AMD SB800-Series Southbridges BIOS Developer's Guide

Technical Reference Manual
Rev. 3.03

PN: 45483_sb800_bdg_pub_3.03

Table of Contents

1 Introduction

1.1 About this Manual

This manual provides guidelines for BIOS developers working with the AMD SB800. It describes the BIOS and software modifications required to fully support the device.

Note that while the term ”SB800” is used in this document to refer to all the SB800-series southbridges, not all information in this book is applicable to all products in the series. Please refer to the databook of any particular southbridge for its features and functionality.

Other documents on the SB800 are available at AMD’s OEM Resource Center or from your AMD FAE representative.

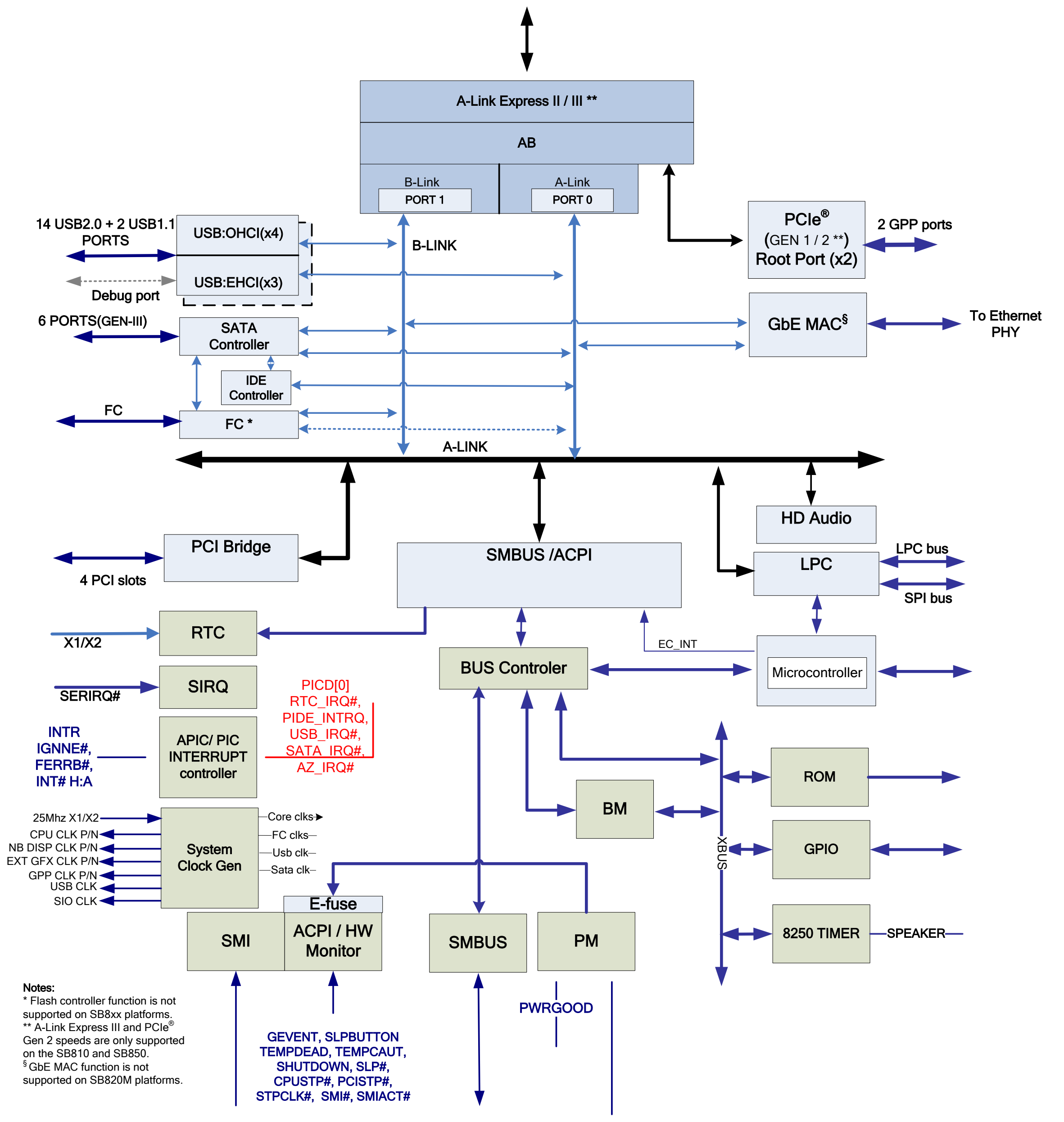
To help the reader to readily identify changes/updates in this document, changes/updates over the previous revision are highlighted in red. Refer to *Appendix: Revision History* at the end of this document for a detailed revision history.

1.2 Overview

AMD’s SB800 southbridges integrate the key I/O, communications, and audio features required in a state-of-the-art PC into a single device. These products are specifically designed to operate with AMD’s integrated graphics northbridge products in both desktop and mobile PCs.

1.3 Block Diagram

This section contains a diagram for the SB800. *Figure 1* below shows the SB800 internal PCI devices and the major function blocks.



A-Link Express II / III **
AB
B-Link
PORT 1
A-Link
PORT 0
14 USB2.0 + 2 USB1.1 PORTS
USB:OHCI(x4)
USB:EHCI(x3)
Debug port
B-LINK
PCIe® (GEN 1 / 2 **) Root Port (x2)
2 GPP ports
6 PORTS(GEN-III)
SATA Controller
IDE Controller
GbE MAC§
To Ethernet PHY
FC
FC *
A-LINK
PCI Bridge
4 PCI slots
SMBUS /ACPI
HD Audio
LPC
LPC bus
SPI bus
X1/X2
RTC
EC_INT
BUS Controler
Microcontroller
SERIRQ#
SIRQ
PICD[0] RTC_IRQ#, PIDE_INTRQ, USB_IRQ#, SATA_IRQ#, AZ_IRQ#
INTR IGNNE#, FERRB#, INT# H:A
APIC/ PIC INTERRUPT controller
ROM
BM
XBUS
GPIO
25Mhz X1/X2
CPU CLK P/N
NB DISP CLK P/N
EXT GFX CLK P/N
GPP CLK P/N
USB CLK
SIO CLK
System Clock Gen
Core clks
FC clks
Usb clk
Sata clk
E-fuse
SMI
ACPI / HW Monitor
SMBUS
PM
8250 TIMER
SPEAKER
PWRGOOD
GEVENT, SLPBUTTON TEMPDEAD, TEMPCAUT, SHUTDOWN, SLP#, CPUSTP#, PCISTP#, STPCLK#, SMI#, SMIACT#
Notes:
* Flash controller function is not supported on SB8xx platforms.
** A-Link Express III and PCIe® Gen 2 speeds are only supported on the SB810 and SB850.
§ GbE MAC function is not supported on SB820M platforms.

Figure 1 SB800 PCI Internal Devices and Major Function Blocks

1.4 Internal PCI Devices

Note: The SB800 internal PCI devices are listed in *Figure 2* below. The sub-sections that follow provide descriptions of the PCI configuration space, the I/O space, and the memory space registers for each device. PCI configuration space registers are only accessible with configuration Read or configuration Write cycles and with the target device selected by setting its corresponding IDSEL bit in the configuration cycle address field.

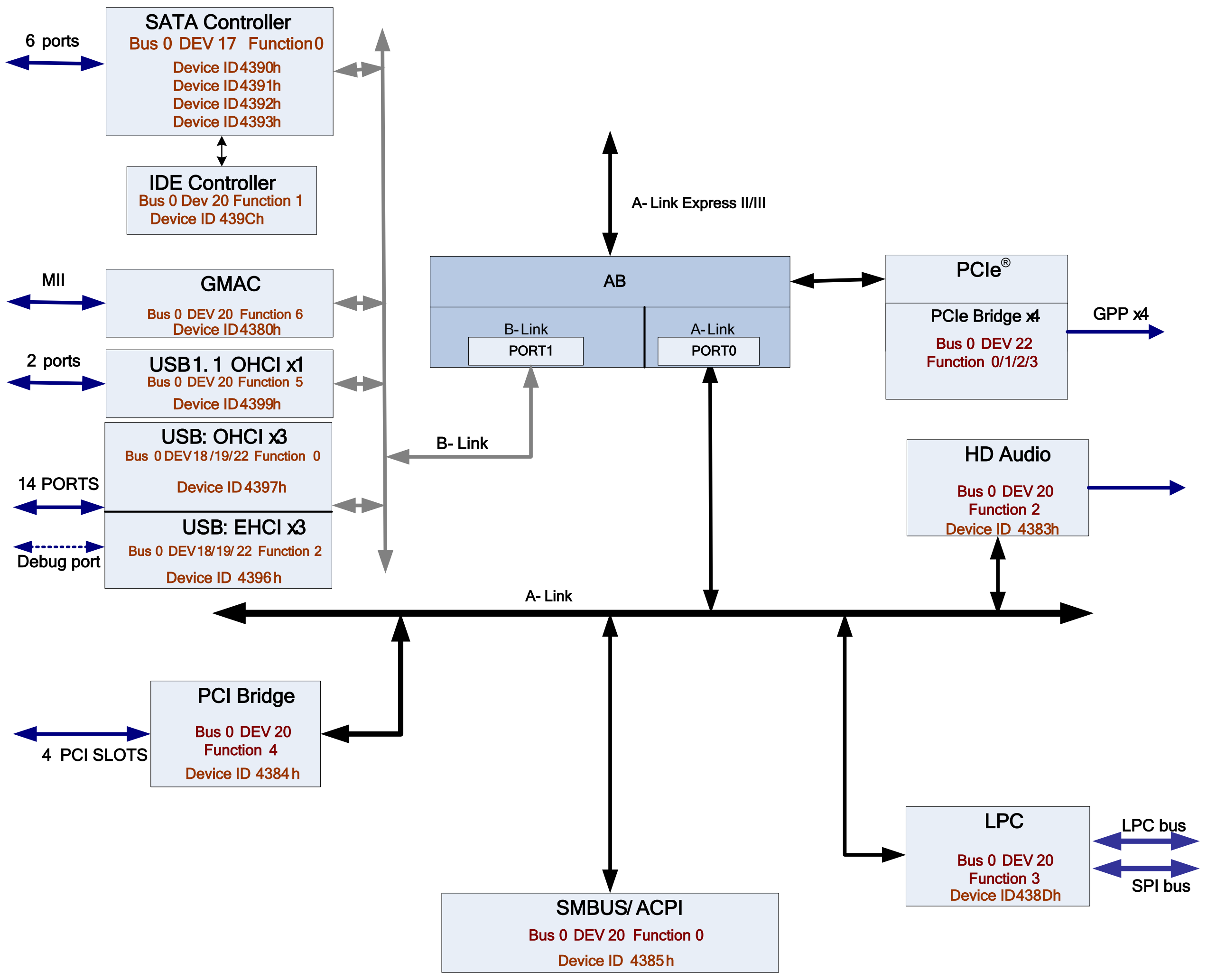


Figure 2: SB800 PCI Internal Devices

1.5 Major Differences between SB800 and SB700

The following table lists the major differences between SB800 and SB700 (previous generation SB). There may be other differences not mentioned here since they may not be important to a BIOS developer.

