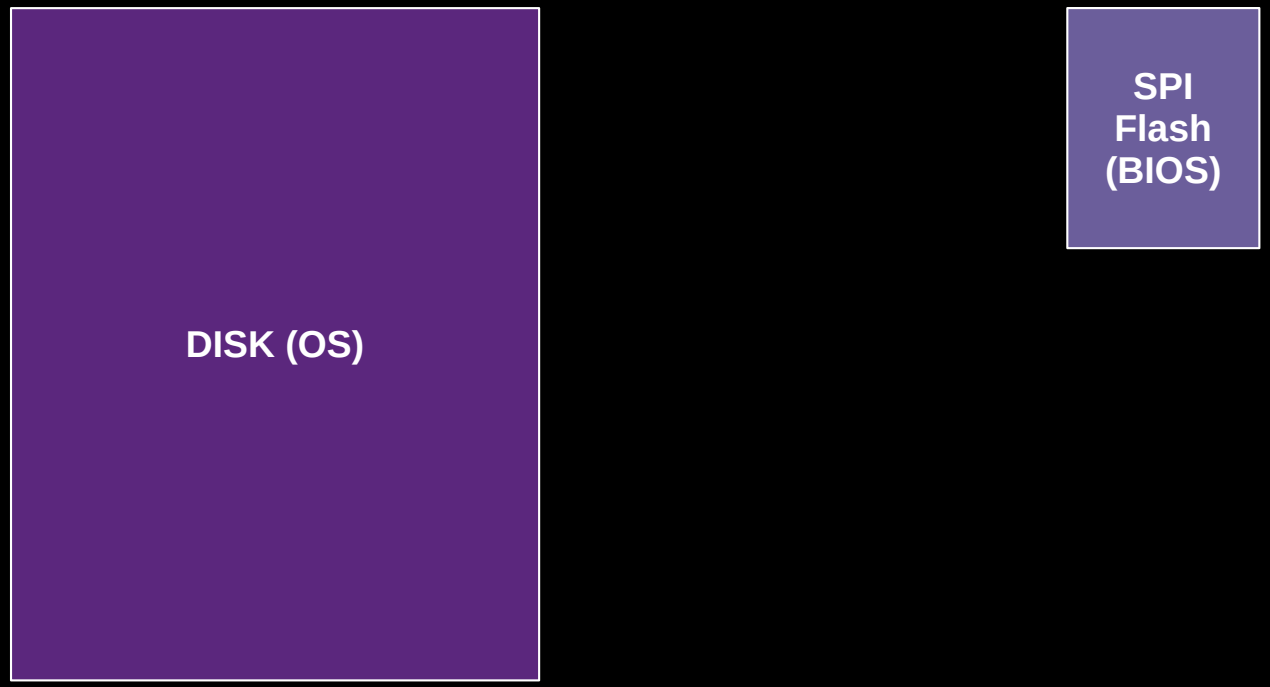
BIOS **Bootkits**

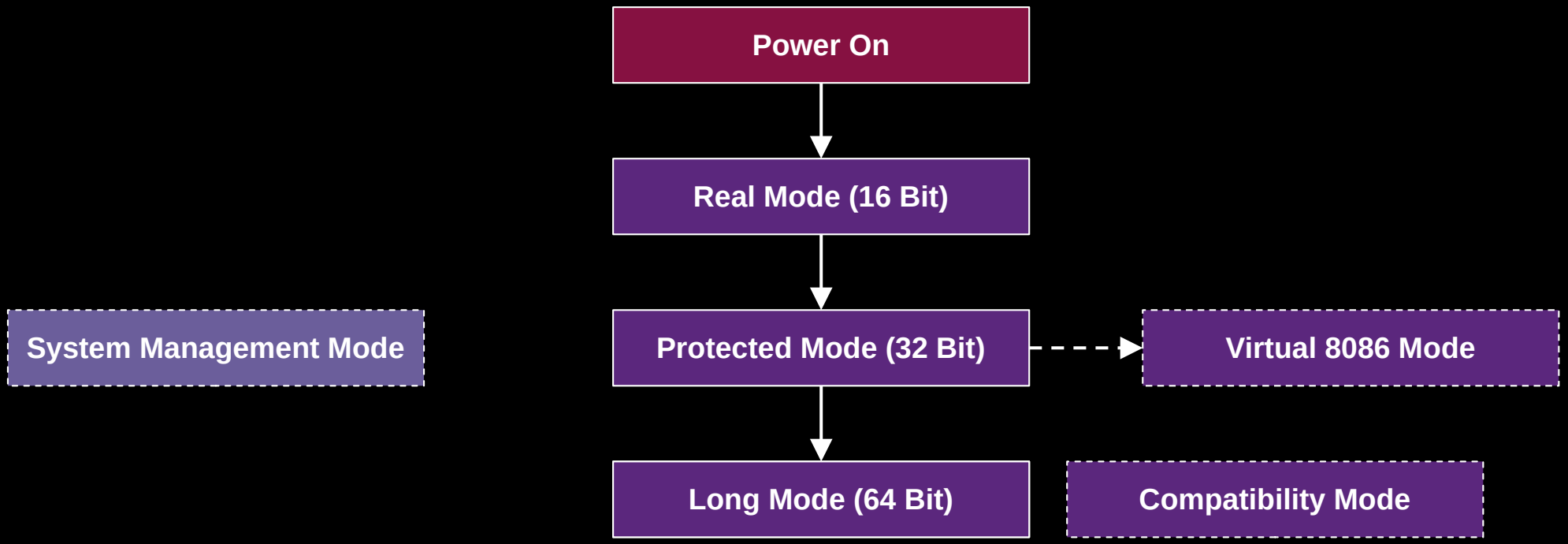
Agenda

- Leveraging **user-space** paths
- Analyzing **motherboards**
- Identifying **peripherals**
- **PCI**, **LPC**, **SPI** Communication
- Reading from the SPI **Flash**
- The **Reset Vector** address
- **UEFI** vs **Legacy** BIOS
- UEFI **Phases**
- Summary

