KS8695P

Integrated Multi-Port Gateway Solution

Programming Guide

Version 1.41

February 2005

Contents

1 General Information

1.1 Revision History

Revision	Date	Description
1.0	05-15-03	Created
1.1	06-25-03	Modified PCI Device Configuration section
1.2	07-15-03	Added Linux PCI data coherent information
1.3	09-17-03	Modified Linux kernel Porting section
1.4	02-04-04	Clarified various sections
1.41	02-15-05	Clarified various sections

1.2 References

1. *KS8695P Functional Specification.* San Jose, CA: Micrel, Inc., 2003.
2. *ARM Architecture Reference Manual*. ARM Limited, 2000.
3. *PCI Local Bus Specification.* Revision 2.3. PCI Special Interest Group, 2002.
4. Shanley, Tom and Anderson, Don. *PCI System Architecture*, third edition. Reading, MA: Addison-Wesley, 1995.

2 Overview

The KS8695P Programming Guide covers the entire KS8695P-based embedded system software. It assumes that the user is familiar with ARM development tools, such as ARM assembler and C/C++, and Debug tools, such as Multi-ICE and others.

This Guide focuses on how to program the KS8695P-specific modules. Some of the areas it covers are: how to set up a memory map, how to configure the WAN and LAN DMA engines, how to program the UART to communicate with a remote hyper terminal, and exception handling.

The ARM processor specific programming information is not included in this guide except for some key KS8695P-related elements, such as enabling the MMU, I-Cache and D-Cache. These tasks are important to system performance. Sample code is also included in the guide for the purpose of examples and demonstrations.

Appendix A contains important tips on the KS8695P DMA engine programming. This is a key element for KS8695P system performance and reliability. **All developers should read *Appendix A* before working on the DMA drivers.**

The complete source code tree is included on the KS8695P SOHO CD. This tree contains the `ksinit.s` source codes, which are provided as part of the KS8695P evaluation kit. The source code subdirectories are:

```
Soho/loader/diag/ksinit.s
Soho/loader/common/tlb.s
```

2.1 Embedded Application Codes

The embedded application codes written for an ARM CPU-based system reside in ROM and execute on reset. When writing an embedded operating system, or embedded applications that execute from reset without an operating system, the following factors need to be considered:

- Remapping from ROM to RAM to improve the execution speed
- Initializing the environment and application
- Linking an embedded executable image to place the code and data in specific memory locations

Boot-up Sequence for the KS8695P

The boot-up sequence for the KS8695P commences with the ARM CPU core executing instructions from address 0 at reset. There must be a ROM at address 0 when the system is reset.

Typically, ROM is narrow and slow when compared to RAM. This affects the speed of exception handling (the exception vectors cannot be modified in this case). A common strategy

is to remap ROM to RAM, and then copy the exception vectors from ROM to RAM after startup.

After reset, an embedded application or operating system initializes the system including:

- Initializing the execution environment, such as exception vector, stacks, I/O
- Initializing the application, such as copying initialization values for nonzero write-able data to the write-able data region and zeroing the ZI data region

Boot-up Exceptions

The application may need to handle following exceptions:

RESET	Occurs when the KS8695P is powered-up. Branching to the reset vector at 0x0 can do a soft reset.
INTERRUP (IRQ)	Occurs when any of the KS8695P's 32 interrupt sources are enabled and asserted and the I bit in the CPSR is cleared.
FAST INTERRUPT (FIQ)	Occurs when any of the KS8695P's 32 interrupt sources are enabled and asserted, and the F bit in the CPSR is cleared. This exception is typically used where interrupt latency must be kept to a minimum.
UNDEFINED INSTRUCTION	Occurs if neither the processor, nor any attached co-processor, recognizes the currently executing instruction.
SOFTWARE INTERRUPT	This is a user-defined synchronous interrupt instruction. For example, it allows a program running in User mode to request privileged operations that run in Supervisor mode, such as an RTOS function.
PREFETCH ABORT	Occurs when the processor attempts to execute an instruction that has prefetched from an illegal address.
DATA ABORT	Occurs when a data transfer instruction attempts to load or store data at an illegal address.

By default the KS8695P clock is set at 25 MHz both for the system bus and CPU. All the KS8695P registers are mapped onto the memory area starting from 0x03FF0000.

At power up, the KS8695P starts a series of self-booting. The switch core boots itself. WAN and LAN also start their logic at power up. Software has to wait for 4.7 msec before accessing these function blocks.

If the WAN and LAN DMA get reset (soft reset), there should be 1.7 msec waiting period before starting frame transmission. Failing to wait the 1.7 msec will cause the DMA engine to generate frames containing CRC errors.

Please refer to the KS8695P Data Sheet or Registration Description for all register descriptions.

3 Memory Configuration

Configuring memory is the first step to perform when bringing up your KS8695P-based system.

Many embedded systems often implement complex memory configurations. For example, an embedded system might use a fast 32-bit RAM for performance-critical code, such as interrupt handlers and the stack, slower 16-bit RAM for application RW data, and ROM for normal application code. Linker scatter loading mechanisms are used to construct executable images suitable for complex systems.

The KS8695P supports a flexible memory map. The chip supports up to two ROM/SRAM/FLASH devices and up to two SDRAM devices. There are two registers to control ROM/SRAM/FLASH devices: ROM/SRAM/FLASH control registers 0 and 1 (ROMCON0 0x4010 and ROMCON1 0x4014); and two registers to configure two banks of SDRAM: SDRAM control registers 0 and 1 (SDCON0 0x4030 and SDCON1 0x4034). By default, only the first ROM/SRAM/FLASH bank is enabled. It is set at 32 MB. The default memory map is illustrated in Figure 3.1

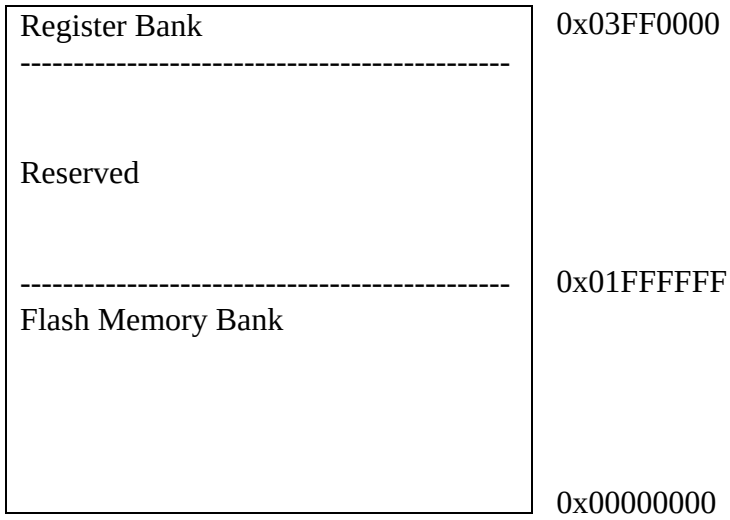


Figure 3.1. Default Memory Map

3.1 Memory Map

The KS8695P is designed so that SDRAM banks are disabled by default, and the KS8695P registers are mapped to the memory address starting at 0x03FF0000. The register bank takes up 64KB. The maximum memory space that the KS8695P can support is 64MB.

Reconfiguring the Memory Map

The memory map has to be arranged in order to put SDRAM in the memory map, and the ROM/FLASH memory must be moved to a starting address other than 0x0. Use the following steps to reconfigure the memory map.

Procedure

1. Reprogram the ROM/SRAM/FLASH bank register to reduce its size from 32 MB to its designed size (4 MB for the KS8695P demo board) if the device does not take up a 32 MB space. This is to allow some space for SDRAM and to make the memory map continuous.
2. Enable the SDRAM banks and set them on top of the ROM/SRAM/FLASH bank.
3. Copy the instructions that are used to reprogram the ROM/SRAM/FLASH control register into the enabled SDRAM.
4. From the SDRAM, execute the newly copied instructions for reprogramming the ROM/SRAM/FLASH control register.

   This action moves its address to a higher address and makes space for exception vectors to be put at 0x0.
5. From the newly configured ROM/SRAM/FLASH memory address, reprogram the SDRAM control register to move it down to 0x0 and continue to boot.
6. Continue booting to initialize the execution environment (such as exception vector, stacks, I/O).
7. Initialize the application (for example copying initialization values for nonzero write-able data to the write-able data region and zeroing the ZI data region).

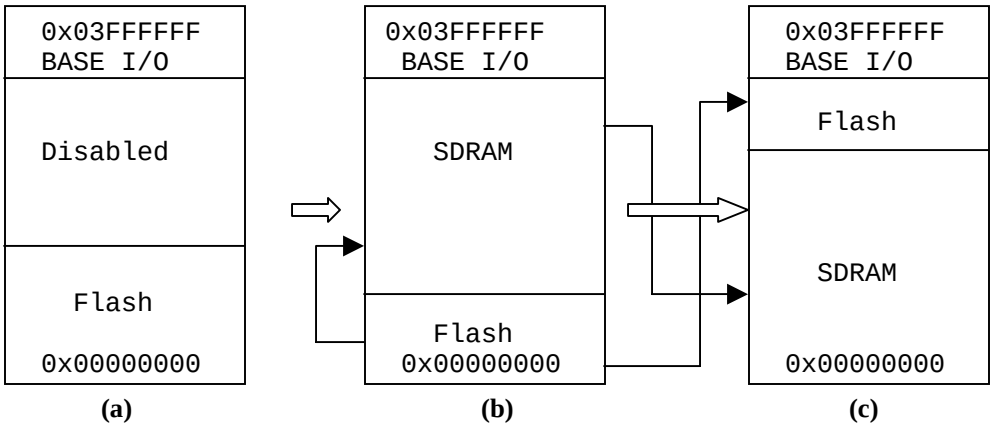


Figure 3.2. Memory Remapping: (a) Default Memory Map, (b) Move Code to SDRAM, (c) Remap Flash and SDRAM.

3.2 Enable SDRAM

Before SDRAM can be read and written, it must be configured and enabled.

Two SDRAM control registers are used to set up SDRAM. The KS8695P demo board uses only one of them. The SDRAM control registers hold the memory address ranges and memory parameters such as column address bits, number of banks and data width. These parameters are set to match that of the SDRAM device supports and are given in the SDRAM's data sheet.

SDRAM General Control Register (SDGCN 0x4038) sets the SDRAM RAS to CAS latency. Usually these values are programmed to match those values supported by the SDRAM device. These parameters are given in the SDRAM data sheet.

Procedure

1. After the SDRAM control and general control registers are set, a NOP command has to be issued through the SDRAM Buffer Control Register (SDBCON 0x403C).
2. Upon completion of the NOP command, a PRECHARGE command is sent.

   Wait for two memory refresh cycles to complete.

   A shorter value can be written into SDRAM Refresh Timer Register (REFTIM 0x4040) to set the refresh quicker. The timer has to be resumed after two quick refreshes. **Ensure that the refresh timer meets the minimum requirement of the SDRAM**.
3. The MODE command is sent at the final MEMORY refresh cycle, initializing the SDRAM for use.

Enabling the Read Buffer and the Write FIFO for the SDRAM access is optional. The SDRAM Read Buffer and Write FIFO improve the memory access time by performing burst access. Use the FIFO as in programming flash devices since the CPU can read old values such as the toggle bits. It is the user's responsibility to maintain data coherency when the Read Buffer and Write FIFO are enabled.