SB700	SB800
Supports five OHCI controllers (Bus 0 Dev 12h Fn 0, 1, Bus 0 Dev 13h Fn 0, 1, Bus 0 Dev 14h Fun 5) and two EHCI controllers (Bus 0 Dev 12h Fn 2, Bus 0 Dev 13h Fn 2).	Supports four OHCI controllers (Bus 0 Dev 12h Fn 0, Bus 0 Dev 13h Fn 0, Bus 0 Dev 14h Fun 5, Bus 0 Dev 16h Fn 0) and three EHCI controllers (Bus 0 Dev 12h Fn 2, Bus 0 Dev 13h Fn 2, Bus 0 Dev 16h Fn 2)
Does not support Combined Mode.	Supports GEN3, and SATA (IDE) controller can be hidden once Combined Mode is disabled.
Alternate software interface to PMIO and legacy registers (per customer's request). Use MMIO instead of IO CD6/CD7 and other legacy IO addresses.	Forces software interface to use MMIO instead of IO CD6/CD7 and other legacy IO addresses. Also regroups all the SMI events, GPIO/IOMux, etc, into this MMIO interface.
No GPP support.	GPP port support.

2 SB800 Programming Architecture

2.1 PCI Devices and Functions

Bus:Device:Function	Function Description	Dev ID	Enable/Disable
Bus 0:Device 14h:Function 0	SMBus Controller	4385h	Always enabled
Bus 0:Device 14h:Function 1	SATA (IDE) Controller	438Ch	PM_Reg: 0xDA bit3
Bus 0:Device 14h:Function 2	Azalia Controller	4383h	PM_Reg: 0xEB bit0
Bus 0:Device 14h:Function 3	LPC Controller	438Dh	PM_Reg: 0xEC bit0
Bus 0:Device 14h:Function 4	PCIB Bridge	4384h	Always enabled
Bus 0:Device 14h:Function 6	GEC Controller	1699h	PM_Reg: 0xF6 bit0 cleared
Bus 0:Device 12h:Function 2 Bus 0:Device 13h:Function 2 Bus 0:Device 16h:Function 2	USB #1EHCI Controller USB #2EHCI Controller USB #3EHCI Controller	4396h 4396h 4396h	PM_Reg: 0xEF bit1, bit3, bit5
Bus 0:Device 12h:Function 0 Bus 0:Device 13h:Function 0 Bus 0:Device 14h:Function 5 Bus 0:Device 16h:Function 0	USB #1 OHCI Controller USB #2 OHCI Controller USB #4 OHCI Controller USB #3 OHCI Controller	4397h 4397h 438Bh 4397h	PM_Reg: 0xEF bit 0, bit 2, bit 4, bit 6
Bus 0:Device 11h:Function 0	SATA Controller: Native/Legacy IDE Mode AHCI Mode Non-Raid-5 Mode Raid5 Mode	 4390h 4391h 4392h 4393h	PM_Reg: 0xDA bit0

2.2 I/O Map

The I/O map is divided into Fixed and Variable address ranges. Fixed ranges cannot be moved, but can be disabled in some cases. Variable ranges are configurable.

2.2.1 Fixed I/O Address Ranges

2.2.1.1 Fixed I/O Address Ranges – SB800 Proprietary Ports

I/O Address	Description	Enable Bit
C00h-C01h	IRQ Routing Index/Data register	PM_Reg: 0x00 [1]
C14h	PCI Error Control register	PM_Reg: 0x00 [20]
C50h-C51h	Client Management Index /Data registers	PM_Reg: 0x00 [27]
C52h	Gpm Port	PM_Reg: 0x00 [22]
C6Fh	Flash Rom Program Enable	PM_Reg: 0x00 [24]
CD0h-CD1h	PM2 Index/Data	Always enable
CD4h-CD5h	BIOS RAM Index/Data	PM_Reg: 0x20 [0] (Default enabled)
CD6h-CD7h	Power Management I/O register	PM_Reg: 0x00 [25]

2.2.2 Variable I/O Decode Ranges

I/O Name	Description	Configure Register	Range Size (Bytes)
PM1_EVT	ACPI PM1a_EVT_BLK	PM_Reg: 0x60 & 0x61	4
PM1_CNT	ACPI PM1a_CNT_BLK	PM_Reg: 0x62 & 0x63	2
PM_TMR	ACPI PM_TMR_BLK	PM_Reg: 0x64 & 0x65	4
P_BLK	ACPI P_BLK	PM_Reg: 0x66 & 0x66	6
GPE0_EVT	ACPI GPE0_EVT_BLK	PM_Reg: 0x68 & 0x67	8
SMI CMD Block *	SMI Command Block	PM_Reg: 0x6A & 0x6B	2
Pma Cnt Block	PMa Control Block	PM_Reg: 0x6E & 0x6F	1
SMBus	SMBus IO Space	PM_Reg: 0x2C & 0x2D	16

Note:

- The SMI CMD block must be defined on the 16-bit boundary, i.e., the least significant nibble of the address must be zero (for example, B0h, C0h, etc.)
- The SMI CMD block consists of two ports – the SMI Command port at base address, and the SMI Status port at base address+1.
- The writes to SMI Status port will not generate an SMI. The writes to the SMI Command port will generate an SMI.

- The SMI Command and SMI Status ports may be written individually as 8-bit ports, or together as a 16-bit port.

2.3 Memory Map

Memory Range	Description	Enable Bit
0000 0000h-000D FFFFh 0010 0000h-TOM	Main System Memory	
000E 0000h-000F FFFFh	Either PCI ROM or LPC ROM	PCI ROM: Enabled by ROM strap only, no register to program. LPC ROM: LPC_Reg: 0x68 & LPC_Rom strap
FEC0 0000h-FEC0 00EFh	IOAPIC	
FEC0 00F0h-FEC0 00F4h	Watch Dog Timer Base Address * Recommended	PM_Reg: 0x48[0]
FED1 0000h-FED1 0100h	BIOS RAM base address * Recommended	PM_Reg: 0x20[0]
FED4 0000h-FED4 3FFFh	TPM	Depends on configuration
FED6 1000h-FED6 1100h	GEC SHADOW ROM	LPC Reg9Ch [0]
FED8 0000h-FED8 0EFF	Acpi MMIO address * Recommend	PM_Reg: 0x24[0]
FFC0 0000h-FFC7 FFFFh FF80 0000h-FF87 FFFFh	FWH	LPC Reg: 0x70[3:0]
FFC8 0000h-FFCF FFFFh FF88 0000h-FF8F FFFFh	FWH	LPC Reg: 0x70[7:4]
FFD0 0000h-FFD7 FFFFh FF90 0000h-FF97 FFFFh	FWH	LPC Reg: 0x70[11:8]
FFD8 0000h-FFDF FFFFh FF98 0000h-FF9F FFFFh	FWH	LPC Reg: 0x70[15:12]
FFE0 0000h-FFE7 FFFFh FFA0 0000h-FFA7 FFFFh	FWH	LPC Reg: 0x70 [19:16]
FFE8 0000h-FFEF FFFFh FFA8 0000h-FFAF FFFFh	FWH	LPC Reg: 0x70[23:20]
FFF0 0000h-FFF7 FFFFh FFB0 0000h-FFB7 FFFFh	FWH	LPC Reg: 0x70[27:24]
FFF8 0000h-FFFF FFFFh FFB8 0000h-FFBF FFFFh	FWH	LPC Reg: 0x70[31:28]

2.3.1 MMIO Programming for Legacy Devices

The SB legacy devices LPC, IOAPIC, ACPI, TPM and Watchdog Timer require the base address of the Memory Mapped I/O registers to be assigned before these logic blocks are accessed. The Memory Mapped I/O register base address and its entire range should be mapped to non-posted memory region by programming the CPU register.