Where is the BIOS

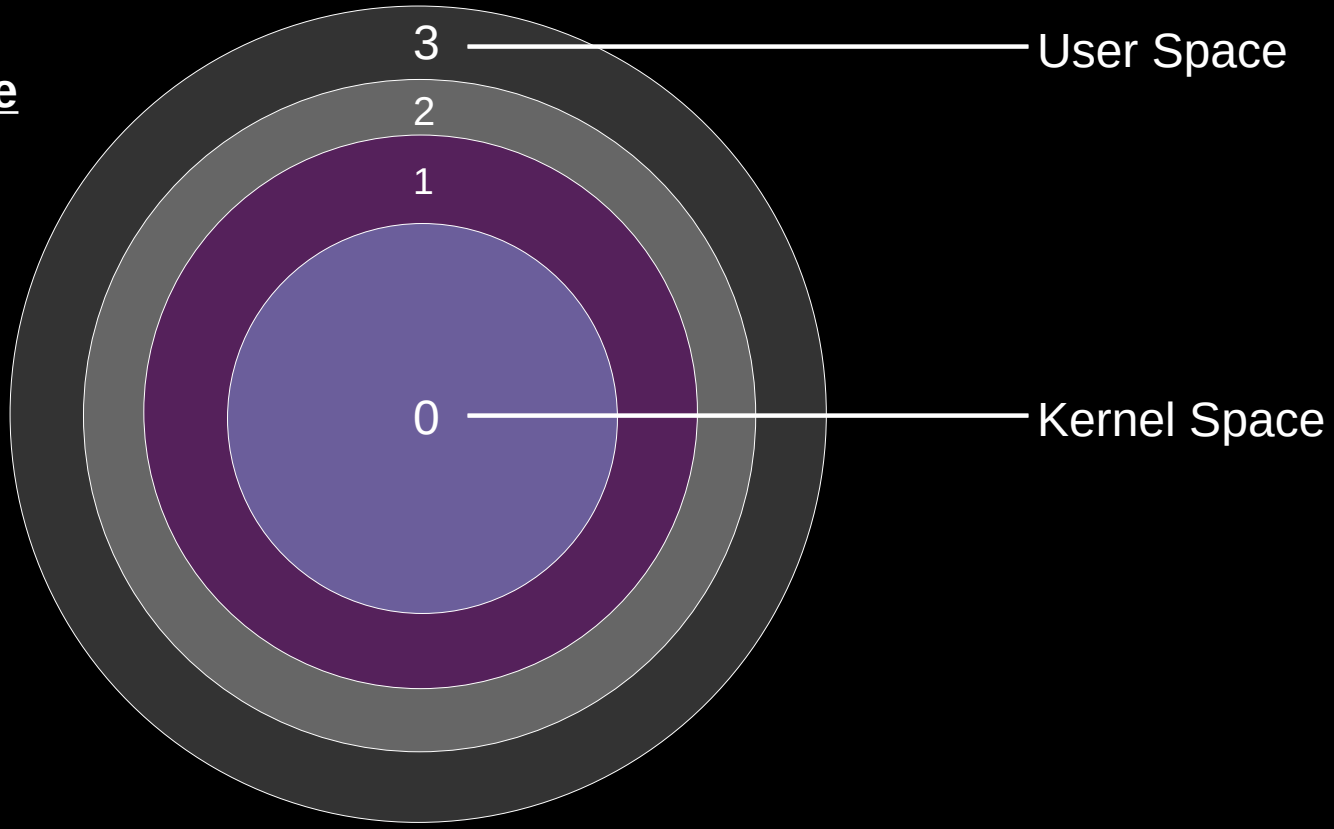


The x86 Modes

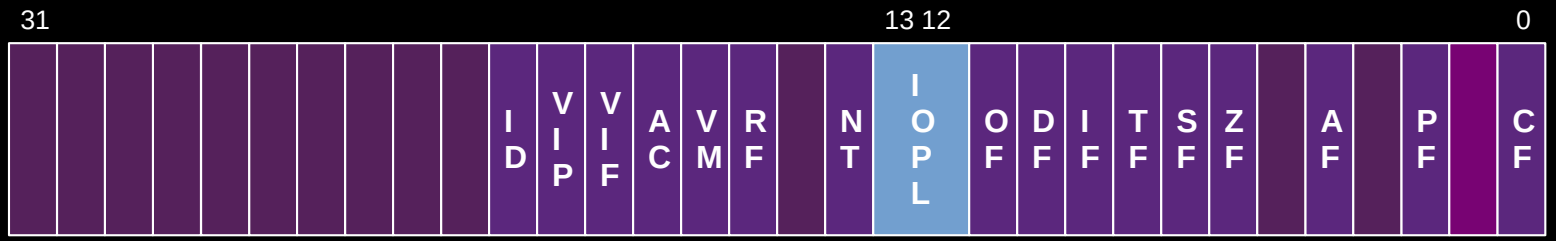


Protection **Rings**

Available from >32bit mode



EFLAGS Register



IOPL (I/O Privilege Level) - Indicates the I/O privilege level of the currently running program or task. The **current privilege level (CPL)** of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space. The POPF and IRET instructions can modify this field only when operating at a CPL of 0.

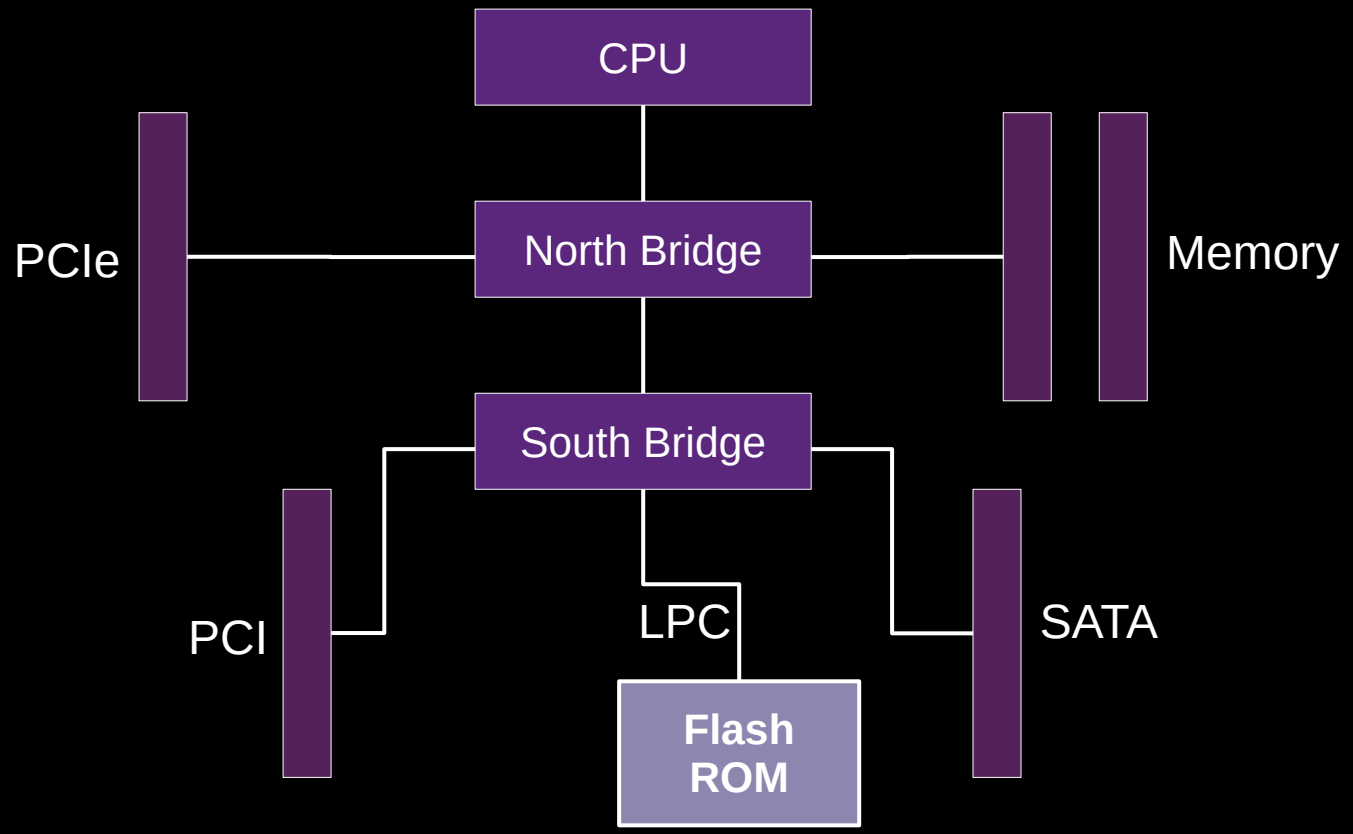
I/O Instructions

```
IF ((PE = 1) and ((CPL > IOPL) or (VM = 1))) THEN
    IF (Any I/O Permission Bit for I/O port being accessed = 1) THEN
        #GP(0);
    ELSE
        DEST := SRC;
    FI;
ELSE (Real Mode or Protected Mode with CPL ≤ IOPL *)
    DEST := SRC;
FI;
```