3.3 Remapping the KS8695P Demo Board Memory

The following section gives an example of remapping the KS8695P demo board. The source code is in `ksinit.s`, which can be found on the accompanying KS8695P evaluation kit CD. The source code subdirectory is:

```
Soho/loader/diag/ksinit.s
```

The KS8695P board has a flexible memory map. The memory addressing space is limited to 64 MB. There is one flash memory device and one SDRAM memory device. At power up, the ARM922T does not see any SDRAM, it only sees the Flash and the Base I/O register bank space as shown in Figure 3.3. During system reset, the memory banks are remapped as shown in Figure 3.3 (b).

Some designs may use the second bank of ROM/SRAM/FLASH and the second bank of the SDRAM; these registers have to be programmed to the correct value. The memory map can be overlapped in the same manner as part of the SDRAM overlaps FLASH. In this case, FLASH memory has a higher priority so that the overlapped SDRAM is not accessible.

Memory can be non contiguous by leaving some memory holes between the SDRAM space and FLASH memory space for future SDRAM expansion.

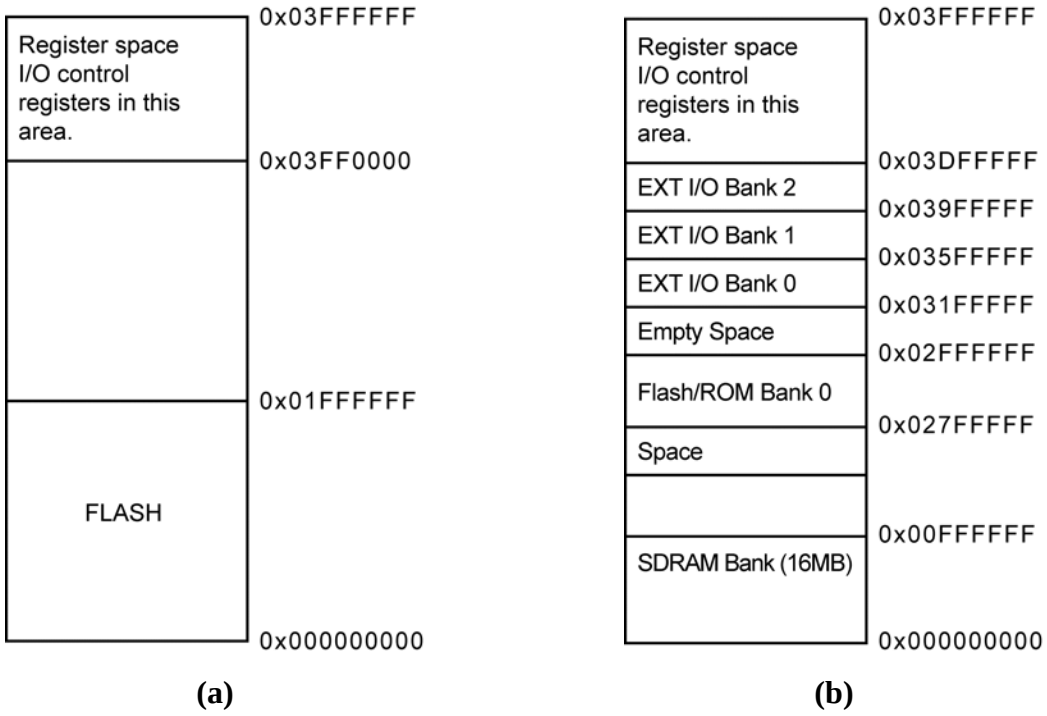


Figure 3.3. KS8695P Demo Board Memory Remap:
(a) Power On Default and (b) After Remap

4 Setting Up UART to Communicate with a Remote Host

Since the KS8695P does not support a display device, it is very important to set up the UART at the earliest stage of the system booting process. This allows boot messages to be sent to a remote host connected to the target through the serial port. The Hyper Terminal, or equivalent, running on the remote host can be used to receive messages and also to send commands to the target. The UART can also be used to download and debug the program from the host by running an Angel debugger or other debug software on the target.

The UART can be initialized at the very beginning of the booting process. The simplest method is to enable the UART and configure it in polling mode so that it does not require SDRAM to be enabled first. The following parameters are necessary:

```
Baud Rate               ;divisor register
Word Length             ;line control register
Parity                  ;line control register
Stop bit                ;line control register

LDR  r4, =REG_IO_BASE               ;set register base address
LDR  r5, =REG_UART_FIFO_CTRL        ;get FIFO register offset
ADD  r5, r5, r4                     ;get FIFO register address
MOV  r1, #0x6                       ;reset value
STR  r1, [r5]                       ;reset FIFO
NOP                                 ;wait the FIFO reset
NOP
NOP

;enable FIFO, set word length = 8
MOV  r1, #0x81
STR  r1, [r5]

LDR  r5, =REG_UART_LINE_CTRL        ;get line control register
                                    ;offset
ADD  r5, r5, r4                     ;get line control register
                                     ;address
MOV  r1, #0x3                       ;no parity, 1 stop bit
STR  r1, [r5]                       ;write the register

;set baud rate at 38400
LDR  r5, =REG_UART_DIVISOR          ;get divisor's register
                                    ;offset
ADD  r5, r5, r4                      ;get divisor register address
LDR  r1, =0x28B                     ;baud rate 38400
STR  r1, [r5]                       ;set baud rate
```

To read the data, check the Line Status Register bit 0. If it is set, the program can read the data directly from the Receive Buffer Register. To send data, write the character directly to the Transmit holding register if bit 5 of the Line Status Register is set.

The UART can be configured to be in character mode or FIFO mode. In character mode, the software can only read one byte at a time from the receive buffer register and send only one byte to the transmit holding register. In contrast, FIFO mode supports reading up to the trigger level of characters or sending 16 bytes of characters to the transmission FIFO.

1.1 Polling Mode

The UART can be set to polling mode at the boot stage. At this point, there is no IRQ service routine installed, no stack, and SDRAM is not enabled. In polling mode, all UART interrupts are disabled. Refer to Interrupt Mode on page 15 for details. After the UART is initialized, use the following routine to read/write characters to UART.

Macro to read from UART

```
    MACRO
     UART_READ_CHAR
     LDR  r4, =REG_IO_BASe                ;set register base address
     LDR  r0, =REG_UART_LINE_STATUS       ;get line status register offset
     ADD  r0, r0, r4                      ;get line status register address
     LDR  r2, =REG_UART_RX_BUFFER         ;get receive buffer register
     ADD  r2, r2, r4                      ;get receive buffer register address
     05  LDR  r12, [r0]                   ;read status
     ANDS r12, r12, #0x1                  ;check bit 0
     BEQ  %B05                            ;if data available
     LDR  r0, [r2]                        ;load data to r0
    MEND
```

Macro to write to UART, the character is stored in r0.

```
    MACRO
     UART_OUTPUT_CHARACTER
     LDR  r4, =REG_IO_BASE                ;set register base address
     LDR  r1, =REG_UART_LINE_STATUS       ;get line status register offset
     ADD  r1, r1, r4                      ;get line status register address
     LDR  r2, =REG_UART_TX_HOLDING        ;get transmission holding register
     ADD  r2, r2, r4                      ;address
     MOV  r3, r0                          ;move character to r3
     05  LDR  r12, [r1]                   ;load line status
     ANDS r12, r12, #0x20                 ;check bit 5
     BEQ  %B05                            ;check if the transmission holding
                                          ;register is empty.
     STRB  r3, [r2]                       ;send out the character
    MEND
```

The advantage of polling mode is that it is very simple and reliable. It does not require any dynamic storage or stack, and there is no interrupt involved. The disadvantage is that the UART is relatively slow when compared with CPU and WAN or LAN DMAs. The macros given on page 14 spend most of their time waiting for the desired status. If this happens too often, the UART could block other tasks and degrade DMA performance. The UART can also be blocked if a task takes too long to complete. To overcome this, program the UART to interrupt mode as soon as the memory map is configured and the stack is initialized.

1.1 Interrupt Mode

Baud rate, word length, stop bit, and parity are programmed in the same manner as polling mode and interrupt mode. To program the UART to be interrupt driven, use the following steps.

Procedure

1. Set the interrupt bits in Interrupt Enable Register (INTEN 0xE204):

   UART Receive Interrupt Enable (bit 9)
   UART Transmit Interrupt Enable (bit 8)
   UART Line Error Status Enable (bit 10)

2. Create an interrupt service routine for these events.

   A buffer is also needed for storing the data received and the data to be sent out. Since interrupt mode is much more complicated than polling mode, be extremely careful in handling these interrupts so as to avoid entering a race condition.

Receive Interrupt

The Receive Interrupt status is cleared once data is read out of the receive buffer.

Transmit Interrupts

Transmit Interrupts are handled differently. This edge-triggered interrupt has to be cleared by writing 1 to the status bit. **Resetting the bit in the wrong order could cause unpredictable results**.

It is very important to clear the Transmit Interrupt Status before putting any data into the Transmit Holding Register. **Failure to do so may lead to unpredictable behavior for transmission**. However, do not clear the status if there is no data to transfer.

Refer to the diagnostic program source code on the CD for information on how to create the interrupt service routine and how to handle the two interrupts. In FIFO mode, software has to check the reason for the interrupt by checking the UART Status Register (USR 0xE020). The characters inside the FIFO reaching the trigger level, or simply timing out, can trigger the receiving interrupt. In the latter case, when the FIFO waits too long to reach the trigger level, the software needs to decide how many characters to read.

5 Interrupt Configuration

The KS8695P supports multiple interrupt sources with programmable priority. Interrupt requests can be generated by internal functional blocks as well as external pins, which are shared with GPIO pins.

There are two types of interrupts: a normal Interrupt Request (IRQ), and a Fast Interrupt Request (FIQ). The KS8695P interrupt controller has an interrupt status bit for each interrupt source. The following registers are used to control interrupt generation and handling.

Interrupt Mode Control Register (INTMC 0xE200): Defines each of the interrupt sources to be FIQ (set to 1) or IRQ (set to 0). This is the first register that has to be programmed before any interrupt occurs.

Interrupt Priority Register for WAN DMA (INTPW 0xE20C): There are 16 levels that can be assigned to any interrupt source. The index of 15 is the highest priority and 0 is the lowest. FIQ always has precedence over IRQ. If this register is not set, all the interrupts have the same priority. There are interrupt priority registers for WAN, LAN DMA, timer, UART interrupts, external interrupt, communication channel, and bus error (listed below). Different interrupts can be set at the same level of the priority.

Interrupt Priority Register for LAN DMA (INTPL 0xE214): Defines interrupt priorities for all LAN DMA interrupts.

Interrupt Priority Register for TIMER (INTPT 0xE218): Defines interrupt priorities for timer 0 and timer 1 interrupts.

Interrupt Priority Register for UART (INTPU 0xE21C): Defines interrupt priorities for all UART interrupts.

Interrupt Priority Register for EXTERNAL INTERRUPT (INTPE 0xE220): Defines interrupt priorities for all External interrupts.

Interrupt Priority Register for COMMUNICATION CHANNEL (INTPC 0xE224): Defines interrupt priorities for communications channel interrupts.

Interrupt Priority Register for BUS ERROR (INTBE 0xE228): Defines interrupt priority for bus error interrupt.

Interrupt Status Register (INTST 0xE208): Indicates the interrupt status, bit value 1 means the source has generated an interrupt request. The register value is not masked by the value of the Interrupt Mask Register or the value of the Interrupt Enable Register.

Some of the interrupts are level-triggered thus there is no need to clear the status. Some of the interrupts are edge-triggered and can only be cleared by writing 1 to the interrupt status bits. Special attention has to be paid to these types of interrupts. The program could be in an infinite loop if the status is not cleared, or program behavior could be unpredictable if cleared improperly.

This register is where the program has to write 1 to clear the edge-triggered interrupts.