Below is a sample code for FCH MMIO Range calculation in System BIOS.

```
#include "SbPlatform.h"

//
// Declaration of local functions
//

typedef struct _OPTIMUM_FCH_MMIO_STRUCT {
  UINT16 Range0Base;
  UINT16 Range0Limit;
  UINT16 Range1Base;
  UINT16 Range1Limit;
  UINT16 Range2Base;
  UINT16 Range2Limit;
} OPTIMUM_FCH_MMIO_STRUCT;

/**
 * The FCH MMIO non-POST Range
 */
typedef struct _MMIO_RANGE_STRUCT {
  UINT16 Lpc0Base;
  UINT16 Lpc0Limit;
  UINT16 Lpc1Base;
  UINT16 Lpc1Limit;
  UINT16 SpiBase;
  UINT16 SpiLimit;
  UINT16 TmpBase;
  UINT16 TmpLimit;
  UINT16 HpetBase;
  UINT16 HpetLimit;
  UINT16 BiosRamBase;
  UINT16 BiosRamLimit;
  UINT16 WatchDogBase;
  UINT16 WatchDogLimit;
  UINT16 IoapicBase;
  UINT16 IoapicLimit;
  UINT16 AcpiMmioBase;
  UINT16 AcpiMmioLimit;
} MMIO_RANGE_STRUCT;

//
// Declaration of local functions
//

VOID  fchMmioRangeCalculation (IN AMDSBCFG* pConfig, OUT OPTIMUM_FCH_MMIO_STRUCT* TempRange);

/**
```

```
 * fchMmioRangeCalculation - Calculatw FCH none-POST Mmio resource
 *
 *
 *   - Private function
 *
 * @param[in] pConfig - FCH configuration structure pointer.
 * @param[out] CFGMmioTableDescription - Optimum range for non-POST FCH MMIO range for IBV
 *
 */
VOID
fchMmioRangeCalculation (
  IN       AMDSBCFG*   pConfig,
  OUT      OPTIMUM_FCH_MMIO_STRUCT*  TempRange
  )
{
  MMIO_RANGE_STRUCT fchTemp;
  UINT16 TempRange1BaseH;
  UINT16 TempRange1BaseL;
  UINT8 Rang2Flag;

  // Fill all FCH Mmio range
  // Lpc ROM 1 Base read from FCH
  ReadPCI ((LPC_BUS_DEV_FUN << 16) + SB_LPC_REG68, AccWidthUint16, &fchTemp.Lpc0Base);
  // Lpc ROM 1 Limit read from FCH
  ReadPCI ((LPC_BUS_DEV_FUN << 16) + SB_LPC_REG6A, AccWidthUint16, &fchTemp.Lpc0Limit);
  // Lpc ROM 2 Base read from FCH
  ReadPCI ((LPC_BUS_DEV_FUN << 16) + SB_LPC_REG6C, AccWidthUint16, &fchTemp.Lpc1Base);
  // Lpc ROM 2 Limit read from FCH
  ReadPCI ((LPC_BUS_DEV_FUN << 16) + SB_LPC_REG6E, AccWidthUint16, &fchTemp.Lpc1Limit);
  // Spi Base Address read from FCH
  ReadPCI ((LPC_BUS_DEV_FUN << 16) + SB_LPC_REGA0 + 2, AccWidthUint16, &fchTemp.SpiBase);
  // Spi base address limit is less then 64K

  // Tpm Base Address read from FCH
  ReadPCI ((LPC_BUS_DEV_FUN << 16) + 0x86, AccWidthUint16, &fchTemp.TmpBase);
  // Tpm Limit Address read from FCH
  ReadPCI ((LPC_BUS_DEV_FUN << 16) + 0x8A, AccWidthUint16, &fchTemp.TmpLimit);
  // HPET Base Address read from FCH
  ReadMEM (ACPI_MMIO_BASE + PMIO_BASE + SB_PMIOA_REG50 + 2, AccWidthUint16, &fchTemp.HpetBase);
  // HPET base address limit is less then 64K

  // BIOS RAM base Address read from FCH
  ReadMEM (ACPI_MMIO_BASE + PMIO_BASE + SB_PMIOA_REG20 + 2, AccWidthUint16,
&fchTemp.BiosRamBase);
  // BIOS RAM address limit is less then 64K

  // WatchDog base address read from FCH
  ReadMEM (ACPI_MMIO_BASE + PMIO_BASE + SB_PMIOA_REG48 + 2, AccWidthUint16,
&fchTemp.WatchDogBase);
  // WatchDog address limit is less then 64K

  // IoApic base address read from FCH
  ReadMEM (ACPI_MMIO_BASE + PMIO_BASE + SB_PMIOA_REG34 + 2, AccWidthUint16, &fchTemp.IoapicBase);
  // IoApic address limit is less then 64K
  // ACPI Mmio base address read from FCH
  ReadMEM (ACPI_MMIO_BASE + PMIO_BASE + SB_PMIOA_REG24 + 2, AccWidthUint16,
&fchTemp.AcpiMmioBase);
  // ACPI Mmio address limit is less then 64K

  // Reserved Range0Base for LPC ROM location for CPU specific ROM cycle.
  // In CIMx usually set LPC ROM2 for LPC ROM base address
  TempRange->Range0Base = fchTemp.Lpc1Base;
  TempRange->Range0Limit = fchTemp.Lpc1Limit;

  // Intent all other filed (except LPC) combine to one big MMIO range.
  TempRange1BaseL = 0xFEC0;        // FCH default value for Watchdoag base address (lowerest)
  TempRange1BaseH = 0xFED8;        // FCH default value for ACPI MMIO base address (highest)
  TempRange->Range1Base = 0xFEC0;
  TempRange->Range1Limit = 0xFED8;
  TempRange->Range2Base = 0;
  TempRange->Range2Limit = 0;
  Rang2Flag = 0x00;

  if  (( fchTemp.SpiBase < TempRange1BaseL ) || ( TempRange1BaseH < fchTemp.SpiBase)) {
    Rang2Flag = 1;
    TempRange->Range2Base = fchTemp.SpiBase;
    TempRange->Range2Limit = fchTemp.SpiBase;
  }

  if (( fchTemp.TmpBase != 0) && ( Rang2Flag != 1)) {
    if  (( fchTemp.TmpBase < TempRange1BaseL ) || ( TempRange1BaseH < fchTemp.TmpBase)) {
      Rang2Flag = 1;
      TempRange->Range2Base = fchTemp.TmpBase;
      TempRange->Range2Limit = fchTemp.TmpLimit;
    }
  }
  if (( fchTemp.HpetBase != 0) && ( Rang2Flag != 1)) {
    if  (( fchTemp.HpetBase < TempRange1BaseL ) || ( TempRange1BaseH < fchTemp.HpetBase)) {
      Rang2Flag = 1;
      TempRange->Range2Base = fchTemp.HpetBase;
      TempRange->Range2Limit = fchTemp.HpetBase;
    }
  }
  if (( fchTemp.BiosRamBase != 0) && ( Rang2Flag != 1)) {
    if  (( fchTemp.BiosRamBase < TempRange1BaseL ) || ( TempRange1BaseH < fchTemp.BiosRamBase))
{
      Rang2Flag = 1;
      TempRange->Range2Base = fchTemp.BiosRamBase;
      TempRange->Range2Limit = fchTemp.BiosRamBase;
    }
  }
  if (( fchTemp.WatchDogBase != 0) && ( Rang2Flag != 1)) {
    if  (( fchTemp.WatchDogBase < TempRange1BaseL ) || ( TempRange1BaseH < fchTemp.WatchDogBase))
{
      Rang2Flag = 1;
      TempRange->Range2Base = fchTemp.WatchDogBase;
      TempRange->Range2Limit = fchTemp.WatchDogBase;
    }
  }
  if (( fchTemp.IoapicBase != 0) && ( Rang2Flag != 1)) {
    if  (( fchTemp.IoapicBase < TempRange1BaseL ) || ( TempRange1BaseH < fchTemp.IoapicBase)) {
      Rang2Flag = 1;
      TempRange->Range2Base = fchTemp.IoapicBase;
      TempRange->Range2Limit = fchTemp.IoapicBase;
    }
  }
  if (( fchTemp.AcpiMmioBase != 0) && ( Rang2Flag != 1)) {
    if  (( fchTemp.AcpiMmioBase < TempRange1BaseL ) || ( TempRange1BaseH < fchTemp.AcpiMmioBase))
{
      Rang2Flag = 1;
      TempRange->Range2Base = fchTemp.AcpiMmioBase;
      TempRange->Range2Limit = fchTemp.AcpiMmioBase;
    }
  }
  if (( Rang2Flag != 1) && ( fchTemp.Lpc0Base != 0 )) {
    TempRange->Range2Base = fchTemp.Lpc0Base;
    TempRange->Range2Limit = fchTemp.Lpc0Limit;
  }
}
```

3 SB800 Early-POST Initialization

The system BIOS needs to configure the SB800 at the very beginning of POST. Some of the settings will change depending on the OEM design, or on the newer revision chipset.

3.1 512K/1M ROM Enable

With the SB800 design, there can be two possible ROM sources: PCI ROM and LPC ROM. Two pin straps (UseLpcRom, FWHDisable) decide where the ROM is (see the SB800 databook). Upon system power on, the SB800 enables 256K ROM by default. The BIOS needs to enable 512K ROM or up to 1M for LPC ROM, if required.

3.1.1 *PCI ROM*

Control Bit	Description	256K ROM Setting (Default)	512K ROM Setting
PM_Reg: 0x04 [12]	When set to 1, the address between FFF80000h to FFFDFFFFh will be directed to the PCI ROM interface.	0	1
PM_Reg: 0x04 [13]	When set to 1, the address between 0E0000h to 0EFFFFh will be directed to the PCI ROM interface.	0	1

3.1.2 *LPC ROM*

To use the LPC ROM, the pin straps UseLpcRom, FWHDisable must be set accordingly.

Control Bit(s)	Description	Default	512K ROM Setting	1M ROM Setting
LPC PCI Reg: 0x68	16-bit starting & end address of the LPC ROM memory address range 1.	000E0000h	000E0000h	000E0000h
LPC PCI Reg: 0x6C	16-bit starting & end address of the LPC ROM memory address range 2.	FFFE0000h	FFF80000h	FFF00000h
LPC PCI Reg: 0x48 [4:3]	Enable bits for LPC ROM memory address range 1 & 2. Note: with pins straps set to LPC ROM, these two bits have no effect on Reg68 & Reg6C.	00b	11b	11b

3.1.3 *LPC ROM Read/Write Protect*

The SB800 allows all or a portion of the LPC ROM addressed by the firmware hub to be read protected, write protected, or both read and write protected. Four dword registers are provided to select up to 4 LPC ROM ranges for read or write protection. The ROM protection range is defined by the base address and the length. The base address is aligned at a 2K boundary. The address length can be from 1K to 512K in increments of 1K.

Register 50h, 54h, 58h, 5ch of Device 14h, Function 3

Field Name	Bits	Description
Base Address	31:11	ROM Base address. The most significant 21 bits of the base address are defined in this field. Bits[10:0] of the base address are assumed to be zero. Base address, therefore, is aligned at a 2K boundary.
Length	10:2	These 9 bits (0-511) define the length from 1K to 512K in increments of 1K.
Read Protect	1	When set, the memory range defined by this register is read-protected. Reading any location in the range returns FFh.
Write Protect	0	When set, the memory range defined by this register is write-protected. Writing to the range has no effect.

Example:

Protect 32K LPC ROM starting with base address FFF80000.

Base address bits 31:11   1111 1111 1111 1000 0000 0 b

Length 32K   bit 10:2 = 31h = 000 0111 11 b

Read protect bit 1 = 1

Write protect bit 0 = 1

Register 50h = 1111 1111 1111 1000 0000   0000 0111 1111 b = FFF8007F h

Note**:**

1. Registers 50h ~ 5Fh can be written once after the hardware reset. Subsequent writes to them have no effect.
2. Setting sections of the LPC ROM to either read or write protect will not allow the ROM to be updated by a flash programming utility. Most flash utilities write and verify ROM sectors, and will terminate programming if verification fails due to read protect.

3.1.4 *SPI ROM Controller*

The SPI ROM interface is a new feature added to the SB800. Refer to the *AMD SB800 Register Reference Manual* for more information on this feature. AMD will provide reference code for this feature.

Note: The LPC ROM Read/Write Protect mentioned in the previous paragraph also applies to SPI. Two strap pins, PCICLK0 and PCICLK1, determine the SB800 boot up from LPC ROM or SPI ROM. There is no register status to reflect whether the current ROM interface is LPC or SPI.

3.2 Real Time Clock (RTC)

3.2.1 RTC Access

The internal RTC is divided into two sections: the clock and alarm function (registers 0h to 0Dh), and CMOS memory (registers 0Eh to FFh). The clock and alarm functions must be accessed through I/O ports 70h/71h. The CMOS memory (registers 0Eh to FFh) should be accessed through I/O ports 72h/73h.

3.2.1.1 Special Locked Area in CMOS

Some CMOS memory locations may be disabled for read/write. Register 6Ah of SMBus (Bus 0, Device 14h, Function 0) has bits to disable these CMOS memory locations. Once set, the area is protected. It can only be disabled by cycling the system from S0 to G3 to S0 (RSM_RST# toggled) or by doing a system cold reset (SYS_Reset# toggled).

