_base_address+AMCC_OP_REG_MCSR);
    outpd(op_reg_base_address+AMCC_OP_REG_MCSR, temp|RESET_A2P_FLAGS|
        RESET_P2A_FLAGS|
        A2P_HI_PRIORITY|
        P2A_HI_PRIORITY|
        EN_A2P_TRANSFERS|
        EN_P2A_TRANSFERS);
}

```

```

/*****
/*      Function:      handler                               */
/*      Purpose:      Check interrupt source and service S5933 int's */
/*      Inputs:       None                                   */
/*      Outputs:      None                                   */
*****/

void interrupt_handler(void)
{
    byte    status;

    status = inportb(op_reg_base_address+AMCC_OP_REG_INTCSR+2);

    if((status & ANY_S5933_INT) != 0){
        /* Disable bus mastering */
        outportb(op_reg_base_address+AMCC_OP_REG_MCSR+1,0x11);

        /* Identify AMCC Interrupt Source(s) */
        /* AMCC Hardware Interrupt Source */
        if((status & READ_TC_INT) != 0)
            /* Read TC Interrupt Code Here */
            if((status & WRITE_TC_INT) != 0)
                /* Write TC Interrupt Code Here */
                if((status & MASTER_ABORT_INT) != 0)
                    /* Master Abort Interrupt Code Here */
                    if((status & TARGET_ABORT_INT) != 0)
                        /* Target Abort Interrupt Code Here */

                /* Clear Interrupt Enables */
                outportb(op_reg_base_address+AMCC_OP_REG_INTCSR+1,0);

            /* Clear all active interrupt sources */
            outportb(op_reg_base_address+AMCC_OP_REG_INTCSR+2,status);

        /* Reenable transfer count interrupts */
        outportb(op_reg_base_address+AMCC_OP_REG_INTCSR+1,0xC0);
    }
    else{
        /* Not an S5933 Interrupt Source */
        _chain_intr(oldhandler);
    }

    /* Specific End of interrupt to clear in-service bit(s) */
    /* Mask upper 5 bits of int. line register */
    if(interrupt_line<8)
        outportb(0x20,0x60|(interrupt_line&0x07));
    else{
        /* Issue master then slave EOI */
        outportb(0xa0,0x60|((interrupt_line-8)&0x07));
        outportb(0x20,0x62);
    }
}
}

```

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

```

/*****
/* SETUP_INT_VECT      */
/*      */
/* Purpose:   Install the interrupt vector for the S5933 interrupt      */
/*            handler and reference the previous handler routine.      */
/* Inputs:    S5933 Operation registers base address      */
/* Outputs:    None      */
/*****

void setup_int_vect(byte bus_num,byte dev_func)

{
    byte  int_vector;
    int  mask;

    if(read_configuration_byte(bus_num,dev_func,PCI_CS_INTERRUPT_LINE,
        &interrupt_line)==SUCCESSFUL){

        if(interrupt_line != 0xff){
            if(interrupt_line<8){
                int_vector=0x08+interrupt_line;
            }
            else{
                int_vector=0x70+(interrupt_line-8);
            }
        }

        /* Make sure the system 8259 enables the IRQ with OCW1 @ I/O 21h      */
        /* for IRQ0-7 or @ I/O A1h for IRQ8-15      */
        if(interrupt_line < 8){
            mask = inportb(0x21);
            mask = mask & ~(1<<interrupt_line);
            outportb(0x21,mask);
        }
        else {
            mask = inportb(0xA1);
            mask = mask & ~(1<<(interrupt_line-8));
            outportb(0xA1,mask);
        }

        /* If AMCC interrupt service routine vector is not already installed,      */
        /* install it and save the old vector for chaining.      */
        if(getvect(int_vector) != handler){
            /* Save the old interrupt vector */
            oldhandler=getvect(int_vector);

            /* Install the new interrupt vector */
            setvect(int_vector,handler);
        }
    }
}

```


APPENDIX B - AMCLIB.C LIBRARY FILE

```

/*****
/* Module: AMCLIB.C
/* Purpose: Define a C interface to the PCI BIOS
/* Functions Defined:
/*
/* PCI_BIOS_PRESENT
/* FIND_PCI_DEVICE
/* FIND_PCI_CLASS_CODE
/* GENERATE_SPECIAL_CYCLE
/* READ_CONFIGURATION_BYTE
/* READ_CONFIGURATION_WORD
/* READ_CONFIGURATION_DWORD
/* WRITE_CONFIGURATION_BYTE
/* WRITE_CONFIGURATION_WORD
/* WRITE_CONFIGURATION_DWORD
/* OUTPD
/* INPD
/* INSB
/* INSW
/* INSD
/* OUTSB
/* OUTSW
/* OUTSD
/*
/* Local Functions
/* READ_CONFIGURATION_AREA
/* WRITE_CONFIGURATION_AREA
*****/