Interrupt Masked Status Register (0xE22C): The value of this register is the logical AND of the interrupt Enable Register and the Interrupt Status Register. It always reflects the enabled interrupt status. The interrupt service routine should read this register to determine which interrupt has arrived and then handle it. Since this register is read only, the program cannot write to the register to clear the interrupt status. This has to be done by writing to the Interrupt Status Register.

Interrupt Enable Register (0xE204): Is the register where different interrupt requests can be enabled or disabled, meaning the logic is going to allow the interrupt request go to the CPU or not go to the CPU (disabled).

Interrupt Pending Highest Priority Registers (0xE230 and 0xE234): There are two Interrupt Pending Highest Priority Registers: one for FIQ and one for IRQ. If the program is designed to have interrupt priorities and serves the interrupt according to its priority, the interrupt service routine has to read this register to identify the pending interrupts with the highest priority. It is normal to have more than one highest interrupts pending if they are set at the same priority level. Thus, the hardware will guarantee that the interrupts are served by priority. In this case, the program should not read the Masked Interrupt Status Register.

Special attention has to be paid to some of the DMA interrupts; for example, Frame Receive Interrupt, which is edge-triggered and has to be cleared before processing any received frame. The interrupt should not be cleared if the received frame cannot be processed. Failure to do so will cause the Receive DMA receive interrupt to be lost. If the receive interrupt is cleared and the frames do not get processed, the DMA will not have receive buffers available. As a result the DMA will not trigger any interrupts after it finds that there are no more buffers for the received frames.

6 Setting Up WAN and LAN DMAs

The KS8695P has two DMA engines: the LAN DMA and the WAN DMA. Each DMA engine can transmit and receive packets independently. Descriptor lists are used for host software to transmit and receive packets to and from the DMA engines.

The KS8695P supports a multi-frame, multi-fragment memory gathering process. Descriptors representing frames are built and linked in system memory by a host processor. The TxDMA Logic is responsible for transferring the multi-fragment frame data from the host memory into the TxFIFO.

The TxDMA Logic monitors the amount of free space in the TxFIFO, and uses this value to decide when to request a TxDMA.

Each of the KS8695P transmit DMA's use 3 KB of transmit data buffer between the TxDMA Logic and the Transmit MAC. When the TxDMA Logic determines there is enough space available in the TxFIFO, the TxDMA Logic moves any pending data frame into the TxFIFO.

The RxDMA Logic also supports a multi-frame, multi-fragment DMA scatter process. Descriptors representing frames are built and linked in system memory by the host processor. The RxDMA Logic is responsible for transferring the frame data from the RxFIFO to the host memory. The RxDMA Logic monitors the number of bytes in the RxFIFO. After the entire frame has been received, the frame becomes "visible." Then the RxDMA Logic issues a request to the arbiter to transfer the frame data to the memory buffer.

Each DMA engine works at three different states: idle, suspend, and stop. The idle state is the normal state for a DMA ready to transmit or receive frames. The suspend state is entered if the descriptors are not available. The suspend state can change to idle state once the above conditions are no longer valid and a RxDMA start or TxDMA start command has been sent.

6.1 Frame Transmission and Reception

The TxDMA Logic transfers frame data from the system memory to the TxFIFO based on linked list frame descriptors called TFDs. Each TFD contains the memory location of one fragment of a frame as shown in Figure 6.1

TFD

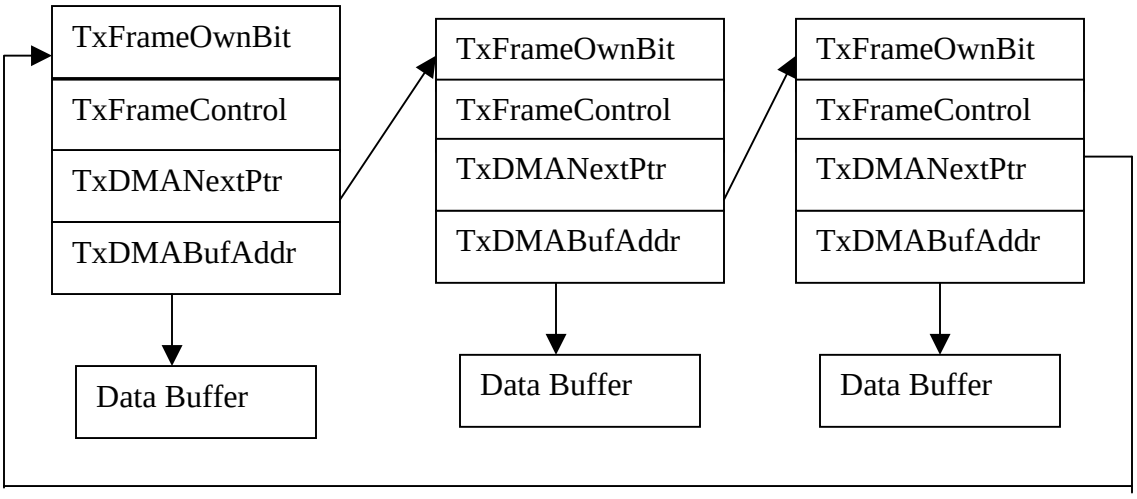


Figure 6.1. TxDMA Descriptor List

RxDMA mechanism is similar to the TxDMA. RxDMA is structured around a linked list of frame descriptors, called RFDs. RFDs contain pointers to the fragment buffers into which the KS8695P places received data, as shown in Figure 6.2. One RxDMA receive interrupt is generated for each frame received.

A receive descriptor list and the associated buffers have to be created prior to receiving a frame. One approach is to allocate buffers large enough to hold a maximum size Ethernet frame (1536 bytes for full duplex mode and 1600 bytes for half duplex mode) and have the descriptors point to them. The other approach is for the system to request the buffers from the protocol ahead of time.

The DMA engines are in the stop stat after reset, activating the DMA engines by setting up the descriptor list, issuing start commands, and enable DMA.

RFD

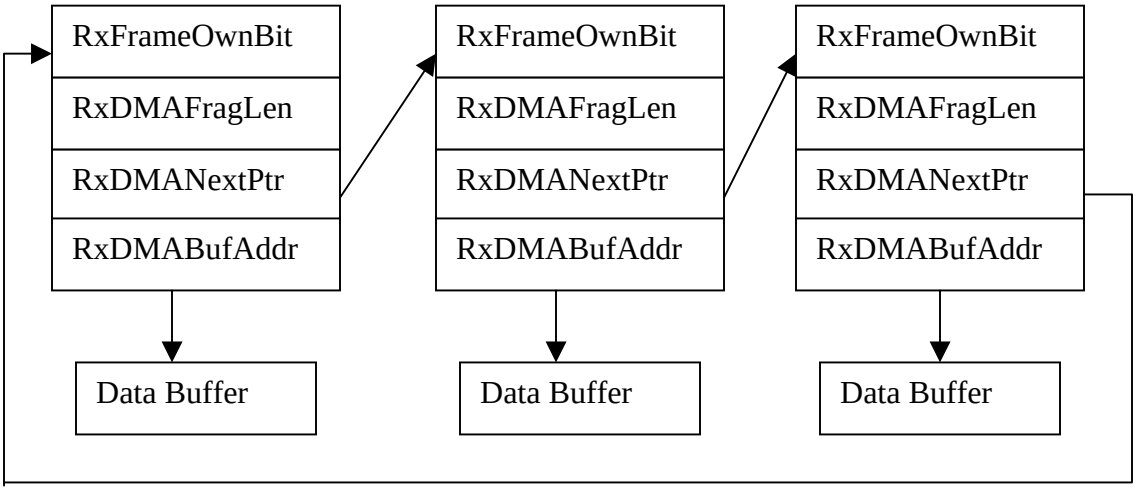


Figure 6.2. RxDMA Descriptor List

One way for host software to efficiently utilize the DMA engines is to create a descriptor link list. The “next descriptor pointer” of the last descriptor in the list should point to the address of the first descriptor to form a ring. Host software sets the OWN BIT accordingly after chaining the descriptors together. Before starting the DMA engine, the descriptors for transmitting packets should have OWN BIT cleared (owned by host software), and the descriptors for receiving packets should have OWN BIT set (owned by DMA engine). For this manner, the software sets bit 31 of TDES0 register.

6.2 Set Descriptor List Base Address

Several DMA registers have to be programmed in order to start DMA engines. The first step is to let DMA engines know where to access the descriptor list. This is accomplished by writing the physical address of the first transmission and receiving descriptor to the Transmit Descriptor List Base Address Registers and Receive Descriptor List Base Address Registers, respectively, for the WAN DMA and LAN DMA, as listed below:

WAN Transmit Descriptor List Base Address Register (WTDLB 0x6010)
WAN Receive Descriptor List Base Address Register (WTDLB 0x6014)

LAN Transmit Descriptor List Base Address Register (WTDLB 0x8010)
LAN Receive Descriptor List Base Address Register (WTDLB 0x8014)

6.3 Configure DMA Operation Mode and Enable DMA Engines

Each DMA engine has control registers for setting transmission and receiving operation modes. The register offsets are 0x6000 and 0x6004 for the WAN DMA engine and 0x8000 and 0x8004 for the LAN DMA engine.

Set bit 0 of these registers to put DMA engines in a running state. At this point DMA is ready to transmit and receive frames. It is waiting a start command to be issued and then goes through the descriptor list until all the descriptors are consumed. The RxDMA Logic enters suspend state if there are no more receive descriptors available. The TxDMA Logic enters suspend state if there are no more frames to transmit.

6.4 Start DMA Engines

Start the DMA engine after descriptor list is prepared and DMA operation mode is configured. Write any non-zero value to Transmit Start Command Register at offset 0x8008 or 0x6008 to start transmission operation of LAN DMA or WAN DMA, respectively. Write any non-zero value to Receive Start Command Register at offset 0x800C or 0x600C to start receiving operation of LAN DMA or WAN DMA, respectively.

6.5 Stop DMA Engines

Clear bit 0 of DMA control registers (WMDTXC 0x6000, LMDTXC 0x8000) stops the DMA engine after the completion the current transmission or receiving operation.

Once the DMA is put into stop state, there are two ways to restart the DMA engine: soft reset and DMA re-enable followed by DMA start command. Each DMA engine can be soft reset by writing 1 to the Transmit Control Register bit 31. In this way, both Transmit and Receive DMA are reset and all the DMA registers are put back to the default values. This means that all the DMA registers have to be programmed again after reset. Be sure to wait at least 1.7 msec after reset before reprogramming and re-enabling the DMA engine.

For each DMA, there should be one MAC address written into its Station Address Registers (WMAL 0x6018, WMAH 0x601C for WAN and LMAL 0x8018, LMAH 0x801C for LAN). Another way to restart DMA is to write 1 to the bit 0 in DMA control register and then issue a start command.

6.6 Sample Code

For an example of how to set up DMA for frame transmission, refer to the source code of the KS8695P Diagnostic Program and Linux DMA driver. The complete Linux 2.4 kernel and ARM patch source code are also included.

7 Program Timers

There are two timers in the KS8695P: timer0 and timer1. Timer 0 can be programmed as watchdog timer by putting 0xFF into the byte 0 of the Timeout Count register (T0TC 0xE408). The timer has to be disabled first in order to reprogram the time count value. The watchdog timer will reset the KS8695P once it is expired.

The timer can be used to generate waveform output from TOUT1 and TOUT0 pins. The waveform can be configured by setting the pulse value to the Pulse Count Register.

The timer counter starts to decrease itself as soon as it has been enabled through the Timer Control Register. It can generate interrupt once the counter gets to zero. It is up to the user to determine the need for using an interrupt or not. If an interrupt is not used, the user needs to poll for timer expiration. The watchdog timer will reset the system automatically regardless of whether the timer interrupt is enabled or not.