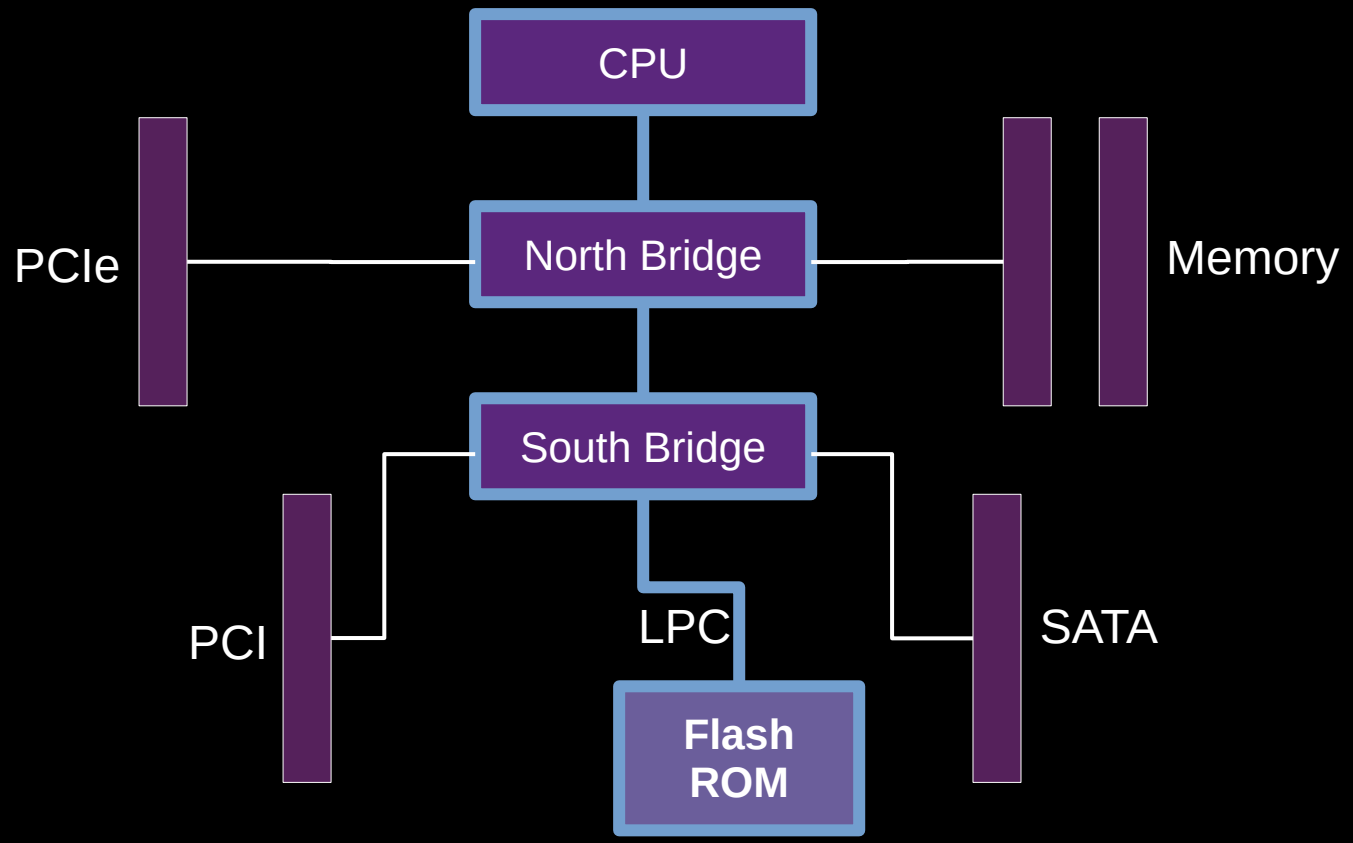
How to set **IOPL** bits

- In Linux, use the **iopl(3)** function
- In Windows, use the **NtSetInformationProcess(...)** function
- ...

The Architecture



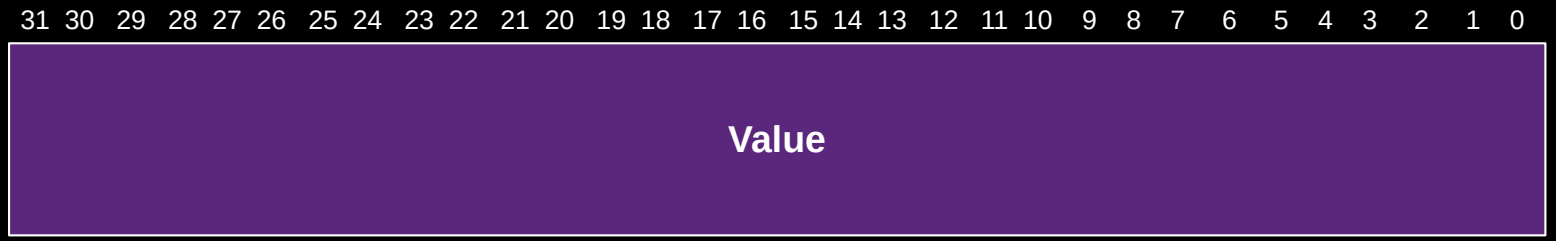
The Architecture



PCI Configuration MMIO Registers

CFGADR
0xCF8

CFGDAT
0xCFC



Example

Device(**0x1**, **0x2**, **0x3**), Register(**0x4**)

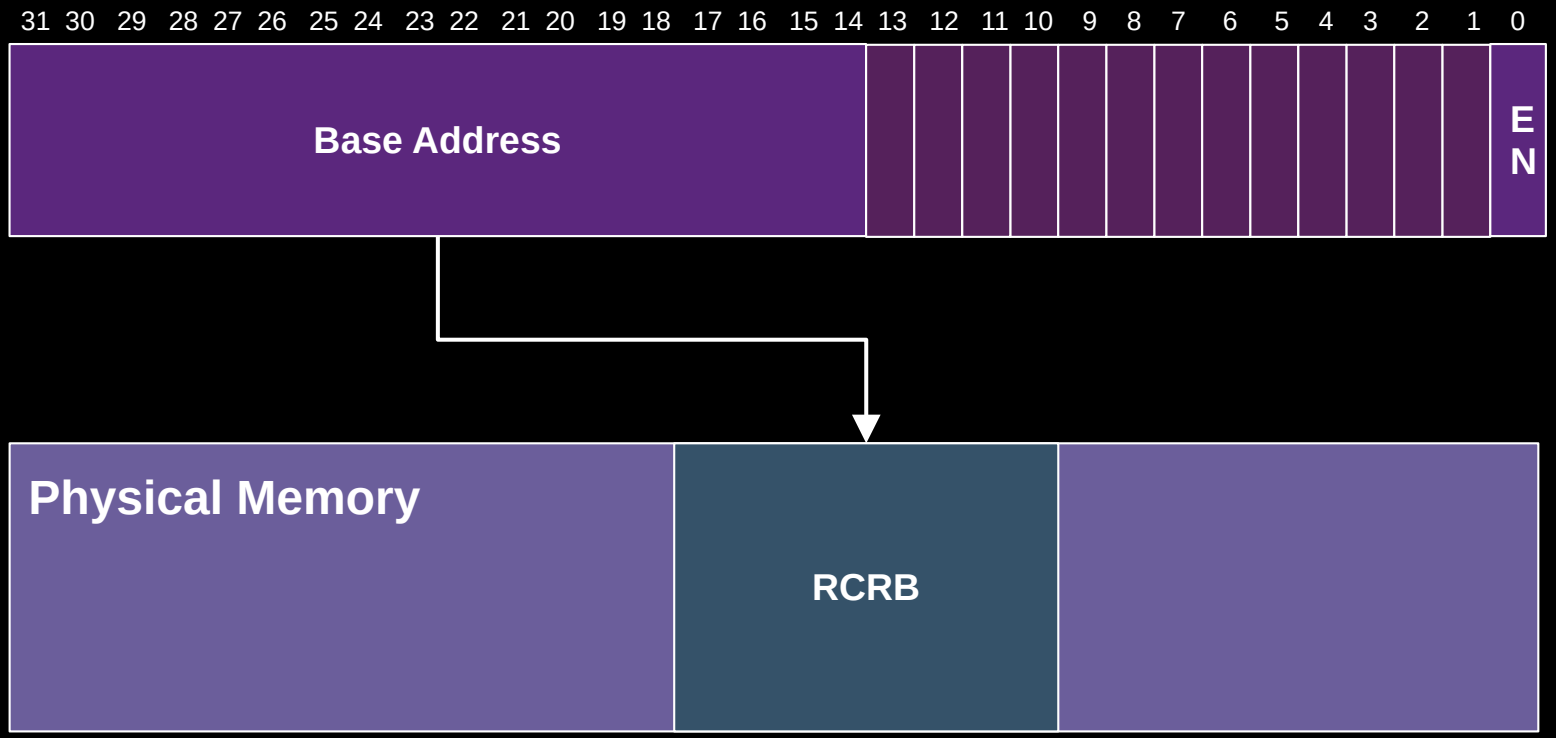
```
#include <sys/io.h>

#define CFGADR      0xCF8
#define CFGDAT      0xCFC

u32 adr;
adr = (1 << 31) | (0x1 << 16) | (0x2 << 11) | (0x3 << 8) | 0x4;

outl(adr, CFGADR);

u32 dat;
dat = inl(CFGDAT);
```

