EDK II SMM call topology

SMBASE Relocation

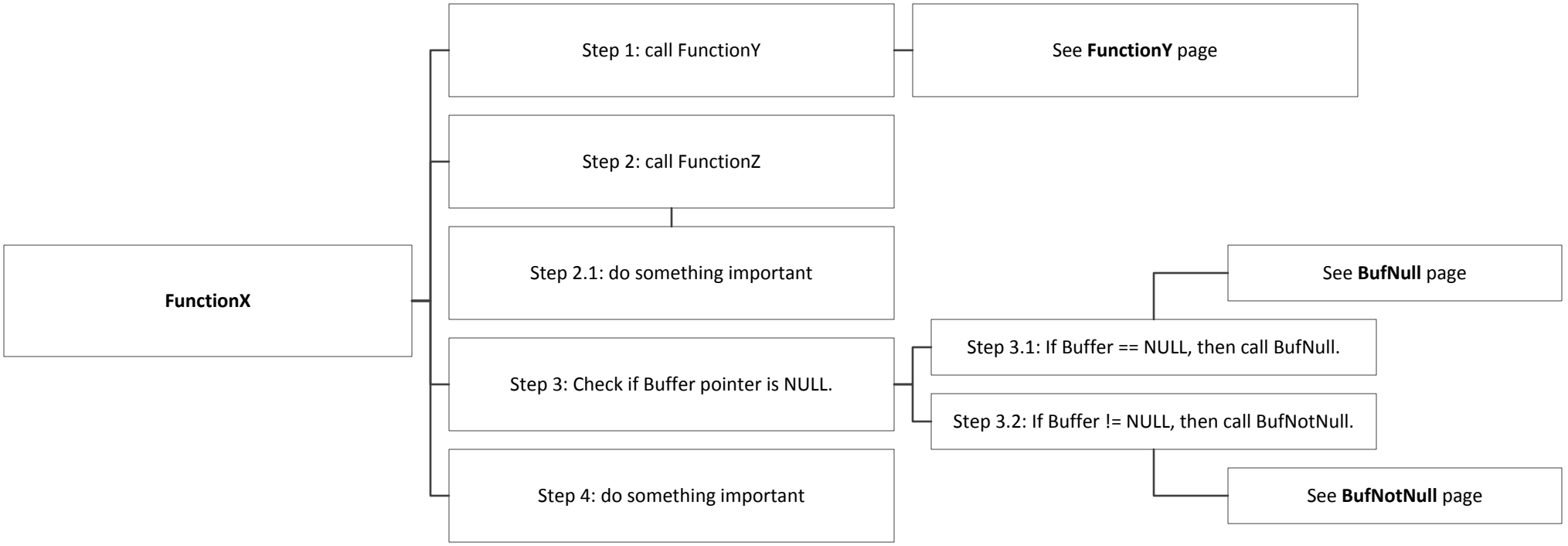
SMI Handler

SMI Exception Handler

The function being discussed always starts in the only box on the far left. Boxes represent steps in a function, a branch evaluation in a funtion, or a note to see details on another page about a function being called at a step. Connectors between boxes indicate code flow (who called what) and should be read left-to-right. Text in each box will indicate if it's a call, a branch evaluation, or a note. This format was chosen to fit important function details on 1 page.

For the example below, the equivalent C code (note connectors from Step 1-4 to FunctionX):

```
FunctionX() {
        FunctionY();    // see details on FunctionY page
        FunctionZ();    // Step 2; Step 2.1 is in FunctionZ() and is listed because it is important; note the connector between Step 2 and 2.1
        If (Buffer == NULL)     // Step 3
                BufNull();      // Step 3.1
        else
                BufNotNull();   // Step 3.2
        Step 4
}
```



How protocol services are defined so one can find the protocol code to examine it