RTCProtect- RW - 8 bits - [PCI_Reg: 6Ah]			
Field Name	**Bits**	**Default**	**Description**
RTCProtect	0	0h	When set, RTC RAM index 38h:3Fh will be locked from read/write.
RTCProtect	1	0h	When set, RTC RAM index F0h:FFh will be locked from read/write.
RTCProtect	2	0h	When set, RTC RAM index E0h:EFh will be locked from read/write.
RTCProtect	3	0h	When set, RTC RAM index D0h:DFh will be locked from read/write.
RTCProtect	4	0h	When set, RTC RAM index C0h:CFh will be locked from read/write.
Reserved	7:5	0h	

3.2.1.2 Century Byte

The RTC has a century byte at CMOS location 32h. Century is stored in a single byte and the BCD format is used for the century (for example, 20h for the year 20xx). This byte is accessed using I/O ports 70h and 71h. (The BIOS must set PMIO register 7Ch bit 4 to 1 to use this century byte at CMOS location 32h

3.2.1.3 Date Alarm

The RTC has a date alarm byte. This byte is accessed as follows:

1. Set to 1 the RTC register 0Ah, bit 4, using I/O ports 70h and 71h.
2. Write Date Alarm in BCD to register 0Dh using I/O ports 70h and 71h.
3. Clear to 0 the RTC register 0Ah bit 4 using I/O ports 70h and 71h.

Note: It is important to clear RTC register 0Ah bit 4 to zero; otherwise, the CMOS memory may not be accessed correctly from this point onward.

3.3 BIOS RAM

The SB800 has 256 bytes of BIOS RAM. Data in this RAM is preserved until RSMRST# or S5 is asserted, or until power is lost.

This RAM is accessed using index and data registers at CD4h/CD5h.

3.4 Serial IRQ

The SB800 supports serial IRQ, which allows one single signal to report multiple interrupt requests. The SB800 supports a message for 21 serial interrupts, which include 15 IRQs, SMI#, IOCHK#, and 4 PCI interrupts.

SMBus PCI Reg69h is used for setting serial IRQ.

Bits in SMBus PCI Reg: 69h	Description	Power-on Default	Recommended Value
7	1 – Enables the serial IRQ function 0 – Disables the serial IRQ function	0	1
6	1 – Active (quiet) mode 0 – Continuous mode	0	0
5:2	Total number of serial IRQs = 17 + NumSerIrqBits 0 – 17 serial IRQs (15 IRQs, SMI#, IOCHK#) 1 – 18 serial IRQs (15 IRQs, SMI#, IOCHK#, INTA#) ... 15 - 32 serial IRQ's The SB800 serial IRQ can support 15 IRQs, SMI#, IOCHK#, INTA#, INTB#, INTC#, and INTD#.	0	0100b
1:0	Number of clocks in the start frame	0	00b

Note**:** BIOS should enter the continuous mode first when enabling the serial IRQ protocol, so that the SB800 can generate the start frame.

3.5 SubSystemID and SubSystem Vendor ID

SubSytem ID and SubSystem Vendor ID can be programmed in various functions of SB800 register 2Ch. These registers are write-once registers. For example, to program a SubSytem vendor ID of 1002h and SubSystem ID of 4341h in AC97 device 14h, function 5, use the following assembly language sample code:

```
mov     eax,8000A52Ch
mov     dx,0CF8h
out  dx,eax
mov     dx,0CFCh
mov     eax,43411002h
out  dx,eax
```

3.6 System Restart after Power Fail

The way the system restarts following a power-fail/power-restore cycle depends on the setting of PMIO register 5Bh [bits 1:0].

PMIO Register 5Bh bits[1:0]	Description
00b or 10b	The system will remain off until the power button is pressed.
01b	The system will always restart after the power is restored.
11b	At power-up the system will either restart or remain off depending on the state of the system at power failure. If the system was on when the power failed, the system will restart at power-up. If the system was off when the power failed, the system will remain off after the power is restored. Pressing the power button is required to restart the system.

Notes on programming the PMIO register 74h:

1. Bits[3:0] should be used for programming. Bits[7:4] are read-only bits and reflect the same values as bits[3:0].
2. Bit 2 is used by the hardware to save the power on/off status. This bit should not be modified during software/BIOS programming, however, its value should be restored (effectively setting a value of '1' as will be the case after every power up) upon every access into this register. The BIOS programmer should always read PMIO register 5Bh, modify bit3 and bits[1:0] as required, and write back the PMIO register 5Bh.
3. PM_Reg:5Bh is initialized on every cold boot (G3->S5->S0 transition); however, it is also required to restore the settings in bits[7, 5:4] to bits[3, 1:0] following any SYS_RST# or RSMRST# assertion.

3.6.1 *Power Fail and Alarm Setup*

The state of the machine after the power-fail/power-restore cycle is controlled by PMIO register 5Bh bits[1:0] as described above. This programming can be over-ridden for the special case when the alarm is set. When both the alarm and the PMIO register 5Bh bit3 are set, the system will restart after the power is restored, regardless of how register 5Bh bits[1:0] are defined.

4 PCI IRQ Routing

4.1 PCI IRQ Routing Registers

The SB800 uses one pair of I/O ports to do the PCI IRQ routing. The ports are at C00h/C01h.

Address	Register Name	Description
C00h	PCI_Intr_Index	PCI interrupt index. Selects which PCI interrupt to map 0h: INTA# 1h: INTB# 2h: INTC# 3h: INTD# 4h: INTE# 5h: INTF# 6h: INTG# 7h: INTH# 8h: Misc 9h: Misc0 Ah: Misc1 Bh: Misc2 Ch: INTA from serial irq Dh: INTB from serial irq Eh: INTC from serial irq Fh: INTD from serial irq 10h: SCI 11h: SMBUS0 12h: ASF 13h: HD audio 14h: FC 15h: GEC 16h: PerMon 20h: IMC INT0 21h: IMC INT1 22h: IMC INT2 23h: IMC INT3 24h: IMC INT4 25h: IMC INT5 30h: Dev18 (USB) IntA# 31h: Dev18 (USB) IntB# 32h: Dev19 (USB) IntA# 33h: Dev19 (USB) IntB# 34h: Dev22 (USB) IntA# 35h: Dev22 (USB) IntB# 36h: Dev20 (USB) IntC# 40h: IDE pci interrupt 50h: GPPInt0 51h: GPPInt1 52h: GPPInt2 53h: GPPInt3
C01h	PCI_Intr_Data	0 ~ 15 : IRQ0 to IRQ15 IRQ0, 2, 8, 13 are reserved

4.2 PCI IRQ BIOS Programming

PCI IRQs are assigned to interrupt lines using I/O ports at C00h and C01h in index/data format. The register C00h is used for index as written with index number 0 through 0Ch as described in section *4.1* above. Register C01h is written with the interrupt number as data.

The following assembly language example assigns INTB# line to interrupt 10 (0Ah).

```
mov     dx,0C00h        ; To write to IO port C00h
mov     al,02h          ; Index for PCI IRQ INTB# as defined in section 4.1
out     dx,al           ; Index is now set for INTB#
mov     dx,0C01h        ; To write interrupt number 10 (0Ah)
mov     al,0Ah          ; Data is interrupt number  10 (0Ah )
out     dx,al           ; Assign IRQB# to interrupt 10
```

4.3 Integrated PCI Devices IRQ Routing

In the SB800, the AC'97 and USB need PCI IRQ. Internally, they are routed to different PCI INT#s.

Device	Reg3Dh of PCI Device	PCI INT#	Description
Bus 0:Device 14h:Function 1	02	INTB#	IDE Controller*
Bus 0:Device 14h: Function 2	01	INTA#	High Definition Audio
Bus 0:Device 14h: Function 5	03	INTC#	USB #4 OHCI Controller
Bus 0:Device 12h:Function 0	01	INTC#	USB #1 OHCI Controller #0
Bus 0:Device 12h:Function 2	02	INTB#	USB #1 EHCI Controller
Bus 0:Device 13h: Function 0	01	INTC#	USB #2 OHCI Controller #0
Bus 0:Device 13h: Function 2	02	INTB#	USB #2 EHCI Controller
Bus 0:Device 16h: Function 0	01	INTC#	USB #3 OHCI Controller #0
Bus 0:Device 16h: Function 2	02	INTB#	USB #3 EHCI Controller
Bus 0:Device 11h:Function 0	01	INTD#	SATA Controller #2
* This is implement in current CIMx module reference code			

4.4 PCI IRQ Routing for APIC Mode

PCI IRQ	APIC Assignment
INTA#	16
INTB#	17
INTC#	18
INTD#	19
INTE#	20
INTF#	21
INTG#	22
INTH#	23

5 SMBus Programming

The SB800 SMBus (System Management Bus) complies with SMBus Specification Version 2.0.

5.1 SMBus Timing

The SMBus frequency can be adjusted using different values in an 8-bit I/O register at the SMBus base + 0Eh location.

The SMBus frequency is set as follows:

SMBus Frequency = (Primary Alink Clock)/(Count in index 0Eh * 4)

The power-up default value in register 0Eh is A0h, therefore the default frequency is (66MHz)/(160 * 4), or approximately 103 KHz.

The minimum SMBus frequency can be set with the value FFh in the register at index 0Eh, which yields the following: (66MHz)/(255*4) = 64.7 KHz.

5.2 SMBus Host Controller Programming

Step	Descriptions	Register in SMBus I/O Space	Comments
1	Wait until SMBus is idle.	Reg00h[0]	0 – Idle 1 – Busy
2	Clear SMBus status.	Reg00h[4:1]	Write all 1's to clear
3	Set SMBus command.	Reg03h	The command will go to SMBus device.
4	Set SMBus device address with read/write protocol	Reg04h	Bit7:1 – address Bit0 – 1 for read, 0 for write
5	Select SMBus protocol	Reg02h[4:2]	
6	Do a read from Reg02 to reset the counter if it's going to be a block read/write operation	Reg02h	
7	Set low byte when write command	Reg05h	Byte command – It is the written data Word command – It is the low byte data Block command – It is block count Others – Don't care
8	Set high byte when write command	Reg06h	Word command – It is the high byte data Others – Don't care
9	Write the data when block write	Reg07h	Block write – write data one by one to it Others – Don't care
10	Start SMBus command execution	Reg02h[6]	Write 1 to start the command
11	Wait for host not busy	Reg00h[0]	
12	Check status to see if there is any error	Reg00h[4:2]	With 1 in the bit, there is error

13	Read data	Reg05h	Byte command – It is the read data Word command – It is the low byte data Block command – It is block count Others – Don't care
14	Read data	Reg06h	Word command – It is the high byte data Others – Don't care
15	Read the data when block write	Reg07h	Block read – read data one by one. Others – Don't care

The following flow chart illustrates the steps in programming the SMBus host controller.