LPC Controller (00.1F.00)

- Is at address **0x00**, **0x1F**, **0x00**
  - **Bus: 0x00**
  - **Device: 0x1F**
  - **Function: 0x00**
- Contains an interesting register called **RCBA (0xF0)**
  - **Offset: 0xF0**

Root Complex Base Address

RCBA (Root Complex Base Address Register) - 0xF0



Root Complex Register Block

- Contains **SPI** Registers (at offset **SPIBAR**)
- Is mapped at some Physical address
  - **/dev/mem** in Linux
  - **mmap** syscall
- Pointed by the **RCBA** which is in LPC Controller
- The size is **16KB**

SPI Base Address

- The offset starting from **RCRB** beginning
- Points to **SPI** register block
- Is a **hard-coded** offset (based on PCH, ICH etc. version)

Finding the SPI Register Block

```
u32 lpc_adr;
lpc_adr = (1 << 31) | (0x00 << 16) | (0x1F << 11) | (0x00 << 8) | 0xF0;

outl(lpc_adr, CFGADR);

u32 rcba;
rcba = inl(CFGDAT);

int mem;
mem = open("/dev/mem", O_RDWR);

void *rcrb;
rcrb = mmap(NULL, 0x4000, PROT_READ | PROT_WRITE, MAP_SHARED, mem, rcba & 0xFFFFC000);

void *spirb;
spirb = ((u8*)rcrb + 0x3800);
```

Finding the SPI Register Block

Read RCBA from the LPC Controller

```
u32 lpc_adr;
lpc_adr = (1 << 31) | (0x00 << 16) | (0x1F << 11) | (0x00 << 8) | 0xF0;

outl(lpc_adr, CFGADR);

u32 rcba;
rcba = inl(CFGDAT);

int mem;
mem = open("/dev/mem", O_RDWR);

void *rcrb;
rcrb = mmap(NULL, 0x4000, PROT_READ | PROT_WRITE, MAP_SHARED, mem, rcba & 0xFFFFC000);

void *spirb;
spirb = ((u8*)rcrb + 0x3800);
```

Finding the SPI Register Block

```
u32 lpc_adr;
lpc_adr = (1 << 31) | (0x00 << 16) | (0x1F << 11) | (0x00 << 8) | 0xF0;

outl(lpc_adr, CFGADR);

u32 rcba;
rcba = inl(CFGDAT);
```

Ask OS to give us a virtual address for that RCBA address

```
int mem;
mem = open("/dev/mem", O_RDWR);

void *rcrb;
rcrb = mmap(NULL, 0x4000, PROT_READ | PROT_WRITE, MAP_SHARED, mem, rcba & 0xFFFFC000);

void *spirb;
spirb = ((u8*)rcrb + 0x3800);
```

Finding the SPI Register Block

```
u32 lpc_adr;
lpc_adr = (1 << 31) | (0x00 << 16) | (0x1F << 11) | (0x00 << 8) | 0xF0;

outl(lpc_adr, CFGADR);

u32 rcba;
rcba = inl(CFGDAT);

int mem;
mem = open("/dev/mem", O_RDWR);

void *rcrb;
rcrb = mmap(NULL, 0x4000, PROT_READ | PROT_WRITE, MAP_SHARED, mem, rcba & 0xFFFFC000);

void *spirb;
spirb = ((u8*)rcrb + 0x3800);
```

Move RCRB by SPIBAR to get the SPI Registers

SPI Registers

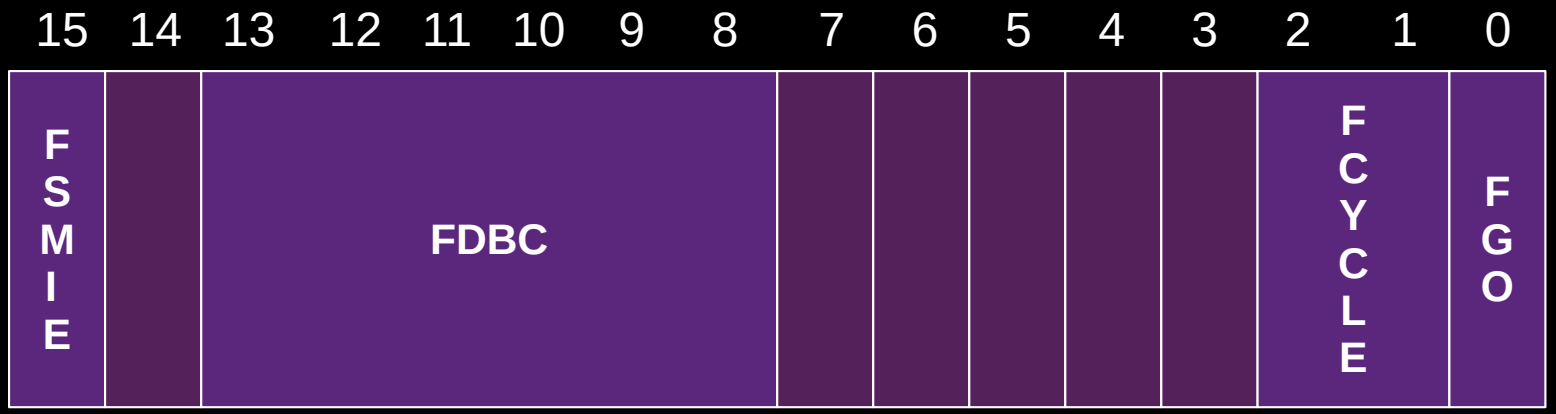
- **HSFSTS:** Hardware Sequencing Flash Status Register
- **HSFCTL:** Hardware Sequencing Flash Control Register
- **FADDR:** Flash Address Register
- **FDATAx:** Flash Data Register **[0..15]**
- **...**

The HSFSTS (0x04) Register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FLOCKDN	FDV	FDOPSS								SCIP	BERASE		AEL	FCERR	FDONE

- **SCIP:** SPI Cycle In Progress. Set by the HW. Can be used to check the status
- **AEL:** Access Error Log. Needs to be cleared. Can be used to check errors
- **FCERR:** Flash Cycle Error. Needs to be cleared. Can be used to check errors
- **FDONE:** Flash Cycle Done. Tells whether the operation is done or not
- ...