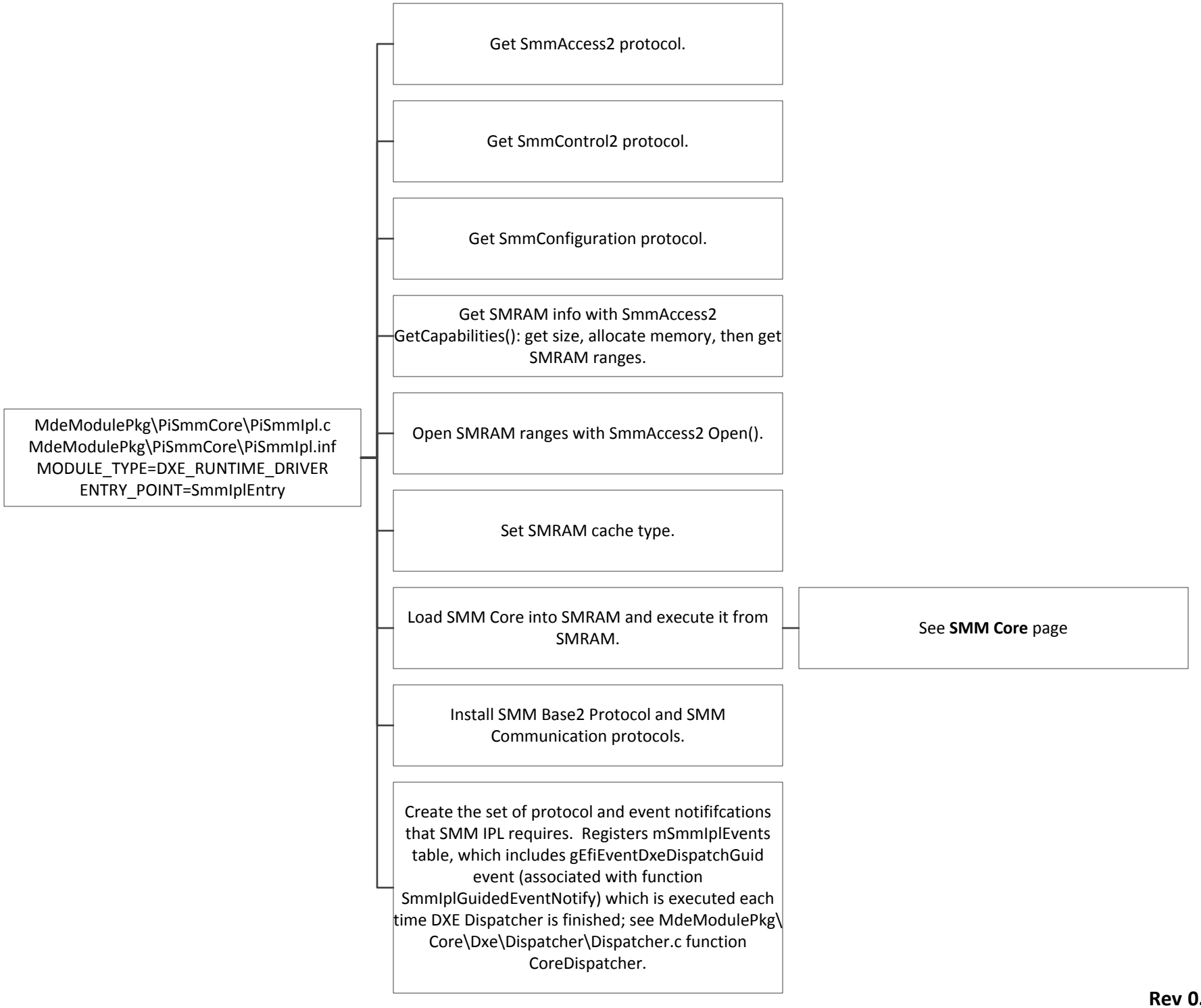
Summary: if you want to examine the code for a protocol function, you should find the structure definition for the protocol, then find the declaration of the structure, then find the structure member that corresponds with the protocol in the structure definition because they may have different names. EFI_BOOT_SERVICES defines LocateProtocol, mBootServices is of type EFI_BOOT_SERVICES, and the structure member CoreLocateProtocol corresponds with the structure definition LocateProtocol.

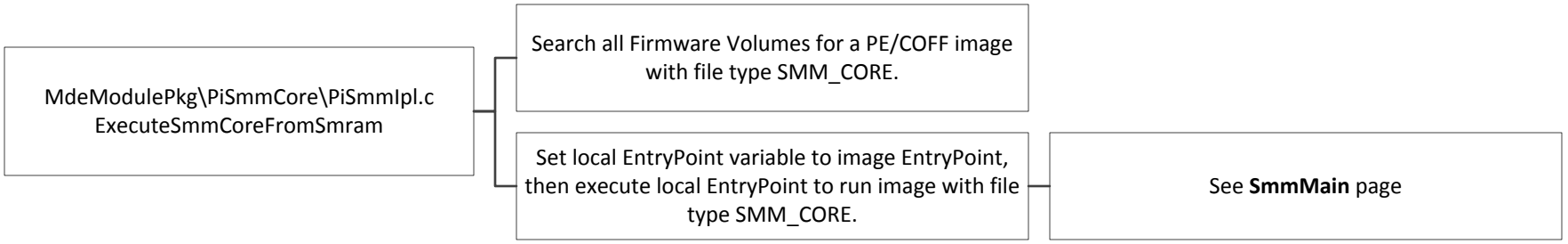
MdePkg\Include\UefiSpec.h defines EFI_BOOT_SERVICES structure, and has structure members for protocol services (LocateProtocol, InstallProtocolInterface, etc). MdeModulePkg\Core\Dxe\DxeMain.c has a variable mBootServices of type EFI_BOOT_SERVICES. mBootServices sets function pointers for functions such as LocateProtocol to CoreLocateProtocol and InstallMultipleProtocolInterfaces to CoreInstallMultipleProtocolInterfaces. These functions are defined in MdeModulePkg\Core\Dxe\Handle.c.

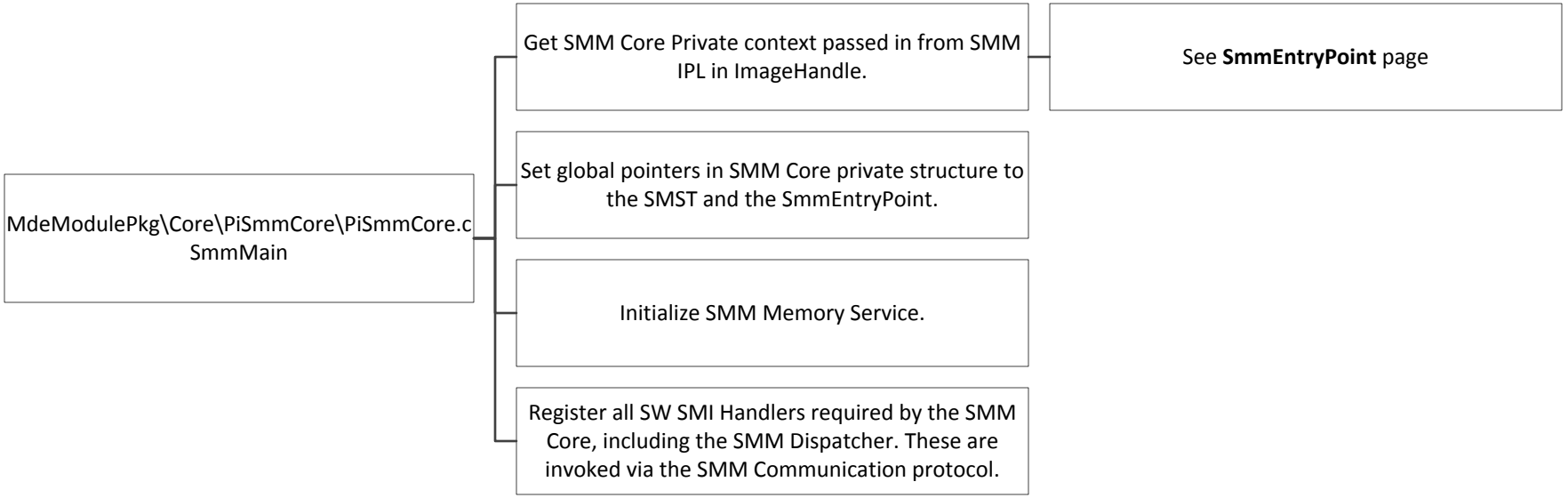
How protocols are loaded from flash into memory

