How does a Malware modify the BIOS

Hrant Tadevosyan, Levon Martirosyan

With the advanced security measurements the malware authors started get creative. It is best to avoid any dependencies, when trying to hide the malicious activities. For this particular reason malware authors almost always seek to find alternative ways for writing regular software code, meaning, they try to avoid libraries, helper tools, etc. The lower is the level of the malware source code, the harder it is to monitor and reason about the activities. This document will focus only on, how can a software reach the BIOS from the user-space. The attention will be drawn to the x86 architecture, although it is worth mentioning that the steps should be applicable for any architecture/operating system combination.

The first thing to note is the **RFLAGS** register in x86 Architecture. The Intel Architecture Manual[1] shows that the register contains a lot of interesting fields, one of them is called *IOPL (I/O Privilege Level)*. This bit can be used to make the I/O instructions such as **IN** or **OUT** to be accessible from the user space (ring 3).

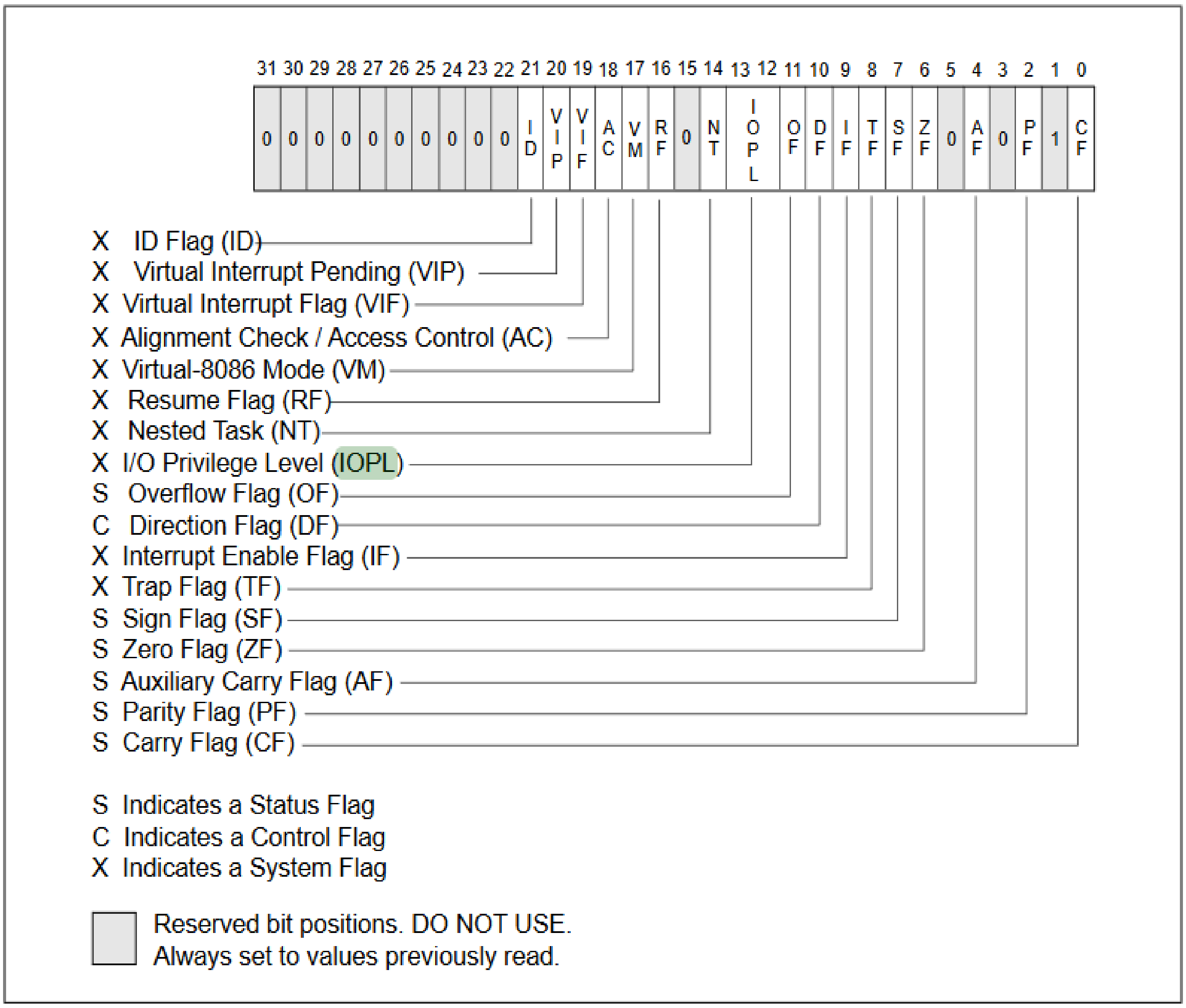

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check / Access Control (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE. Always set to values previously read.