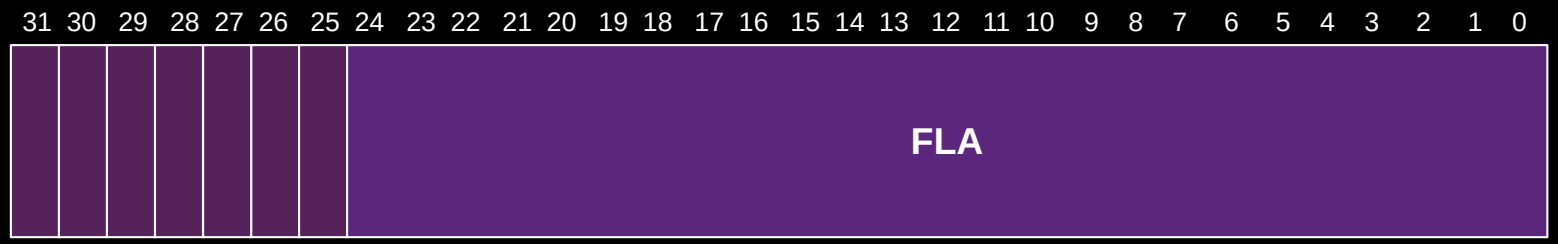
The HSFCTL (0x06) Register



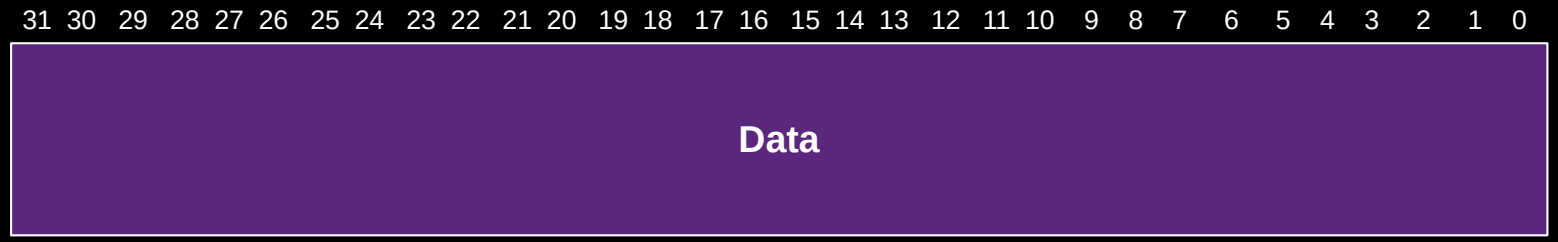
- **FDBC:** Block count. The value is zero based.
- **FCYCLE:** **0b00** is read and **0b10** is write
- **FGO:** Start the operation
- ...

The FADDR/FDATAx Registers

FADDR
0x08



FDATAx
0x10+



- **FLA:** Flash Linear Address
- ...

Reading from the **Flash ROM**

```
void *spirb;
spirb = ((u8*)rcrb + 0x3800);

for (u32 addr = 0; addr <= 0x1000; addr += 64) {
        u16 hsfsts;
        hsfsts = *(u16*)((u8*)spirb + 0x04);
        // clear FCERR, AEL, FDONE bits
        hsfsts &= ~((1 << 2) | (1 << 1) | (1 << 0));
        *(u16*)((u8*)spirb + 0x04) = hsfsts;

        u32 faddr;
        faddr = *(u32*)((u8*)spirb + 0x08);
        faddr &= 0xFE000000;
        faddr |=   addr;
        *(u32*)((u8*)spirb + 0x08) = faddr;

        u16 hsfctl;
        hsfctl = *(u16*)((u8*)spirb + 0x06);
        hsfctl &= ~(0b11 << 1);  // READ request
        hsfctl |=   ((64 – 1) << 8) | (1 << 0); // FGO, FDBC
        *(u16*)((u8*)spirb + 0x06) = hsfctl;

        // print FDATAx values
}
```

Reading from the **Flash ROM**

```
void *spirb;
spirb = ((u8*)rcrb + 0x3800);

for (u32 addr = 0; addr <= 0x1000; addr += 64) {
        u16 hsfsts;
        hsfsts = *(u16*)((u8*)spirb + 0x04);
        // clear FCERR, AEL, FDONE bits
        hsfsts &= ~((1 << 2) | (1 << 1) | (1 << 0));
        *(u16*)((u8*)spirb + 0x04) = hsfsts;

        u32 faddr;
        faddr = *(u32*)((u8*)spirb + 0x08);
        faddr &= 0xFE000000;
        faddr |=   addr;
        *(u32*)((u8*)spirb + 0x08) = faddr;

        u16 hsfctl;
        hsfctl = *(u16*)((u8*)spirb + 0x06);
        hsfctl &= ~(0b11 << 1);  // READ request
        hsfctl |=   ((64 – 1) << 8) | (1 << 0); // FGO, FDBC
        *(u16*)((u8*)spirb + 0x06) = hsfctl;

        // print FDATAx values
}
```

From 0x0000 to 0x1000, 64 byte blocks

Reading from the **Flash ROM**

```
void *spirb;
spirb = ((u8*)rcrb + 0x3800);

for (u32 addr = 0; addr <= 0x1000; addr += 64) {
    u16 hsfsts;
    hsfsts = *(u16*)((u8*)spirb + 0x04);
    // clear FCERR, AEL, FDONE bits
    hsfsts &= ~((1 << 2) | (1 << 1) | (1 << 0));
    *(u16*)((u8*)spirb + 0x04) = hsfsts;

    u32 faddr;
    faddr = *(u32*)((u8*)spirb + 0x08);
    faddr &= 0xFE000000;
    faddr |=   addr;
    *(u32*)((u8*)spirb + 0x08) = faddr;

    u16 hsfctl;
    hsfctl = *(u16*)((u8*)spirb + 0x06);
    hsfctl &= ~(0b11 << 1);  // READ request
    hsfctl |=   ((64 – 1) << 8) | (1 << 0); // FGO, FDBC
    *(u16*)((u8*)spirb + 0x06) = hsfctl;

    // print FDATAx values
}
```

Clear status bits

Reading from the **Flash ROM**

```
void *spirb;
spirb = ((u8*)rcrb + 0x3800);

for (u32 addr = 0; addr <= 0x1000; addr += 64) {
        u16 hsfsts;
        hsfsts = *(u16*)((u8*)spirb + 0x04);
        // clear FCERR, AEL, FDONE bits
        hsfsts &= ~((1 << 2) | (1 << 1) | (1 << 0));
        *(u16*)((u8*)spirb + 0x04) = hsfsts;

        u32 faddr;
        faddr = *(u32*)((u8*)spirb + 0x08);
        faddr &= 0xFE000000;
        faddr |=   addr;
        *(u32*)((u8*)spirb + 0x08) = faddr;

        u16 hsfctl;
        hsfctl = *(u16*)((u8*)spirb + 0x06);
        hsfctl &= ~(0b11 << 1);  // READ request
        hsfctl |=   ((64 – 1) << 8) | (1 << 0); // FGO, FDBC
        *(u16*)((u8*)spirb + 0x06) = hsfctl;

        // print FDATAx values
}
```

