SMMU – System MMU

Contents

[1. SMMU Introduction 2](#_Toc187266583)

[1.1. SMMU versus MMU: - 3](#_Toc187266584)

[1.2. Advantages of SMMU: - 3](#_Toc187266585)

[1.3. Disadvantages of using the IOMMU:- 4](#_Toc187266586)

[1.4. SMMU stream 4](#_Toc187266587)

[1.5. Stream ID and Stream Table Entry (STE) 5](#_Toc187266588)

[2. Queues in SMMU driver: - 5](#_Toc187266589)

[2.1. SMMU Command Queue, cmdq: - 5](#_Toc187266590)

[2.2. Event Queue, evtq: - 6](#_Toc187266591)

[2.3. Priority Queue, priq: - 6](#_Toc187266592)

[3. Registering the SMMU with the IOMMU 6](#_Toc187266593)

[4. IOMMU and DMA 9](#_Toc187266594)

[4.1. Function code flow in case of DMA mapping 9](#_Toc187266595)

[References: - 12](#_Toc187266596)

# SMMU Introduction

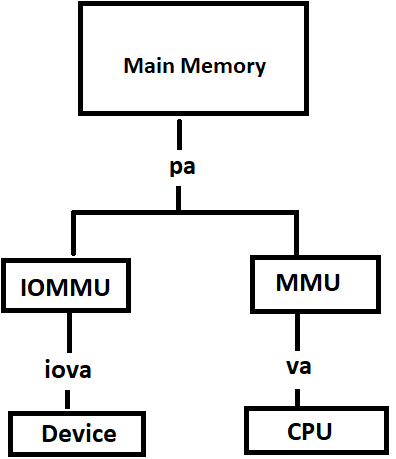
SMMU converts virtual addresses to physical addresses for external peripheral devices. This allows multiple external devices to perform direct memory access (DMA) to the entire range of the system physical memory.

As an example, certain peripheral devices limited to accessing only 24 bits of address space would now be able to access all 64 bits addressing through the memory translation tables of the SMMU.

SMMU registers are configured through ARM Trusted firmware (ATF) BL31 SMC calls by the Cryptography device drivers. The ATF performs default SMMU initializations of the stream IDs through the system manager and SMMU secure registers configuration during the boot-up process.

In computing, an input–output memory management unit (IOMMU) or SMMU in case of ARM is a memory management unit (MMU) that connects a direct-memory-access–capable (DMA-capable) I/O bus to the physical memory. Like a traditional MMU, the IOMMU maps device-visible virtual addresses (also called I/O virtual address, IOVA) to physical addresses (PAs). Different platforms have different IOMMUs, such as System Memory Management Unit (SMMU) used by the ARM platform.

The CPU and devices access the physical memory in the following way:

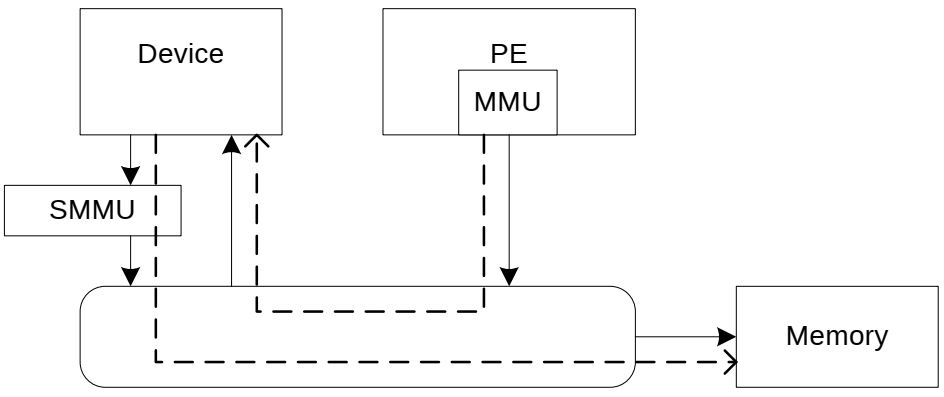


## SMMU versus MMU: -

Traffic from DMA capable I/O device to Memory is handled by SMMU.

