Advanced x86: BIOS and System Management Mode Internals

*SPI Flash Programming*

Xeno Kovah & Corey Kallenberg

LegbaCore, LLC
All materials are licensed under a Creative Commons “Share Alike” license.  
http://creativecommons.org/licenses/by-sa/3.0/

Attribution condition: You must indicate that derivative work
“Is derived from John Butterworth & Xeno Kovah’s ‘Advanced Intel x86: BIOS and SMM’ class posted at http://opensecuritytraining.info/IntroBIOS.html”
SPI note:

• We’re not really going to care about low level SPI protocol details
• We’re just going to care about the way that it’s exposed to the BIOS, so that we can understand the BIOS’s view of the world, and therefore interpret its actions accordingly
SPI (Serial Peripheral Interface) Flash

- Intel provides a programmable interface to the SPI flash device
  - System BIOS lives here
  - Other stuff does too
- Copernicus programs this interface to dump a binary of the SPI flash
How you should think of reading & writing the flash chip

All you see/need to care about:

- CPU
- RAM
  ("SPIBAR" = RCBB + X MMIO to ICH or PCH)
- ICH or PCH
- Flash Chip

Behind the scenes:

- SPI
Programming the SPI Flash

• *RCBA holds RCRB*
• RCRB + X = SPIBAR
  — (for X = 0x3800 on all newer systems, but not old ones)
• SPIBAR + Y = flash programming registers
• An app can choose either “Hardware Sequencing” (meaning the hardware picks the actual SPI commands that get sent for read/write) or Software Sequencing (meaning you pick the actual SPI commands)
  — For simplicity of discussion, we’ll be referring to only those operations/details pertaining to Hardware Sequencing
    • Software Sequencing just offers a little more fine-grain control

All SPI registers in the following slides are from:

You’re welcome!
Bwahahaha!
Since I didn’t have anywhere better to put this...

21.1 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 are in the RCBA Register (see Section 12.1.40). The individual registers are then accessible at SPIBAR + Offset as indicated in the following table.

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

Table 21-1. Serial Peripheral Interface (SPI) Register Address Map (SPI Memory Mapped Configuration Registers) (Sheet 1 of 2)

<table>
<thead>
<tr>
<th>SPIBAR + Offset</th>
<th>Mnemonic</th>
<th>Register Name</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h–03h</td>
<td>BFPR</td>
<td>BIOS Flash Primary Region</td>
<td>00000000h</td>
</tr>
<tr>
<td>04h–05h</td>
<td>HSFS</td>
<td>Hardware Sequencing Flash Status</td>
<td>0000h</td>
</tr>
<tr>
<td>06h–07h</td>
<td>HSFC</td>
<td>Hardware Sequencing Flash Control</td>
<td>0000h</td>
</tr>
<tr>
<td>08h–0Bh</td>
<td>FADDR</td>
<td>Flash Address</td>
<td>00000000h</td>
</tr>
<tr>
<td>0Ch–0Fh</td>
<td>—</td>
<td>Reserved</td>
<td>00000000h</td>
</tr>
<tr>
<td>10h–13h</td>
<td>FDATA0</td>
<td>Flash Data 0</td>
<td>00000000h</td>
</tr>
<tr>
<td>14h–4Fh</td>
<td>FDATAN</td>
<td>Flash Data N</td>
<td>00000000h</td>
</tr>
</tbody>
</table>
SPI Programming Flash Address Register

• Specifies starting address of the SPI I/O cycle
  – Flash address, not a system RAM address
  – Valid range is 0 to <size of flash chip – 1>

FADDR—Flash Address Register
(SPI Memory Mapped Configuration Registers)

<table>
<thead>
<tr>
<th>Memory Address:</th>
<th>Attribute:</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIBAR + 08h</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Default Value:</th>
<th>Size:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000h</td>
<td>32 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:25</td>
<td>Reserved</td>
</tr>
<tr>
<td>24:0</td>
<td>Flash Linear Address (FLA) — R/W. The FLA is the starting byte linear address of a SPI Read or Write cycle or an address within a Block for the Block Erase command. The Flash Linear Address must fall within a region for which BIOS has access permissions. Hardware must convert the FLA into a Flash Physical Address (FPA) before running this cycle on the SPI bus.</td>
</tr>
</tbody>
</table>
SPI Programming Data Registers

- Contains the data we just read from the SPI flash (up to 64 bytes), or data we’re about to write to the flash chip
- R/W (since it can be used to specify data to write to flash)
SPI Programming Control Register

**HSFC—Hardware Sequencing Flash Control Register**  
(SPI Memory Mapped Configuration Registers)

- **Memory Address:** SPIBAR + 06h  
- **Default Value:** 0000h  
- **Attribute:** R/W, R/WS  
- **Size:** 16 bits