Set the address

Reading from the **Flash ROM**

```
void *spirb;
spirb = ((u8*)rcrb + 0x3800);

for (u32 addr = 0; addr <= 0x1000; addr += 64) {
        u16 hsfsts;
        hsfsts = *(u16*)((u8*)spirb + 0x04);
        // clear FCERR, AEL, FDONE bits
        hsfsts &= ~((1 << 2) | (1 << 1) | (1 << 0));
        *(u16*)((u8*)spirb + 0x04) = hsfsts;

        u32 faddr;
        faddr = *(u32*)((u8*)spirb + 0x08);
        faddr &= 0xFE000000;
        faddr |=   addr;
        *(u32*)((u8*)spirb + 0x08) = faddr;

        u16 hsfctl;
        hsfctl = *(u16*)((u8*)spirb + 0x06);
        hsfctl &= ~(0b11 << 1);  // READ request
        hsfctl |=   ((64 – 1) << 8) | (1 << 0); // FGO, FDBC
        *(u16*)((u8*)spirb + 0x06) = hsfctl;

        // print FDATAx values
}
```

READ request, set the block length and go

Reading from the **Flash ROM**

```
void *spirb;
spirb = ((u8*)rcrb + 0x3800);

for (u32 addr = 0; addr <= 0x1000; addr += 64) {
        u16 hsfsts;
        hsfsts = *(u16*)((u8*)spirb + 0x04);
        // clear FCERR, AEL, FDONE bits
        hsfsts &= ~((1 << 2) | (1 << 1) | (1 << 0));
        *(u16*)((u8*)spirb + 0x04) = hsfsts;

        u32 faddr;
        faddr = *(u32*)((u8*)spirb + 0x08);
        faddr &= 0xFE000000;
        faddr |=   addr;
        *(u32*)((u8*)spirb + 0x08) = faddr;

        u16 hsfctl;
        hsfctl = *(u16*)((u8*)spirb + 0x06);
        hsfctl &= ~(0b11 << 1);  // READ request
        hsfctl |=   ((64 – 1) << 8) | (1 << 0); // FGO, FDBC
        *(u16*)((u8*)spirb + 0x06) = hsfctl;

        // print FDATAx values
}
```

Print, dump ....

The Final **BIOS Dump File**

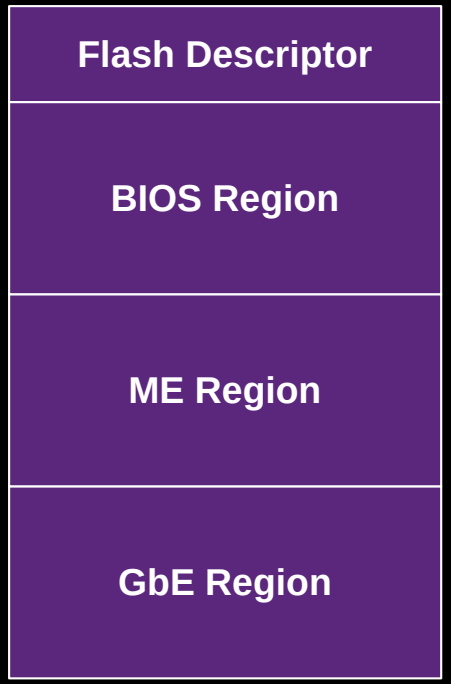
- The **FDATAx** should contain the actual BIOS data
  - It can be reverse engineered
- The **WRITE** command can be used to patch some areas
- But to keep the malware persistent, we have to understand the internal workings of the BIOS

The Flash ROM

UEFI vs **Legacy** BIOS

- Unified Extensible Firmware Interface (UEFI) is **standardized**
- **EDK II** is a generic reference implementation
- **IBVs** and **OEMs** extend it on their own
- UEFI was started in 2005
- Nowadays almost every x86 machine has an UEFI firmware installed

The Flash Regions

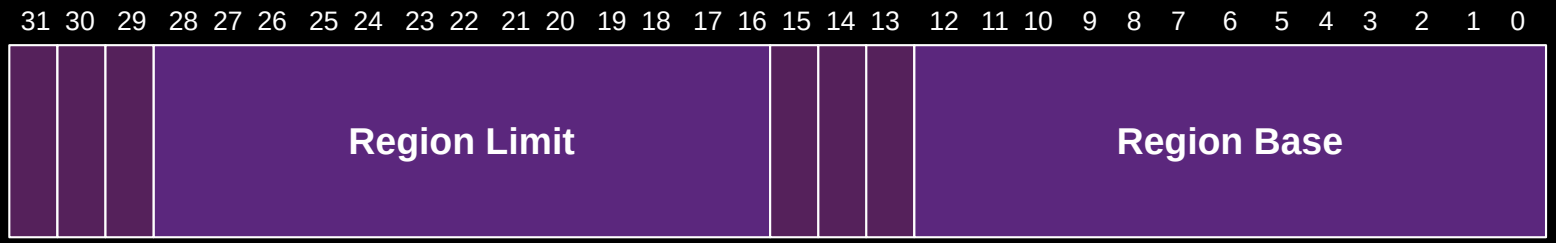


The FREGx Registers

- They are in SPIRB (**RCRB** + **SPIBAR**)
- Point to regions in the flash memory
- They have predefined meanings
  - **FREG0:** Flash Descriptor
  - **FREG1:** BIOS Descriptor
  - **FREG2:** ME Descriptor
  - **FREG3:** GbE Descriptor
  - **FREG4:** Platform Data
- The special address **0x1FFF** indicates that the region is not being used