The timer programming sequence should be as follows:

1. Programming timer value to timer counter register
2. Programming timer pulse counter register
3. Enable timer interrupt if necessary
4. Enable timer to let it to start to work
5. Processing timer interrupt if necessary

The following is an example for programming timer0 to be a one second watchdog timer and timer1 to be 10ms timer.

Procedure

1. Determine the values to fill the timer registers. The formula is

   T(time interval) = (Timer data value + Pulse data value)/25000000

To set Timer0 as watchdog timer. The lowest byte of the Timeout Count register should be 0xFF so that the equation could be

1 = (Timer data value + Pulse data value)/25000000 for 1 second watchdog timer.

Timer data value + Pulse data value = 25000000

You can pick Timer data value = 1249967 or 0xBEBBFF to be the watchdog setting.
thus

Pulse data value = 25000000 - 1249967 = 12500033 or 0xBEBEC41

then program these values into the timer data registers as below

```
/* 1000 MS WATCHDOG TIMER*/

REG_TIMER0 = 0xBEBBFF
REG_TIMER0_PCOUNT = 0xBEBEC41;
```

In the same way, you can set 10 ms timer as below. (not a watchdog timer setting):

Assume

Timer data value = Pulse data value

so that

0.01 = 2 * Timer data value/25000000

Timer data value = 125000 or 0x1E848

```
REG_TIMER1 =  0x1E848 ;
REG_TIMER1_PCOUNT =  0x1E848;
```

then enable timer 0 and timer 1

```
REG_TIMER_CTRL =  3
```

enable KS8695P timer interrupt if necessary

```
REG_INT_ENABLE = 0xC0;
```

enable CPU interrupt

Your ISR for timer1 should look something like:

```
__irq void UartIrqHandler(void)
{
        unsigned int intstatus, linestatus, check, index, ch, i;

        DisableIRQ();
```

```
        intstatus = IO_READ(REG_INT_STATUS);

        if ( intstatus & 0x80 ) /*timer 1*/
        {
                IO_WRITE(REG_INT_STATUS, 0x80); /*clear int*/
                /*put your timer handling code here*/
        }
        EnableIRQ();
}
```

8 GPIO

The KS8695P provides eight GPIO pins for input or output. These pins can be used to help debug both the hardware and software by writing specific bit patterns to them when there is no other means of display to see any debug information.

Programming GPIO pins is straightforward. There are three registers to program. The first one is the GPIO mode register. This register controls each GPIO pin. Each pin can be configured as either an output or input pin but they cannot be configured identically. The default configuration is input for all the pins. Pin 4 and 5 are shared with timer 0 and 1. When these pins are configured to be the timer output, it overrides the I/O mode configuration. Pin 0 to pin 3 are shared with external interrupts. When these pins are configured as output pins, the CPU can generate a soft interrupt by writing appropriate data to the Port Data Register.

The I/O Port Control Register (IOPC 0xE604) determines each of the GPIO pins to be the normal GPIO or for timer output or external soft interrupt pins. There are bits for external interrupt trigger mode settings.

The last register is the I/O Port Data Register. It is used to send data out on the GPIO pins.

9 Switch Programming

The KS8695P switch block provides different ways to program the switch. All the major functions blocks are covered in this chapter.

The switch block includes four LAN ports and one LAN DMA port connected to the CPU internally. There is a switch engine to handle all incoming and outgoing traffic through all of these ports. The switch is disabled at reset, which means that the switch block will not receive and transmit any frames.

Note that there is a need for a 200 nsec delay between switch register reads and writes (registers 0xE8xx). Failure to do so will lead to unpredictable switch settings. Please see the KS8695P Functional Specification for switch register details.

The first step is to configure each port to set features such as Auto Negotiation, VLAN tag, spanning tree learning, broadcast storm protection, and others.

The next step is to turn on the switch by setting the bit 0 of the Switch Engine Control 0 register (SEC0 0xE800). This register also contains the bits to set LED functions for each port.

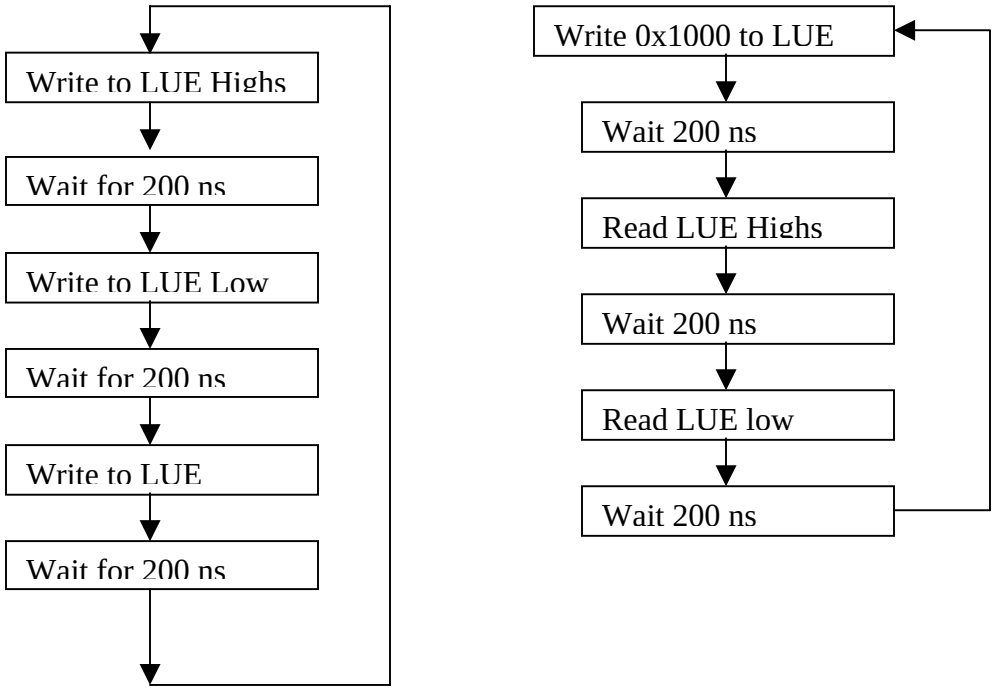
The Auto Negotiation (AN 0xE848-0xE84C) Register specifies the capability for each port and also keeps the result of the auto negotiation. The program should also always check the Link Status (bit 23 and 7) for each of these registers to see if the AN is completed, or not, since the link-up signal comes after the AN is completed. The AN complete bit will never be set if the link partner is in force mode; waiting for this bit to become 1 may hang the program in force mode.

The switch look-up engine supports static table dynamic table and a VLAN table. Each entry in the static table specifies a MAC address as the frame source and a port map, which specifies the frame with the matched MAC address that will be sent to the port number specified in the port map. Following steps are necessary to set up a static table.

Procedure

1. Prepare the table and write one entry at a time to the Look-Up Engine Indirect Access Data Register High I and II and Look-Up Engine Indirect Access Data Register Low (SEIADH1 and 2 0xE854 and 0xE858 and SEIADLEIL 0xE85C).
2. Then write another entry to the Look-Up Engine Control Register (SELUEC 0xE850) bit 11 and bit 10 = (0, 0).
3. Wait for 200 ns, and then repeat writing another entry.

   Figure 9.1 on page 27 shows the static table write/read.



(a) **(b)**

Figure 9.1. Static Table Write (a) /Read (b)

This dynamic table cannot be modified by an application program. The switch engine will learn and set it up automatically. The program can only read dynamic table entries.

Figure 9.2 on page 28 shows the steps for reading dynamic table entries.

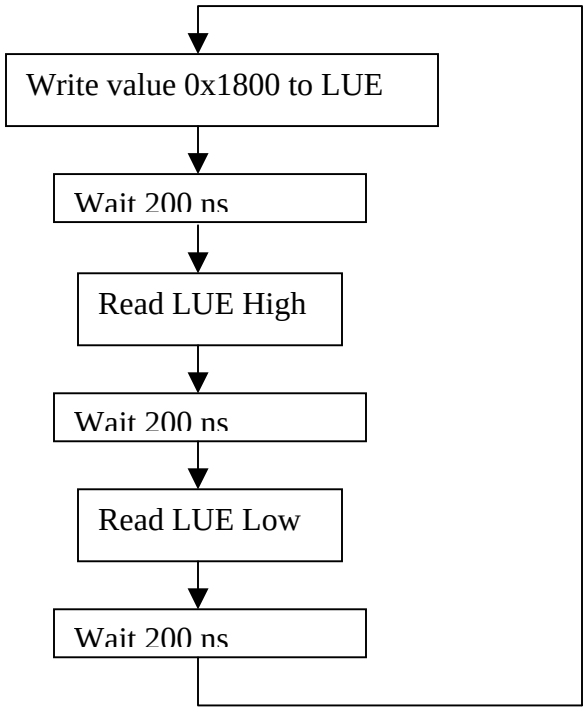


Figure 9.2. Dynamic Table Read

9.1 Direct Mode

The KS8695P switch engine can be programmed to Direct Mode to bypass the MAC table lookup. This is done through the VLAN tag setting in the port configuration register and also by the Switch Engine Destination Port Map setting in the transmit descriptor.

The KS8695P Receive Direct Mode and Transmit Pre-tag Mode are control modes that the switch engine uses to determine how to route the packet between the control port (port 5, which is connected to the CPU) and the four LAN ports. The control modes are mainly used for configuring the VLAN for Ingress and Egress packets.

When the Receive Direct Mode bit (bit 14 Port 5 Configuration Register) is set, the switch engine bypasses the dynamic and static lookup tables. This means that the packet switching is not dependent on the MAC address for each LAN port; rather, it depends on the port map.

The user has to specify the ports that port 5 can transmit and receive packets from; the user has to specify the transmission and communication ports for each port. Once a packet arrives at the switch engine from any LAN port, it is examined and passed to the specified port(s). When transmitting a packet to any LAN port through port 5, the port map has to be given in the transmit descriptor for this packet (bit 23:20 for the Transmit Descriptors).

The bits definition is listed below:

bit 23 22 21 20
- 1 1 1 1 broadcast, the packet is sent to all ports
- 0 0 0 1 the packet is sent to port one
- 0 0 1 0 the packet is sent to port two
- 0 1 0 0 the packet is sent to port three
- 1 0 0 0 the packet is sent to port four

You can put any combination of above table in the port map.

There is also a port map in the receive descriptor for each packet, which specifies the origin of the packet. The bit definition is as below:

bit 21 20
- 0 0 the packet is from port 1
- 0 1 the packet is from port 2
- 1 0 the packet is from port 3
- 1 1 the packet is from port 4

In our diagnostic program, the direct mode is set and each port can only talk to port 5; however, port 5 can talk to all four LAN ports.

The following registers have to be programmed in order to use direct mode.

Port 1 Configuration Register (0xE808)
Port 2 Configuration Register (0xE80C)
Port 3 Configuration Register (0xE810)
Port 4 Configuration Register (0xE814)
Port 5 Configuration Register (0xE818)

The port map must also be set for each packet in its transmit descriptor.

As a rule, do not use direct mode unless you know exactly what you are doing.

Do not put the OS kennel in the non-bufferable and non-cacheable areas. By doing so, you treat the ARM as an ARM processor. Only put memory I/O access and DMA buffers in these areas.

The diagnostic program is an example of how to use direct mode to do the port loop back.