/*****
/* Include Files
*****/
#include <dos.h>
#include <stddef.h>
#include "amcc.h"

/*****
/* Local Prototypes
*****/

static int read_configuration_area(byte function,
                                byte bus_number,
                                byte device_and_function,
                                byte register_number,
                                dword *data);

static int write_configuration_area(byte function,
                                byte bus_number,
                                byte device_and_function,
                                byte register_number,
                                dword value);

/*****
/* Define macros to obtain individual bytes from a word register
*****/

#define HIGH_BYTE(ax) (ax >> 8)
#define LOW_BYTE(ax) (ax & 0xff)

```

```

/*****
/*  PCI_BIOS_PRESENT                                     */
/*                                                     */
/* Purpose: Determine the presence of the PCI BIOS     */
/* Inputs: None                                         */
/* Outputs:                                             */
/*   byte *hardware_mechanism                          */
/*       Identifies the hardware characteristics used by the platform. */
/*       Value not assigned if NULL pointer.           */
/*       Bit 0 - Mechanism #1 supported                */
/*       Bit 1 - Mechanism #2 supported                */
/*                                                     */
/*   word *interface_level_version                    */
/*       PCI BIOS Version - Value not assigned if NULL pointer. */
/*       High Byte - Major version number              */
/*       Low Byte - Minor version number               */
/*                                                     */
/*   byte *last_pci_bus_number                         */
/*       Number of last PCI bus in the system. Value not assigned if NULL*/
/*       pointer                                        */
/*                                                     */
/* Return Value - Indicates presence of PCI BIOS      */
/*   SUCCESSFUL - PCI BIOS Present                    */
/*   NOT_SUCCESSFUL - PCI BIOS Not Present            */
/*                                                     */
*****/

int pci_bios_present(byte *hardware_mechanism,
                    word *interface_level_version,
                    byte *last_pci_bus_number)
{
    int ret_status;          /* Function Return Status. */
    byte bios_present_status; /* Indicates if PCI bios present */
    dword pci_signature;     /* PCI Signature ('P', 'C', 'I', ' ') */
    word ax, bx, cx, flags; /* Temporary variables to hold register values */

    /* Load entry registers for PCI BIOS */
    _AH = PCI_FUNCTION_ID;
    _AL = PCI_BIOS_PRESENT;

    /* Call PCI BIOS Int 1Ah interface */
    geninterrupt(0x1a);

    /* Save registers before overwritten by compiler usage of registers */
    ax = _AX;
    bx = _BX;
    cx = _CX;
    pci_signature = _EDX;
    flags = _FLAGS;
    bios_present_status = HIGH_BYTE(ax);

    /* First check if CARRY FLAG Set, if so, BIOS not present */
    if ((flags & CARRY_FLAG) == 0) {

        /* Next, must check that AH (BIOS Present Status) == 0 */
        if (bios_present_status == 0) {

            /* Check bytes in pci_signature for PCI Signature */
            if ((pci_signature & 0xff) == 'P' &&
                ((pci_signature >> 8) & 0xff) == 'C' &&
                ((pci_signature >> 16) & 0xff) == 'I' &&
                ((pci_signature >> 24) & 0xff) == ' ') {

                /* Indicate to caller that PCI bios present */
            }
        }
    }
}

```

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

```

ret_status = SUCCESSFUL;

/* Extract calling parameters from saved registers */
if (hardware_mechanism != NULL) {
    *hardware_mechanism = LOW_BYTE(ax);
}
if (hardware_mechanism != NULL) {
    *interface_level_version = bx;
}
if (hardware_mechanism != NULL) {
    *last_pci_bus_number = LOW_BYTE(cx);
}
}
}
else {
    ret_status = NOT_SUCCESSFUL;
}
}
else {
    ret_status = NOT_SUCCESSFUL;
}
}

return (ret_status);
}

/*****
/* FIND_PCI_DEVICE
/*
/* Purpose: Determine the location of PCI devices given a specific Vendor
/*           Device ID and Index number. To find the first device, specify
/*           0 for index, 1 in index finds the second device, etc.
/* Inputs:
/*   word device_id
/*       Device ID of PCI device desired
/*
/*   word vendor_id
/*       Vendor ID of PCI device desired
/*
/*   word index
/*       Device number to find (0 - (N-1))
/* Outputs:
/*   byte *bus_number
/*       PCI Bus in which this device is found
/*
/*   byte *device_and_function
/*       Device Number in upper 5 bits, Function Number in lower 3 bits
/*
/* Return Value - Indicates presence of device
/*   SUCCESSFUL - Device found
/*   NOT_SUCCESSFUL - BIOS error
/*   DEVICE_NOT_FOUND - Device not found
/*   BAD_VENDOR_ID - Illegal Vendor ID (0xffff)
/*
*****/

int find_pci_device(word device_id,
                   word vendor_id,
                   word index,
                   byte *bus_number,
                   byte *device_and_function)
{
    int ret_status; /* Function Return Status */
    word ax, bx, flags; /* Temporary variables to hold register values */