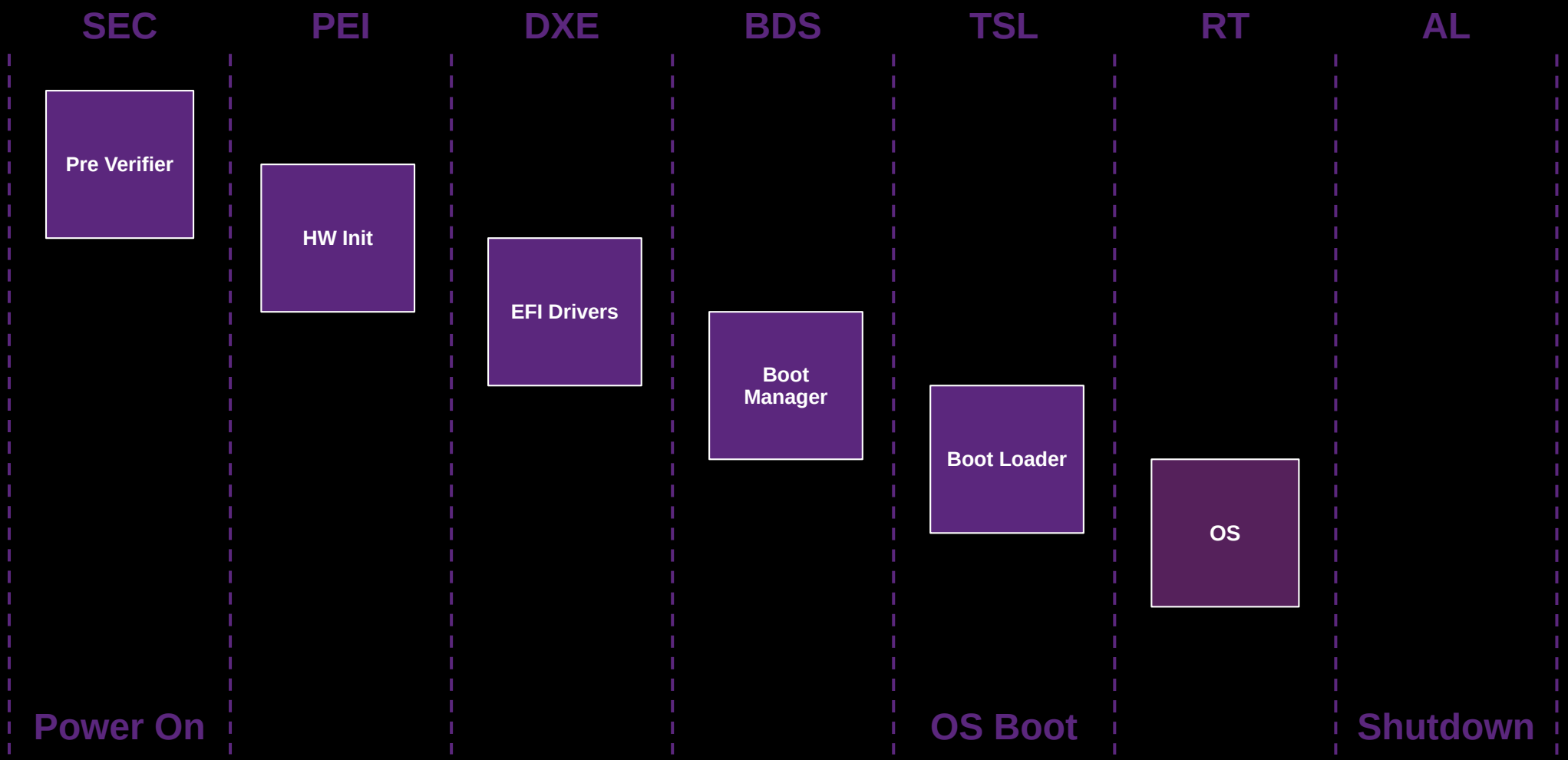
FREGx Registers

Flash Region Register X - 0x54+

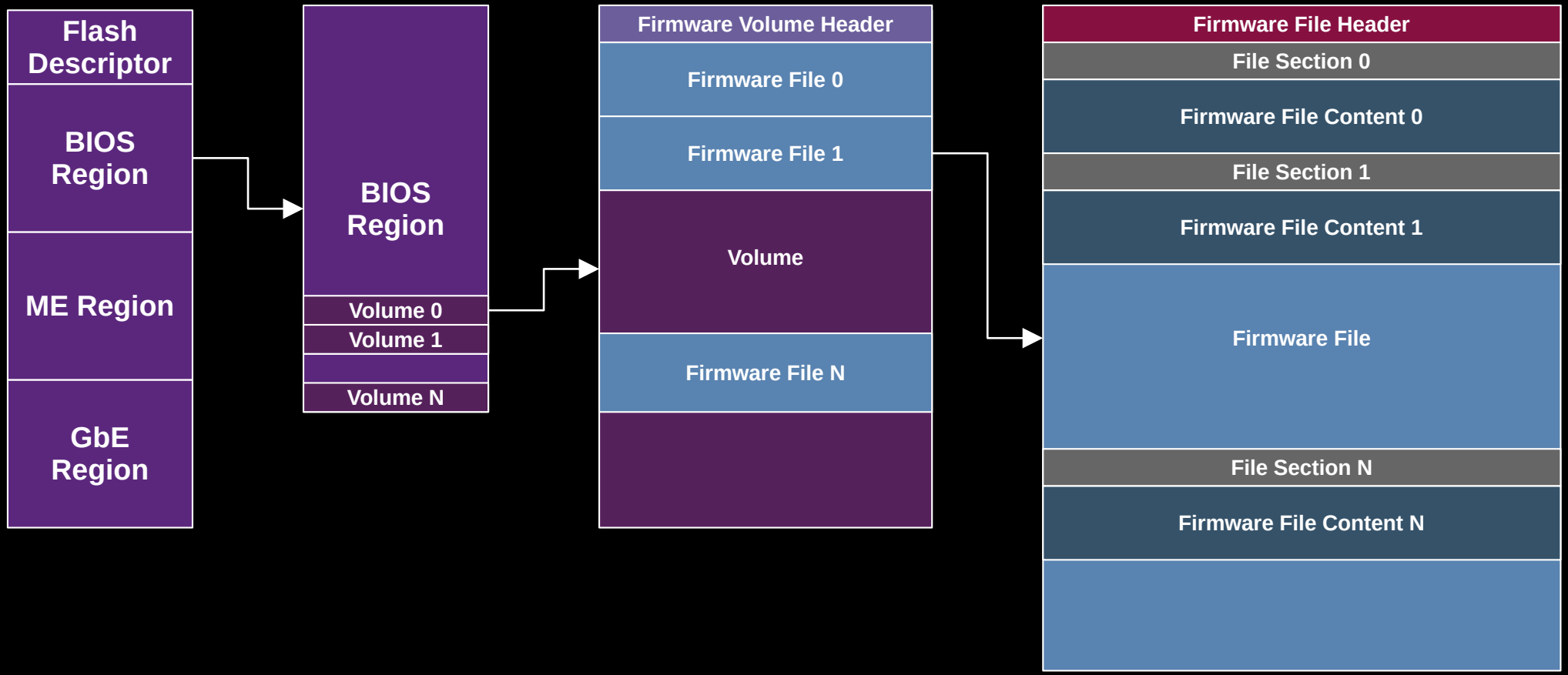


- **Region Base:** needs to be shifted left by **12**
- **Region Limit:** needs to be shifted right by **4**

BOOT Phases


SEC
PEI
DXE
BDS
TSL
RT
AL
Pre Verifier
HW Init
EFI Drivers
Boot Manager
Boot Loader
OS
Power On
OS Boot
Shutdown