Traffic in the other direction, from the system or PE to the device, is managed by other means – for example, the PE MMUs.

PE – Programming element/CPU



## 1.2. Advantages of SMMU: -

* Large regions of memory can be allocated without the need to be contiguous in physical memory. The IOMMU maps contiguous virtual addresses (VAs) to the fragmented PAs.
* Devices that do not support memory addresses long enough to address the entire physical memory can still address the entire memory through the IOMMU. For example, x86 computers can address more than 4 GB of memory with the Physical Address Extension (PAE) feature. But an ordinary 32-bit PCI device cannot address the memory above 4 GB. With IOMMU, the device can address the entire physical memory.
* Memory is protected from malicious devices that are attempting DMA attacks and faulty devices that are attempting errant memory transfers because a device cannot read or write to the mapped physical memory.
* In virtualization, a guest OS can use hardware that is not specifically made for virtualization. Higher performance hardware such as graphics cards uses DMA to access memory directly. In a virtual environment, all memory addresses are re-mapped by the virtualization software (such as QEMU), which causes the guest OS fails to access memory using DMA. The IOMMU handles this re-mapping, allowing the drivers to use DMA to access memory in the guest OS.
* In some architecture IOMMU also performs interrupt re-mapping, in a manner similar to address re-mapping.
* Peripheral memory paging can be supported by an IOMMU. A peripheral using the PCI-SIG PCIe Address Translation Services (ATS), Page Request Interface (PRI) extension can detect and signal the need for memory manager services.

## 1.3. Disadvantages of using the IOMMU:-

* Extra performance and memory overhead. Address translation and page fault processing increase performance overhead.
* In addition, the IOMMU needs to allocate space for the I/O page table in the memory. In some cases, the IOMMU and CPU share the page table to avoid this memory overhead. For example, the device and CPU share the virtual address.

## 1.4. SMMU stream

To associate device traffic with translations and to differentiate different devices behind an SMMU, requests have an extra property, alongside address, read/write, permissions, to identify a stream.

Different streams are logically associated with different devices and the SMMU can perform different translations or checks for each stream.

## 1.5. Stream ID and Stream Table Entry (STE)

The SMMU provides the capability of accessing the physical memory by using the IOVA visible to the device. In the system architecture, multiple devices may use the IOVA to access the physical memory through the IOMMU.

The IOMMU needs to distinguish different devices, so each device is assigned with a stream ID (SID), which points to the corresponding stream table entry (STE). All STEs exist in the memory as arrays. The SMMU records the start addresses of the STE arrays. When scanning a device, the OS allocates a unique SID to the device. All configurations for the device to access the memory through the IOMMU are written into the STE corresponding to the SID.

**Virtualization Scenario: -**

The STE stores the address translation process from the IOVA to the PA. To adapt to the memory access requirement in the virtualization scenario, the SMMU supports two stages of address translation in a similar way to extended page tables (EPTs). Stage 1 translates the VA into an intermediate physical address (IPA), and stage 2 translates the IPA into a PA.

**Non-virtualization Scenario: -**

In non-virtualization scenarios, if a device uses the IOVA to perform DMA through the IOMMU, only stage 1 address translation is required.

# Queues in SMMU driver: -

The SMMU driver offers three queues: -

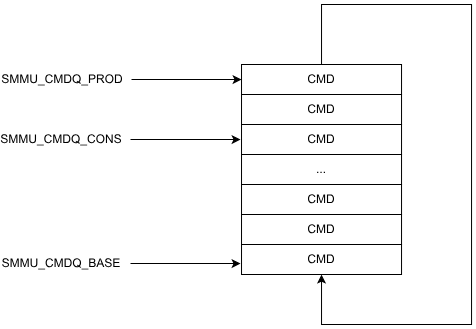
cmdq , evtq and priq

## SMMU Command Queue, cmdq: -

The command queue is used by the SMMU driver to send commands to hardware, such as refreshing the TLB and writing to CD.

The SMMU is controlled via a circular Command queue in memory. For example, when software changes an STE or a translation, it needs to invalidate the related caching in the SMMU. This can be done by issuing corresponding invalidation commands to the Command queue.