    /* Load entry registers for PCI BIOS */
    _CX = device_id;
    _DX = vendor_id;

```

```
_SI = index;
_AH = PCI_FUNCTION_ID;
_AL = FIND_PCI_DEVICE;

/* Call PCI BIOS Int 1Ah interface */
geninterrupt(0x1a);

/* Save registers before overwritten by compiler usage of registers */
ax = _AX;
bx = _BX;
flags = _FLAGS;

/* First check if CARRY FLAG Set, if so, error has occurred */
if ((flags & CARRY_FLAG) == 0) {

    /* Get Return code from BIOS */
    ret_status = HIGH_BYTE(ax);
    if (ret_status == SUCCESSFUL) {

        /* Assign Bus Number, Device & Function if successful */
        if (bus_number != NULL) {
            *bus_number = HIGH_BYTE(bx);
        }
        if (device_and_function != NULL) {
            *device_and_function = LOW_BYTE(bx);
        }
    }
}
else {
    ret_status = NOT_SUCCESSFUL;
}

return (ret_status);
}

/*****
/* FIND_PCI_CLASS_CODE
/*
/* Purpose: Returns the location of PCI devices that have a specific Class
/* Code.
/*
/* Inputs:
/* word class_code
/* Class Code of PCI device desired
/*
/* word index
/* Device number to find (0 - (N-1))
/*
/* Outputs:
/* byte *bus_number
/* PCI Bus in which this device is found
/*
/* byte *device_and_function
/* Device Number in upper 5 bits, Function Number in lower 3 bits
/*
/* Return Value - Indicates presence of device
/* SUCCESSFUL - Device found
/* NOT_SUCCESSFUL - BIOS error
/* DEVICE_NOT_FOUND - Device not found
/*
/*
/*****/

int find_pci_class_code(dword class_code,
                       word index,
                       byte *bus_number,
                       byte *device_and_function)
```

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

```

{
    int ret_status;      /* Function Return Status */
    word ax, bx, flags; /* Temporary variables to hold register values */

    /* Load entry registers for PCI BIOS */
    _ECX = class_code;
    _SI = index;
    _AH = PCI_FUNCTION_ID;
    _AL = FIND_PCI_CLASS_CODE;

    /* Call PCI BIOS Int 1Ah interface */
    geninterrupt(0x1a);

    /* Save registers before overwritten by compiler usage of registers */
    ax = _AX;
    bx = _BX;
    flags = _FLAGS;

    /* First check if CARRY FLAG Set, if so, error has occurred */
    if ((flags & CARRY_FLAG) == 0) {

        /* Get Return code from BIOS */
        ret_status = HIGH_BYTE(ax);
        if (ret_status == SUCCESSFUL) {

            /* Assign Bus Number, Device & Function if successful */
            if (bus_number != NULL) {
                *bus_number = HIGH_BYTE(bx);
            }
            if (device_and_function != NULL) {
                *device_and_function = LOW_BYTE(bx);
            }
        }
    }
    else {
        ret_status = NOT_SUCCESSFUL;
    }

    return (ret_status);
}

/*****
/* GENERATE_SPECIAL_CYCLE                                     */
/*                                                         */
/* Purpose: Generates a PCI special cycle                 */
/* Inputs:                                               */
/*   byte bus_number                                     */
/*     PCI bus to generate cycle on                     */
/*                                                         */
/*   dword special_cycle_data                           */
/*     Special Cycle Data to be generated               */
/* Outputs:                                              */
/*   Return Value - Indicates presence of device        */
/*     SUCCESSFUL - Device found                         */
/*     DEVICE_NOT_FOUND - Device not found              */
/*                                                         */
*****/

int generate_special_cycle(byte bus_number,
                          dword special_cycle_data)
{
    int ret_status; /* Function Return Status */
    word ax, flags; /* Temporary variables to hold register values */

```

```
/* Load entry registers for PCI BIOS */
_BH = bus_number;
_EDX = special_cycle_data;
_AH = PCI_FUNCTION_ID;
_AL = FIND_PCI_CLASS_CODE;

