How to Develop Embedded Software Using the QEMU Machine Emulator

Written by:

Apriorit Inc.

Author:

Artem Kotovsky,

Software Analyst,
Driver Development Team

www.apriorit.com

info@apriorit.com

How to Develop Embedded Software Using The QEMU Machine Emulator

What is it all about?

This e-book has been written for embedded software developers by Apriorit experts. It goes in-depth on how to save time when developing a Windows device driver by emulating a physical device with QEMU and explores the details of device driver emulation based on QEMU virtual devices.

Embedded devices are characterized by complex software that should provide stable and secure communication between operating systems and hardware. However, developing a device driver significantly increases the time to market for peripheral devices. Fortunately, virtualization technologies like QEMU allow developers to emulate a physical device and start software development before hardware is manufactured.

The QEMU machine emulator and visualizer also allow developers to securely test device drivers, find and fix defects which can crash the entire operating system. Developing and debugging drivers on an emulator makes working with them similar to working with user-space applications. At worst, bugs can lead to the emulator crashing.

In this e-book, we explain our approach to developing Windows drivers using a QEMU virtual device. You'll find out what are the benefits and limitations of device emulation for driver development and get a clear overview on how you can establish communication between a device and its driver.

The e-book includes detailed steps to create a virtual hardware device and develop a Windows driver for it. You'll discover how QEMU can be used for building running, testing, and debugging the whole environment and how embedded software can be developed for new virtual hardware even before a physical device becomes available.

We've been using QEMU virtual devices to facilitate embedded software development for quite a long time, so this approach has already confirmed its value and effectiveness.

Table of Contents

Introduction

Developing Windows device drivers and device firmware are difficult and interdependent processes. In this book, we consider how to speed up and improve device driver development from the earliest stages of the project, prior to or alongside the development of the device and its firmware.

To begin, let's consider the main stages of hardware and software development:

1. Setting objectives and analyzing requirements
2. Developing specifications
3. Testing the operability of the specifications
4. Developing the device and its firmware
5. Developing the device driver
6. Integrating software and hardware, debugging, and stabilizing

To speed up the time for driver development, we propose using a mock device that can be implemented in a QEMU virtual machine.

Why do we use QEMU?

QEMU has all the necessary infrastructure to quickly implement virtual devices. Additionally, QEMU provides an extensive list of APIs for device development and control. For a guest operating system, the interface of such a virtual device will be the same as for a real physical device. However, a QEMU virtual device is a mock device, most likely with limited functionality (depending on the device's capabilities) and will definitely be much slower than a real physical device.

Pros and cons of using a QEMU virtual device

Let's consider the pros and cons of this approach, beginning with the pros:

1. The driver and device are implemented independently and simultaneously, provided that there already is a device communication protocol.
2. You get proof of driver-device communication before implementing the device prototype. When implementing a QEMU virtual device and driver, you can test their specifications and find any issues in the device-driver communication protocol.

3. You can detect logical issues in the device communication specifications at early stages of development.
4. QEMU provides driver developers with a better understanding of the logic of a device's operation.
5. You can stabilize drivers faster due to simple device debugging in QEMU.
6. When integrating a driver with a device, you'll already have a fairly stable and debugged driver. Thus, integration will be faster.
7. Using unit tests written for the driver and QEMU device, you can iteratively check the specification requirements for a real physical device as you add functionality.
8. A QEMU virtual device can be used to automatically test a driver on different versions of Windows.
9. Using a QEMU virtual device, you can practice developing device drivers without a real device.

Now let's look at the cons of this approach:

1. It takes additional time to implement a QEMU virtual device, debug it, and stabilize it.
2. Since a QEMU virtual device isn't a real device but is only a mock device with limited capabilities, not all features can be implemented. However, it's enough to implement stubs for functionality.
3. A QEMU virtual device is much slower than a real physical device, so not everything can be tested on it. Particularly, it's impossible to test synchronization and boundary conditions that cause device failure.
4. Driver logic functionality can't be fully tested. Some parts remain to be finished during the device implementation stage.