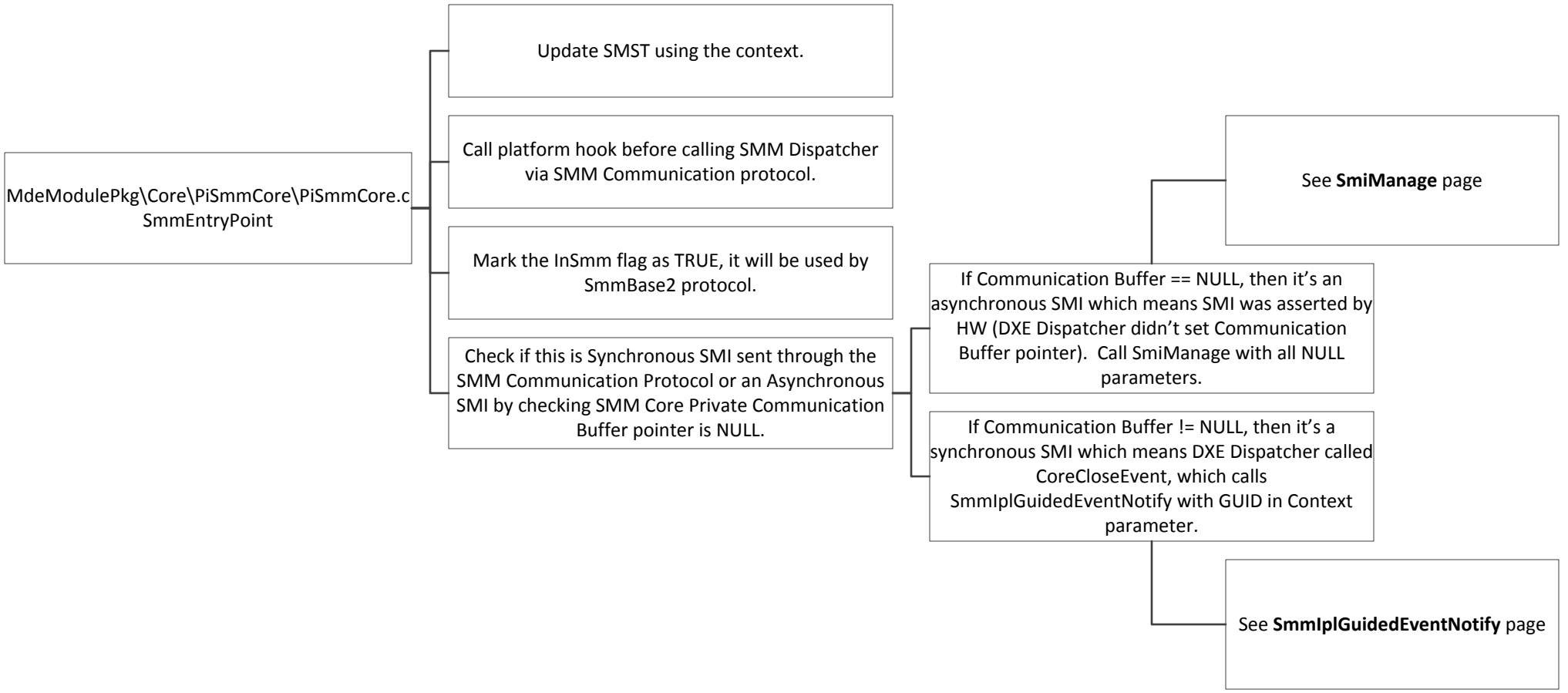
Summary: drivers are loaded from flash into memory by some mechanisms into a linked list during the PEI phase.