<table>
<thead>
<tr>
<th>2:1</th>
<th>FLASH Cycle (FCYCLE) — R/W. This field defines the Flash SPI cycle type generated to the FLASH when the FGO bit is set as defined below:</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Read (1 up to 4 bytes by setting FDBC)</td>
</tr>
<tr>
<td>01</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Write (1 up to 4 bytes by setting FDBC)</td>
</tr>
<tr>
<td>11</td>
<td>Block Erase</td>
</tr>
</tbody>
</table>

- Set the type to read (bits 2:1 == 00b)
**SPI Programming Control Register**

- Initiates the SPI I/O cycle
  - Used by programming app (Copernicus)
- Defines the number of bits to read (or write) in the I/O cycle

**HSFC—Hardware Sequencing Flash Control Register**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Address</td>
<td>SPIBAR + 06h</td>
</tr>
<tr>
<td>Default Value</td>
<td>0000h</td>
</tr>
<tr>
<td>Attribute</td>
<td>R/W, R/WS</td>
</tr>
<tr>
<td>Size</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

- **Flash Data Byte Count (FDBC)** — R/W. This field specifies the number of bytes to shift in or out during the data portion of the SPI cycle. The contents of this register are 0s based with 0b representing 1 byte and 111111b representing 64 bytes. The number of bytes transferred is the value of this field plus 1. This field is ignored for the Block Erase command.

- **Flash Cycle Go (FGO)** — R/W/S. A write to this register with a 1 in this bit initiates a request to the Flash SPI Arbiter to start a cycle. This register is cleared by hardware when the cycle is granted by the SPI arbiter to run the cycle on the SPI bus. When the cycle is complete, the FDONE bit is set. Software is forbidden to write to any register in the HSFLCTL register between the FGO bit getting set and the FDONE bit being cleared. Any attempt to violate this rule will be ignored by hardware. Hardware allows other bits in this register to be programmed for the same transaction when writing this bit to 1. This saves an additional memory write. This bit always returns 0 on reads.
SPI Programming Status Register

- Indicates that an SPI I/O cycle is in progress
- Set automatically by hardware, nothing we need to really care much about

**HSFS—Hardware Sequencing Flash Status Register (SPI Memory Mapped Configuration Registers)**

<table>
<thead>
<tr>
<th>Memory Address:</th>
<th>SPIBAR + 04h</th>
<th>Attribute:</th>
<th>RO, R/WC, R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value:</td>
<td>0000h</td>
<td>Size:</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

**SPI Cycle In Progress (SCIP)—** RO. Hardware sets this bit when software sets the Flash Cycle Go (FGO) bit in the Hardware Sequencing Flash Control register. This bit remains set until the cycle completes on the SPI interface. Hardware automatically sets and clears this bit so that software can determine when read data is valid and/or when it is safe to begin programming the next command. Software must only program the next command when this bit is 0.

**NOTE:** This field is only applicable when in Descriptor mode and Hardware sequencing is being used.
SPI Programming Status Register 2

- Indicates the SPI I/O cycle has completed
- Software must poll on this bit to determine when the hardware is done reading/writing data

**HSFS—Hardware Sequencing Flash Status Register (SPI Memory Mapped Configuration Registers)**

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Attribute</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIBAR + 04h</td>
<td>RO, R/WC, R/W</td>
<td>16 bits</td>
</tr>
<tr>
<td>Default Value</td>
<td>0000h</td>
<td></td>
</tr>
</tbody>
</table>

| 0 | Flash Cycle Done (FDONE) — R/W/C. The ICH sets this bit to 1 when the SPI Cycle completes after software previously set the FGO bit. This bit remains asserted until cleared by software writing a 1 or hardware reset due to a global reset or host partition reset in an Intel ME enabled system. When this bit is set and the SPI SMI# Enable bit is set, an internal signal is asserted to the SMI# generation block. Software must make sure this bit is cleared prior to enabling the SPI SMI# assertion for a new programmed access. |

**NOTE:** This field is only applicable when in Descriptor mode and Hardware sequencing is being used.
Reading the flash chip

- BIOS reading software sets up the location it wants to read (as part of reading the entire chip) and how many bytes to read
Reading the flash chip

- BIOS reading software says to start the read

FADDR=0x1000
FDATA0=00000…
FDONE=0
SCIP=0
FCYCLE=0
FGO=0

Ring 0

HSFC |= 1 (aka FGO=1)
Reading the flash chip

- Cycle in progress
- We need to poll on FDONE, waiting for it to be set to 1

FADDR=0x1000
FDATA0=00000…

FDONE=0
SCIP=1
FCYCLE=0
FGO=1
Reading the flash chip

• Once the cycle is done (FDONE=1) we can do read the FDATA registers

FADDR=0x1000
FDATA0=0xf005ba11

FDONE=1
SCIP=0
FCYCLE=0
FGO=0
Reading the flash chip

- BIOS reading software will get the contents out of the FDATA register(s) and store to memory and/or disk
Lab: RYOFCC!
(Read Your Own Flash Chip!)