The Firmware Volumes



Definitions

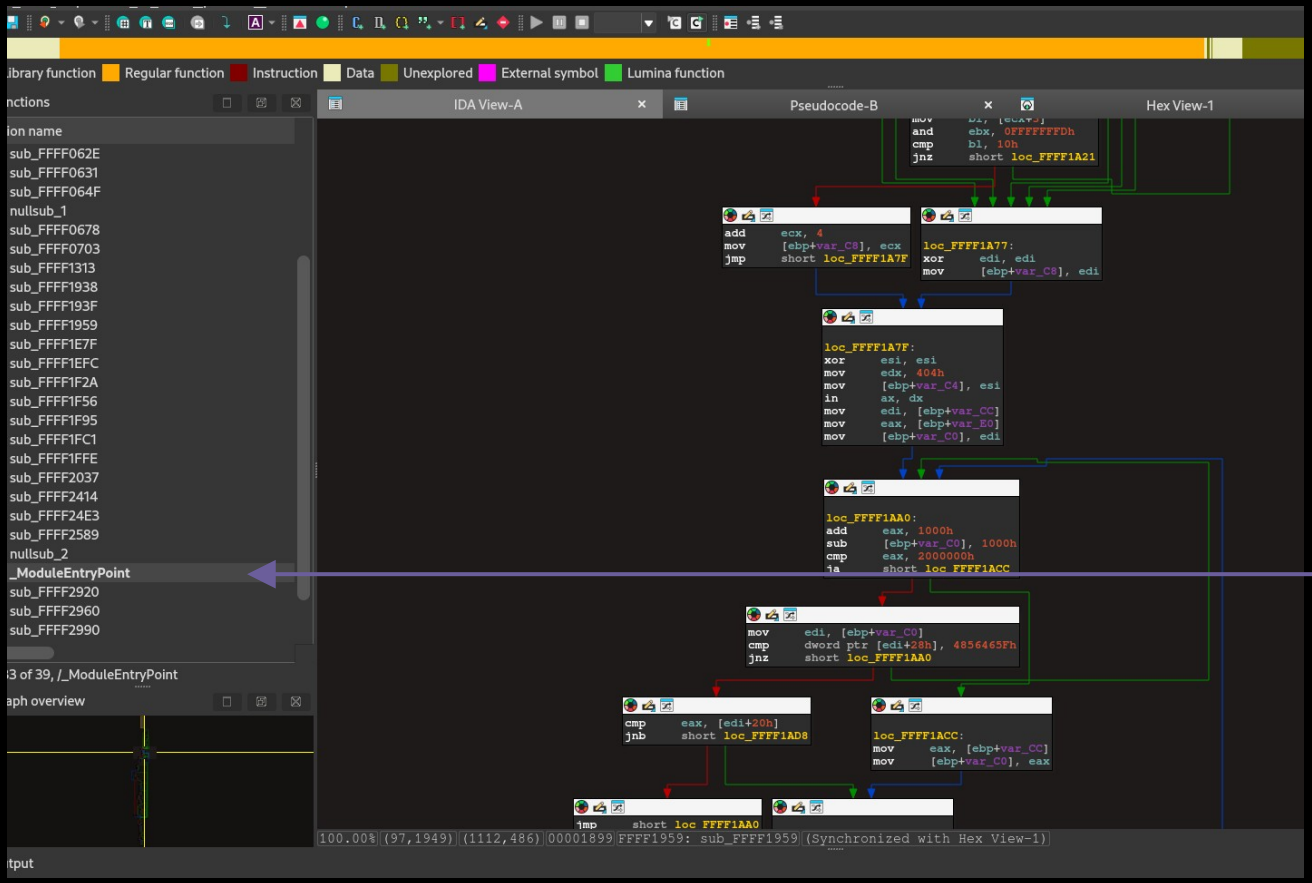
```
typedef struct {
        UINT8 ZeroVector[16];
        EFI_GUID FileSystemGuid;
        UINT64 FvLength;
        UINT32 Signature;
        EFI_FVB_ATTRIBUTES_2 Attributes;
        UINT16 HeaderLength;
        UINT16 Checksum;
        UINT16 ExtHeaderOffset;
        UINT8 Reserved[1];
        UINT8 Revision;
        EFI_FV_BLOCK_MAP BlockMap[];
} EFI_FIRMWARE_VOLUME_HEADER;
```

```
typedef struct {
        EFI_GUID Name;
        EFI_FFS_INTEGRITY_CHECK IntegrityCheck;
        EFI_FV_FILETYPE Type;
        EFI_FFS_FILE_ATTRIBUTES Attributes;
        UINT8 Size[3];
        EFI_FFS_FILE_STATE State;
} EFI_FFS_FILE_HEADER;
```

```
typedef struct {
        UINT8 Size[3];
        EFI_SECTION_TYPE Type;
} EFI_COMMON_SECTION_HEADER;
```

- First file can be found at **Volume Start + HeaderLength**
- Consecutive files can be found by moving **Size** amount
- Similarly file sections can be found by using the section **Size**
- The core files are **PE** or **TE** formatted files

The Result



Entry Point

The **UEFI Images**

- **Drivers:** Stays in the System Memory
  - **Service Drivers:** Provides Protocols
    - **Boot Service:** Removed after Boot Loader exits
    - **Runtime:** Remains after Boot Loader exits
  - **Initialization Drivers:** Performs Platform initialization
  - ...

- **Applications:** Cleanup is performed after exit
  - **Boot Loaders:** Grub, Winbmgr etc.
  - **EFI Shell Commands:** Utility tools
  - ...

The **UEFI Image Entry**

- The entry point accepts 2 arguments
- The **ImageHandle**, firmware installed handle
- The **SystemTable**, a pointer to service APIs and more

The System Table

```
typedef struct {
        EFI_TABLE_HEADER                    Hdr;
        CHAR16                              *FirmwareVendor;
        UINT32                              FirmwareRevision;
        EFI_HANDLE                          ConsoleInHandle;
        EFI_SIMPLE_TEXT_INPUT_PROTOCOL      *ConIn;
        EFI_HANDLE                          ConsoleOutHandle;
        EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL     *ConOut;
        EFI_HANDLE                          StandardErrorHandle;
        EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL     *StdErr;
        EFI_RUNTIME_SERVICES                *RuntimeServices;
        EFI_BOOT_SERVICES                   *BootServices;
        UINTN                               NumberOfTableEntries;
        EFI_CONFIGURATION_TABLE             *ConfigurationTable;
} EFI_SYSTEM_TABLE;
```

Runtime APIs

Boot APIs

The Runtime Services

```
typedef struct {
        EFI_TABLE_HEADER                    Hdr;
        EFI_GET_TIME                        GetTime;
        EFI_SET_TIME                        SetTime;
        EFI_GET_WAKEUP_TIME                 GetWakeupTime;
        EFI_SET_WAKEUP_TIME                 SetWakeupTime;
        EFI_SET_VIRTUAL_ADDRESS_MAP         SetVirtualAddressMap;
        EFI_CONVERT_POINTER                 ConvertPointer;
        EFI_GET_VARIABLE                    GetVariable;
        EFI_GET_NEXT_VARIABLE_NAME          GetNextVariableName;
        EFI_SET_VARIABLE                    SetVariable;
        EFI_GET_NEXT_HIGH_MONO_COUNT        GetNextHighMonotonicCount;
        EFI_RESET_SYSTEM                    ResetSystem;
        EFI_UPDATE_CAPSULE                  UpdateCapsule;
        EFI_QUERY_CAPSULE_CAPABILITIES      QueryCapsuleCapabilities;
        EFI_QUERY_VARIABLE_INFO             QueryVariableInfo;
} EFI_RUNTIME_SERVICES;
```

Allows OS to change EFI Vars

Allows updating the Firmware

The Boot Services

```
typedef struct {
    EFI_TABLE_HEADER                        Hdr;
    ...
    EFI_INSTALL_PROTOCOL_INTERFACE          InstallProtocolInterface;
    ...
    EFI_HANDLE_PROTOCOL                     HandleProtocol;
    ...
    EFI_IMAGE_LOAD                          LoadImage;
    EFI_IMAGE_START                         StartImage;
    ...
    EFI_EXIT_BOOT_SERVICES                  ExitBootServices;
    ...
    EFI_OPEN_PROTOCOL                       OpenProtocol;
    EFI_CLOSE_PROTOCOL                      CloseProtocol;
    ...
    EFI_LOCATE_PROTOCOL                     LocateProtocol;
    ...
} EFI_BOOT_SERVICES;
```

The **UEFI Protocols**

- Serve as Library APIs
- There are predefined APIs which can be used at the beginning
- Drivers can create their own interfaces
- They're used to abstract away from the hardware
- Identified by a **GUID**
- They ***HAVE*** to contain the GUID, but Functions and/or Data is optional

UEFI Application example

```
EFI_STATUS EFIAPI UefiMain(
    EFI_HANDLE        ImageHandle,
    EFI_SYSTEM_TABLE  *SystemTable)
{
    MY_INTERFACE *MyInterface;
    SystemTable->BootServices->LocateProtocol(
        MY_PROTOCOL_GUID,
        NULL,
        &MyInterface);
    //  ...

    MyInterface->MyFunction0();
    //  ...

    return EFI_SUCCESS;
}
```

Questions?

LinkedIn: Hrant Tadevosyan, Levon Martirosyan
Email: hrant.tadevosyan@protonmail.com, levonmartirosyan2019@gmail.com
GitHub: https://github.com/0x0000z3r0/bios

Website: astronaut.am

ASTRONAUT

per aspera ad astra