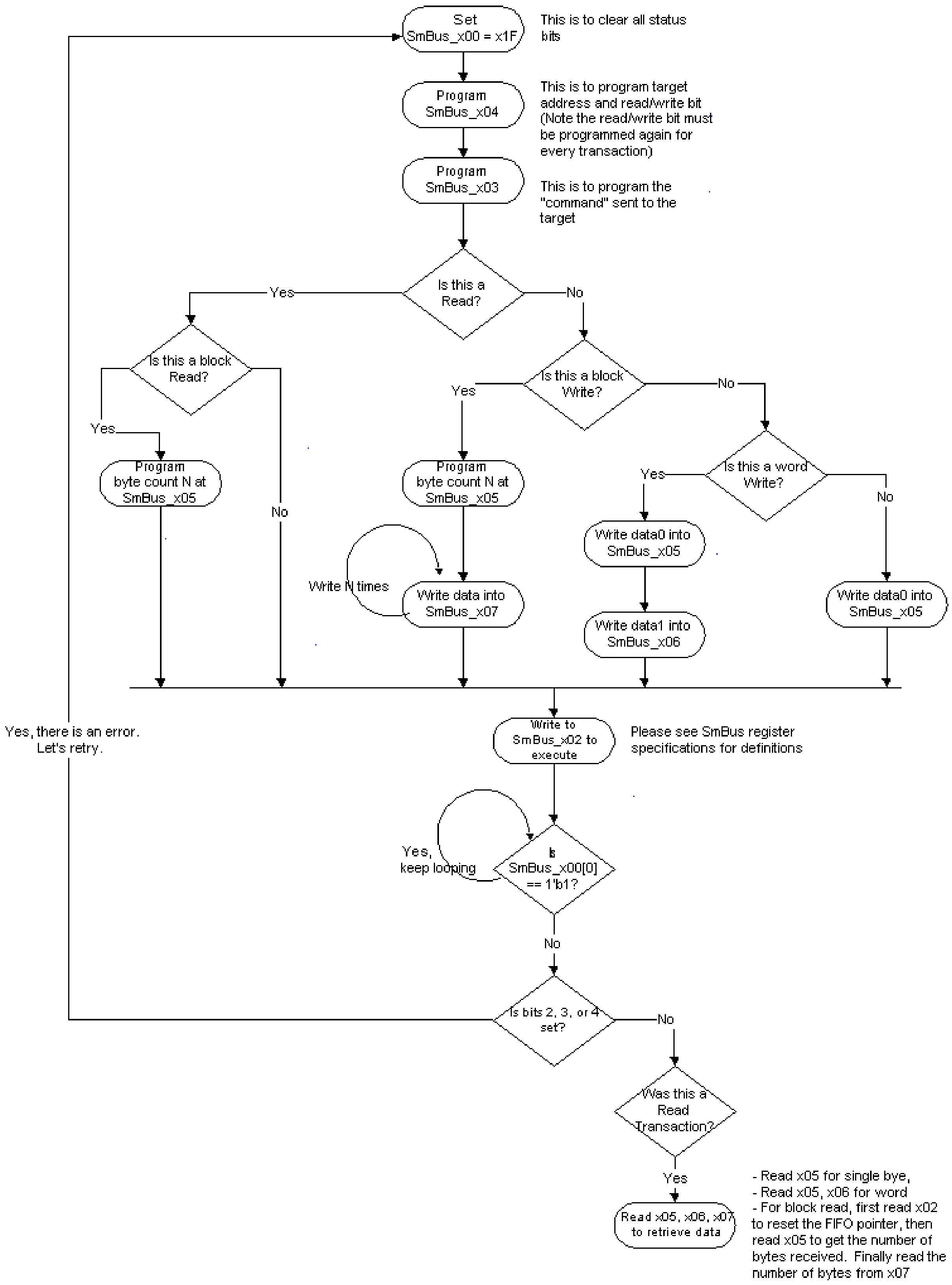

Set SmBus_x00 = x1F
This is to clear all status bits
Program SmBus_x04
This is to program target address and read/write bit (Note the read/write bit must be programmed again for every transaction)
Program SmBus_x03
This is to program the "command" sent to the target
Is this a Read?
Yes
No
Is this a block Read?
Is this a block Write?
Is this a word Write?
Program byte count N at SmBus_x05
Write N times
Write data into SmBus_x07
Write data0 into SmBus_x05
Write data1 into SmBus_x06
Yes, there is an error. Let's retry.
Write to SmBus_x02 to execute
Please see SmBus register specifications for definitions
Yes, keep looping
Is SmBus_x00[0] == 1'b1?
Is bits 2, 3, or 4 set?
Was this a Read Transaction?
Read x05, x06, x07 to retrieve data
- Read x05 for single bye,
- Read x05, x06 for word
- For block read, first read x02 to reset the FIFO pointer, then read x05 to get the number of bytes received. Finally read the number of bytes from x07

Fig 3: Programming the SMBus Host Controller

6 Serial ATA (SATA)

The SB800 SATA controller compared to the previous generation Southbridges differs in two areas:

1. Supports two additional ports providing a total of six SATA ports.
2. Supports a unique architecture that allows the user to configure the SATA controller to work in conjunction with the IDE (PATA) controller to provide configurations that cannot be supported with the SATA controller alone. This feature is referred to as "Combined Mode" in this document.

In the Combined Mode, the SATA controller can be configured as either AHCI mode or RAID mode and supports up to four SATA ports. Ports 0:3 are assigned for this configuration. The other two SATA ports will be configured as PATA ports and function in IDE mode. Two SATA ports (port 4 and port 5) share one IDE channel (could be either Primary or Secondary channel) from the IDE (PATA) controller.

Alternatively, the SATA controller can be configured to operate in IDE mode supporting up to four IDE channels. In this configuration the SATA ports 0 to 3 will be assigned to the Primary / Secondary channels as defined in Table 1 below. Together with the PATA controller, a total of six IDE ports can be supported, where one controller is configured to work in Legacy mode while the other configured to work in Native (or Compatibility) mode, or both controllers configured to work in Native mode.

Table 1 SATA Port Assignment in Combined IDE Mode

Port Number	Primary , Secondary , Master / Slave Assignment	SATA Drive Controlled by
Port 0	Primary master	SATA controller
Port 1	Secondary master	SATA controller
Port 2	Primary slave	SATA controller
Port 3	Secondary slave	SATA controller
Port 4	Primary (Secondary) master	PATA controller
Port 5	Primary (Secondary) slave	PATA controller

The following figure shows the various combined mode configurations.

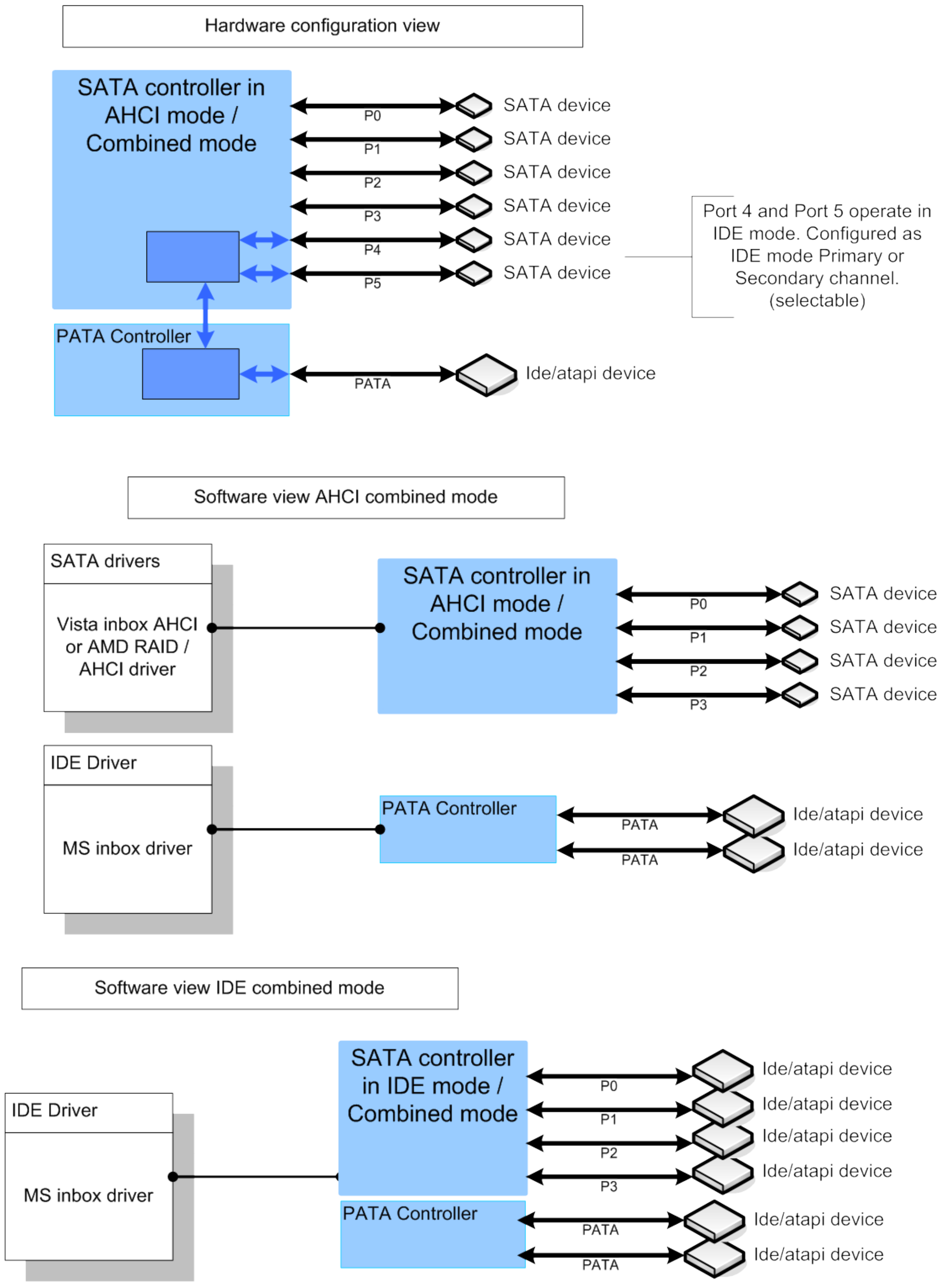


Fig 4: Combined Mode Configurations

Note: In the IDE combined mode, the MS inbox driver will control all PATA drives showing all devices under two physical IDE controllers.

6.1 Device ID

The SB800 SATA will have different device IDs for different drivers, because they are totally different devices from the driver point of view. It's not sufficient for the OS to know whether to load IDE, AHCI, or RAID driver. In a non-fresh installed condition, Windows® will match the 4 IDs (vendor ID, device ID, sub-system ID and sub-vendor ID) first, and if they are matched, it will load the driver and will not check the sub-class code. This will cause blue screen in Windows® XP if SATA RAID driver is loaded, SATA controller is in IDE mode, and device ID is shared.

Device ID	Device
4390	SATA in IDE mode
4391	SATA in AHCI mode with Microsoft® driver
4392	SATA in RAID mode with Promise non-Raid 5 driver
4393	SATA in RAID mode with Promise Raid 5 driver

6.2 SATA Controller Operating Modes

Whenever SATA is set to any of the IDE modes (native IDE, legacy IDE, IDE->AHCI, IDE->HFS), and the Combined Mode is set to OFF, only four ports (0-3) can be supported by the SATA controller, while the other two ports (4-5) cannot be used (it may work under OS but will not work under BIOS). Note that IDE->Hyperflash mode is intended only for driver testing and debugging. In IDE->Hyperflash mode, ports 1 & 3 will not work under BIOS POST and this is a limitation of the hardware.