Driver implementation stages

To understand when we can use a QEMU virtual device, let's consider the stages of driver implementation:

1. Developing device specifications and functionality, including the device communication protocol
2. Implementing a mock device in QEMU (implementing the real physical device can begin simultaneously)
3. Implementing the device and debugging it, including writing tests and providing the proof of driver-device communication
4. Integrating and debugging the driver when running on a real device

5. General bug fixing, changing the requirements and functionality of both the device and its driver
6. Releasing the device and its driver

For a Windows guest operating system, a virtual device will have all the same characteristics and interfaces as a real device because the driver will work identically with both the virtual device and the real device (aside from bugs in any of the components). However, the Windows guest operating system itself will be limited by the resources allocated by QEMU.

We've successfully tested this approach on Apriorit projects, confirming its value and effectiveness. Driver profiling can be used at early stages of working with a QEMU virtual device. This allows you to determine performance bottlenecks in driver code when working with high-performance devices (not all issues are possible to detect, however, because virtual device performance is several times slower). That's why it's essential to use Driver Verifier and the Windows Driver Frameworks (WDF) Verifier when developing any drivers for any environment.

Communication between a device and its driver

Let's consider how a peripheral component interconnect (PCI) device and its operating system driver communicate with each other. The PCI specification describes all possible channels of communication with a device, while the device PCI header indicates the resources necessary for communication and the operating system or BIOS allocates or initializes these specified resources. In this book, we discuss only two types of communication resources:

1. I/O address space
2. Interrupts

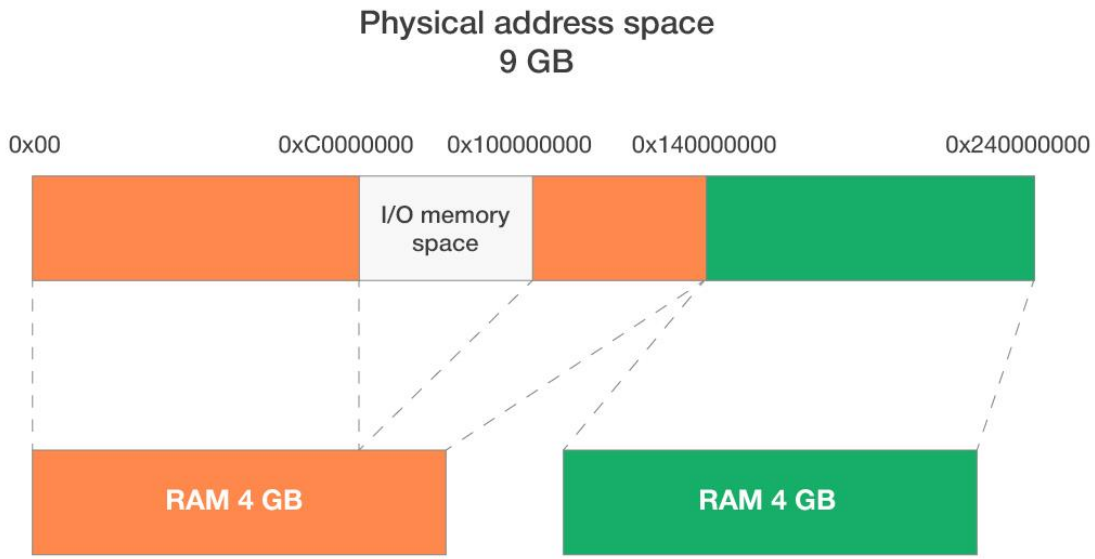
We'll take a brief look at these resources, discussing work with them only at the level on which they'll be used to implement communication functionality.

I/O address space

I/O address space is a region of addresses in a device (not necessarily on the physical memory of the device, but simply a region of the address space). When the operating system accesses these addresses, it generates a data access request (to read or write data) and sends it to the device. The device processes the request and sends its response. Access to the I/O address