/* Call PCI BIOS Int 1Ah interface */
geninterrupt(0x1a);

/* Save registers before overwritten by compiler usage of registers */
ax = _AX;
flags = _FLAGS;

/* First check if CARRY FLAG Set, if so, error has occurred */
if ((flags & CARRY_FLAG) == 0) {

    /* Get Return code from BIOS */
    ret_status = HIGH_BYTE(ax);
}
else {
    ret_status = NOT_SUCCESSFUL;
}

return (ret_status);
}

/*****
/* READ_CONFIGURATION_BYTE
/*
/* Purpose: Reads a byte from the configuration space of a specific device
/* Inputs:
/*   byte bus_number
/*     PCI bus to read configuration data from
/*
/*   byte device_and_function
/*     Device Number in upper 5 bits, Function Number in lower 3 bits
/*
/*   byte register_number
/*     Register Number of configuration space to read
/* Outputs:
/*   byte *byte_read
/*     Byte read from Configuration Space
/*
/* Return Value - Indicates presence of device
/* SUCCESSFUL - Device found
/* NOT_SUCCESSFUL - BIOS error
/* BAD_REGISTER_NUMBER - Invalid Register Number
/*
/*****/

int read_configuration_byte(byte bus_number,
                           byte device_and_function,
                           byte register_number,
                           byte *byte_read)
{
    int ret_status; /* Function Return Status */
    dword data;

    /* Call read_configuration_area function with byte data */
    ret_status = read_configuration_area(READ_CONFIG_BYTE,
                                        bus_number,
                                        device_and_function,
                                        register_number,
                                        &data);

    if (ret_status == SUCCESSFUL) {
        /* Extract byte */
        *byte_read = (byte)(data & 0xff);
    }
}
```

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

```

}

return (ret_status);
}

/*****
/*  READ_CONFIGURATION_WORD                                     */
/*                                                                 */
/* Purpose: Reads a word from the configuration space of a specific device */
/* Inputs:                                                                 */
/*   byte bus_number                                             */
/*     PCI bus to read configuration data from                   */
/*                                                                 */
/*   byte device_and_function                                    */
/*     Device Number in upper 5 bits, Function Number in lower 3 bits */
/*                                                                 */
/*   byte register_number                                        */
/*     Register Number of configuration space to read           */
/* Outputs:                                                                 */
/*   word *word_read                                            */
/*     Word read from Configuration Space                       */
/*                                                                 */
/* Return Value - Indicates presence of device                 */
/*   SUCCESSFUL - Device found                                  */
/*   NOT_SUCCESSFUL - BIOS error                               */
/*   BAD_REGISTER_NUMBER - Invalid Register Number             */
/*                                                                 */
*****/

int read_configuration_word(byte bus_number,
                           byte device_and_function,
                           byte register_number,
                           word *word_read)
{
    int ret_status; /* Function Return Status */
    dword data;

    /* Call read_configuration_area function with word data */
    ret_status = read_configuration_area(READ_CONFIG_WORD,
                                         bus_number,
                                         device_and_function,
                                         register_number,
                                         &data);

    if (ret_status == SUCCESSFUL) {

        /* Extract word */
        *word_read = (word)(data & 0xffff);
    }

    return (ret_status);
}

/*****
/*  READ_CONFIGURATION_DWORD                                   */
/*                                                                 */
/* Purpose: Reads a dword from the configuration space of a specific device */
/* Inputs:                                                                 */
/*   byte bus_number                                             */
/*     PCI bus to read configuration data from                   */
/*                                                                 */
/*   byte device_and_function                                    */
/*     Device Number in upper 5 bits, Function Number in lower 3 bits */
/*                                                                 */
/*   byte register_number                                        */
/*     Register Number of configuration space to read           */
/* Outputs:                                                                 */
/*   dword *dword_read                                          */
/*                                                                 */
*****/

```

```
/*      Dword read from Configuration Space      */
/*      */
/*      Return Value - Indicates presence of device      */
/*      SUCCESSFUL - Device found      */
/*      NOT_SUCCESSFUL - BIOS error      */
/*      BAD_REGISTER_NUMBER - Invalid Register Number      */
/*      */
/*****/

int read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read)
{
    int ret_status; /* Function Return Status */
    dword data;