An SMMU that implements SMMUv3.3 can optionally support multiple Command queues for reducing contention between multiple PEs submitting Commands to the SMMU simultaneously.



## Event Queue, evtq: -

The event queue is used by the platform device mounted to the SMMU to send exception messages to the driver.

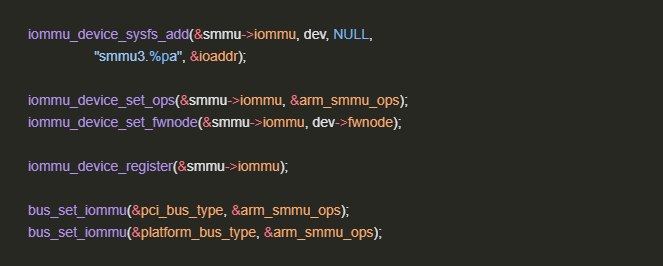
## Priority Queue, priq: -

The priority queue has similar functions to the event queue except that the priority queue is used by the mounted PCI device.

# Registering the SMMU with the IOMMU

IOMMU devices on different platforms have a unified IOMMU interface in the Linux kernel. During SMMU initialization, an smmu->iommu device node is registered in the **sys** directory, and arm\_smmu\_ops is registered with the device, system PCI bus, and platform device bus. In this way, when the IOMMU public interface is used, the functions provided by the SMMU are invoked. For details, see the implementation of various IOMMU interfaces provided in arm\_smmu\_ops.

File – drivers/iommu/arm/arm-smmu-v3/arm-smmu-v3.c



static struct iommu\_ops arm\_smmu\_ops = {

//...

.domain\_alloc\_paging = arm\_smmu\_domain\_alloc\_paging,

.domain\_alloc\_sva = arm\_smmu\_sva\_domain\_alloc,

.domain\_alloc\_user = arm\_smmu\_domain\_alloc\_user,

.probe\_device = arm\_smmu\_probe\_device,

.release\_device = arm\_smmu\_release\_device,

.device\_group = arm\_smmu\_device\_group,

//...

.owner = THIS\_MODULE,

.default\_domain\_ops = &(const struct iommu\_domain\_ops) {

.attach\_dev = arm\_smmu\_attach\_dev,

.set\_dev\_pasid = arm\_smmu\_s1\_set\_dev\_pasid,

.map\_pages = arm\_smmu\_map\_pages,

.unmap\_pages = arm\_smmu\_unmap\_pages,

.flush\_iotlb\_all = arm\_smmu\_flush\_iotlb\_all,

.iotlb\_sync = arm\_smmu\_iotlb\_sync,

.iova\_to\_phys = arm\_smmu\_iova\_to\_phys,

.enable\_nesting = arm\_smmu\_enable\_nesting,

.free = arm\_smmu\_domain\_free\_paging,

}

};

static int arm\_smmu\_device\_probe(struct platform\_device \*pdev)

{

//...

ret = iommu\_device\_sysfs\_add(&smmu->iommu, dev, NULL,

"smmu3.%pa", &ioaddr);

//...

ret = iommu\_device\_register(&smmu->iommu, &arm\_smmu\_ops, dev);

//...

return 0;

}

# IOMMU and DMA

A DMA capable device while doing DMA directly to physical memory can create attack. So, in such case if IOMMU is used, then it prevents device to directly access physical address. Therefore IOVA is generated.

When the peripheral performs DMA, only the IOVA needs to be used.

When the dma\_alloc series functions are called to allocate memory, the iommu\_dma\_alloc function is called. The iommu\_dma\_alloc function allocates the IOVA and actual physical memory, and uses iommu\_map to establish the mapping between the IOVA and the physical memory, that is, locates the STE corresponding to the device, locates the CD (generally CD 0), locates the corresponding page table in the memory, and writes the mapping between the IOVA and the physical memory to the page table. The page table operations related to the ARM SMMU are performed in io-pgtable.c.

## Function code flow in case of DMA mapping

Example: -

typedef u64 **dma\_addr\_t**;

dma\_addr\_t **dma\_map\_single** ( struct device \*dev, void \*cpu\_addr, size\_t size, enum dma\_data\_direction dir);

The **dma\_map\_single( )** API is used in Linux kernel programming to prepare a single buffer for DMA (Direct Memory Access) operations. This function maps a memory buffer so that a DMA-capable device can access it.