• We’re going to be cool and manually program the flash registers in RWE to read the reset vector from our flash chip
• DISCLAIMER! DISCLAIMER! DISCLAIMER!
• Don’t follow along unless you’re super careful! You could accidentally brick your system! :D
• I am not responsible if you brick your system (or your enemies’ :P)
• DISCLAIMER! DISCLAIMER! DISCLAIMER!

I TOO LIKE TO LIVE
DANGEROUSLY
Register Access: SPI Base Address Register (SPIBAR)

- SPI extends its Base Address Registers (BARs) to memory starting at an address called SPIBAR.
- SPIBAR is offset from the Root Complex Register Block (RCRB).

<table>
<thead>
<tr>
<th>Offset</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0h</td>
<td>RCBA</td>
<td>FED18000</td>
</tr>
</tbody>
</table>

- The offset from RCRB is chipset-dependent, but will be listed in the SPI section of the datasheet.
Lab: Locate SPIBAR

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 13.1.35. The individual registers are then accessible at SPIBAR + Offset as indicated in Table 22-1.

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

- Per your datasheet, the SPI host interface registers (SPIBAR) is located at an offset from RCRB.
- In the Mobile 4-Series chipset on our lab laptop, SPIBAR is located at:
- $\text{RCRB} + 3800h = \text{FED1}_h \_\_B800h$
Back to “Fun things to do in SMM”!
Attack 1 – Manipulate Copernicus output

- From within the OS, targeted hooks into Copernicus code
- From within the OS with “DDefy” [20] rootkit style hooks into file writing routines
- From within the HD controller firmware [21][22][23]
- From within the OS with a network packet filter driver
- From within the NIC firmware [24][25]
- Etc. Lots more options
Attack 2 – A new generic attack.

- It is possible for SMM to be notified when SPI reads or writes occur.
- An attacker who controls the BIOS controls the setup of SMM.
- In this way a BIOS-infecting attacker can perform a SMM MitM attack against those who would try to read the BIOS to integrity check it.
- We call our SMM MitM “Smite’em, the Stealthy”.
Eye of the dragon - FSMIE - hw sequencing

- This is what allows an attacker in SMM to know when someone is trying to access the flash chip

**HSFC—Hardware Sequencing Flash Control Register (SPI Memory Mapped Configuration Registers)**

<table>
<thead>
<tr>
<th>Memory Address: SPIBAR + 06h</th>
<th>Attribute:</th>
<th>R/W, R/WS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value: 0000h</td>
<td>Size:</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

This register is only applicable when SPI device is in descriptor mode.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td><strong>Flash SPI SMI# Enable (FSMIE)</strong> — R/W. When set to 1, the SPI asserts an SMI# request whenever the Flash Cycle Done bit is 1.</td>
</tr>
</tbody>
</table>

- The Flash Cycle Done bit is set to 1 after every read and write
Reading the flash chip in the presence of Smite'em

- BIOS reading software sets up the location it wants to read (as part of reading the entire chip) and how many bytes to read

FADDR = 0x1000, FCYCLE = 00 (read)
FDBC = 111111b (64 byte read)

FADDR=00000…
FDATA0=00000…
FDONE=0
SCIP=0
FCYCLE=0
FGO=0
FSMIE=1 (already set)
Reading the flash chip in the presence of Smite'em

- BIOS reading software says to start the read

FADDR=0x1000
FDATA0=00000…
FDONE=0
SCIP=0
FCYCLE=0
FGO=0
FSMIE=1
Reading the flash chip in the presence of Smite'em

- Cycle in progress

FADDR=0x1000
FDATA0=00000…
FDONE=0
SCIP=1
FCYCLE=0
FGO=1
FSMIE=1
Reading the flash chip in the presence of Smite'em

- Once the cycle is done, and the data is available for reading, if the FSMIE = 1, an SMI is triggered, giving Smite'em the first look

FADDR=0x1000
FDATA0=0x1badd00d

FDONE=1
SCIP=0
FCYCLE=0
FGO=0
FSMIE=1
Reading the flash chip in the presence of Smite'em

- Smite'em can change any data that would reveal its presence to the original benign data

FADDR=0x1000
FDATA0=0x1badd00d
FDONE=1
SCIP=0
FCYCLE=0
FGO=0
FSMIE=1

FADDR=0xf005ba11
Reading the flash chip in the presence of Smite'em

- BIOS reading software will be misled!
Think it can't happen?