    /* Call read_configuration_area function with dword data */
    ret_status = read_configuration_area(READ_CONFIG_DWORD,
                                         bus_number,
                                         device_and_function,
                                         register_number,
                                         &data);

    if (ret_status == SUCCESSFUL) {

        /* Extract dword */
        *dword_read = data;
    }

    return (ret_status);
}

/*****/
/*  READ_CONFIGURATION_AREA      */
/*  */
/*  Purpose: Reads a byte/word/dword from the configuration space of a      */
/*            specific device      */
/*  Inputs:      */
/*  byte bus_number      */
/*  PCI bus to read configuration data from      */
/*  */
/*  byte device_and_function      */
/*  Device Number in upper 5 bits, Function Number in lower 3 bits      */
/*  */
/*  byte register_number      */
/*  Register Number of configuration space to read      */
/*  Outputs:      */
/*  dword *dword_read      */
/*  Dword read from Configuration Space      */
/*  */
/*  Return Value - Indicates presence of device      */
/*  SUCCESSFUL - Device found      */
/*  NOT_SUCCESSFUL - BIOS error      */
/*  BAD_REGISTER_NUMBER - Invalid Register Number      */
/*  */
/*****/

static int read_configuration_area(byte function,
                                   byte bus_number,
                                   byte device_and_function,
                                   byte register_number,
                                   dword *data)
{
    int ret_status; /* Function Return Status */
    word ax, flags; /* Temporary variables to hold register values */
    dword ecx; /* Temporary variable to hold ECX register value */
}
```

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

```

/* Load entry registers for PCI BIOS */
_BH = bus_number;
_BL = device_and_function;
_DI = register_number;
_AH = PCI_FUNCTION_ID;
_AL = function;

/* Call PCI BIOS Int 1Ah interface */
geninterrupt(0x1a);

/* Save registers before overwritten by compiler usage of registers */
ecx = _ECX;
ax = _AX;
flags = _FLAGS;

/* First check if CARRY FLAG Set, if so, error has occurred */
if ((flags & CARRY_FLAG) == 0) {

    /* Get Return code from BIOS */
    ret_status = HIGH_BYTE(ax);

    /* If successful, return data */
    if (ret_status == SUCCESSFUL) {
        *data = ecx;
    }
}
else {
    ret_status = NOT_SUCCESSFUL;
}

return (ret_status);
}

/*****
/* WRITE_CONFIGURATION_BYTE                                     */
/*                                                             */
/* Purpose: Writes a byte to the configuration space of a specific device */
/*                                                             */
/* Inputs:                                                     */
/*   byte bus_number                                           */
/*       PCI bus to write configuration data to                 */
/*                                                             */
/*   byte device_and_function                                   */
/*       Device Number in upper 5 bits, Function Number in lower 3 bits */
/*                                                             */
/*   byte register_number                                       */
/*       Register Number of configuration space to write       */
/*                                                             */
/*   byte byte_to_write                                         */
/*       Byte to write to Configuration Space                  */
/* Outputs:                                                    */
/*   Return Value - Indicates presence of device              */
/*       SUCCESSFUL - Device found                             */
/*       NOT_SUCCESSFUL - BIOS error                          */
/*       BAD_REGISTER_NUMBER - Invalid Register Number        */
*****/

int write_configuration_byte(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             byte byte_to_write)
{
    int ret_status; /* Function Return Status */

```

```
/* Call write_configuration_area function with byte data */
ret_status = write_configuration_area(WRITE_CONFIG_BYTE,
                                     bus_number,
                                     device_and_function,
                                     register_number,
                                     byte_to_write);

return (ret_status);
}

/*****
/* WRITE_CONFIGURATION_WORD
/*
/* Purpose: Writes a word to the configuration space of a specific device
/* Inputs:
/*   byte bus_number
/*     PCI bus to read configuration data from
/*
/*   byte device_and_function
/*     Device Number in upper 5 bits, Function Number in lower 3 bits
/*
/*   byte register_number
/*     Register Number of configuration space to read
/*
/*   word word_to_write
/*     Word to write to Configuration Space
/* Outputs:
/*   Return Value - Indicates presence of device
/*     SUCCESSFUL - Device found
/*     NOT_SUCCESSFUL - BIOS error
/*     BAD_REGISTER_NUMBER - Invalid Register Number
/*
*****/
int write_configuration_word(byte bus_number,
                            byte device_and_function,
                            byte register_number,
                            word word_to_write)
{
    int ret_status; /* Function Return Status */