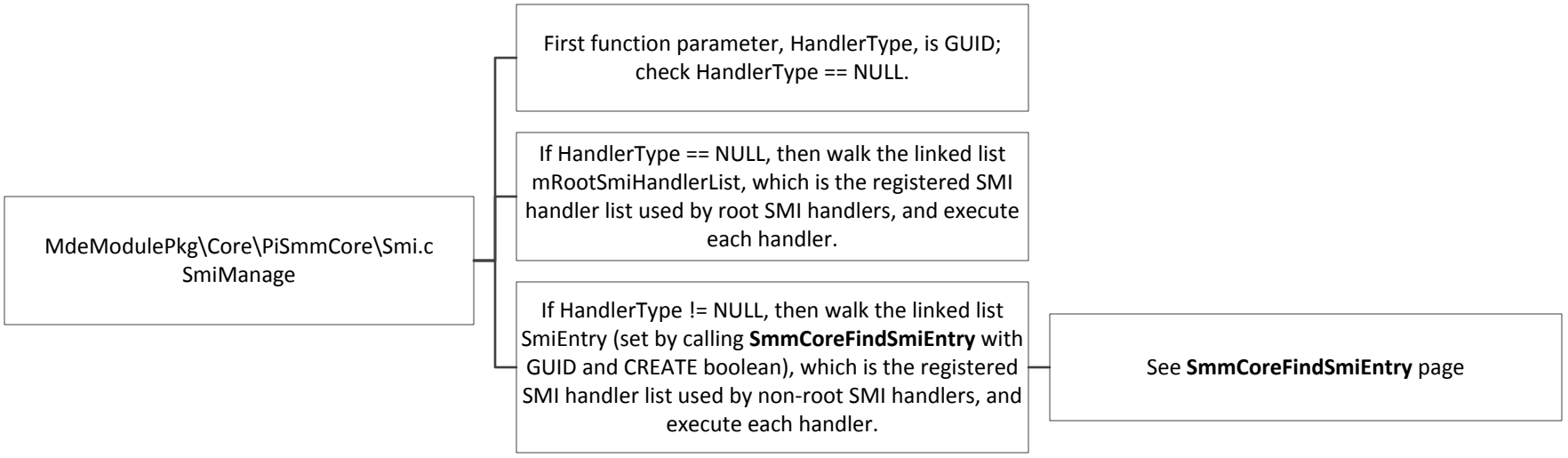
MdeModulePkg\Core\Dxe\DxeMain.c has DxeMain function which is called when the DXE Core driver is loaded. MdeModulePkg\Core\Dxe\DxeMain.inf has MODULE_TYPE=DXE_CORE and ENTRY_POINT=DxeMain. The end of DxeMain calls CoreInstallMultipleProtocolInterface with the GUID for the HOB that was populated with drivers from the flash part during PEI. PEI phase calls ReadSection (associated with FvReadFileSection in Universal\FirmwareVolume\FwVolDxe\FwVol.c), which eventually gets to a call to LocateProtocol with gEfiDecompressProtocolGuid as a parameter.

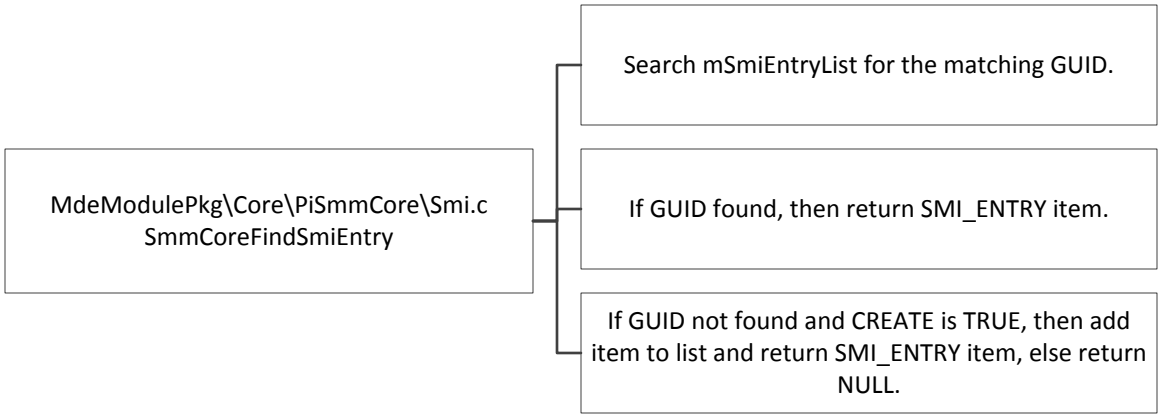

MdeModulePkg\PiSmmCore\PiSmmIpl.c
MdeModulePkg\PiSmmCore\PiSmmIpl.inf
MODULE_TYPE=DXE_RUNTIME_DRIVER
ENTRY_POINT=SmmIplEntry
Get SmmAccess2 protocol.
Get SmmControl2 protocol.
Get SmmConfiguration protocol.
Get SMRAM info with SmmAccess2 GetCapabilities(): get size, allocate memory, then get SMRAM ranges.
Open SMRAM ranges with SmmAccess2 Open().
Set SMRAM cache type.
Load SMM Core into SMRAM and execute it from SMRAM.
See **SMM Core** page
Install SMM Base2 Protocol and SMM Communication protocols.
Create the set of protocol and event notifications that SMM IPL requires. Registers mSmmIplEvents table, which includes gEfiEventDxeDispatchGuid event (associated with function SmmIplGuidedEventNotify) which is executed each time DXE Dispatcher is finished; see MdeModulePkg\Core\Dxe\Dispatcher\Dispatcher.c function CoreDispatcher.