- **Flashrom 0.9.7 source**

  ```c
  ichspi.c:    hsfc = REGREAD16(I9H9_REG_HSFC);
  ichspi.c:    hsfc &= ~HSFC_FCYCLE; /* set read operation */
  ichspi.c:    hsfc &= ~HSFC_FDBC; /* clear byte count */
  ichspi.c:    hsfc |= (((block_len - 1) << HSFC_FDBC_OFF) & HSFC_FDBC);
  ichspi.c:    hsfc |= HSFC_FGO; /* start */
  ```

- If you don't account for hw/sw sequencing's FSMIE bit (as no previous software did), you will just lose and provide false assurances of a lack of BIOS compromise
What you don't know can bite you

- The basic solution would seem to be just for querying tools to set FSMIE = 0 before trying to read

- Multiple ways for an adversary to counter
  - Kernel agent continuously setting FSMIE = 1
    - So you just clear it and check if it's getting re-set, and if so...?
  - VMX interception of MMIO to SPI space, falsifying that you successfully cleared FSMIE
    - But then if they're using VMX too, they can also just directly forge FDATA
  - Target your security software specifically
    - If your tool is good enough to detect attacker, he's incentivized to go after you specifically
Terror at 35,000 feet (high level overview)
Another attack…

- Let's assume that Smite'em wants to pick another generic, low-effort way to avoid detection (i.e. doesn't want to use VMX until absolutely necessary)
- Smite'em recruits a *Dragon Knight* avatar
  - Could be kernel-based code or a DMA device and independent of CPU
- Avatar polls SPI configuration registers to detect if an SPI cycle is in progress
- Upon detecting an SPI cycle in progress, the avatar triggers an SMI
- Smite’em running in SMM has exclusive access to the CPU, and can stall until the cycle completes and then replace the data read from flash before Copernicus can read it
Reading the flash chip in the presence of Smite'em

- Copernicus sets up the location it wants to read (as part of reading the entire chip) and how many bytes to read

FADDR = 0x1000, FCYCLE = 0(read)
FDBC = 111111b (64 byte read)

FADDR=00000...
FDATA0=00000...
FDONE=0
SCIP=0
FCYCLE=0
FGO=0
FSMIE=0
Reading the flash chip in the presence of Smite'em

- Then says go

SMM

FADDR=0x1000
FDATA0=00000…

FDONE=0
SCIP=0
FCYCLE=0
FGO=0
FSMIE=0

Ring 0
Reading the flash chip in the presence of Smite'em

- Copernicus sets up the location it wants to read (as part of reading the entire chip) and how many

FADDR=0x1000
FDATA0=00000...
FDONE=0
SCIP=1
FCYCLE=0
FGO=1
FSMIE=0

Poll for SCIP=1 and once found poll faster for FDONE=1
Reading the flash chip in the presence of Smite'em

- Once it sees the data, it tries not to race with Copernicus, but instead stops itself and Copernicus by signaling Smite'em with an SMI
Reading the flash chip in the presence of Smite'em

- Smite'em then cleans up as usual

FADDR=0x1000
FDATA0=0x1badd00d
FDONE=1
SCIP=0
FCYCLE=0
FGO=0
FSMIE=0
Better luck next time?

• We then implemented a basic “Copernicus 2” that used Intel TXT to try and defeat this
  – We care about *trustworthy* detection tools. It’s no good to have detection tools if your attacker can easily bypass them and give you a *false sense of trust* that your system is clean
    • But we know that we have to start somewhere
• We thought it would work because the manuals implied that TXT automatically suppresses SMIs until they’re explicitly turned back on
  – It turned out that it does on *old* hardware, but newer hardware turns on SMIs automatically, so our TXT code is back to being vulnerable to SMM subversion until STMs are available
    • That’s part of why we want STMs so much
  – :'(  
  – TT
• Our only consolation is that Copernicus 2 *does* take OS level attackers off the table...But really if we’re trying to detect BIOS level attackers, we need to counter SMM in order for it to matter
Coming soon?(ever?)

• “Copernicus 3” would take our existing research on “timing-based attestation” and port it into TXT-land

• Unfortunately since I said that idea while I was at MITRE, even though I didn’t implement it, probably MITRE owns the idea and I can’t do it without their permission/cooperation :-/

• So for today, your only option is...
Trustworthy BIOS malware detection!

(note: doesn’t measure SMM :))