    /* Call read_configuration_area function with word data */
    ret_status = write_configuration_area(WRITE_CONFIG_WORD,
                                         bus_number,
                                         device_and_function,
                                         register_number,
                                         word_to_write);

    return (ret_status);
}

/*****
/* WRITE_CONFIGURATION_DWORD
/*
/* Purpose: Reads a dword from the configuration space of a specific device
/* Inputs:
/*   byte bus_number
/*     PCI bus to read configuration data from
/*
/*   byte device_and_function
/*     Device Number in upper 5 bits, Function Number in lower 3 bits
/*
/*   byte register_number
/*     Register Number of configuration space to read
/*
/*   dword dword_to_write
/*     Dword to write to Configuration Space
/* Outputs:
/*   Return Value - Indicates presence of device
/*     SUCCESSFUL - Device found
*****/
```

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

```

/*      NOT_SUCCESSFUL - BIOS error                                     */
/*      BAD_REGISTER_NUMBER - Invalid Register Number                 */
/*                                                                 */
/*****
int write_configuration_dword(byte bus_number,
                              byte device_and_function,
                              byte register_number,
                              dword dword_to_write)
{
    int ret_status; /* Function Return Status */

    /* Call write_configuration_area function with dword data */
    ret_status = write_configuration_area(READ_CONFIG_DWORD,
                                         bus_number,
                                         device_and_function,
                                         register_number,
                                         dword_to_write);

    return (ret_status);
}

/*****
/*  WRITE_CONFIGURATION_AREA                                         */
/*                                                                 */
/* Purpose: Writes a byte/word/dword to the configuration space of a */
/*          specific device                                           */
/* Inputs:                                                                 */
/*   byte bus_number                                                 */
/*     PCI bus to read configuration data from                       */
/*                                                                 */
/*   byte device_and_function                                         */
/*     Device Number in upper 5 bits, Function Number in lower 3 bits */
/*                                                                 */
/*   byte register_number                                             */
/*     Register Number of configuration space to read                */
/*                                                                 */
/*   dword value                                                      */
/*     Value to write to Configuration Space                         */
/* Outputs:                                                                 */
/*   Return Value - Indicates presence of device                    */
/*   SUCCESSFUL - Device found                                       */
/*   NOT_SUCCESSFUL - BIOS error                                     */
/*   BAD_REGISTER_NUMBER - Invalid Register Number                 */
/*                                                                 */
/*****

static int write_configuration_area(byte function,
                                   byte bus_number,
                                   byte device_and_function,
                                   byte register_number,
                                   dword value)
{
    int ret_status; /* Function Return Status */
    word ax, flags; /* Temporary variables to hold register values */

    /* Load entry registers for PCI BIOS */
    _BH = bus_number;
    _BL = device_and_function;
    _ECX = value;
    _DI = register_number;
    _AH = PCI_FUNCTION_ID;
    _AL = function;

    /* Call PCI BIOS Int 1Ah interface */
    geninterrupt(0x1a);

```

```
/* Save registers before overwritten by compiler usage of registers */
ax = _AX;
flags = _FLAGS;

/* First check if CARRY FLAG Set, if so, error has occurred */
if ((flags & CARRY_FLAG) == 0) {

    /* Get Return code from BIOS */
    ret_status = HIGH_BYTE(ax);
}
else {
    ret_status = NOT_SUCCESSFUL;
}

return (ret_status);
}

/*****
/*  OUTPD                                     */
/*                                     */
/* Purpose: Outputs a DWORD to a hardware port */
/* Inputs:                                     */
/*   word port                                 */
/*   hardware port to write DWORD to         */
/*                                     */
/*   dword value                               */
/*   value to be written to port            */
/* Outputs:                                     */
/*   None                                     */
/*                                     */
*****/

void outpd(word port, dword value)
{
    _DX = port;
    _EAX = value;

    /* Since asm cannot generate OUT  DX, EAX must force in */
    __emit__(0x66, 0xEF);
}

/*****
/*  INPD                                     */
/*                                     */
/* Purpose: Inputs a DWORD from a hardware port */
/* Inputs:                                     */
/*   word port                                 */
/*   hardware port to write DWORD to         */
/* Outputs:                                     */
/*   dword value                               */
/*   value read from the port                */
/*                                     */
*****/

dword inpd(word port)
{
    /* Set DX register to port number to be input from */
    _DX = port;

    /* Since asm cannot generate IN  EAX, DX, must force in */
    __emit__(0x66, 0xED);

    return(_EAX);
}
```

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

```

/*****
/*  INSB                                                    */
/*                                                    */
/* Purpose: Inputs a string of BYTES from a hardware port */
/* Inputs:                                                    */
/*   word port                                                    */
/*   hardware port to read from                                */
/*                                                    */
/*   void *buf                                                    */
/*   Buffer to read data into                                  */
/*                                                    */
/*   int count                                                    */
/*   Number of BYTES to read                                  */
/* Outputs:                                                    */
/*   None                                                        */
/*                                                    */
/*****

void insb(word port, void *buf, int count)
{
    _ES = FP_SEG(buf);    /* Segment of buf */
    _DI = FP_OFF(buf);    /* Offset of buf */
    _CX = count;          /* Number to read */
    _DX = port;           /* Port */
    asm  REP INSB;
}