Figure 3-8. EFLAGS Register

These 2 bits from IOPL will be used by the hardware for an internal check with the *Current Privilege Level (CPL)*. As an example we can take a look at the official pseudo-code of the *IN* instruction from the Intel Reference Manuals. We can safely ignore the *PE (Protected Mode Enabled)* bit since it is there to tell us explicitly that the *IOPL* bits are available in 32bit+ modes (Protected Mode+). The original Intel reference manual for 16bit 8086 does not mention anything about the I/O privilege levels, hence, the protection rings are available from protected mode only.

The *VM/VME (Virtual 8086)* bit can also be ignored since it also does not play a major role for this discussion. When this bit is set then the operating system can access the 16bit (real mode) functionality from the 32bit+ (protected mode+). It is a protected mode feature. One usage for it is to jump back in real mode and access legacy BIOS functions.

It is also interesting to observe that the reference manual description for **IN** instruction does not explain everything very precisely. From the pseudo code, one might think when the *IOPL* register bits are set to **0b11 = 3** then in ring 3 where the *Current Privilege Level (CPL) = 3* we still can't access the I/O ports, however, this is not true.

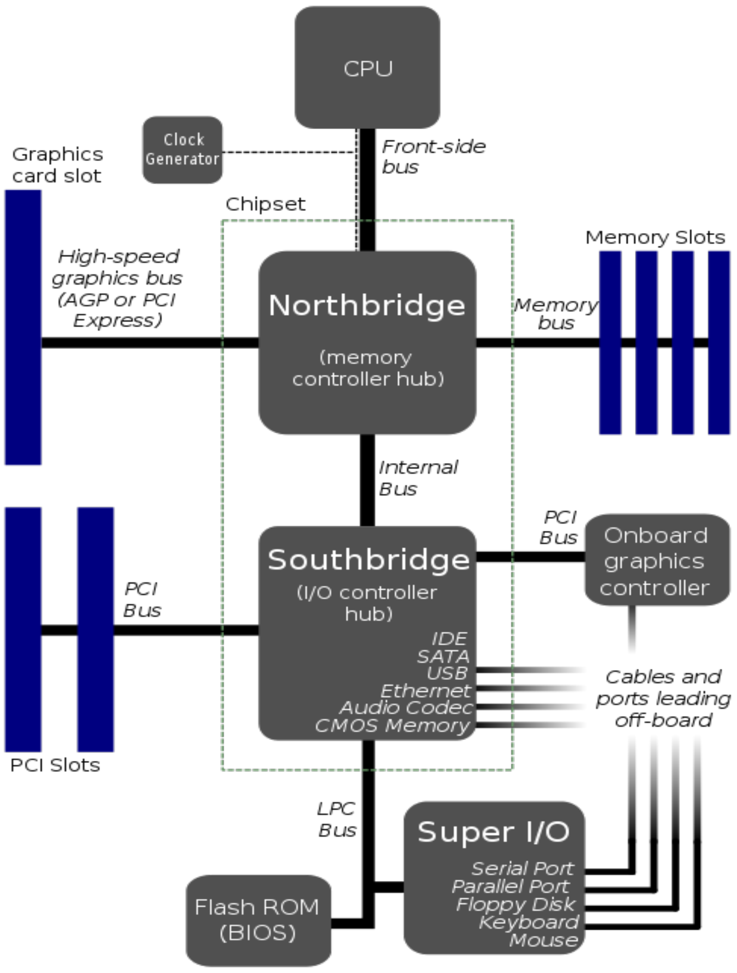
To 'prove' this, one can simply create a kernel driver and execute manually change the IOPL bits, which after he would need to create a user space program containing I/O instructions. It is easy to see that the program would not crash after the changes.