9.2 Enabling Half-Duplex Back Pressure Flow Control as the Default Setting

Back pressure flow control should be enabled as a default for the half-duplex mode. The register bits for enabling this mode for the Switch Engine, WAN, and LAN ports are as follows:

WAN MAC Port	Bit 9 of offset 0x6000
LAN MAC Port	Bit 9 of offset 0x8000
Switch Engine	Bit 5 of offset 0xE800

10 System Performance Enhancement

Chapter 10 addresses how the KS8695P hardware can be enhanced to accelerate the execution of application programs; however, code optimization is not covered in this chapter since it is usually customer dependent. Areas covered are writing fast IRQ or FIQ service routine, choosing which interrupts to handle and setting up interrupt priorities, and scheduling tasks.

Most of the Ethernet devices support auto negotiation and flow control, but the software still needs to configure (disable or enable) flow control according to the auto-negotiation results of the WAN and LAN ports. To reduce the auto-negotiation time and make the program start faster with auto MDIX mode, clear the following register bits:

Register 0xE84C bit 7 and 23.
Register 0xE850 bit 7 and 23.
Register 0xEA10 bit 7.

10.1 Higher System Clock Speed

The KS8695P runs on the system clock from 25 MHz to 125 MHz. The System Clock and Bus Control Register (CLKCON 0x0004) set the system clock. Table 10.1 on page 32 includes all the clock speeds that the KS8695P supports.

By default, system bus and CPU are running in synchronous mode; that is., the system clock is the same as the CPU clock. The speed of the default clock is 25 MHz. Changing the values of the System Clock and Bus Control Register will change the clock speeds for both the system bus and the CPU.

The KS8695P also supports asynchronous mode; that is, the system bus and the CPU running at different clock speeds. Modifying register values will change both the bus clock and the CPU clock speeds although they can run at different speeds.

If the system bus and CPU are in asynchronous mode, the program has to put the system bus and CPU back to the synchronous mode before changing the clock value. This means that changing the system clock should be always done in synchronous mode. After the change, the program can set the system clock to asynchronous mode.

Bit 31:30 of the Coprocessor 15 control register determines the CPU mode. The CPU can run at a higher clock speed than the system bus clock in asynchronous mode.

Register Value	System Bus Clock	CPU Clock in Asynchronous Mode
0	125 MHz	166 MHz
1	100 MHz	
2	62.5 MHz	83 MHz
3	50 MHz	
4	41.7 MHz	55.3 MHz
5	33.3 MHz	
6	31.3 MHz	41.5 MHz
7	25 MHz	

Table 10.1 KS8695P System Clocks

The following ARM instruction sets can be used to read and write co-processor control registers where Rd is the ARM register that contains the value to the co-processor or reading value from the co-processor. .

```
MRC P15, 0, Rd, C1, C0, 0    ;Read control register
MCR P15, 0, Rd, C1, C0, 0    ;Write control register
```

10.2 I-Cache, D-Cache, and MMU

The KS8695P has an embedded ARM 922T core that has an 8KB instruction cache (I-Cache) and an 8KB Data cache (D-Cache). The I-Cache can be turned on without enabling the MMU. However, the MMU has to be enabled in order to use the D-Cache. Refer to the *ARM Architecture Reference Manual* for more details about the memory access sequence.

Register 1 of Coprocessor 15 is used for configuring the MMU, I-Cache and D-Cache.

Set the bit 12 of register 1 to enable I-Cache. The following code segment shows how to enable I-Cache.

```
MRC p15, 0, r1, c1, c0 ,0       ;Read control register
MOV r5, #0x1000                 ;Set enable bit
ORR r1, r1, r5                  ;Or enable bit into the read value
MCR p15, 0, r1, c1, c0 ,0       ;enable I cache

Make sure the pipeline is clear of any cached entries
NOP
NOP
NOP
```

Program performance can be significantly improved by turning on the I-Cache, which is 8KB. This is enough to store most of the frequently used codes, thereby boosting system performance.

To disable the I cache, use the following code segment:

```
STMFD    r13!, {r1-r5, r14}
MCR p15, 0, r1, c7, c5, 0
MRC p15, 0, r1, c1, c0 ,0
BIC   r1, r1, #0x1000
MCR p15, 0, r1, c1, c0 ,0
LDMFD    r13!,{r1-r5,pc}
```

Enabling the D-Cache is more complicated. The MMU has to be enabled and then the Address Translation Table has to be created. The Address Translation Table converts a virtual address into a physical address. Each entry in the TLB has an attribute that decides whether the region is cacheable or/and bufferable.

There are two kinds of tables: a first level table and a second level table.

The *first level table* is 4096 bytes and has a 16KB boundary. This table has 1000 entries; each entry controls the mapping of 1MB of virtual memory space.

The coarse page controls 4KB of memory in each entry of the *second level table*. There are 256 entries in a coarse second level table. If the second level table is a fine table (1KB for each entry), there should be 1000 entries for each fine second level table. Please refer to the *ARM Architecture Reference Manual*, chapter B3, for more information about memory management and how the translation table works.

After the TLB is created, the base of the first level table has to be written into register 2 and the Translation Table base register. Then the register 3 Domain Access Control Register has to be set accordingly. Flush both I-Cache and D-Cache before enabling them by programming register 8, the TLB function register. Enable the MMU and the D-Cache by programming the control register after all the tables are properly configured.

The following code segment is an example of the KS8695P Diagnostic Program. It demonstrates how to set up the MMU and enable the D-Cache and Read/Write buffer for some of the SDRAM area.

The TLB will be stored at SDRAM address 0x00040000. The code generates a flat memory map, which means that both the physical and virtual addresses have the same values.

/* MMU Level 1 Table generation data */

LEVEL 1 TABLE

VIRTUAL ADDRESS	PHYSICAL ADDRESS	ENTRY TYPE	ACCESS RIGHT	CACHE, BUFFER
0x00000000 -- 0x001FFFFF	0x00000000	SECTION	FULL_ACCESS	C & B
/*normal SDRAM area used to store code, data, and ISR.*/				
0x00200000 -- 0x002FFFFF	0x00200000	SECTION	FULL_ACCESS	NOT C, NOT B
/*a SDRAM area specified to be used as frame buffer for DMA purpose, must set to be not cacheable and not bufferable*/				
0x00300000 -- 0x00FFFFFF	0x00300000	SECTION	FULL_ACCESS	C & B
0x01000000 -- 0x027FFFFF	0x01000000	FAULT		
/*memory hole, access to this area will cause data abort*/				
0x02800000 -- 0x02FFFFFF	0x02800000	SECTION	FULL_ACCESS	NOT C, NOT B
/*space for flash memory, must set to be not cacheable and not bufferable in order to program flash*/				
0x03000000 -- 0x03EFFFFF	0x03000000	FAULT		
/*memory hole, access to this area will cause data abort*/				
0x03F00000 -- 0x03FFFFFF	0x03F00000	PAGES		
/*KS8695P register bank , must set to be not cacheable and not bufferable, set as second level table in this example*/				
0x04000000 -- 0xFFFFFFFF	0x04000000	FAULT		
/*memory hole, access to this area will cause data abort*/				

In the table above, the first MB of memory space is set as a section entry and is cache-able and bufferable.

The second MB is used to store the DMA descriptor list; it should not be cache-able and bufferable. It is a section page.

Memory 3MB to 16 MB is a section page and is cache-able and bufferable. This area is used to store program code and variables.

Memory space 16 to 40MB is a memory hole meaning that there is no physical memory in this area and thus set as a FAULT page.

Memory space 40 to 48 MB is reserved for flash memory. It is not cache-able or bufferable.

Memory 64 MB is reserved for register I/O, which is the page entry.

Memory 65MB to 4 GB does not exist. It is set to be a FAULT page.

Since the register I/O space is a page entry, it involves second level table. The requirement of this table is shown in the following Table:

LEVEL 2 TABLE

VIRTUAL ADDRESS	PHYSICAL ADDRESS	ENTRY TYPE	ACCESS RIGHT	Cacheble, Bufferable
0x03F00000 TO 0x03FEFFFF	0x03F00000	FAULT		
/*none used space */				
0x03FF0000 TO 0x03FFFFFF	0x03FF0000	LARGEPAGES	FULL_ACCESS	NOT C, Not B
/*KS8695P register bank*/				

In this page, only memory from 0x03ff0000 to 0x03ffffff can be accessed. It is not cache-able or bufferable.

Two tables are generated according to above requirement. This is given below. The code showing how to load these tables into the ARM co-processor is also given.

NOTE: IT IS VERY IMPORTANT THAT THE ENTRIES SHOWN BELOW IN THE SECOND LEVEL TABLE ARE IN THE ORDER THEY ARE REFERENCED IN THE LEVEL 1 DATA (above). For a completed list of the tables, refer to the code given in the accompanying KS8695 CD.

```
LEVEL1TABLE
  DCD   0x00000c1e, 0x00100c1e, 0x00200c12, 0x00300c1e, 0x00400c1e, 0x00500c1e, 0x00600c1e, 0x00700c1e
  DCD   0x00800c1e, 0x00900c1e, 0x00a00c1e, 0x00b00c1e, 0x00c00c1e, 0x00d00c1e, 0x00e00c1e, 0x00f00c1e
  DCD   0x01000000, 0x01100000, 0x01200000, 0x01300000, 0x01400000, 0x01500000, 0x01600000, 0x01700000
  DCD   0x01800000, 0x01900000, 0x01a00000, 0x01b00000, 0x01c00000, 0x01d00000, 0x01e00000, 0x01f00000
  DCD   0x02000000, 0x02100000, 0x02200000, 0x02300000, 0x02400000, 0x02500000, 0x02600000, 0x02700000
.....
  DCD   0xff800000, 0xff900000, 0xffa00000, 0xffb00000, 0xffc00000, 0xffd00000, 0xffe00000, 0xfff00000

LEVEL2TABLE
  DCD   0x03f00000, 0x03f01000, 0x03f02000, 0x03f03000, 0x03f04000, 0x03f05000, 0x03f06000, 0x03f07000
  DCD   0x03f08000, 0x03f09000, 0x03f0a000, 0x03f0b000, 0x03f0c000, 0x03f0d000, 0x03f0e000, 0x03f0f000
.....
  DCD   0x03fe0000, 0x03fe1000, 0x03fe2000, 0x03fe3000, 0x03fe4000, 0x03fe5000, 0x03fe6000, 0x03fe7000
  DCD   0x03fe8000, 0x03fe9000, 0x03fea000, 0x03feb000, 0x03fec000, 0x03fed000, 0x03fee000, 0x03fef000
  DCD   0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1
  DCD   0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1, 0x03ff0ff1
```

```
        LDR r1, =LEVEL1TABLE
        MCR p15, 0, r1, c2, c0 ,0

        ;Initialize Domain Access Control
        MOV r1 , #1
        MCR p15, 0, r1, c3, c0 ,0

        ;flush I TLB and D TLB
        MCR p15,0,r1,c8,c5,0
        MCR p15,0,r1,c8,c6,0
```

```
MOV r1, #(ENABLE_MMU :OR: ENABLE_DCACHE :OR: ENABLE_WB)
ORR r1, r1, #ENABLE_ICACHE

MRC p15, 0, r2, c1, c0 ,0


BIC r2, r2, r1  ; Clear the bits that we may change
ORR r2, r2, r1  ; Merge in NEW BITS

; Write the following to the control register
 MCR p15, 0, r2, c1, c0 ,0

; Make sure the pipeline is clear of any cached entries
NOP
NOP
NOP
```

To disable the Dcache and MMU, use following code segment:

```
STMFD    r13!, {r1-r5, r14}
MCR p15, 0,  r1, c7, c6, 0
MRC p15, 0, r1, c1, c0 ,0
BIC   r1, r1, #4
MCR p15, 0,  r1, c1, c0 ,0

; Make sure the pipeline is clear of any cached entries
NOP
NOP
NOP
LDMFD    r13!,{r1-r5,pc}
```

10.3 Customize Address Translation Table

With slight modifications, many real applications can use the above TLB. The first instance is when the TLB is not loaded onto memory address 0x00040000. In this case, the section entry value for the KS8695P register bank has to be adjusted. The following code segment can be used to do this. The code segment has included the misc.s as one function call.

```
AdjustLevel2Talbe
        STMFD    r13!, {r1-r2, r14}
        LDR   r1, =LEVEL1TABLE
        LDR   r2, =0x4011
        ADD  r2, r1, r2
        ADD  r1, r1, #252
        STR   r2, [r1]
   LDMFD    r13!,{r1-r2,pc}
```

LEVEL1TABLE has to be given at link time. For example, the diagnostic program's scatter load file explicitly loads the tlb.o to the address 0x00040000, which is the value of LEVEL1TABLE.