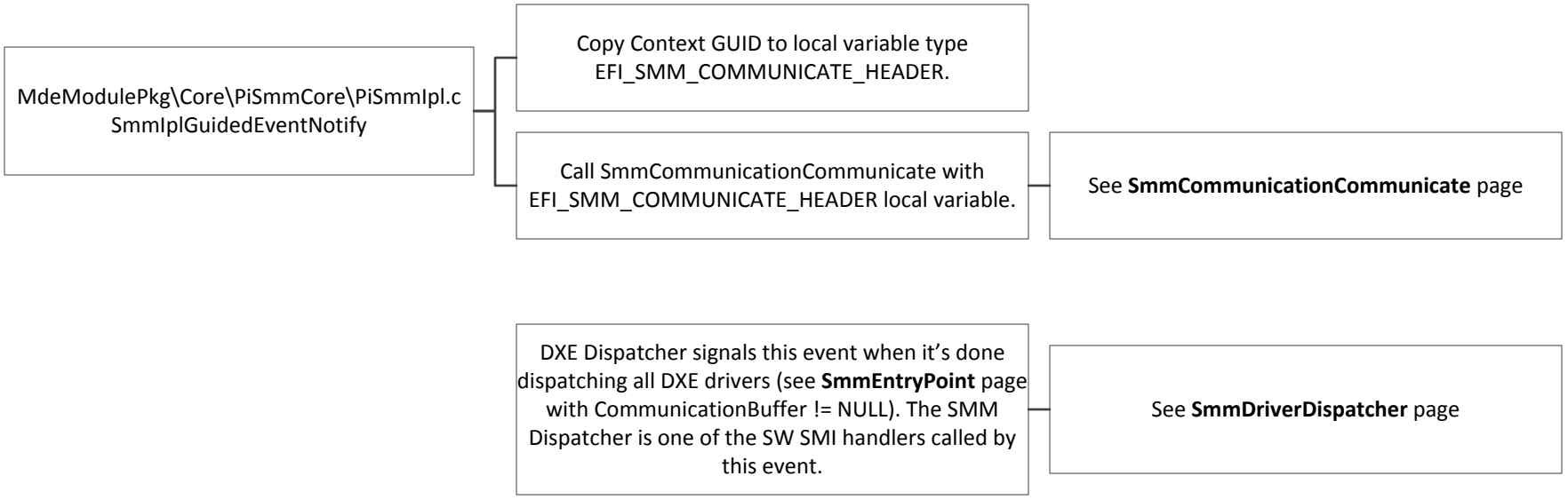

MdeModulePkg\PiSmmCore\PiSmmIpl.c
ExecuteSmmCoreFromSmram
Search all Firmware Volumes for a PE/COFF image with file type SMM_CORE.
Set local EntryPoint variable to image EntryPoint, then execute local EntryPoint to run image with file type SMM_CORE.
See **SmmMain** page

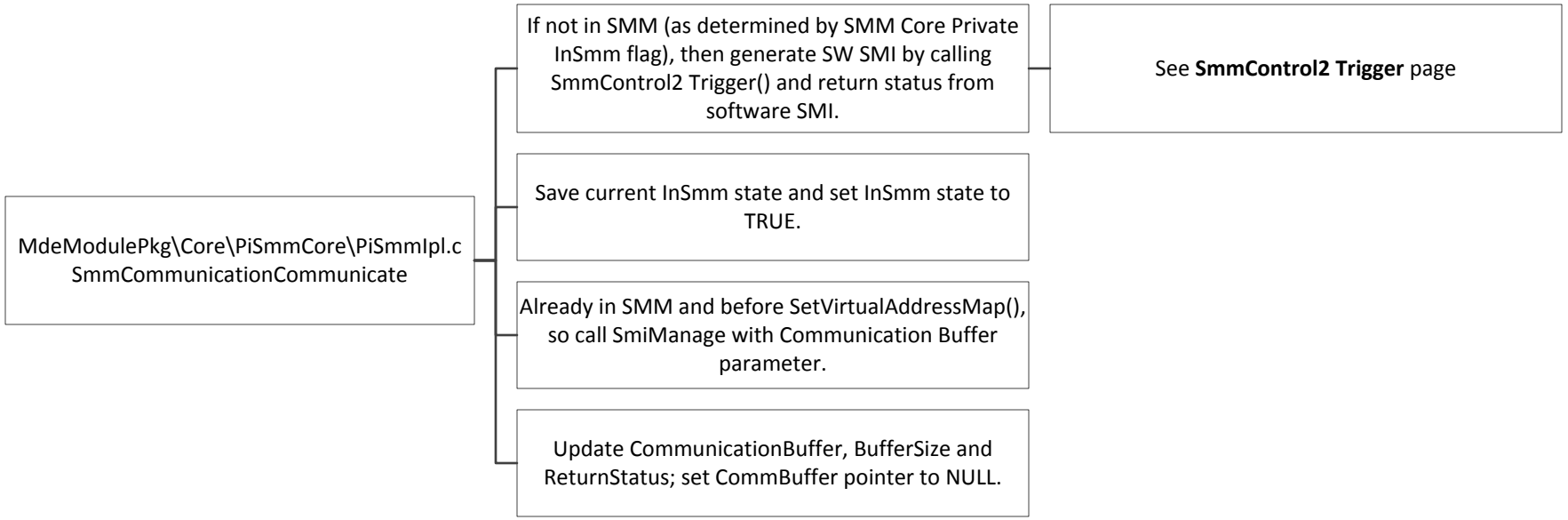

MdeModulePkg\Core\PiSmmCore\PiSmmCore.c
SmmMain
Get SMM Core Private context passed in from SMM IPL in ImageHandle.
See **SmmEntryPoint** page
Set global pointers in SMM Core private structure to the SMST and the SmmEntryPoint.
Initialize SMM Memory Service.
Register all SW SMI Handlers required by the SMM Core, including the SMM Dispatcher. These are invoked via the SMM Communication protocol.