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1)))
    THEN (* Protected mode with CPL > IOPL or virtual-8086 mode *)
        IF (Any I/O Permission Bit for I/O port being accessed = 1)
            THEN (* I/O operation is not allowed *)
                #GP(0);
            ELSE ( * I/O operation is allowed *)
                DEST := SRC; (* Read from selected I/O port *)
        FI;
    ELSE (Real Mode or Protected Mode with CPL  IOPL *)
        DEST := SRC; (* Read from selected I/O port *)
FI;
```

As we can see there is an additional check for that IOPL bit. With this capability we can lower the privilege level of instructions like IN or OUT or their variations. Obviously, allowing to set the IOPL bit from ring 3 would have been a security risk, which is only ring 0 code can set those bits. Most well known operating systems such as Linux or Windows do support system functions which can be used to set the current processes state. In Linux one would use the *iopl* system call, where in Windows we have the undocumented *NtSetInformationProcess* function which accepts an IOPL request. Other operating systems will most likely have their own implementations for it. This paper will focus on Linux (it is worth mentioning that Linux requires the *CAP_SYS_RAWIO* capability to be set). After setting both of the IOPL bits, we can safely use the I/O instruction which would have been throwing an exception in other scenarios.

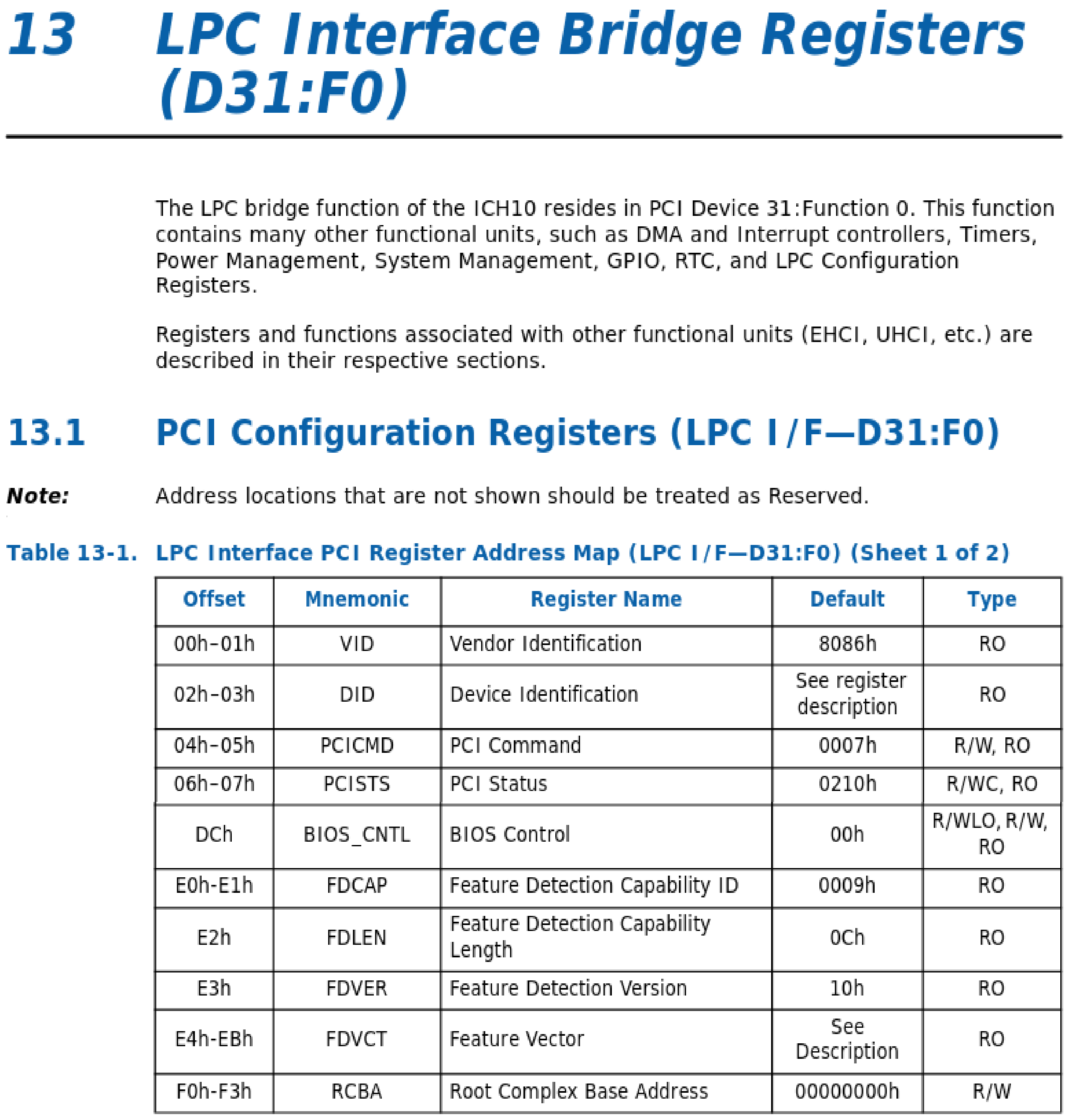
Note that the *iopl* system call is now labeled as deprecated and instead the Linux manuals recommend to use the *ioperm* system call. One can specify the I/O port range to be enabled for ring 3 operations. The system call as with its older alternative requires the *CAP_SYS_RAWIO* capability.