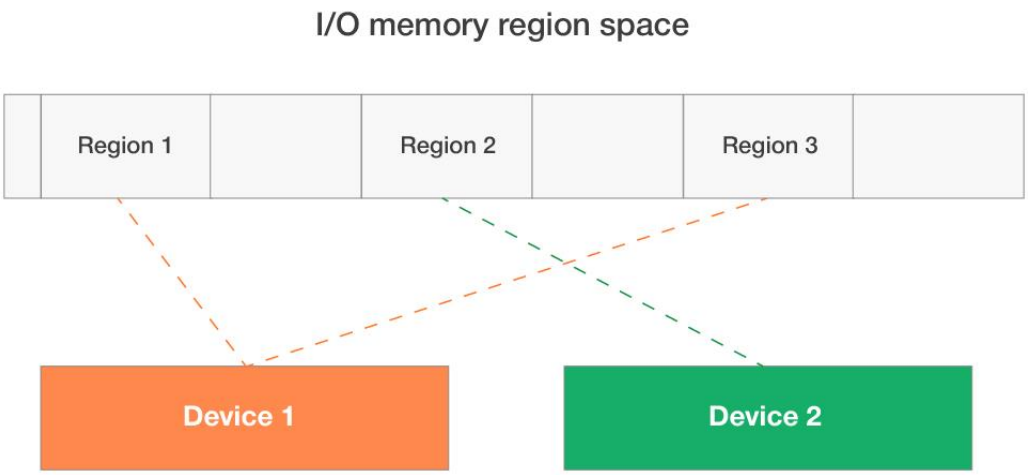
space in the Windows operating system is performed through the WRITE_REGISTER_ * and READ_REGISTER_ * function families, provided that the data size is 1, 2, 4, or 8 bytes. There are also functions that read an array of elements, where the size of one element is 1, 2, 4, or 8 bytes, and allow you to read or write buffers of any data size in one call.

The operating system and BIOS is responsible for allocating and assigning address regions to a device in the I/O address space. The system allocates these addresses from a special physical address space depending on the address dimension requirements. This additional level of abstraction of the device resource initialization eliminates device resource conflicts and relocates the device I/O address space in runtime. Here's an illustration of the physical address space for a hypothetical system:



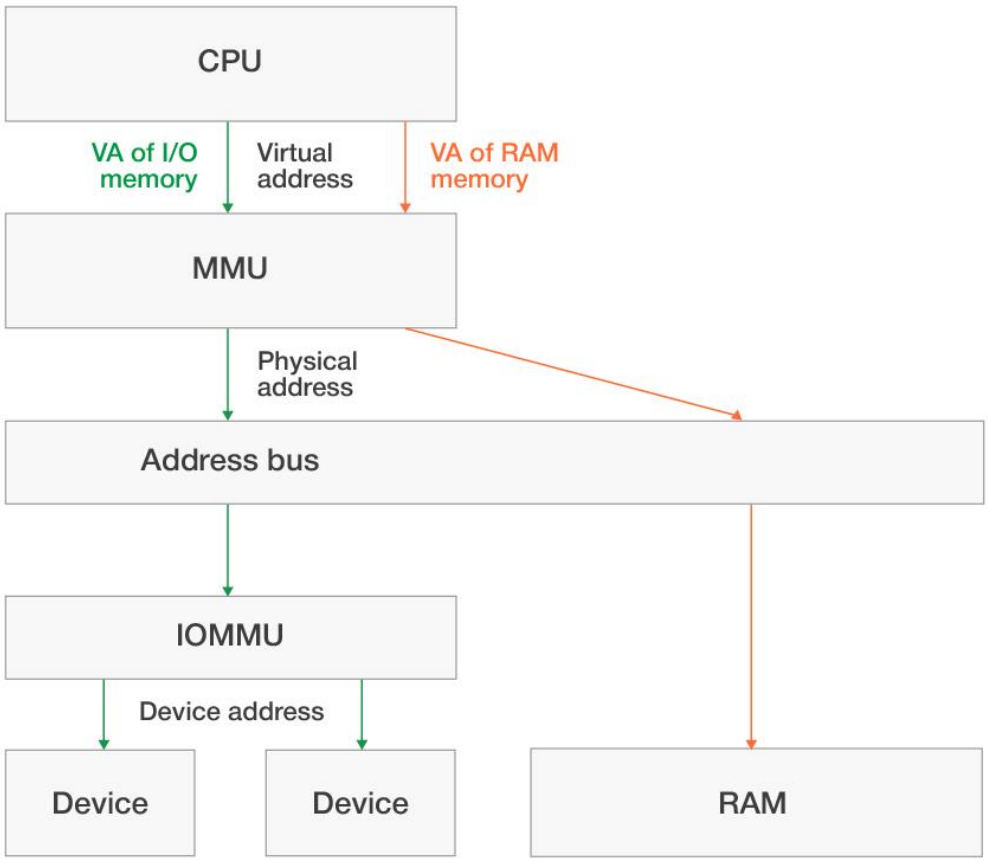
Physical address space: 0x00000000 – 0x23FFFFFF
I/O address space: 0xC0000000 – 0xFFFFFFFF