Value 252 is the relative position of the register bank first level entry in bytes.

In our sample code, this entry has a value of 0x00044011, which is the pointer pointing to the second level table. In this case, the second level table is at:

LEVEL1TABLE + the length of the first level table,

where 0x00040000 + 0x00004000 = 0x00044000. The last two bytes in 0x000044011 need to be specified that the table entry is a coarse table.

The only modification needed here is to change this entry to reflect the changing of the LEVEL1TABLE; for example, if you set the LEVEL1TABLE to 0x60000, then the entry is going to be 0x000064011.

The second case is to configure the register bank as a section table. The entry here should be changed to 0x03f00c12 rather than 0x00044011. In this case, it is not necessary to be concerned about where to load the TLB as there is no second level table. Simply load the TLB to any address, based on the fact that its lower 14 bits are zero. The program will work without any problem.

The third case is if you have more then one page table. In this case, manually add some second level tables. The following procedure gives the steps.

Procedure

1. Modify the entry in the first level table. Refer to the *ARM Architecture Reference Manual* for information on how to set the value for the entry in the first level table.
2. Generate second level table and put it in order in the tlb.s file. Refer to the *ARM Architecture Reference Manual* for information on how to set the value for the entry in the second level table.

If you have a memory map that is very different from our sample memory map, you may need to create your own TLB. Please refer to the *ARM Architecture Reference Manual* and mmugen sample code for how to create a MMU table.

For example, create the TLB by selecting a LEVEL1TABLE as the starting address of the TLB. Then create the Table. This Table has to be loaded onto the address starting at exactly at the LEVEL1TABLE. Make sure that the LEVEL1TABLE is 14 bits aligned, meaning the last 14 bits of it should be "0s" or your code will not work.

It is also possible to generate TLB on the fly by putting the TLB generating code into the program. By doing this, it will not be necessary to select the starting address in advance and create the TLB offline. However, this approach is more difficult to debug due to the fact that if your code has bugs in it, it may generate wrong TLB and this situation is not easy to detect.

10.4 FIQ and IRQ

The ARM processor has two levels of external interrupt, FIQ and IRQ, both of which are level-sensitive active LOW signals into the core. For an interrupt to be taken, the appropriate disable bit in the CPSR must be clear.

FIQ has higher priority than IRQ in two ways:

- FIQs are serviced first when multiple interrupts occur.
- Servicing an FIQ causes IRQs to be disabled, preventing them from being serviced until after the FIQ handler has re-enabled them. This is usually done by restoring the CPSR from the SPSR at the end of the handler.

The FIQ vector is the last entry in the vector table (at address 0x1C) so that the FIQ handler can be placed directly at the vector location and run sequentially from that address. This removes the need for a branch and its associated delays, and also means that if the system has a cache, the vector table and FIQ handler may all be locked down in one block within it. This is important because FIQs are designed to service interrupts as quickly as possible. The five extra FIQ mode banked registers enable status to be held between calls to the handler, again increasing execution speed.

Installing an FIQ service routine is more difficult then installing an IRQ service routine. It involves copying the whole service routine to the address started at 0x1C rather then setting up a jump instruction over there.

10.5 SDRAM Read Buffer and Write FIFO

The KS8695P memory controller also provides a Read Buffer and Write FIFO to accelerate the memory access. Turning on these features will make the memory access in burst fashion. The Read Buffer and Write FIFO are controlled by SDRAM Buffer Control Register (SDBCON 0x403C). The Read Buffer and Write FIFO can be enabled separately. It is necessary to turn off the Read Buffer in some case where the Read Buffer may not reflect the value change in the physical location such as programming flash memory. The program may fail to read the toggle bit since the memory controller does not have any information about the change of the data on the flash memory bus.

10.6 DMA Burst Size

Each DMA engine has bits to control the burst size, the maximum number of words (32 bits/word) to be transferred by one DMA request. The burst size can be 0, 1, 2, 4, 8, 16, and 32. 0 means unlimited size. Changing this number has an impact on system performance but it is not as great as the other factors mentioned above.

10.7 Protocol Engine

The KS8695P has a built-in Protocol Engine to offload CPU utilization by executing IP/TCP/UDP header checksum generation and validation in hardware, but not for ICMP packets. It can be used to improve overall system performance. The following is a Linux driver example describing the procedures to enable the hardware checksum generation:

- The DMA Transmit Control Register has the bits for checksum generation for the packets to transmit, bit 16 for IP checksum generation, bit 17 for TCP checksum generation, and bit 18 for UDP checksum generation, respectively. The default is disable.
- To enable the checksum generation, set the corresponding bit. NOTE that the bits are the same for all DMA Transmit Control Registers, including WAN (0x6000), LAN (0x8000) and HPNA (0xa000).
- The DMA Receive Control Register has the bits for checksum validation for the received packets: bit 16 for IP checksum, bit 17 for TCP checksum and bit 18 for UDP checksum ,respectively. The Default is disable. To enable the checksum checking, set the corresponding bit. NOTE that the bits are same for all DMA Receive Control Registers, including WAN (0x6004), LAN (0x8004) and HPNA (0xa004).

In addition to the checksum generation and validation, the following steps are needed in the Linux protocol stack:

- In driver probe routine, KS8695P_probe, set netdev->features =NETIF_F_HW_CSUM to notify the upper edge net device that the driver enables its hardware checksum offload features to offload stack's work. This flag implies that the driver will do both checksum generation for transmit packets and checksum checking for receive packets.
- If the driver wants to offload the checksum validation for receive packets only, set skb->ip_summed = CHECKSUM_UNNECESSARY will do it. Please refer to Process Rx Interrupts routine for the sample code.
- Several files are being modified to offload the Kernel. See linux/net/core/dev.c, linux/net/ipv4/ip_input.c, ip_output.c, tcp_ipv4.c and udp.c.

  A CONFIG_CSUM_UNNECESSARY flag is added for compilation. Do make menuconfig, go to Network device support/Ethernet (10/100 Mbit)/KS8695P, you can enable/disable IP/TCP/UDP Hardware Checksum.

10.7.1 Gain Setting for Performance Enhancement

To enhance the cable length performance of the KS8695P, simply change the following system control register.

Register Offset (Hex)	Value(Hex)
0xEA14	0x0200B000

11 PCI Interface

The KS8695P has two modes of PCI operation: the Host Bridge mode and the Guest Bridge mode.

In the Host Bridge mode, the CPU acts as the Host of the entire PCI environment. It configures the systems and coordinates all the transactions, including initiating transaction between a PCI device and AHB bus subsystem. In this mode, the CPU accesses all the Bridge registers and configures the PCI device connected.

In Guest Bridge mode, the KS8695P works as a slave on the PCI bus. In either case, the KS8695P memory subsystem is accessible from either the PCI Host or the ARM9 CPU. Communication between the Host and the ARM9 can be accomplished through message passing or shared memory.

Linux also has some requirements for driver development in order to avoid the data incoherent problem when the D-Cache is enabled. The driver's performance can be further improved by implementing the following rules:

1. Always call pci_alloc_consistent() to get a chunk of memory for tx, rx, or data buffer so that the kernel will mark these area as nonbufferable and noncache-able. The data is always coherent. pci_alloc_consistent() returns the virtual address for kernel usage and a physical address for pci device. Call pci_free_consistent() to free the memory region.
2. If your driver needs lots of smaller memory, use pci_pool APIs to do the allocation. The functions used are pci_pool_create(), pci_pool_alloc(). pci_pool_free(), and pci_pool_destroy().
3. Use streaming DMA mappings to improve performance. They can be called from interrupt context. The function calls are pci_map_single(), pci_unmap_single(), and pci_dma_sync_single().

If you use above functions to allocate and de-allocate the memory for DMA, you will not have data incoherent problem.

11.1 PCI Header Structure

The PCI standard has a specified header structure for PCI devices that are non PCI-to-PCI bridge devices. The configuration header allocates the first 16 double words of its configuration space to the PCI device, as shown in Figure 11.1 on page 43. Refer to references 3 and 4 for complete details about the header structure.

11.2 PCI Device Configuration

One of the new features of the KS8695P is the support of PCI bus. The following is the recommended procedure for PCI device configuration.

Step 1: Configure the Host Bridge

a. Set Subsystem ID and Rendor ID (Register Offset 0x202C).
b. Set Prefetch Limit (Register Offset 0x2204).
c. Set PCI-AHB Bridge Function Mode[1] and Bus Mode[2] (Register Offset 0x2200).
d. Set Host Bridge Memory Base Address (Register Offset 0x2208).
e. Set Host Bridge I/O Base Address (Register Offset 0x2218).
f. Set Host Bridge Memory Base Address Mask (Register Offset 0x2210).
g. Set Host Bridge I/O Base Address Mask (Register Offset 0x2220).
h. (Optional) Set Host Bridge Memory Base Address Translation[3] (0x2214).
i. (Optional) Enable Memory Address Translation (Register Offset 0x220C).
j. (Optional) Set Host Bridge I/O Base Address Translation (0x2224).
k. (Optional) Enable I/O Address Translation (Register Offset 0x2224).

The routine “Pci_HostBridgeCfg()” in the “pcidiag.c” of the Diagnostic Program has the complete source code for this step. The source code for the KS8695P Diagnostic Program can be found on the accompanying CD in the Board Support Package (BSP).

Step 2: Scan Devices

Scan the PCI bus looking for a valid device. Build a device node tree with all device found.

The routine “Pci_ScanDevices()” in the “pcidiag.c” of the Diagnostic Program has the complete source code for this step. The source code for the KS8695P Diagnostic Program can be found on the accompanying CD in the BSP.

Step 3: Assign Resources

Based on the device tree nodes to glean resource requirements for the device and assign resources to the device, accordingly.

The routine “Pci_AssignResources()” in the “pcidiag.c” of the Diagnostic Program has the complete source code for this step. The source code for the KS8695P Diagnostic Program is included in the BSP.

[1] The PCI-AHB Bridge Bus mode is selected by the power-up strap option.
[2] Select mini-PCI mode for the KS8695P Demo Board that has one mini-PCI connector.
[3] Enabling Memory and I/O Address Translation is optional if the address translation is not needed.

11.3 More About PCI Device Configuration

The PCI device connected to the PCI bus can be configured through the PCI-AHB Bridge. The bridge provides three windows to access an external PCI device: a configuration window, a memory access window, and an I/O access window as shown in Figure 11.2 on page 44.

The bridge has two registers for the configuration window: the AHB Bridge Configuration Address Register (PBCA, offset 0x2100) and PCI-AHB Bridge Configuration Data Register (PBCD offset 0x2104). The address register acts as the command register that selects the external device and generates type 0 and type 1 configuration cycles. The KS8695P demo board supports one external device; the device numbers are fixed at 0 for the bridge and 5 for the external device. Other values are ignored.