The next thing to do is, accessing the necessary devices on the motherboard. Different laptops/pcs with various OEM, CPU Vendors will have different designs. One has to figure out what peripherals he needs to access in order to reach the final destination. This particular example can be described with the following diagram:



The diagram hints that if someone needs to access the BIOS then the person has to go trough all the devices from CPU to Flash ROM (BIOS). These include the Northbridge and Southbridge controllers. In Intel manuals they are referred to as Memory Controller Hub (MCH) and I/O Controller Hub (ICH) respectively. Note that different models will have different names even if the vendor is the same. For our sample the MCH documentation [3] states that we use the PCI Configuration Address register at *0xCF8* and the PCI Configuration Data register at *0xCFC* to ask for PCIe devices. These values are the same for our analyzed AMD samples as well.

Now, that we have the PCI I/O registers, we can look for a southbridge device, which can talk to the flash. According to the diagram it is done via the LPC. Next step is identifying the ICH family [4]. For this example we have the ICH10, which tells us that we have to look for the *Root Complex Base Address (RCBA)* register, since that is the one register that contains the memory block for the SPI registers. The LPC Interface Bridge can be found at PCI Bus 0x00, Device 0x1F, Function 0x00 (0.1f.0) according to the documentation. This is the common address value for Intel (AMD has it at 0.14.3 usually). Note that the below picture from the reference manual is not completely correct, we had to cut out the majority of rows to make it fit in one place nicely.



13 LPC Interface Bridge Registers (D31:F0)

The LPC bridge function of the ICH10 resides in PCI Device 31:Function 0. This function contains many other functional units, such as DMA and Interrupt controllers, Timers, Power Management, System Management, GPIO, RTC, and LPC Configuration Registers.

Registers and functions associated with other functional units (EHCI, UHCI, etc.) are described in their respective sections.

