1 Interrupt

Before writing any interrupt program, you should keep the following points in mind.

1. Interrupt handlers **can not enter sleep**, so to avoid calls to some functions which has **sleep**.
2. When the interrupt handler has part of the code to enter the critical section, **use spinlocks lock, rather than mutexes**. Because if it can’t take mutex it will go to sleep until it takes the mute.
3. Interrupt handlers **can not exchange data with the userspace**.
4. The interrupt handlers **must be executed as soon as possible**. To ensure this, it is best to split the implementation into two parts, the top half and the bottom half. The top half of the handler will get the job done as soon as possible and then work late on the bottom half, which can be done with **softirq** or **tasklet** or **workqueue**.
5. Interrupt handlers **can not be called repeatedly**. When a handler is already executing, its corresponding IRQ must be disabled until the handler is done.
6. Interrupt handlers **can be interrupted by higher authority handlers**. If you want to avoid being interrupted by a highly qualified handler, you can mark the interrupt handler as a fast handler. However, if too many are marked as fast handlers, the performance of the system will be degraded because the interrupt latency will be longer.

LOgical nad virtual

Kernel logical addresses are mappings accessible to kernel code through normal CPU memory access functions. On 32-bit systems, only 4GB of kernel logical address space exists, even if more physical memory than that is in use. Logical address space backed by physical memory can be allocated with kmalloc.

Virtual addresses do not necessarily have corresponding logical addresses. You can allocate physical memory with vmalloc and get back a virtual address that has no corresponding logical address (on 32-bit systems with PAE, for example). You can then use kmap to assign a logical address to that virtual address

Simply speaking, virtual address would include "high memory", which doesn't do the 1:1 mapping for the physical address,if your RAM size is more than the address range of kernel(typically,For 1G/3G in X86,your RAM is 3G but your kernel addressing range is 1G) and also the address return from kmap() and vmalloc(), which requires the kernel to establish page table for the memory mapping. since logic address is always memory mapped by the kernel(1:1 mapping), you don't need to explicitly call kernel API,like set\_pte to set up the page table entry for the particular page.

so virtual address can't be logic address all the time.

DMA MEMMORY MAPPED AND STREAM

The **AXI Direct Memory Access (AXI DMA)** IP core facilitates efficient data movement between AXI4 memory-mapped interfaces (e.g., system memory) and AXI4-Stream interfaces (e.g., data processing IPs). The description provided outlines key features and operations. Below is an explanation and flow of its working:

### **AXI DMA: Key Features and Operations**

#### **1. MM2S and S2MM Channels (Independent Operation)**

* **MM2S (Memory-Mapped to Stream)**:
  + Transfers data from system memory (AXI4 memory-mapped interface) to an AXI4-Stream target IP.
  + Often used for sending input data to processing cores, e.g., video, audio, or signal processing.
* **S2MM (Stream to Memory-Mapped)**:
  + Transfers data from an AXI4-Stream source IP to system memory.
  + Often used to store processed data from cores back into memory.

Both channels operate independently, enabling simultaneous data transfers in different directions.

#### **2. Scatter/Gather (SG) Mode**

Scatter/Gather mode enables:

* Breaking large transfers into smaller buffer descriptors stored in system memory.
* Fetching and updating buffer descriptors dynamically using the **SG Read/Write Master interfaces**.
* Efficient handling of complex, non-contiguous memory regions.

#### **3. 4 KB Address Boundary Protection**

* Ensures that memory transfers don’t cross 4 KB boundaries.
* Prevents illegal or fragmented accesses that might corrupt data.

#### **4. Automatic Burst Mapping**

* Optimizes memory transactions by aligning burst sizes for efficient AXI4 utilization.
* Maximizes throughput by combining smaller transfers into bursts.

#### **5. Byte-Level Data Realignment**

* Allows DMA transfers to start at any byte offset, supporting unaligned data access.
* Ensures compatibility with memory layouts that don't follow strict alignment.

#### **6. AXI Control Stream and AXI Status Stream**

* **Control Stream (MM2S)**:
  + Sends user-defined control signals or additional metadata alongside the main data stream.
  + Useful for commands or configuration data sent to the target IP.
* **Status Stream (S2MM)**:
  + Reports status or metadata about the received stream data, such as errors, completion, or extra information.
  + Useful for monitoring and verifying transfer correctness.

### **Flow of Working**

Here’s the flow of operations for MM2S and S2MM with Scatter/Gather enabled:

#### **A. MM2S (Memory-Mapped to Stream) Channel**

1. **Buffer Descriptor Setup**:
   * User creates buffer descriptors in system memory.
   * Descriptors define data locations, sizes, and attributes for transfers.
2. **Scatter/Gather Engine**:
   * Fetches buffer descriptors via the SG Read Master interface.
   * Configures the DMA with the source address and transfer details.
3. **Memory Read**:
   * DMA reads data from the specified source memory address via the AXI4 Read Master interface.
4. **Stream Transfer**:
   * Data is sent to the target IP via the AXI4-Stream interface.
   * If configured, a control stream carries metadata alongside the main data.
5. **Status Update**:
   * DMA updates the buffer descriptor with the transfer status via the SG Write Master interface.

#### **B. S2MM (Stream to Memory-Mapped) Channel**

1. **Stream Data Reception**:
   * Data arrives at the AXI4-Stream interface from the source IP.
   * Optional: Status stream provides metadata for incoming data.
2. **Buffer Descriptor Setup**:
   * Scatter/Gather engine retrieves buffer descriptors specifying the destination memory address.
3. **Memory Write**:
   * Data is written to the specified memory location via the AXI4 Write Master interface.
4. **Status Update**:
   * DMA updates the buffer descriptor in system memory with completion details.

### **Diagram: MM2S and S2MM Data Flow**

MM2S (Memory-Mapped to Stream)

--------------------------------------------------------

System Memory --> SG Read Master --> DMA --> AXI4-Stream

(Buffer Descriptors) (Target IP)

S2MM (Stream to Memory-Mapped)

--------------------------------------------------------

AXI4-Stream --> DMA --> SG Write Master --> System Memory

(Source IP) (Buffer Descriptors)

### **Scatter/Gather Mode Overview**

* SG mode allows dynamic fetching and updating of buffer descriptors.
* Facilitates handling of large or scattered data efficiently.
* Ensures DMA doesn't require CPU intervention for every transfer.

### **Summary**

AXI DMA operates as a bridge between AXI4 memory-mapped and AXI4-Stream interfaces, supporting efficient data transfer with high throughput. MM2S and S2MM channels independently handle streaming data to/from memory, with features like scatter/gather, burst optimization, and byte realignment enhancing flexibility and performance.