There is only one PCI bus; the bus number is always zero. For a single function PCI device, the function number is always zero.

In most case, the software is only able to set the subsystem ID, the subsystem Vendor ID, the base address registers, and the interrupt line to enable the PCI device.

31	24	15	7	0
Device ID			Vendor ID	
Status Register			Command Register	
Class Code				Revision ID
BIST	Header Type	Latency Timer		Cache Line Size
Base Address 0				
Base Address 1				
Base Address 2				
Base Address 3				
Base Address 4				
Base Address 5				
Base Address 6				
Subsystem ID			Subsystem Vendor ID	
Card Bus CIS Pointer				
Expansion ROM Base Address				
Reserved				
Reserved				
Max_Lat	Min_Gnt	Interrupt Pin		Interrupt Line

Figure 11.1. PCI Device Configuration Header Space Format

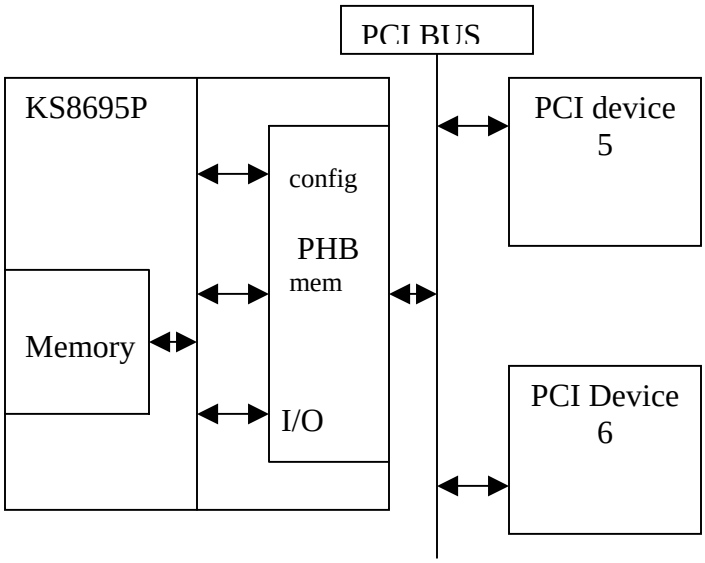


Figure 11.2. PCI-Host Bridge

The KS8695P PCI Bridge supports PCI revisions 2.1 and 2.2 protocols as well as the AHB bus 2.0 interface. The AHB-to-PCI downstream access allows AHB bus masters to access the PCI bus. The PCI-to-AHB upstream access allows PCI bus masters to access system resources on the AHB.

The Host Bridge includes all PCI specific configuration registers. For upstream access, it only support memory access.

The PCI-AHB Bridge (Host Bridge or PAB) implements all configuration registers required by the PCI specification. Since the configuration registers can be accessed from both the PCI and AHB buses, they have two sets of addresses, one for each bus: PCI and AHB. The PCI bus configuration register address range is 0x0000 – 0x00FC, and the AHB bus configuration register offset range is 0x2000 – 0x20FC.

The PAB enables a full software-driven initialization when acting as a host bridge. This allows the software to identity and query the PAB. In the guest bridge mode, the subsystem ID and subsystem vendor ID have been programmed by the CPU. The remaining configuration is completed by the host system. The first step is to set the PAB to work at Host or Guest mode and identify the type of bus that it has to control. This is accomplished by setting the PAB mode register (offset 0x2200). By default, the PAB is working in Host bridge mode, with the PCI bus selected.

Accessing reserved configuration registers is treated as a normal operation; however, the data is discarded. Trying to read all the reserved registers will return zero.

Software reset has no effect on the configuration registers; however, hardware reset sets the configuration registers to their default values.

There are two stages for initializing a PCI device: In stage 1 initialization, the CPU programs the subsystem ID and subsystem Vendor ID register thus eliminating the need for a EEPROM. The Prefetch Limit bits can be initialized in stage 1. Only internal accessing to these registers is done in stage 1. During stage 2 initialization, the software programs the remaining PAB configuration registers, and the PAB specific registers located at 0x2100 – 0x2224.

The PAB registers are located at the host memory address space. They are word aligned, 32 bits long, and must be accessed using word instructions with word-aligned addresses only. Trying to write to reserved bits of these registers will cause incompatibility problems with a future version of the PAB. Reserved bits are undefined on reading accessing.

Two PAB registers (PBCA offset 0x2100) and PBCD (offset 0x2104) are used to configure the PCI device on the PCI bus.

The PAB memory base address register (offset 0x2208) is used to set up the memory window for the PCI device and PAB. Any memory access within this range is treated as PCI memory access. There are three different kinds of memory addresses. The AHB memory address that is seen by CPU is called the host memory address, the PCI address is the memory address thatappears on the PCI bus. There is address translation between host memory address and PCI bus address. The third memory address is the memory base address for the PCI device on the configuration registers. This address is the PCI bus address assigned to the particular PCI device.

The address translation between AHB and PCI is controlled by the PAB memory base control register (offset 0x220C), the PAB memory base memory address mask (offset 0x2210), and the PAB memory base address translation (offset 0x2214).

For example, if you want to translate host memory range 0x60000000 – 0x7fffffff to 0x40000000 – 0x5fffffff on the PCI space, the above registers have to be programmed with following values:

Set PAB Memory Base Address Register (offset 0x2208) to be 0x60000000
Set PAB Memory Base Address Mask Register (offset 0x2210) to be 0xE0000000
Set PAB Memory Base Address Translation Register (offset 0x2214) to be 0x40000000
Set PAB Memory Base Address Control Register (offset 0x220C) to be 0x80000000

The I/O address translation between AHB and PCI is controlled by the PAB I/O Base Control Register (offset 0x221C), the PAB I/O Base Address Mask (offset 0x2220), and the PAB I/O Base Address Translation (offset 0x2224).

For example, if you want to translate host I/O range 0x80000000 – 0x8000ffff to 0x60000000 – 0x6000ffff on the PCI space, the above registers have to be programmed with following values:

Set PAB I/O Base Address Register (offset 0x2218) to be 0x80000000
Set PAB I/O Base Address Mask Register (offset 0x2220) to be 0xFFFF0000
Set PAB I/O Base Address Translation Register (offset 0x2224) to be 0x60000000
Set PAB I/O Base Address Control Register (offset 0x221C) to be 0x80000000

11.4 PCI Device Enumerator

When the KS8695P is first powered on, the configuration software must scan the PCI bus to determine what PCI device exists and what its configuration requirements are. This process is commonly referred to as scanning, waking, probing, or the discovery process. The program that performs the bus scan is frequently referred to as the PCI bus enumerator.

A subset of a device's configuration registers has to be read first to determine the presence of the device and its type. Having determined the presence of the device, the program then accesses the device's other configuration registers to determine how many blocks of memory and/or IO space the device requires. Then it programs the device memory and IO address decoders to respond to memory and IO address ranges that are guaranteed to be mutually exclusive from those assigned to other system devices.

If the device indicates usage of a PCI interrupt request line, the configuration software programs it with routing information indicating what system interrupt request line the device's PCI interrupt request line is routed to by the system. The interrupt line in the KS8695P demo board is fixed to the external interrupt 0 line.

If the device has bus mastering capability, the software can read two of its configuration registers to determine how quickly it requires access to the PCI bus when it asserts REQ# and the average duration of its transfer when it has acquired ownership of the bus. This information can be used to program the bus masters' latency timer register and the PCI bus arbiter to provide the optimal PIC bus utilization.

11.4.1 Configuring a PCI Device

Configuring a PCI device involves the following steps:

- Scan and identify a PCI device on local and bridged PCI buses
- Assign device resources in PCI memory and I/O space
- Allocate interrupt numbers
- Numbering the PCI-PCI bridges

There are three PCI address spaces:

- Configuration space
- I/O space
- Memory space

The device header has to provide the following information:

- Device type
- Device manufacturer
- How much PCI I/O space the device requires
- How much PCI memory address space the device requires

The PCI configuration code is run by the host bridge (i.e., the processor that owns PCI bus 0). Initializing the PCI subsystem is carried out in three phases:

Step 1: Host Bridge Initialization

1. Scan the local PCI bus for a PCI device.

   Some of the PCI devices found are PCI-PCI bridges. In this case, the PCI initialization code also scans for PCI devices.

Step 2: PCI-PCI Bridge Initialization Downstream

1. When initializing the PCI-PCI bridge, the code must number the PCI buses.
2. Then assign resources to the PCI devices.

   These resources are areas of PCI I/O and PCI memory. PCI devices must be given addresses in the PCI I/O and the PCI memory space. Both of these addresses must be enabled.

Step 3: Interrupt Numbers Assignment

1. Give interrupt numbers to the PCI devices that are meaningful to the device drivers in the application or operating system.

11.4.2 How to Find the Requirement for Memory or I/O Space

Scanning the returned reading value of base address upwards from bit four or bit two for I/O base address register, the binary weighted value of the first one bit found indicates the required amount of space.

For example; writing 0xFFFFFFFF to the base address register and the read back value of 0xFFF00000 means that the address decode is a memory address decoder and the range can be anywhere within 4GB. The first non-zero bit is bit 20 so the memory range is one MB.
After determining the size and type of address space the device requires, the programmer can specify the start address by writing the appropriate value to the register.

Since many embedded applications do not need full scale PCI enumeration, hard coding may be used to speed up the PCI device configuration. In this case, the software program needs to have the following data: information about the PCI device connected, how many of them are present, and do they support memory and/or I/O access. Once this is supplied, the software can write the configuration values directly to the PCI configuration header to enable it.

12 Using init.s

Init.s can be easily incorporated into an application with little modification. The code segment supports up to two SDRAM and two FLASH/ROM/SRAM devices as defined in KS8695.h. The memory size can be specified at compiling time through compiler switches. Init.s can be used to build a small boot code to initialize the board: an example is shown in the following memmap.bat.

The following is the batch file:

```
REM Micrel Kendin Operations, Copyright@ October 2004
REM ================================================
REM This batch file demonstrates how to build the memory
REM map for the Kendin KS8695P demo board.following options can be used for
REM different memory configuration:
REM ADD -PD "ROM_RAM_REMAP SETL {TRUE}" will build a remmaped
REM memory map in which SDRAM or SRAM will be start from 0x0
REM SDRAM size have to be specified here so that the code can
REM generate correct configuration for them. KS8695P support up to
REM two SDRAM devices, connected to bank0 and bank1 memory controller.
REM memory size of both banks have to be given here, 0 need to be
REM assigned to the bank 1 if you have only one SDRAM device. Here is an
REM example for two 8MB SDRAM devices.
REM -PD "SDRAM_BANK0_SIZE SETA 0x800000" and -PD
REM "SDRAM_BANK1_SIZE SETA 0x800000"
REM Put SDRAM_BANK1_SIZE to be 0 if you have only one SDRAM device.
REM put -PD "USE_SRAM SETL {TRUE}" if your board has SSRAM or SRAM so REM
that the
REM SSRAM or SRAM will be remmaped to address from 0x0 just ahead of the
REM SDRAM.

REM following setting is for KS8695P demo board with one SDRAM device of
REM16MB on board.

armasm -cpu 4T -PD "ROM_RAM_REMAP SETL {TRUE}" -PD   \
"SDRAM_BANK0_SIZE SETA 0x1000000" -PD "SDRAM_BANK1_SIZE SETA 0"  \  init.s
armlink init.o -o memmap.axf -ro-base 0x0 -rw-base 0x8000 -entry 0x0
fromelf -bin memmap.axf -o memmap.bin
```

To enable memory to work, there are other parameters such as data width (8, 16, or 32 bit), RAS, and CAS latency and column address bits as well as the number of banks of the SDRAM device that need to be correctly set. These parameters are described in the KS8695P Datasheet. Changing all of the above parameters involves changing some of constant definitions in file KS8695.h.