When the Combined Mode is ON, ports 4, 5 will always be connected through the PATA controller, meaning that any device connected to this port will be shown as a PATA IDE device. For the Trevally reference board, SATA port 4 is the one closest to the CPU, while port 5 is the one that has the mobile SATA connector. For the Shiner reference board, ports 4, 5 are the two e-SATA ports.

7 APIC Programming

With the AMD integrated chipset solution, the BIOS needs to program both the Northbridge and the Southbridge in order to support APIC.

7.1 Northbridge APIC Enable

There are three bits in the Northbridge that the BIOS should set before enabling APIC support.

For RS480/RS690:

- Enable Local APIC in K8. (Set bit[11] in APIC_BASE MSR(001B) register.)
- Reg4C bit[1] - This bit should be set to enable. It forces the CPU request with address 0xFECx_xxxx to the Southbridge.
- Reg4C bit[18] - This bit should be set to enable. It sets the Northbridge to accept MSI with address 0xFEEx_xxxx from the Southbridge.

7.2 Southbridge APIC Enable

There are two bits in the Southbridge that the BIOS should set before enabling APIC support.

- Reg64 bit[3] = 1 to enable the APIC function.
- Reg64 bit[7] = 1 to enable the xAPIC function. It is only valid if bit[3] is being set.

7.3 IOAPIC Base Address

The IOAPIC base address can be defined at SMBus PCI Reg: 74h. The power-on default value is FEC00000h.

Note**:** This register is 32-bit access only. The BIOS should not use the byte restore mechanism to restore its value during S3 resume.

7.4 APIC IRQ Assignment

SB800 has IRQ assignments under APIC mode as follows:

- IRQ0~15 – Legacy IRQ
- IRQ 16 – PCI INTA
- IRQ 17 – PCI INTB
- IRQ 18 – PCI INTC
- IRQ 19 – PCI INTD
- IRQ 20 – PCI INTE
- IRQ 21 – PCI INTF
- IRQ 22 – PCI INTG
- INT 23 – PCI INTH
- IRQ 09 – ACPI SCI

SCI is still as low-level trigger with APIC enabled.

7.5 APIC IRQ Routing

During the BIOS POST, the BIOS will do normal PCI IRQ routing through port C00h/C01h. Once APIC is fully enabled by the OS, routing in C00h/C01 must be all cleared to zero.

The following is a sample ASL code that may be incorporated into the BIOS:

```
OperationRegion (PIRQ, SystemIO, 0xC00, 0x2)
Field (PIRQ, ByteAcc, NoLock, Preserve) {
    PIDX, 8,          // Index port
    PDAT, 8           // Data port
}
IndexField (PIDX, PDAT, ByteAcc, NoLock, Preserve) {
  PIRA, 8,            // INT A
  PIRB, 8,            // INT B
  PIRC, 8,            // INT C
  PIRD, 8,            // INT D
  PIRE, 8,            // INT E
  PIRF, 8,            // INT F
  PIRG, 8,            // INT G
  PIRH, 8,            // INT H
  Offset (0x10),
  PIRS, 8,            // SCI
  Offset (0x13),
  HDAD, 8,            // HD Audio
  Offset (0x15),
  GEC_, 8,            // GEC
  Offset (0x30),
  USB1, 8,            // USB1
  USB2, 8,            // USB2
  USB3, 8,            // USB3
  USB4, 8,            // USB4
  USB5, 8,            // USB5
  USB6, 8,            // USB6
  USB7, 8,            // USB7
  Offset (0x40),
  IDE_, 8,            // IDE
  SATA, 8,            // SATA
  Offset (0x50),
  GPP0, 8,            // GPP0
  GPP1, 8,            // GPP1
  GPP2, 8,            // GPP2
  GPP3, 8,            // GPP3
}
```

8 A-Link Bridge

The registers are accessed using an address-register/data-register mechanism. The address register is AB_INDX [31:0], and the data register is AB_DATA [31:0].

31:30	29:17	16:2	1:0
RegSpace[1:0]	Reserved	Register address[16:2]	Reserved

AB_INDX [31:0]

31:0
Data[31:0]

AB_DATA [31:0]

RegSpace[1:0]	
00b	AXINDC Index/Data Registers. (AX_INDXC)
01b	AXINPD Index/Data Registers (AX_INDXP)
10b	Alink Express Configuration (AXCFG)
11b	Alink Bridge Configuration (ABCFG)

Definition of RegSpace[1:0]

In order to read or write a particular register, the software will write the register address and the register space identifier to AB_INDX and then do a read or write to AB_DATA. This is analogous to how PCI configuration reads and writes work through I/O addresses CF8h/CFCh.

The location of AB_INDX in the I/O space is defined by the abRegBaseAddr register located at PMIO, register 0E0h. The AB_DATA register address is offset 4h from the AB_INDX address. The address of the AB_INDX must be 8 byte aligned.

31:3	2:0
BaseAddr[31:3]	Rsv

abRegBAR [31:0] at PMIO, Register 0E0h

AXCFG and ABCFG registers are accessed indirectly through AB_INDX/AB_DATA. To read or write a particular register through AB_INDX/AB_DATA, the register address and the register space identifier is first written to AB_INDX. The specified register is then accessed by doing a read or write to AB_DATA (see the example below).

Access to AXINDC and AXINDP registers requires a second level of indirection. Registers in these spaces are addressed through the following indirection registers: AX_INDEXC/AX_DATAC and AX_INDEXP/AX_DATAP.

Register	Indirect Address
AX_INDXC	30h
AX_DATAC	34h
AX_INDXP	38h
AX_DATAP	3Ch

Example: To write to register 21h in the INDXC space with a data of 00, the following steps are required:

1. Out 30h to AB_INDX. This will prepare to write register from INDXC
2. Out 21h to AB_DATA. This will set register 21h of INDXC
3. Out 34h to AB_INDX. This will prepare to write data to register defined in steps 1 and 2 above
4. Out 00 to AB_DATA. This will write the data to the register defined n steps 1 and 2 above.

8.1 Programming Procedure

Indirect access is required to access both A-Link Express Configuration and A-Link Bridge Configuration register space. The programming procedure is as follows:

Write:

1. Set the A-Link bridge register access address. This address is set at PMIO, register 0E0h. This is an I/O address and needs to be set only once after power-up. The I/O address must be on an 8-byte boundary (i.e., 3 LS bits must be zeroes).

   Example: To set CD8h as an A-Link bridge register access address:

```
mov     dx,0CD6h                ;
mov     al,0E0h         ; Index
out     dx,al
mov     dx,0CD7h                ;
mov     al,0D8h         ; Data
out     dx,al
mov     dx,0CD6h                ;
mov     al,0E1h         ; Index
out     dx,al
mov     dx,0CD7h                ;
mov     al,00Ch         ; Data
out     dx,al
```

Note: Although the 32-bit I/O address is set for the A-Link bridge (e.g., 00000CD8h), the bridge may be accessed by a 16-bit address (i.e., 0CD8h). The MS word is set to 00 by default (see the example below).

2. Write the register address in the AB_INDX.

   Example: To write to the A-Link Bridge configuration register space at 90h:

```
mov     dx, 0CD8h               ; I/O address index assigned to A-Link
mov     eax, 0C0000090h         ; Bits[31:30] = 11 for A-Link Bridge
register
                                ; space
out     dx, eax                 ; Register index is set
mov     dx, 0CDCh               ; I/O address for data
mov     eax, 00000001h          ; Power down 2 lanes to save power
out     dx, eax
```

Read:

Use a similar indirect procedure to read out the register value inside AB and BIF.

8.2 A-Link Express Configuration DMA Access

To enable A-Link Express Configuration DMA access, a specific register space needs to be configured first. This register is in the A-Link Express register space that refers to port-specific configuration registers (see section 8 above for a description of the AB_INDX register). When configuring the register, bit 2 of byte 4 needs to be set to “1” to enable the DMA access.

Follow these steps to initialize A-Link Express configuration DMA access (this initialization has to be performed during S3 wakeup also):

1. Issue an I/O write to AB_INDX. The write data's bit [31:29] should be 100b (binary). The register to be written is in the port-specific configuration register space, and bit [16:0] should be 0x4 (hex).
2. Issue an I/O write to AB_Data. This write data's bits[31:0] should be 0x4h (i.e., 32'b0000_0000_0000_0100 binary).

```
mov         dx,  0CD8h              ; ALINK_ACCESS_INDEX
in   eax, dx
and  eax, NOT (0C001FFFFh)
or   eax, 080000004h
out  dx, eax
mov         dx,  0CDCh              ; ALINK_ACCESS_DATA
mov         eax, 04h
out  dx, eax


            ;Write   AB_INDX   0x30
            ;Write   AB_DATA  0x21
            ;Write    AB_INDX  0x34
            ;Write   AB_DATA  0x00
```

```
    mov         dx,  0CD8h                ; ALINK_ACCESS_INDEX
    mov         eax, 30h
    out  dx, eax
    mov         dx,  0CDCh                ; ALINK_ACCESS_DATA
    mov         eax, 21h
    out  dx, eax
    mov         dx, 0CD8h                 ; ALINK_ACCESS_INDEX
    mov         eax, 34h
    out  dx, eax
    mov         dx, 0CDCh                 ; ALINK_ACCESS_DATA
    mov         eax, 00h
    out  dx, eax
```

8.3 Enabling Non-Posted Memory Write for K8 Platform

The register index 10h of AXINDC bit9 should be set to 1.

```
    mov         dx,AB_INDX                ; AB index register
    mov         eax,30h                   ; Address of AXINDC
    out  dx,eax                           ; Set register address
    mov         dx,AB_DATA                ; To write register address
    mov         eax,10h                   ; Write register address
    out  dx,eax
    mov         dx,AB_INDX
    mov         eax,34h                   ; To write data portion of the AXINDC
    out  dx,eax
    mov         dx,AB_DATA
    in   eax,dx                           ; Read the current data
    or   al,200h                          ; Set bit 9
    out  dx,eax                           ; Write data back.
```

9 CIR Support

The integrated micro controller (IMC) in SB800 provides the interface to connect two IR receivers and two IR transmitters. The two IR receivers are for Learning IR Data and Receiving IR Data and the two IR transmitters are for Transmitting IR Data. The controller has the ability to support wake from S5, S4, S3 or S1 states and is capable of storing and returning the wake identification code to the IR driver.

BIOS Requirements:
1. IMC firmware / CIMx program IMC, CIR base address and IRQ assignment of the IR.
2. CIMx initializes the ASL code with the exception of CIR pin configuration information.
3. Platform BIOS programs the CIR pin configuration in IMC logical Device 5.
4. Platform BIOS then add the information to the ASL code to declare the hardware capabilities (transmitter, receiver, jack, etc.) to the IR driver.