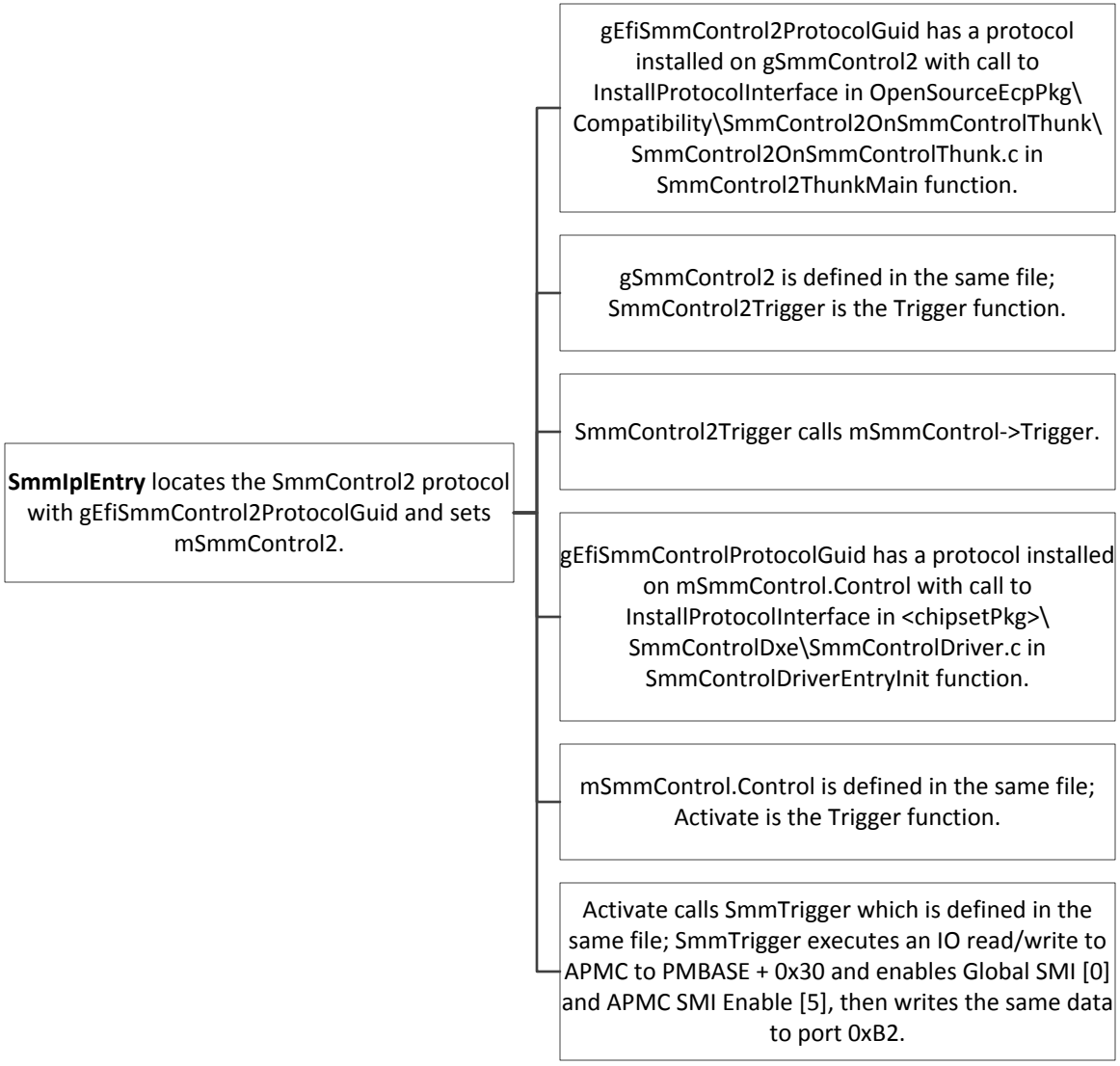

MdeModulePkg\Core\PiSmmCore\PiSmmCore.c
SmmEntryPoint
Update SMST using the context.
Call platform hook before calling SMM Dispatcher via SMM Communication protocol.
Mark the InSmm flag as TRUE, it will be used by SmmBase2 protocol.
Check if this is Synchronous SMI sent through the SMM Communication Protocol or an Asynchronous SMI by checking SMM Core Private Communication Buffer pointer is NULL.
If Communication Buffer == NULL, then it's an asynchronous SMI which means SMI was asserted by HW (DXE Dispatcher didn't set Communication Buffer pointer). Call SmiManage with all NULL parameters.
See **SmiManage** page
If Communication Buffer != NULL, then it's a synchronous SMI which means DXE Dispatcher called CoreCloseEvent, which calls SmmIplGuidedEventNotify with GUID in Context parameter.
See **SmmIplGuidedEventNotify** page

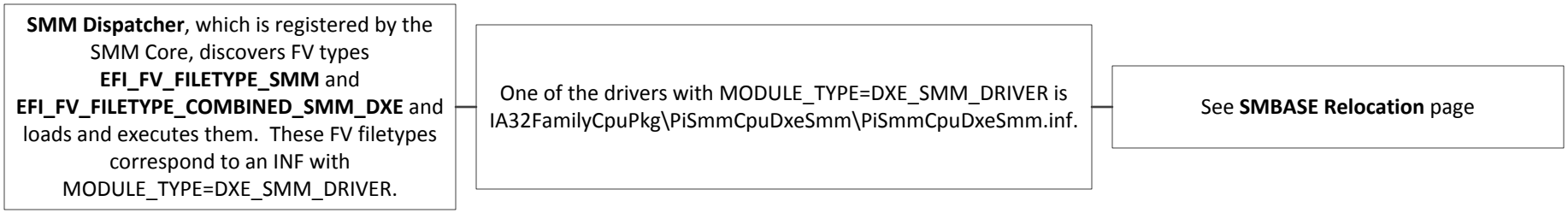

MdeModulePkg\Core\PiSmmCore\Smi.c
SmiManage
First function parameter, HandlerType, is GUID; check HandlerType == NULL.
If HandlerType == NULL, then walk the linked list mRootSmiHandlerList, which is the registered SMI handler list used by root SMI handlers, and execute each handler.
If HandlerType != NULL, then walk the linked list SmiEntry (set by calling **SmmCoreFindSmiEntry** with GUID and CREATE boolean), which is the registered SMI handler list used by non-root SMI handlers, and execute each handler.
See **SmmCoreFindSmiEntry** page


MdeModulePkg\Core\PiSmmCore\Smi.c
SmmCoreFindSmiEntry
Search mSmiEntryList for the matching GUID.
If GUID found, then return SMI_ENTRY item.
If GUID not found and CREATE is TRUE, then add item to list and return SMI_ENTRY item, else return NULL.