/*****
/*  INSW                                                    */
/*                                                    */
/* Purpose: Inputs a string of WORDs from a hardware port */
/* Inputs:                                                    */
/*   word port                                                    */
/*   hardware port to read from                                */
/*                                                    */
/*   void *buf                                                    */
/*   Buffer to read data into                                  */
/*                                                    */
/*   int count                                                    */
/*   Number of WORDs to read                                  */
/* Outputs:                                                    */
/*   None                                                        */
/*                                                    */
/*****

void insw(word port, void *buf, int count)
{
    _ES = FP_SEG(buf);    /* Segment of buf */
    _DI = FP_OFF(buf);    /* Offset of buf */
    _CX = count;          /* Number to read */
    _DX = port;           /* Port */
    asm  REP INSW;
}

```

```

/*****
/*  INSB                                     */
/*                                          */
/* Purpose: Inputs a string of DWORDS from a hardware port */
/* Inputs:                                     */
/*   word port                               */
/*   hardware port to read from              */
/*                                          */
/*   void *buf                               */
/*   Buffer to read data into                */
/*                                          */
/*   int count                              */
/*   Number of DWORDS to read               */
/* Outputs:                                     */
/*   None                                    */
/*                                          */
*****/

```

```

void insd(word port, void *buf, int count)
{
    _ES = FP_SEG(buf); /* Segment of buf */
    _DI = FP_OFF(buf); /* Offset of buf */
    _CX = count;       /* Number to read */
    _DX = port;        /* Port */
    __emit__(0xf3, 0x66, 0x6D); /* asm REP INSD */
}

```

```

/*****
/*  OUTSB                                    */
/*                                          */
/* Purpose: Outputs a string of BYTES to a hardware port */
/* Inputs:                                     */
/*   word port                               */
/*   hardware port to write to              */
/*                                          */
/*   void *buf                               */
/*   Buffer to write data from              */
/*                                          */
/*   int count                              */
/*   Number of BYTES to write              */
/* Outputs:                                     */
/*   None                                    */
/*                                          */
*****/

```

```

void outsb(word port, void *buf, int count)
{
    _SI = FP_SEG(buf); /* Offset of buf */
    _CX = count;       /* Number to read */
    _DX = port;        /* Port */
    asm REP OUTSB;
}

```

BUS MASTERING WITH THE S5933 PCI MATCHMAKER

```

/*****
/*  OUTSW                                                    */
/*                                                    */
/* Purpose: Outputs a string of WORDS to a hardware port  */
/* Inputs:                                                    */
/*   word port                                                    */
/*   hardware port to write to                                */
/*                                                    */
/*   void *buf                                                    */
/*   Buffer to write data from                                */
/*                                                    */
/*   int count                                                    */
/*   Number of WORDS to write                                */
/* Outputs:                                                    */
/*   None                                                        */
/*                                                    */
/*****

void outsw(word port, void *buf, int count)
{
    _SI = FP_SEG(buf);    /* Offset of buf */
    _CX = count;         /* Number to read */
    _DX = port;          /* Port */
    asm  REP OUTSW;
}

/*****
/*  OUTSD                                                    */
/*                                                    */
/* Purpose: Outputs a string of DWORDS to a hardware port  */
/* Inputs:                                                    */
/*   word port                                                    */
/*   hardware port to write to                                */
/*                                                    */
/*   void *buf                                                    */
/*   Buffer to write data from                                */
/*                                                    */
/*   int count                                                    */
/*   Number of DWORDS to write                                */
/* Outputs:                                                    */
/*   None                                                        */
/*                                                    */
/*****

void outsd(word port, void *buf, int count)
{
    _SI = FP_SEG(buf);    /* Offset of buf */
    _CX = count;         /* Number to read */
    _DX = port;          /* Port */
    __emit__(0xf3, 0x66, 0x6F); /* asm  REP OUTSD; */
}

```


APPENDIX C - AMCC.H INCLUDE FILE

The following code includes generic function and constant definitions used for the code in Appendix A and Appendix B.