13.1 PCI Configuration Registers (LPC I/F—D31:F0)

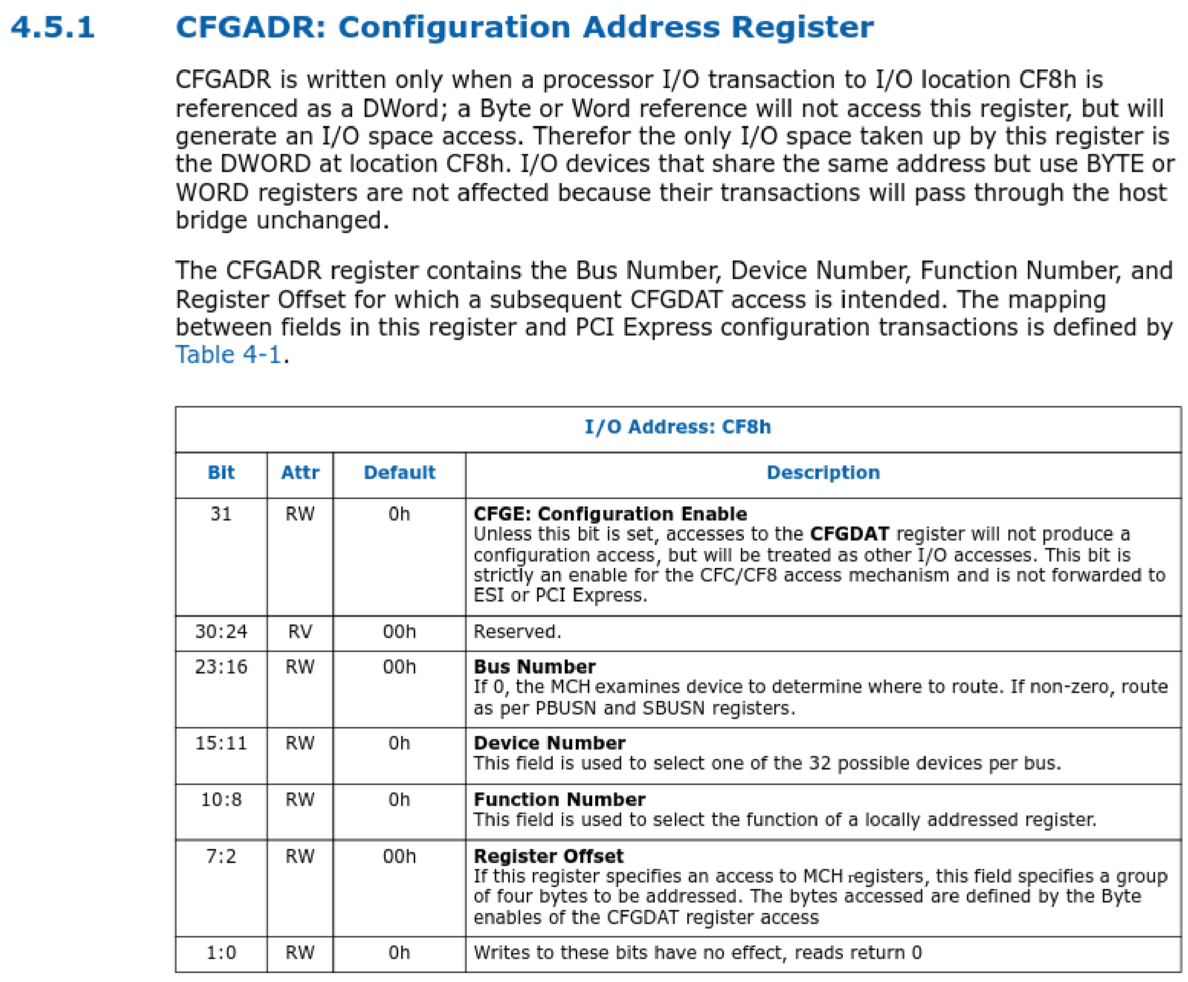
Note: Address locations that are not shown should be treated as Reserved.

Table 13-1. LPC Interface PCI Register Address Map (LPC I/F—D31:F0) (Sheet 1 of 2)

Offset	Mnemonic	Register Name	Default	Type
00h–01h	VID	Vendor Identification	8086h	RO
02h–03h	DID	Device Identification	See register description	RO
04h–05h	PCICMD	PCI Command	0007h	R/W, RO
06h–07h	PCISTS	PCI Status	0210h	R/WC, RO
DCh	BIOS_CNTL	BIOS Control	00h	R/WLO, R/W, RO
E0h-E1h	FDCAP	Feature Detection Capability ID	0009h	RO
E2h	FDLEN	Feature Detection Capability Length	0Ch	RO
E3h	FDVER	Feature Detection Version	10h	RO
E4h-EBh	FDVCT	Feature Vector	See Description	RO
F0h-F3h	RCBA	Root Complex Base Address	00000000h	R/W

As shown above the *RCBA* which points to *Root Complex Register Block (RCRB)* is at offset *0xF0*. In AMD systems, as mentioned earlier the redirections, peripheral names and register names along with their portions will be different. We happen to analyze an AMD laptop from 2014-2016 which had an experimental APU instead of CPU+GPU. On those systems the name Fusion Controller Hub (FCH) was used. It did not have the Root Complex base address in LPC block, instead it was directly pointing towards the SPI base address (register named *SPI_BASE*). It is just an example to compare to the current one, for understanding how different the motherboard architectures can be.

Now that we know the LPC device address we have to understand how we can access it with combination of PCI Configuration Address/Data registers. For that we have to consult the MCH manual:



4.5.1 CFGADR: Configuration Address Register

CFGADR is written only when a processor I/O transaction to I/O location CF8h is referenced as a DWord; a Byte or Word reference will not access this register, but will generate an I/O space access. Therefor the only I/O space taken up by this register is the DWORD at location CF8h. I/O devices that share the same address but use BYTE or WORD registers are not affected because their transactions will pass through the host bridge unchanged.