Driver Requirements:
The CIR driver is available on the AMD NDA site. This driver should be installed for enabling the function of the CIR.

9.1 Southbridge CIR

9.1.1 *Host and CIR Communication*

The platform BIOS needs to configure the CIR pins based on the platform requirement. Platform BIOS will program registers in logical device # 5 of the southbridge.

9.1.2 *Logical Device Number 5 Registers*

The CIR is designed as a logical device in the southbridge that complies with the Plug and Play ISA Specification. The logical device number of the CIR device is 5 and it contains the following registers that need to be updated by the Platform BIOS.

IRC_PIN_CONTROL (INDEX:0xAB): Port to write data to the FIFO.				
FieldName	Bits	Default	Type	Description
IR_OpenDrain	4	0	R/W	Transmitter output pin type. 0: TX are push-pull driver. 1: TX are open-drain.
IR_Tx1	3	0	R/W	0: TX1 emitter output pin held high 1: TX1 pin outputs emitter signal
IR_Tx0	2	0	R/W	0: TX0 emitter output pin held high 1: TX0 pin outputs emitter signal
IR_Enable	1:0	0	R/W	00: IR Function will not control the IR Rx and Tx pins. 01: IR Function controls the RX and TX0 pins. 10: IR Function controls the RX and TX1 pins. 11: IR Function controls the RX and both TX pins.

IR_OpenDrain (bit 4) - Program this bit to 1 or 0 depending on the how it is connected to the CIR emitter.

IR_Tx1 and IR_Tx0 (bits 2 and 3) - Program these bits to '1' to enable the transmit signal for the corresponding Tx that is used on the platform.

IR_Enable (bits 0 and 1]) - Program these bits to ‘1’ to enable the transmit signal for the corresponding Tx that is used on the platform.

IRC_CONFIG1 (INDEX:0xA3): IR Controller Configuration Register 1				
FieldName	Bits	Default	Type	Description
IR_TXINV	1	0	Read/Write	Invert the Transmit signal
IR_RXINV	0	0	Read/Write	Invert the Received signal

IR TX or RX signal can be inverted by programming bit 1:0. Depending on the CIR emitter requirements these bits can be programmed accordingly.

Note that the IMC firmware will set IMC_PortActive to 0x6E once the IMC is enabled because the power-up default value of 0x2E is usually used by most Super IO controllers. Therefore, an I/O conflict may occur when IMC and the Super IO controller are used together. If the value picked by the IMC firmware is still not suitable for the design, platform BIOS developers can change it by programming the IMC_PortActive field of the Southbridge LPC ISA bridge PCI configuration register A4h to the desired address.

LPC ISA Bridge (Device 20, Function 3):

IMC_PortAddress - R/W - 16 bits - [PCI_Reg: A4h]			
Field Name	**Bits**	**Default**	**Description**
IMC_PortActive	0	1b	When set to 1, LPC can decode the address specified in IMC_PortAddress, otherwise LPC ignores it.
Addr15_1	15:1	0017h	When Addr15_1 is non-zero, and if an IO cycle from host has address[15:1] = Addr15_1, the cycle will be routed to IMC instead of to LPC bus. By default, address[15:0] = 002Eh or 002Fh will be routed. Read-only to host if IMC_PortHostAccessEn = 0.

Since the activation of Logical Device Number 5 and the address assignment of the registers are handled by CIMx during the IMC enabling process, the platform software needs only to program the value of the above registers to enable the CIR (refer to the Get_IMC_MsgReg_Base_Addr function in the sample code below).

```
; Find the IMC Config port address to access the configuration registers
; Input:     None
; Output:    DX = IMC Config port address

Get_IMC_Config_Port:
            mov         eax, 8000A3A4h          ; LPC bridge configuration register A4h
            mov         dx, 0CF8h
            out         dx, eax
            mov         dx, 0CFCh
            in                      eax, dx                 ; Read IMC_PortAddress register
            and         ax, 0FEh                ; Mask out bit 0 to retain LPCCfg_A4[15:1]
            mov         dx, ax                  ; DX = Index Port address
            ret

; Find CIR base address
; Input:     None
; Output:    DX =  CIRregister base address

Get_IMC_CIR_Base_Addr:
            call        Get_IMC_Config_Port ; Get IMC Config port address

            mov         al, 5Ah                 ; Enter configuration state
            out         dx, al                  ; Write Index Port

            mov         al, 07h                 ; Select Logical Device Number register
            out         dx, al                  ;
            inc                     dx                      ; DX = Data Port
            mov         al, 05h                 ; Select logical device 5, CIR  Registers
            out         dx, al
            dec         dx                      ; DX = Index Port

            mov         al, 60h                 ; Read high byte of CIR register base address
            out         dx, al
            inc                     dx                      ; DX = Data Port
            in                      al, dx
            mov         bh, al                  ; BX =  CIR base address bit[15:8]
            dec         dx                      ; DX = Index Port

            mov         al, 61h                 ; Read low byte of  CIR register base address
            out         dx, al
            inc                     dx                      ; DX = Data Port
            in                      al, dx
            mov         bl, al                  ; BX =  CIR base address bit[7:0]
            dec         dx                      ; DX = Index Port

            mov         al, 0A5h    ; Exit configuration state
            out         dx, al
            mov         dx, bx                  ; DX =  CIR register base address[15:0]
            ret
```

9.1.3 *Sample Code to Read and Write CIR Registers*

```
; Write an 8-bit value to a CIR register
; Input:      AH = value to write
;                 AL = message register index value
; Output:   None

Write_CIRReg:
              push        ax                          ; Save input parameters
              call        Get_IMC_CIR_Base_Addr; Get IMC CIR register base address
              pop         ax                          ; Restore input parameters
out           dx, al                  ; Write index to CIR index register
              mov         al, ah
              inc                     dx                          ; DX = message data register
              out         dx, al                      ; write data to message data register
              dec         dx
              ret

; Read from a CIR register
; Input:      AL = message register index value
; Output:   AL = value read

Read_CIRReg:
              push        ax                          ; Save input parameters
              call        Get_IMC_CIR_Base_Addr; Get IMC CIR register base address
              pop         ax                          ; Restore input parameters

              out         dx, al                      ; Write index to message index register
              inc                     dx                          ; DX = message data register
              in                      al, dx                      ; AL = value read from message register requested
              ret
```

9.1.4 *Sample Code to Enable Tx0 and Program it as Push Pull Driver*

```
mov       al, ABh    ; AL =  CIR pin config register index value
          call       Read_CIRReg

mov       ah, 05h                ; AH = value to write
          and al, E0h            ; clear bits 4:0
          or al, ah   ; ah = contents to write back
          mov        al, ABh    ;              AL = CIR register index value
          call       Write_CIRReg
```

9.1.5 *Sample ASL Code for CIR*

```
//SB800 Embedded Controller IR device
Device(ECIR)
{
        Name(_HID,EISAID("AMDC001"))

        Method(_STA)
        {
                Store(0x00, Local1)
                If (LEqual(OSTY, 6))                //We need to use IR only for Vista
                {
                        Store(STA(0x05), Local1)
                }
        Return(Local1)
        }

        Method(_CRS)
        {
                Name(RSRC,ResourceTemplate()
                {
                        IO(Decode16,0x00,0x00,0x08,0x08)
                        IRQNoFlags() {}
                })

        CreateByteField(RSRC,0x02,IO1)      //IO port low
        CreateByteField(RSRC,0x03,IO2)      //IO port high
        CreateByteField(RSRC,0x04,IO3)      //IO port low
        CreateByteField(RSRC,0x05,IO4)      //IO port high
        CreateWordField(RSRC,0x09,IRQV)     //IRQ mask

        Acquire(ECMU, 5000)
        CFG(0x05)

        If (ACT)
        {
                Store(IOBL,IO1)
                Store(IOBH,IO2)
                Store(IOBL,IO3)
                Store(IOBH,IO4)
                Store(0x01,LOCAL0)
                ShiftLeft(LOCAL0,INT,IRQV)
        }

        XCFG()
        Release(ECMU)
        Return(RSRC)
        } // Method(_CRS)



Name(_PRS,ResourceTemplate()
        {
                StartDependentFn(0,0)
                {
                        IO(Decode16,0x550,0x550,0x1,0x8)
                        IRQ(Edge,ActiveHigh,Shared) {0x5}
                }
        StartDependentFnNoPri()
        {
                        IO(Decode16,0x650,0x650,0x1,0x8)
                        IRQ(Edge,ActiveHigh,Shared) {0x5}
        }
        StartDependentFnNoPri()
        {
                        IO(Decode16,0x550,0x550,0x1,0x8)
                        IRQ(Edge,ActiveHigh,Shared) {0x3}
```

```
}
StartDependentFnNoPri()
{
                IO(Decode16,0x650,0x650,0x1,0x8)
                IRQ(Edge,ActiveHigh,Shared) {0x3}
}

EndDependentFn()
})

Method(_SRS,1)
{
                //Arg0 = PnP Resource String to set
                CreateByteField(Arg0,0x02,IO1)
                CreateByteField(Arg0,0x03,IO2)
                CreateWordField(Arg0,0x09,IRQV)

                Acquire(ECMU, 5000)
                CFG(0x05)

                Store(IO1,IOBL)
                Store(IO2,IOBH)
                FindSetRightBit(IRQV,Local0)
                Subtract(Local0,1,INT)
                Store(0x01,ACT)

                XCFG()

                Release(ECMU)
        } // Method(_SRS)

Method(_PRW, 0) { Return(GPRW(0x17, 4)) }       // can wakeup from S4 state

Method(_PSW, 1)
{
                If(Arg0)
                {
                        Store(1, IRWF)
                }
                Else
                {
                        Store(0, IRWF)
                }
}


//AMD specific control method to return board related IR info
Method(IRCF, 2)
{

                //Bit 0 - 7
                //Version number of silicon (Supports up to 16 version numbers).
                //For example
                //0x39 for A11, 0x3A for A12

                //Bit 8 v 10(Supports up to 7 transmitters.  (AMD IR device supports 2 transmitters)
                //Number of TX ports
                //For example
                //0 for no emitters, 1 for one emitter

                //Bit 11 v 12(Supports up to 3 receivers. (AMD IR device supports 2 receivers)
                //Number of receivers
                //For example
                //0 for no receivers, 1 for one receiver

                //Bit 13 v 15
                //Receiver number for Learn.
                //0 for no learning receiver.
```

//Otherwise it is the mask value of learning receiver (3 bits to support maximum 3 receivers).