MdeModulePkg\Core\PiSmmCore\PiSmmIpl.c
SmmIplGuidedEventNotify
Copy Context GUID to local variable type EFI_SMM_COMMUNICATE_HEADER.
Call SmmCommunicationCommunicate with EFI_SMM_COMMUNICATE_HEADER local variable.
See **SmmCommunicationCommunicate** page
DXE Dispatcher signals this event when it's done dispatching all DXE drivers (see **SmmEntryPoint** page with CommunicationBuffer != NULL). The SMM Dispatcher is one of the SW SMI handlers called by this event.
See **SmmDriverDispatcher** page


MdeModulePkg\Core\PiSmmCore\PiSmmIpl.c
SmmCommunicationCommunicate
If not in SMM (as determined by SMM Core Private InSmm flag), then generate SW SMI by calling SmmControl2 Trigger() and return status from software SMI.
See **SmmControl2 Trigger** page
Save current InSmm state and set InSmm state to TRUE.
Already in SMM and before SetVirtualAddressMap(), so call SmiManage with Communication Buffer parameter.
Update CommunicationBuffer, BufferSize and ReturnStatus; set CommBuffer pointer to NULL.


SmmIplEntry locates the SmmControl2 protocol with gEfiSmmControl2ProtocolGuid and sets mSmmControl2.
gEfiSmmControl2ProtocolGuid has a protocol installed on gSmmControl2 with call to InstallProtocolInterface in OpenSourceEcpPkg\ Compatibility\SmmControl2OnSmmControlThunk\ SmmControl2OnSmmControlThunk.c in SmmControl2ThunkMain function.
gSmmControl2 is defined in the same file; SmmControl2Trigger is the Trigger function.
SmmControl2Trigger calls mSmmControl->Trigger.
gEfiSmmControlProtocolGuid has a protocol installed on mSmmControl.Control with call to InstallProtocolInterface in <chipsetPkg>\ SmmControlDxe\SmmControlDriver.c in SmmControlDriverEntryInit function.
mSmmControl.Control is defined in the same file; Activate is the Trigger function.
Activate calls SmmTrigger which is defined in the same file; SmmTrigger executes an IO read/write to APMC to PMBASE + 0x30 and enables Global SMI [0] and APMC SMI Enable [5], then writes the same data to port 0xB2.


SMM Dispatcher, which is registered by the SMM Core, discovers FV types **EFI_FV_FILETYPE_SMM** and **EFI_FV_FILETYPE_COMBINED_SMM_DXE** and loads and executes them. These FV filetypes correspond to an INF with MODULE_TYPE=DXE_SMM_DRIVER.
One of the drivers with MODULE_TYPE=DXE_SMM_DRIVER is IA32FamilyCpuPkg\PiSmmCpuDxeSmm\PiSmmCpuDxeSmm.inf.
See **SMBASE Relocation** page

SMBASE Relocation

EDK II SMM call topology

IA32FamilyCpuPkg\PiSmmCpuDxeSmm\PiSmmCpuDxeSmm.c
IA32FamilyCpuPkg\PiSmmCpuDxeSmm\PiSmmCpuDxeSmm.inf
MODULE_TYPE=DXE_SMM_DRIVER
ENTRY_POINT=PiCpuSmmEntry

PiCpuSmmEntry determines the max number of processors (discovered; or if PcdHotPlugSupport is defined, the value of PcdCpuMaxLogicalProcessorNumber), and calculates the tile size of the CPU SMM Save State for the max number of processors.

SmmRelocateBases is called.

SmmRelocateBases copies gcSmmInitTemplate to 0x38000 and issues a SIPI to all APs and then to itself (the BSP).

gcSmmInitTemplate is a LABEL defined in IA32FamilyCpuPkg\PiSmmCpuDxeSmm\X64\SmmInit.asm

The code at gcSmmInitTemplate sets EBP to the address of SmmStartup

SmmStartup sets the GDT, enables Protected Mode and Paging in CR0, and enables LME (Long Mode Enable) in MSR IA32_EFER_MSR.

SmmStartup then calls SmmInitHandler, which is located in IA32FamilyCpuPkg\PiSmmCpuDxeSmm\PiSmmCpuDxeSmm.c

SmmInitHandler gets the executing processor's APIC ID, and then loops over the number of CPUs discovered. If the current processor index == APIC ID, then the SMBASE of the processor is set to the CPU Hot Plug SMBASE. If the current processor is the BSP, then InitializeMpSyncData is called.

SmmStartup executes RSM.

SMM Timer is initialized.

IDT is initialized with call to InitializeIDT, which sets exception handler entry point to _SmiExceptionEntryPoints; see IA32FamilyCpuPkg\PiSmmCpuDxeSmm\X64\SmiException.asm

Exception Handler table mExternalVectorTable (referenced by _SmiExceptionEntryPoints) is initialized with call to InitializeSmmExternalVectorTablePtr.