The operating system reserves a special I/O memory region of various size in the physical address space. This region is usually located within a 4GB address space and ends with 0xFFFFFFFF. This region doesn't belong to RAM memory, but is responsible for accessing the device address space.



Device 1 uses two I/O regions; device 2 uses one I/O region.

A kernel mode driver in Windows OS cannot directly access physical memory addresses. To access the I/O region, a driver needs to map this region to the kernel virtual address space with the special functions *MmMapIoSpace* and *MmMapIoSpaceEx*. These operating system functions return a virtual system address, which is consequently used in the functions of the WRITE_REGISTER_ * and READ_REGISTER_ * families. Schematically, access to the I/O address space looks like this:



RAM isn't used for handling the requests on accessing the virtual I/O address in device memory.

Now let's look at how to use this mechanism for communication with the device.

A driver developer considers the memory of a virtual QEMU device the device memory. The driver can read this memory to obtain information from the device or write to this memory to configure the device and send commands to it.
There's some magic in working with this type of memory, as the device immediately detects changes to it and responds by executing the required operations. For example, to make the device execute any command, it's sufficient to write it to the I/O memory at a certain offset. After this, the device will immediately detect changes in its memory and begin executing the command.

However, this type of memory isn't suitable for transferring large volumes of data due to the following limitations:

- The size of the I/O space is limited.
- Accessing this type of memory is usually slower than accessing RAM.
- The device must contain a comparable amount of internal memory.
- While accessing the I/O space, the CPU performs all required operations, slowing down the performance of the entire system when large volumes of memory are processed.

But such memory can be used to obtain statuses, configure device modes, and do anything else that doesn't require large amounts of memory.
This's a one-way communication mechanism: the driver can access the device memory at any time and the request will be delivered immediately, but the device can't deliver a message to the driver asynchronously by using the I/O memory without constantly polling the device memory from the driver's side.

Interrupts

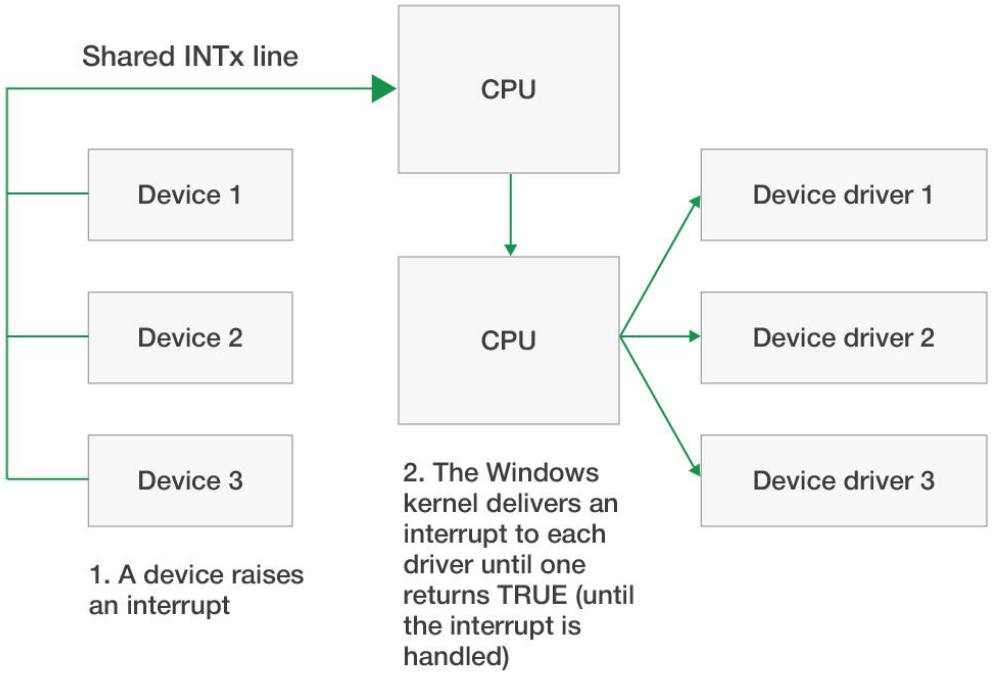
Interrupts are a special hardware mechanism with which a PCI device sends messages to the operating system when it requires the driver's attention or wants to report an event.
A device's ability to work with interrupts is indicated in the PCI configuration space.
There are three types of interrupts:

1. Line-based
2. Message-signaled
3. MSI-X

In this book, we discuss the first two, as we use them for establishing communication between a device and its driver. All these types of interrupts are also well described in other books and articles.

Line-based interrupts

Line-based interrupts (or INTx) are the first type of interrupt that's supported by all versions of Windows. These interrupts can be shared by several devices, meaning that one interrupt line can serve multiple PCI devices simultaneously. When any of these devices use a dedicated pin to trigger an interrupt, the operating system delivers that interrupt to each device driver in succession until one of them handles it.

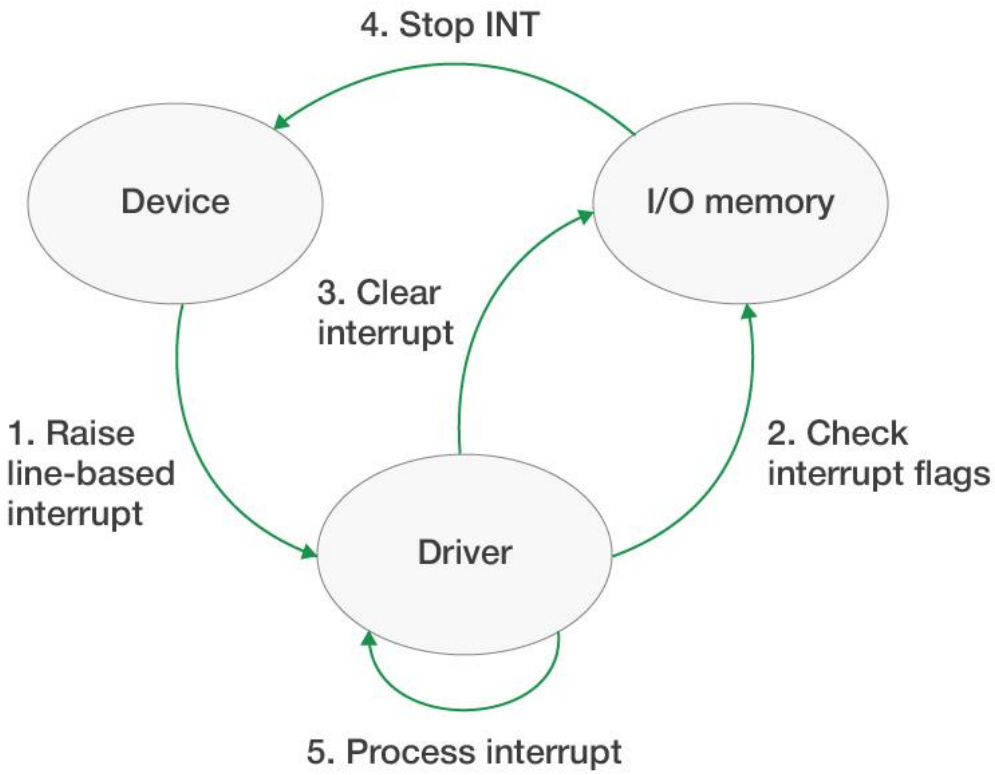


The driver, in turn, requires a mechanism that can determine whether this interrupt was actually raised by its device or came from another device that uses the same INTx line. The device's I/O memory space may contain an interrupt flag, which if set indicates that the interrupt has been raised by this particular device.