//bit16 to represent the presence of receiver LED on the hardware

```
                Name(CONN, 0)
                    If (LEqual(ARG0, 0x01))
                    {           //Query platform configuration
                                Or(\RVID, CONN, CONN)
                                Or(0x13200, CONN, CONN)
                                Return(CONN)
                    }
                    If (LEqual(ARG0, 0x02))
                    {
                                //Emitter Jack connection info
                                ShiftLeft(\GP64, 1, CONN)
                                Or(\GP51, CONN, CONN)
                                Return(CONN)
                    }
                    If (LEqual(ARG0, 0x03))
                    {           //Flash LED function
                                If (LEqual(ARG1, 0x00))
                                {           //Don't flash the LED
                                            Store(1, \G31O)
                                }
                                else
                                {           //Flash the LED
                                            Store(0, \G31O)
                                            Sleep(100)
                                            Store(1, \G31O)
                                }
                    }
                    If (LEqual(ARG0, 0x04))
                    {           //Set EC-ACPI Interrupt Config
                                If (LEqual(ARG1, 0x00))
                                {
                                            //Set as edge triggered
                                            Store(0, \ACIR)
                                }
                                else
                                {
                                            //Set as level triggered
                                            Store(1, \ACIR)
                                }
                    }
            If (LOr (LEqual(ARG0, 0x00),LGreater(ARG0, 0x05) ))
            {
                                Store(0xFFFFFFFF, CONN)         //Return error is invalid function
            }
                    Return(CONN)
        }//_IRS
            } //Device(ECIR)
```

}

10 SMI Programming

10.1 Workaround for SMI Command Port Status Byte

This workaround is required as per erratum # 20 in the SB8xx Family Product Errata (PID # 47481).

If an IMC-enabled platform is using the SMI command port base +1 with byte (8 bit) access, all such access should be changed to use word (16 bit) access starting at the SMI command port base address.

For platforms that use the SMI command port status register, typically software will write to the command port status byte (base +1) and then write to the command port base consecutively to set up the software SMI. The following code example shows how to use word access instead of byte access:

```
      //
      // Issue command port SMI
      //
      if (*ArgumentBufferSize == 2) {
            //WriteIO (ACPI_SMI_DATA_PORT, AccWidthUint8, &bData);
            wValue = bIndex + (bData << 8);
            WriteIO (ACPI_SMI_CMD_PORT, AccWidthUint16, &wValue);
      } else {
            WriteIO (ACPI_SMI_CMD_PORT, AccWidthUint8, &bIndex);
      }
```

Reading from the command port status register will also require using word access:

```
      Get SMI.
      VOID
      SwGetContext(
        IN  DATABASE_RECORD    *Record,
        OUT SMM_CONTEXT        *Context
        )
      {
        EFI_STATUS  Status;
        UINT8       ApmCnt;

        Status = mSmst->SmmIo.Io.Read (
                                   &mSmst->SmmIo,
                                   SMM_IO_UINT16,
                                   SmiCmdPort,
                                   0,
                                   &ApmCnt
                                   );
        ASSERT_EFI_ERROR (Status);

        Context->Sw.SwSmiInputValue = ApmCnt;
}
```

Note: The above examples assume that the setting up of SMI is done by writing to the Command port and Command port+1 consecutively when setting up a software SMI. If the Status byte is read independently, then care must be taken to disable the SMI command port before doing a word access to command port base address in order to prevent an SMI interrupt from being generated. The SMI command port can be enabled after the write is completed. The SMIControl75 field of the SmiControl4 register (bit [23:22] of SMI_Reg: B0h) is used to enable/disable SMI.

11 Legacy BIOS Implementation for Chipset Integration Module Extensive (SB800 CIMx)

11.1 Introduction

CIMx-SB800 introduces a similar interface and distribution model as the CIMx-SB7xx to help quickly integrate SB800 Southbridge family support in customer products. It differs from the CIMx-SB7xx by using a new Push High interface which is the same as the AMD AGESA^{TM} module used for the V5 series model. Since now CPU/NB/SB use the same interface file, it helps reducing the coding size of the interface file of the IBV (i.e., individual BIOS company or BIOS vendor).

11.2 CIMx Interface Calls Environment

Prior to calling any CIMx interface, the following is required:

1. Place CPU into 32-bit protected mode.
2. Set CS as 32-bit code segment with Base/Limit – 0x00000000/0xffffffff.
3. Set DS/ES/SS as 32-bit data segment with Base/Limit – 0x00000000/0xffffffff.

11.3 Interface Definition

All interface calls to CIMx-SB800 binary are C-like calls to the Entry Point of the binary image.

```
void (*ImageEntryPointPtr)(void* Config)
```

11.3.1 *Southbridge Power-On/Reset Initialization*

Upon system power-on, or cold reset, there is minimal initialization required like SMBus base address programming, enabling the legacy IO (like port 60/64 etc) decoding etc to bring the system to a working state. BIOS should call this entry to B1 module at very early stage during power on initialization.

CIMx has assumed a set of default values for all the build parameters such as the BIOS size, SMbus base address, power management base addresses etc. If you want to use your own set of values for these configurable options, you can define the build parameters structure and give the 32-bit physical pointer as input to the CIMx module. If the pointer is NULL or set to all 1, then the CIMx module uses the default parameters which are built in. Also the BIOS developer has the flexibility of changing the oem.h file to redefine the values but in this case, the CIMx module needs to be rebuilt for the new values to be included in the binary.

Parameters

Refer to SB800 CIMx implementation note.chm for further description.

11.3.2 *SouthBridge BIOS POST Initialization*

SB800 BIOS POST initialization is divided into 3 stages as: “Before PCI Enumeration”, “After PCI Enumeration” and “Late POST Initialization”. All these 3 calls need the same inputs except that the function id should be set up properly. The “Before PCI Enumeration” routine should be called before PCI devices are enumerated and resources are assigned. It is recommended to call the “Before PCI Enumeration” routine at very early POST after memory detection. The “After PCI Enumeration” routine should be called after the resources are assigned to the PCI devices since this routine initializes MMIO spaces of some of the devices. The “Late POST Initialization” routine should be called at the end of BIOS POST just before giving control to OS.

Parameters

Refer to SB800 CIMx implementation note.chm for further description.

11.3.3 *S3 Resume Initialization*

CIMx will do the necessary programming to prepare SB to resume from sleep state. It is highly recommended to use same copy AMDSBCFG for BIOS post and S3 resume initialization. This will allow CIMx to exchange data between interface calls if necessary. There are two calls necessary during S3 resume time to restore SB to the previous state: The first call should be done before PCI devices config spaces are restored and the other call should be done after PCI devices config spaces are restored.

Parameters

Refer to SB800 CIMx implementation note.chm for further description.

11.3.4 *Callback Interface Definition*

Callback functions are supported in CIMx module to enable the OEMs to hook at a specific place in the CIMx module. This will enable the OEMs to do some specific initialization in between the CIMx functions. OEMs who require some callbacks can request AMD to add this call in CIMx.

Parameters

Refer to SB800 CIMx implementation note.chm for further description.

12 UEFI BIOS Implementation for Chipset Integration Module Extensive (SB800 CIMx)

12.1 Introduction

The platform interface and configuration driver (AMD platform interface) is the interface between the UEFI core and the AMD chipset driver. It has 2 purposes: The first is to allow the platform to configure the chipset driver behavior (device enabling/disabling, internal configuration change). The second is to provide a means to the platform to abstract some platform specific behavior (GPIO toggling, hardware location, hardware support, etc) and allow the driver to request interaction and information from the platform. This driver is written both by AMD and the UEFI core provider.

The goal is to provide a driver that can be added to any code base with the minimum amount of work and modification. Most calls are done through the entry points or callbacks and all the platform code is centralized in one location.

The driver is built using the TianoCore tool chain (EDK) and verified in a variety of platform BIOS. The EDK build files will be provided as part of the driver package to make it easy to integrate it in any codebase.

12.2 CIMx Interface Calls Environment

The AMD chipset driver is composed of four parts:

1. PEI driver
2. DXE driver
3. AMD UEFI driver library
4. Core chipset driver

The first three are unique to the AMD UEFI chipset driver. The core chipset driver is shared with the legacy driver and is not covered by this document. It is integrated in the UEFI driver as a library since both PEI and DXE need to access its functions.

12.2.1 *Southbridge PEI Module*

The southbridge is controlled by the *AmdSbPei* PEI. This PEI is responsible for the power-on initialization of the southbridge including recovery support, boot-mode update, SMBus control and the production of a reset service.

The boot mode is only updated if the boot mode seen by the southbridge has a higher priority, according to the PI specification, than the current mode. Since the boot mode might not be final when the SB PEIm is entered, a function is provided in AMD_PEI_SB_INIT_PPI to allow the platform to query the module for the SB boot mode. The resulting configuration is used by the SB PEI module for power-on initialization and then passed to DXE through a HOB.

Parameters

Refer to SB800 CIMx implementation note.chm for further description.

12.2.2 *Southbridge DXE Driver*

The southbridge is initialized in three phases:

5. Before PCI initialization
6. After PCI initialization
7. Late SB Init

The first phase is done on entry to the *AmdSbDxe* driver and this is assumed to happen before the PCI subsystem is initialized. This allows the driver to perform some early initialization on peripheral controllers before they are discovered by the core and publish the protocols supported by the southbridge. The runtime southbridge driver to support is also initialized and the necessary callbacks for the subsequent phases are registered.

The second phase happens once PCI has been initialized by the core. It is performed upon the first installation of the PciIo protocol (`EFI_PCI_IO_PROTOCOL_GUID`) by the core, through a callback on the protocol installation with a TPL of `TPL_NOTIFY`. At that point the peripheral controllers have their resources assigned and the SB driver can start using them. Based on platform hardware, procedures such as USB MMIO assignment, SATA port enumeration and audio configuration are performed.

The final phase happens when the system is ready to boot and the ready to boot (`EFI_EVENT_SIGNAL_READY_TO_BOOT`) event has been signaled. A callback on the event with a TPL of `TPL_NOTIFY` is used to initiate this phase. The final southbridge configuration, including HPET and SATA, is performed at this point. The southbridge driver also publishes an “end of SB init” protocol at the end of its initialization to allow other drivers to trigger on it.

Parameters

Refer to SB800 CIMx implementation note.chm for further description.

Appendix: Revision History

Date	Rev.	Description
May, 2012	3.03	• Updated section 3.2.1.1 Special Locked Area in CMOS. • Fixed typo in section 3.6 System Restart after Power Fail, where register 74h is corrected to 5Bh.
April, 2011	3.02	• Added new section 10 SMI Programming.
February, 2011	3.01	• Added new section 9 CIR Support.
November, 2010	3.00	• Public release • Updated section 8 A-Link Bridge with the proper addresses for abRegBAR, AB_INDX, and AB_DATA.
June, 2010	1.01	• Updated Figure 1 and Figure 2 to include IDE controller. • Removed previous section 1.3 Features of the SB800. • Updated decscription of PCI ROM in Section 2.3 Memory Map. • Corrected typos 22h to 16h in Section 2.1 PCI Devices and Functions. • Updated section 3.6 System Restart after Power Fail. • Added missing text to section 3.6.1 Power Fail and Alarm Setup. • Updated paragraph in Section 6 Serial ATA (SATA) to clarify how 6 IDE devices can be supported.
Feb 20, 2009	1.00	• First release.