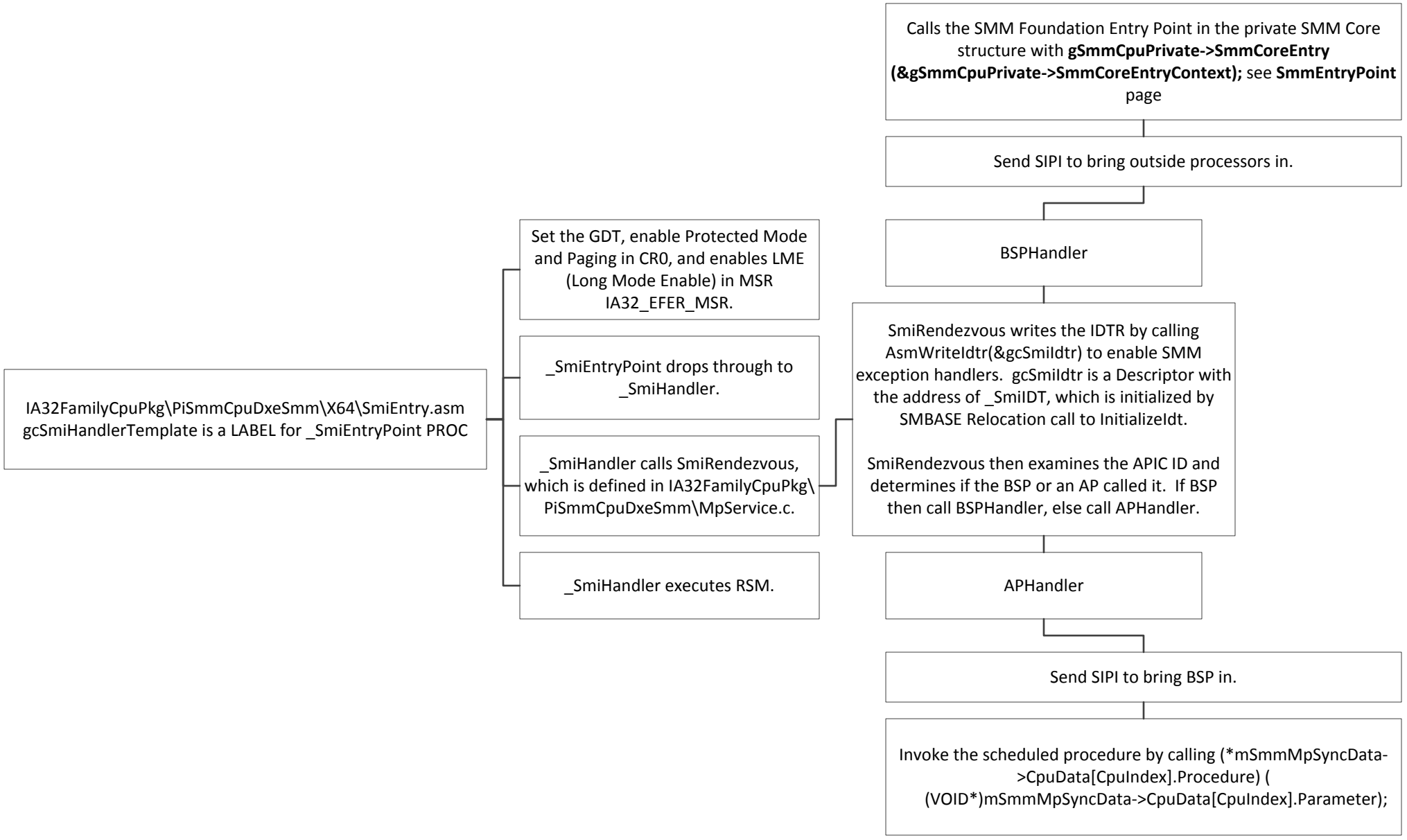
InitializeMpServiceData is called.

SMI Page Tables are created with call to SmmInitPageTable.

Loop over max number of processors: copy gcSmiHandlerTemplate to each processor SMBASE. See **SMI Handler** page for what happens when a HW SMI is asserted.

InitializeMpSyncData is called, sets mSmmMpSyncData to NULL, initializes a few structure items, but leaves CpuData (which has Procedure function pointer used by APHandler) set to NULL.

Rev 0.3.1 October 2012


IA32FamilyCpuPkg\PiSmmCpuDxeSmm\X64\SmiEntry.asm
gcSmiHandlerTemplate is a LABEL for _SmiEntryPoint PROC
Set the GDT, enable Protected Mode and Paging in CR0, and enables LME (Long Mode Enable) in MSR IA32_EFER_MSR.
_SmiEntryPoint drops through to _SmiHandler.
_SmiHandler calls SmiRendezvous, which is defined in IA32FamilyCpuPkg\PiSmmCpuDxeSmm\MpService.c.
_SmiHandler executes RSM.
SmiRendezvous writes the IDTR by calling AsmWriteIdtr(&gcSmiIdtr) to enable SMM exception handlers. gcSmiIdtr is a Descriptor with the address of _SmiIDT, which is initialized by SMBASE Relocation call to InitializeIdt.
SmiRendezvous then examines the APIC ID and determines if the BSP or an AP called it. If BSP then call BSPHandler, else call APHandler.
BSPHandler
APHandler
Send SIPI to bring outside processors in.
Send SIPI to bring BSP in.
Calls the SMM Foundation Entry Point in the private SMM Core structure with gSmmCpuPrivate->SmmCoreEntry (&gSmmCpuPrivate->SmmCoreEntryContext); see SmmEntryPoint page
Invoke the scheduled procedure by calling (*mSmmMpSyncData->CpuData[CpuIndex].Procedure) ((VOID*)mSmmMpSyncData->CpuData[CpuIndex].Parameter);