The CFGADR register contains the Bus Number, Device Number, Function Number, and Register Offset for which a subsequent CFGDAT access is intended. The mapping between fields in this register and PCI Express configuration transactions is defined by Table 4-1.

I/O Address: CF8h			
Bit	**Attr**	**Default**	**Description**
31	RW	0h	**CFGE: Configuration Enable** Unless this bit is set, accesses to the **CFGDAT** register will not produce a configuration access, but will be treated as other I/O accesses. This bit is strictly an enable for the CFC/CF8 access mechanism and is not forwarded to ESI or PCI Express.
30:24	RV	00h	Reserved.
23:16	RW	00h	**Bus Number** If 0, the MCH examines device to determine where to route. If non-zero, route as per PBUSN and SBUSN registers.
15:11	RW	0h	**Device Number** This field is used to select one of the 32 possible devices per bus.
10:8	RW	0h	**Function Number** This field is used to select the function of a locally addressed register.
7:2	RW	00h	**Register Offset** If this register specifies an access to MCH registers, this field specifies a group of four bytes to be addressed. The bytes accessed are defined by the Byte enables of the CFGDAT register access
1:0	RW	0h	Writes to these bits have no effect, reads return 0

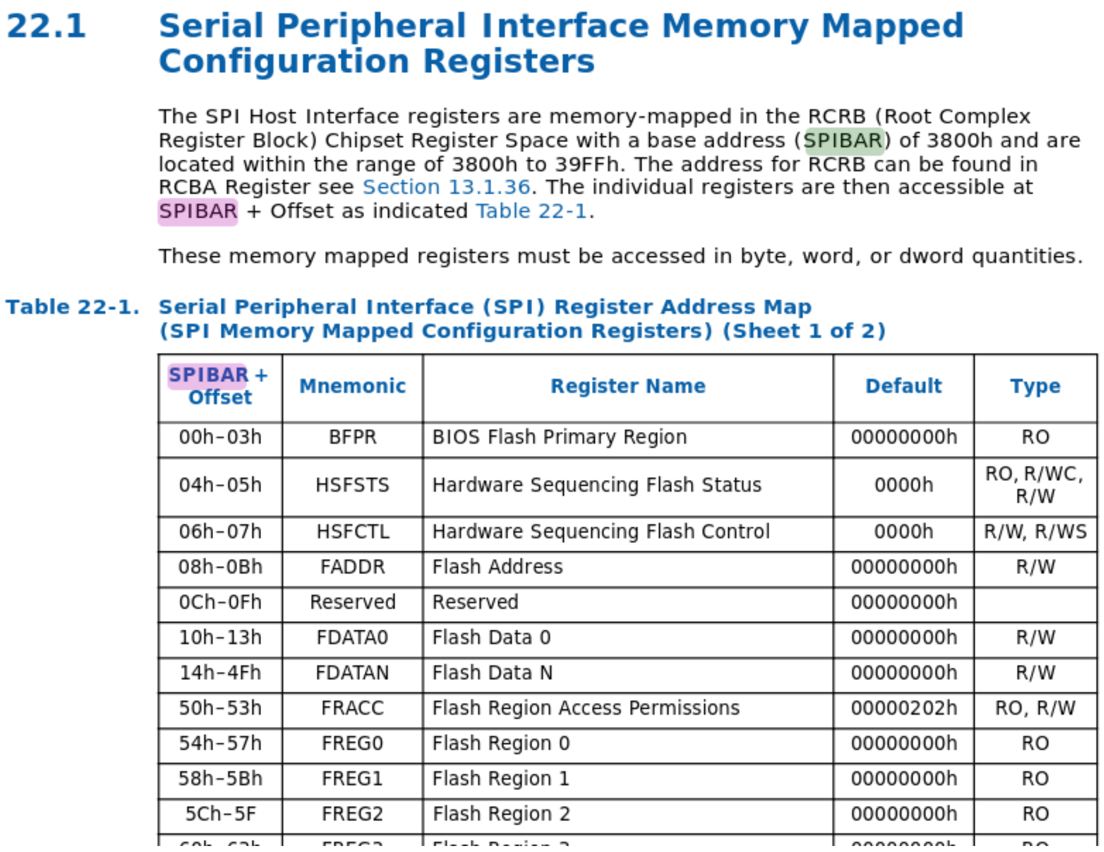
There is the table, describing individual portions of that register. We can use it to put the LPC device address with regitser offset at *RCBA (0xF0)*. Meaning, the *CFGADR* register should contain **1.0.0.1f.0.f0.0** (note that each portion is separated by a dot sign) respectively.