```

/*****/
/* Module: AMCC.H */
/* Purpose: Definitions for AMCC PCI Library */
/*****/

/*****/
/* General Constants */
/*****/

#define TRUE 1
#define FALSE 0

typedef unsigned char byte; /* 8-bit */
typedef unsigned short word; /* 16-bit */
typedef unsigned long dword; /* 32-bit */

#define CARRY_FLAG 0x01 /* 80x86 Flags Register Carry Flag bit */

/*****/
/* PCI Functions */
/*****/

#define PCI_FUNCTION_ID 0xb1
#define PCI_BIOS_PRESENT 0x01
#define FIND_PCI_DEVICE 0x02
#define FIND_PCI_CLASS_CODE 0x03
#define GENERATE_SPECIAL_CYCLE 0x06
#define READ_CONFIG_BYTE 0x08
#define READ_CONFIG_WORD 0x09
#define READ_CONFIG_DWORD 0x0a
#define WRITE_CONFIG_BYTE 0x0b
#define WRITE_CONFIG_WORD 0x0c
#define WRITE_CONFIG_DWORD 0x0d

/*****/
/* PCI Return Code List */
/*****/

#define SUCCESSFUL 0x00
#define NOT_SUCCESSFUL 0x01
#define FUNC_NOT_SUPPORTED 0x81
#define BAD_VENDOR_ID 0x83
#define DEVICE_NOT_FOUND 0x86
#define BAD_REGISTER_NUMBER 0x87

/*****/
/* PCI Configuration Space Registers */
/*****/

#define PCI_CS_VENDOR_ID 0x00
#define PCI_CS_DEVICE_ID 0x02
#define PCI_CS_COMMAND 0x04
#define PCI_CS_STATUS 0x06
#define PCI_CS_REVISION_ID 0x08
#define PCI_CS_CLASS_CODE 0x09
#define PCI_CS_CACHE_LINE_SIZE 0x0c
#define PCI_CS_MASTER_LATENCY 0x0d
#define PCI_CS_HEADER_TYPE 0x0e
#define PCI_CS_BIST 0x0f

```

```
#define PCI_CS_BASE_ADDRESS_0      0x10
#define PCI_CS_BASE_ADDRESS_1      0x14
#define PCI_CS_BASE_ADDRESS_2      0x18
#define PCI_CS_BASE_ADDRESS_3      0x1c
#define PCI_CS_BASE_ADDRESS_4      0x20
#define PCI_CS_BASE_ADDRESS_5      0x24
#define PCI_CS_EXPANSION_ROM        0x30
#define PCI_CS_INTERRUPT_LINE       0x3c
#define PCI_CS_INTERRUPT_PIN        0x3d
#define PCI_CS_MIN_GNT              0x3e
#define PCI_CS_MAX_LAT              0x3f

/*****/
/*  AMCC Operation Register Offsets  */
/*****/

#define OMB1      0x00
#define OMB2      0x04
#define OMB3      0x08
#define OMB4      0x0c
#define IMB1      0x11
#define IMB2      0x14
#define IMB3      0x18
#define IMB4      0x1c
#define FIFO      0x20
#define MWAR      0x24
#define MWTC      0x28
#define MRAR      0x2c
#define MRTC      0x30
#define MBEF      0x34
#define INTCSR    0x38
#define MCSR      0x3c
#define MCSR_NVDATA (AMCC_OP_REG_MCSR + 2) /* Data in byte 2 */
#define MCSR_NVCMD (AMCC_OP_REG_MCSR + 3) /* Command in byte 3 */

/*****/
/*  AMCCLIB Prototypes  */
/*****/

int pci_bios_present(byte *hardware_mechanism,
                    word *interface_level_version,
                    byte *last_pci_bus_number);

int find_pci_device(word device_id,
                  word vendor_id,
                  word index,
                  byte *bus_number,
                  byte *device_and_function);

int find_pci_class_code(dword class_code,
                      word index,
                      byte *bus_number,
                      byte *device_and_function);

int generate_special_cycle(byte bus_number,
                          dword special_cycle_data);

int read_configuration_byte(byte bus_number,
                          byte device_and_function,
                          byte register_number,
                          byte *byte_read);

int read_configuration_word(byte bus_number,
                          byte device_and_function,
                          byte register_number,
                          word *word_read);
```

```
int read_configuration_dword(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             dword *dword_read);

int write_configuration_byte(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             byte byte_to_write);

int write_configuration_word(byte bus_number,
                             byte device_and_function,
                             byte register_number,
                             word word_to_write);

int write_configuration_dword(byte bus_number,
                              byte device_and_function,
                              byte register_number,
                              dword dword_to_write);

void outpd(word port, dword value);

dword inpd(word port);

void insb(word port, void *buf, int count);
void insw(word port, void *buf, int count);
void insd(word port, void *buf, int count);

void outsb(word port, void *buf, int count);
void outsw(word port, void *buf, int count);
void outsd(word port, void *buf, int count);
```