@Arg1: **dev**:

The device performing the DMA operation. It provides the context for the mapping.

The return type of **dma\_map\_single( )** is **dma\_addr\_t**.

* **dma\_addr\_t** is typically an unsigned integer type that represents the physical DMA address of the mapped buffer. This address is used by the device to access the memory.

**Code flow: -**

dma\_map\_single() -> dma\_map\_single\_attrs() -> dma\_map\_page\_attrs() -> iommu\_dma\_map\_page() -> \_\_iommu\_dma\_map() ->

\_\_iommu\_dma\_map(…)

{

//…

dma\_addr\_t iova;

iova = iommu\_dma\_alloc\_iova(…);

//…

iommu\_map();

//…

}

iommu\_map() -> \_\_iommu\_map() -> ops->map\_pages() -> arm\_smmu\_map\_pages();

**Code extract: -**

void \*iommu\_dma\_alloc(struct device \*dev, size\_t size, dma\_addr\_t \*handle,

gfp\_t gfp, unsigned long attrs)

{

//...

cpu\_addr = iommu\_dma\_alloc\_pages(dev, size, &page, gfp, attrs);

//...

\*handle = \_\_iommu\_dma\_map(dev, page\_to\_phys(page), size, ioprot,

dev->coherent\_dma\_mask);

//...

return cpu\_addr;

}

static dma\_addr\_t \_\_iommu\_dma\_map(struct device \*dev, phys\_addr\_t phys,

size\_t size, int prot, u64 dma\_mask)

{

//...

dma\_addr\_t iova;

//...

iova = iommu\_dma\_alloc\_iova(domain, size, dma\_mask, dev);

if (iommu\_map(domain, iova, phys - iova\_off, size, prot, GFP\_ATOMIC)) {

iommu\_dma\_free\_iova(cookie, iova, size, NULL);

return DMA\_MAPPING\_ERROR;

}

return iova + iova\_off;

}

int iommu\_map(struct iommu\_domain \*domain, unsigned long iova,

phys\_addr\_t paddr, size\_t size, int prot, gfp\_t gfp)

{

const struct iommu\_domain\_ops \*ops = domain->ops;

//…

ret = \_\_iommu\_map(domain, iova, paddr, size, prot, gfp);

if (ret == 0 && ops->iotlb\_sync\_map) {

ret = ops->iotlb\_sync\_map(domain, iova, size);

if (ret)

goto out\_err;

}

//...

return ret;

}

static int \_\_iommu\_map(struct iommu\_domain \*domain, unsigned long iova,

phys\_addr\_t paddr, size\_t size, int prot, gfp\_t gfp)

{

//...

pr\_debug("map: iova 0x%lx pa %pa size 0x%zx\n", iova, &paddr, size);

while (size) {

//...

pr\_debug("mapping: iova 0x%lx pa %pa pgsize 0x%zx count %zu\n",

iova, &paddr, pgsize, count);

ret = ops->map\_pages(domain, iova, paddr, pgsize, count, prot,

gfp, &mapped);

//...

}

//...

}

File - /drivers/iommu/arm/arm-smmu-v3/arm-smmu-v3.c

struct iommu\_ops arm\_smmu\_ops = {

//...

.owner = THIS\_MODULE,

.default\_domain\_ops = &(const struct iommu\_domain\_ops) {

.attach\_dev = arm\_smmu\_attach\_dev,

.set\_dev\_pasid = arm\_smmu\_s1\_set\_dev\_pasid,

.map\_pages = arm\_smmu\_map\_pages,

.unmap\_pages = arm\_smmu\_unmap\_pages,

//...

}

};

# References: -

1. <https://www.scribd.com/document/750859791/Learn-the-Architecture-Smmu-Software-Guide-109242-0100-01-En>
2. <https://altera-fpga.github.io/rel-24.2/linux-embedded/smmu/smmu/>
3. <https://www.openeuler.org/en/blog/wxggg/2020-11-21-iommu-smmu-intro.html>
4. SMMU spec

https://developer.arm.com/documentation/ihi0070/latest/