The corresponding data can be read from the *CFGDAT (0xCFC)* register. In summary, to tell the CPU that we want to access the LPC device we have to first put its address in *CFGADR (0xCF8)* by using the **OUT** command (C equivalent would be calling the *outl(req, 0xCF8)* function from *sys/io.h* header file) then to retrieve the data we would use the **IN** instruction with *CFGDAT (0xCFC)* port (In C, again, use the *val = inl(0xCFC) function*).

The obtained value should contain the value inside the LPC RCBA register. This physical base address can be mapped via *mmap* to make it accessible from the user program. In Linux this is done via the */dev/mem* (note, if the device node does not exist then you can make one either with *mknod* system function or the *mknod* command, to see the device of physical memory, one needs to dump the */proc/devices* file and look for *mem*).

Once the RCRB got mapped, we can move on to the next step. In this step we should find out the value of the *SPI Base Address (SPIBAR)*. This offset value will shows where the SPI registers are located. The offset needs to be added to the newly mapped RCRB.

The ICH10 reference manual, tells us that the SPIBAR is at offset value 0x3800.



22.1 Serial Peripheral Interface Memory Mapped Configuration Registers

The SPI Host Interface registers are memory-mapped in the RCRB (Root Complex Register Block) Chipset Register Space with a base address (SPIBAR) of 3800h and are located within the range of 3800h to 39FFh. The address for RCRB can be found in RCBA Register see Section 13.1.36. The individual registers are then accessible at SPIBAR + Offset as indicated Table 22-1.

These memory mapped registers must be accessed in byte, word, or dword quantities.

Table 22-1. Serial Peripheral Interface (SPI) Register Address Map (SPI Memory Mapped Configuration Registers) (Sheet 1 of 2)

SPIBAR + Offset	Mnemonic	Register Name	Default	Type
00h–03h	BFPR	BIOS Flash Primary Region	00000000h	RO
04h–05h	HSFSTS	Hardware Sequencing Flash Status	0000h	RO, R/WC, R/W
06h–07h	HSFCTL	Hardware Sequencing Flash Control	0000h	R/W, R/WS
08h–0Bh	FADDR	Flash Address	00000000h	R/W
0Ch–0Fh	Reserved	Reserved	00000000h	
10h–13h	FDATA0	Flash Data 0	00000000h	R/W
14h–4Fh	FDATAN	Flash Data N	00000000h	R/W
50h–53h	FRACC	Flash Region Access Permissions	00000202h	RO, R/W
54h–57h	FREG0	Flash Region 0	00000000h	RO
58h–5Bh	FREG1	Flash Region 1	00000000h	RO
5Ch–5F	FREG2	Flash Region 2	00000000h	RO

Starting from circa 2009, most of the flash data is very well structured. This mode is often called 'Descriptor Mode'. There are numerous ways of identifying whether a flash data is in descriptor mode or not. One can look inside the SPI region register values, but the most reliable way is finding the magic bytes **0x0FF0A55A**. They appear at the signature region of a BIOS, which is located at the end. The sample BIOS we use happen to be an older one. For a non-descriptor mode BIOS the ICH10 falls backup into compatibility mode with ICH7.

The only part left, is to use the Flash registers for reading or writing into allowed regions. The **FADDR** can be used to specify the address, where the **FDATAx** registers store the actual data. The read or write mode is determined by the **HSFC.FCYCLE** register. Operation progress status can be retrieved from the **HSFS** register.

It is also worth mentioning that there are some portions which can not be accessed from any component except from SMM.

References

[1] Intel, Intel® 64 and IA-32 Architectures Software Developer Manuals.

[2] Intel, Intel® X58 Express Chipset.

[3] Intel, Intel® Memory Controller Hub.

[4] Intel, Intel® I/O Controller Hub 10 (ICH10) Family.

[5] Intel, Intel® I/O Controller Hub 7 (ICH7) Family.

[6] Intel, The 8086 Family Users Manual.