Physically, a line-based interrupt is a special contact to which the device sends a signal until the interrupt is processed by the driver. Thus, the driver must not only check the interrupt flag in the I/O memory but also reset it as soon as possible in order to let the device stop sending a signal to the interrupt contact.

Verifying and clearing the interrupt flag is necessary because several devices can simultaneously raise an interrupt using the same INTx. This approach allows processing interrupts from all devices.

The whole process of handling line-based interrupts looks as follows:



Line-based interrupts are full of flaws and limitations and require unnecessary references to the I/O memory. Fortunately, all these problems are solved with the following interrupt technique.

Message-signaled interrupts

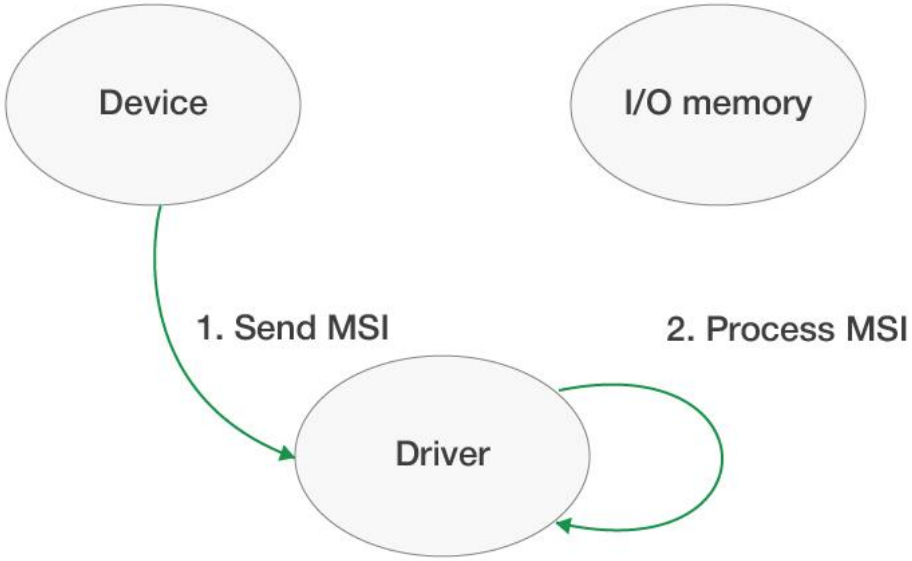
Message-signaled interrupts, or MSIs, are based on messages recorded by the device at a specific address. In other words, instead of maintaining the voltage on the interrupt line, the interrupt is sent simply by writing a few bytes to special memory. MSIs have many advantages compared to line-based interrupts. Improved performance is the major one, as this type of interrupt is much easier and cheaper to handle. MSIs also can be assigned to a specific core number.

The major difference between handling MSIs and line-based interrupts in the driver is that MSIs aren't shared. For instance, if the operating system allocates an MSI interrupt for a device, then this interrupt is guaranteed to be used only by this device provided that all devices in the system work correctly. Because of this, the driver no longer needs to check the interrupt flag in the device I/O space, and the device doesn't need to wait for the driver to process the interrupt.

The operating system can allocate only one line-based interrupt but multiple MSIs for a single device function (see the PCI function number). A driver can **request** the operating system to allocate 1, 2, 4, 8, 16, or 32 MSIs. In this case, the device can send different types of messages to the driver, which allows developers to optimize driver code and interrupt handling.

Each MSI contains information about the message number (the interrupt vector, or the logical type of event on the device's side). All MSI message numbers start with 0 in WDF. After the operating system allocates MSIs for a device, it records the number of interrupts allocated and all the information necessary for sending them to the PCI configuration space. The device uses this information to send different types of MSI messages. If the device is expecting 8 MSIs but the operating system allocates only one message, then the device should send only MSI number 0. At the physical level, the operating system tries to allocate the number of sequential interrupt vectors that were requested by the driver (1, 2, 4, 8, 16, 32) and sets the first interrupt vector number in the PCI configuration space. The device uses this vector as the base for sending different MSI messages.