13 Linux Kernel Porting

Linux Kernel 2.4.16 has been ported to the KS8695P demo board as well as the firewall software Iptables.

The Linux Kernel 2.4.16 porting includes Boot Loader, which handles hardware initialization and loads the kernel and ramdisk as given below.

- Copy a compressed ramdisk image into RAM at 0x00800000
- Copy a compressed kernel image into RAM at 0x8000
- Set up parameters for Linux, starting address of parameters at 0x100

The following parameters are needed:

- Ramdisk starting virtual address, which is 0xC0800000
- Compressed ramdisk size
- Page size (RAM)
- Number of pages

Before the loader can pass the control to the Linux Kernel, it has to put 0 to r0, Architecture ID 180, for the KS8695P, to r1. Then control is passed to the Linux Kernel by jumping to address 0x8000. Then the Linux kernel takes over the system.

RamDisk, includes a shell (ASH), libraries, and all the applications such as web server, firewall, and device drivers. It serves as root file system

After control is passed to the Linux Kernel, Linux goes through the following steps to bring up the OS.

Procedure

1. Uncompress the Linux image.
2. Start-up code at linux/arch/arm/head-armv.S, which sets up the processor, initial page tables, clears BSS, and other similar events.

   Then the linux/init/main.c/start_kernel() takes over and initializes and sets up all the needed tools and functions such as memory, page table, stack, devices, and other similar items.
3. Upon completion, create a shell and pass control to root file's init.

13.1 Configure Linux Kernel 2.4.16 for the KS8695P Demo Board

The KS8695P BSP CD includes a complete Linux Kernel source code. Refer to the following source code subdirectories in the source tree of Linux Kernel for the KS8695P Demo Board specific code.

- linux/include/asm-arm/arch-ks8695 for platform specific headers

- linux/arch/arm/mach-ks8695 for platform/processor specific code such as irq

The following directories also have platform specific codes:
- linux/arch/arm/boot/compressed – for kernel decompression
- linux/arch/arm/mm – MMU for specific processor
- linux/drivers/serial – platform specific serial driver
-linux/drivers/net/ks8695p – DMA driver

Use MAPIO(iomap_func) in linux/arch/arm/mach-ks8695/arch.c to specify the IO mapping function.

The iomap_func should be defined in linux/arch/arm/mach-ks8695/mm.c which initializes virtual spaces to IO spaces.

Use the IO_ADDRESS in linux/include/asm-arm/arch-ks8695/hardware.h to convert the IO address into a virtual address.

The BOOT_MEM(_pram,_pio,_vio) where _pram is the physical ram starting address, _pio is the physical IO starting address, and _vio is the virtual IO starting address.

The physical RAM starting address must match with the PHYS_OFFSET definition in linux/include/asm-arm/arch-ks8695/memory.h

The IO space must match the IO_BASE, IO_SIZE, and IO_START definitions in linux/include/asm-arm/arch-ks8695/hardware.h

Use __virt_to_phys, __phys_to_virt, __virt_to_bus and __bus_to_virt definitions in linux/include/asm-arm/arch-ks8695/memory.h to convert between virtual and physical RAM addresses.

13.2 Build Linux Kernel

To configure the Linux Kernel under a Linux PC such as RedHat Linux, copy the source onto a directory, uncompress the source and follow the steps listed below:

1. Run "make menuconfig" (be sure to save even if there is no change), see menuconfig below.
2. Go to the Network device support/Ethernet(10/100), and select KS8695P as modules(m) or buildin(*).
3. Run "make dep zImage modules"

13.3 Enable Cache Memory with Linux

To enable cache with Linux, do "make menuconfig", go to "System Type" and enable/disable the appropriate cache. Save the new change, do "make clean dep zImage" to build a new kernel image.

13.4 Build File System for Linux

A root file system must contain everything needed to support an embedded Linux system. To be able to do this, the disk must include the minimum requirements for a Linux system:

- The basic file system structure,
- Minimum set of directories: */dev*, */proc*, */bin*, */etc*, */lib*, */sbin*, */web*, */tmp*,
- Basic set of utilities*: sh, ls, cp, mv,* etc. from busybox,
- Minimum set of config files: *rc*, *inittab*, *fstab*, etc.,
- Devices: */dev/tty**, */dev/ram0*, etc.,
- Runtime library to provide basic functions used by utilities.

13.5 Creating the File System

The following section describes how to build a *compressed* file system; it is referred to as this because it is compressed on disk, and when booted, is uncompressed onto a ramdisk. With a compressed file system you can accommodate many more files.

In order to build this type of a root file system, you will need a spare device that is large enough to hold all the files before compression. This device needs to hold about four megabytes.

Use a loopback device that allows a disk file to be treated as a device.

You can use a command like,

dd if=/dev/zero of=/tmp/initrd bs=1k count=*nnnn*

to create a *nnnn*-block file.

When you issue a mount command, you must include the option ``-o loop'' to tell mount to use a loopback device. For example,

```
mount -o loop -t ext2 /tmp/fsfile /mnt
```

will mount /tmp/fsfile (via a loopback device) at the mount point /mnt. A df will confirm this.

After you've chosen one of these options, prepare the DEVICE with,

```
dd if=/dev/zero of=DEVICE bs=1k count=4000
```

This command zeroes out the device. This step is important because the file system on the device will be compressed later. All unused portions should be filled with zeroes to achieve maximum compression.

The mke2fs command will automatically detect the space available and configure itself accordingly. The -m 0 parameter prevents it from reserving space for root, and hence provides more usable space on the disk. Next, mount the device:

```
mount -t ext2 DEVICE /mnt
```

(You must create a mount point /mnt if it does not already exist.) In the remaining sections, all destination directory names are assumed to be relative to /mnt.

13.6 Populating the File System

Here is a reasonable minimum set of directories for your root file system:

- /dev – Devices, required to perform I/O
- /proc – Directory stub required by the proc file system
- /etc – System configuration files
- /sbin – Critical system binaries
- /bin – Basic binaries considered part of the system
- /lib – Shared libraries to provide run-time support
- /usr – Additional utilities and applications

Three of these directories will be empty on the root file system, so they only need to be created with *mkdir*. The /proc directory is basically a stub under which the proc file system is placed. The directories /mnt and /usr are only mount points for use after the boot/root system is running. Hence again, these directories only need to be created.

The remaining four directories are described in the following sections.

13.6.1 /dev

A /dev directory containing a special file for all devices to be used by the system is mandatory for any Linux system. The directory itself is a normal directory and can be created with mkdir in

the normal way. The device special files, however, must be created in a special way, using the mknod command.

13.6.2 /etc

This directory must contain a number of configuration files. On most systems, these can be divided into three groups:

1. Required at all times, e.g, rc, fstab, passwd.
2. May be required, but no-one is too sure.
3. Junk that crept in.

The following files are included in this directory

fstab -- list of file systems to be mounted
inittab -- parameters for the init process, the first process started at boot time.
passwd -- list of users, home directories, etc.
group -- user groups.

The *inittab* should be changed so that its sysinit line runs whatever basic boot script will be used.

13.6.3 /bin and /sbin

The /bin directory is a convenient place for extra utilities you need to perform basic operations, utilities such as ls, mv, cat and dd. All of them are included in the busybox.

13.6.4 /lib

In /lib, place the necessary shared libraries and loaders. If the necessary libraries are not found in the /lib directory, the system will be unable to boot. An error message may appear telling you why.

Almost every program requires at least the libc library, libc.so.*N*, where *N* is the current version number. Check your /lib directory. libc.so.N is usually a symlink to a filename with a complete version number:

*% ls -l /lib/libc**

-rwxr-xr-x 1 root root 4016683 Apr 16 18:48 libc-2.1.1.so*
*lrwxrwxrwx 1 root root 13 Apr 10 12:25 libc.so.6 -> libc-2.1.1.so**

In this case, you want libc-2.1.1.so. To find other libraries, go through all the binaries you plan to include and check their dependencies with the ldd command. For example,

% ldd /sbin/mke2fs
libext2fs.so.2 => /lib/libext2fs.so.2 (0x40014000)
libcom_err.so.2 => /lib/libcom_err.so.2 (0x40026000)
libuuid.so.1 => /lib/libuuid.so.1 (0x40028000)
libc.so.6 => /lib/libc.so.6 (0x4002c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)

Each file on the right-hand side is required. The file may be a symbolic link.
Note that some libraries are *quite large* and will not fit easily on your root file system. For example, the *libc.so* listed above is about 4 meg. You will probably need to strip libraries when copying them to your root file system. Refer to the section on reducing root file system size for instructions.

In /lib, you must also include a loader for the libraries. The loader will be either ld.so (for a.out libraries) or ld-linux.so (for ELF libraries). Newer versions of ldd tell you exactly which loader is needed, as in the previous example, but older versions may not.

13.7 Modules

If you have a modular kernel, you must consider which modules you want to load from your boot disk after booting. You should also include insmod, rmmod and lsmod, depending on whether you want to load modules automatically. The main advantage to using modules is that you can move non-critical modules to a utility disk and load them when needed, thus using less space on your root disk. If you may have to deal with many different devices, this approach is preferred over building a huge kernel with many built-in drivers.

13.8 Build Ramdisk

Once you have finished constructing the root file system, unmount it, copy it to a file and compress it:

```
umount /mnt
dd if=DEVICE bs=1k | gzip -v9 > rootfs.gz
```

When this finishes, you will have a file rootfs.gz that is your compressed root file system. Check its size to make sure it will fit on a diskette; if it doesn't fit you will need to go back and remove some files.

Appendix A: KS8695P DMA Driver Programming Tips

1. Race Condition

In some stress test cases, such as using SmartBits to send small packets continuously (e.g., packet size <256), there could be a race condition between the DMA driver's ISR routines and the KS8695P DMA engines. This could happen in the packet-receiving process. It is caused by the DMA engines running faster than what ISR routines can handle, namely, the DMA engines consuming all the receiving DMA descriptors (and buffers) before these descriptors get a chance to be reprogrammed (refilled in the driver level). In other words, the DMA engines already 'round' around the descriptor link list before the ISR routines finish the descriptor refilling process.

Solution:

Make the code segment that reprograms the receiving descriptors as an atomic process. In other words, disable all the interrupt sources before reprogramming the receiving descriptors. Re-enable the interrupt sources afterward. Please refer to the ProcessRxInterrupts() and ReceiveBufferFill() routines in the KS8695P Linux Ethernet driver for the implementation.

2. Under-Size Packet and Over-Size Packet

When a half-duplex collision occurs, the KS8695P WAN DMA engine may segment the frames incorrectly at half-duplex modes (including both 10Mps Half-Duplex and 100 Mbps Half-Duplex modes). As a result, a pair of four-byte under size packets and a 1550-byte oversized packets could be **randomly** generated. The probability of this phenomenon is very low, and the DMA driver can be programmed to easily screen out the erratic packets.

Solution:

The DMA driver needs to allocate the receiving buffer with size greater than 1550 bytes. Discard the erratic packets at driver level.

3. Random Under-Size Packet.

Random-sized undersize packets are detected in the KS8695P Linux DMA driver version 1.0.0.10 at 10 Mbps Half-Duplex and 100 Mbps Half-Duplex modes.

Solution:

Discard the erratic packets at the driver level.