When a request is sent by a device to allocate the necessary number of interrupts, the operating system will allocate the requested number only if there are free resources. If the operating system is unable to process this request, then it will allocate only one MSI message, which will be number 0. The device and device driver must be ready for this event. Schematically, MSI interrupt processing looks like this:



MSIs are available beginning with Windows Vista, and the maximum number of MSIs supported by Vista is 16. In earlier Windows versions, it was necessary to use line-based interrupts, and because of these drivers must support three modes of interrupt handling:

1. Line-based interrupt (if the system doesn't support MSIs)
2. One MSI interrupt (if the system can't allocate more than one MSI)
3. Multiple MSIs (if the system can allocate all requested MSIs and more than one is requested)

Interrupts are also a one-way communication instrument. They're used by a device to send notifications to the driver. At the same time, interrupts received from devices have the highest priority for the operating system. When an interrupt is received, the system interrupts the execution of one of the processor threads and calls the driver interrupt handler, or interrupt service routine (ISR), callback.

Working with DMA memory

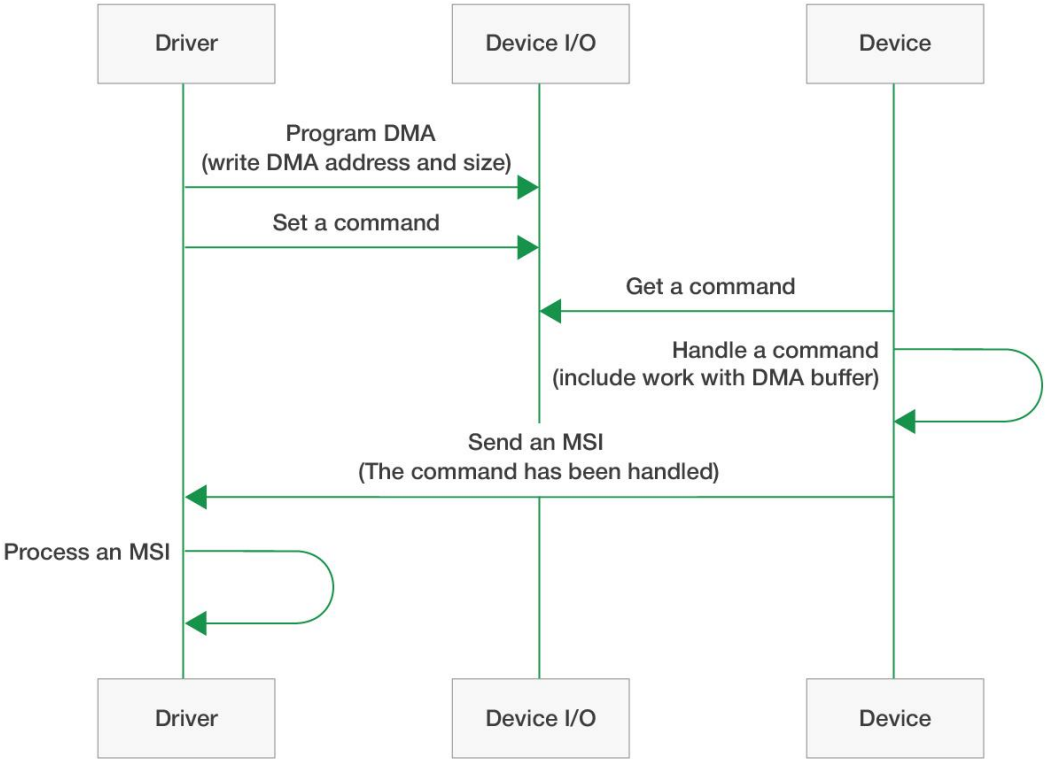
Some PCI devices need to exchange large volumes of data with the driver (for example, audio, video, network, and disk devices). It's not the best option to use I/O memory for these purposes because the processor will be directly involved in copying data, which slows down the entire system.

The direct memory access (DMA) mechanism is used to avoid utilizing the processor when transferring data between a driver and a device. DMA has several operating modes and

selects among them depending on which a device supports. Let's take a look at only one of them: bus mastering.

Bus mastering

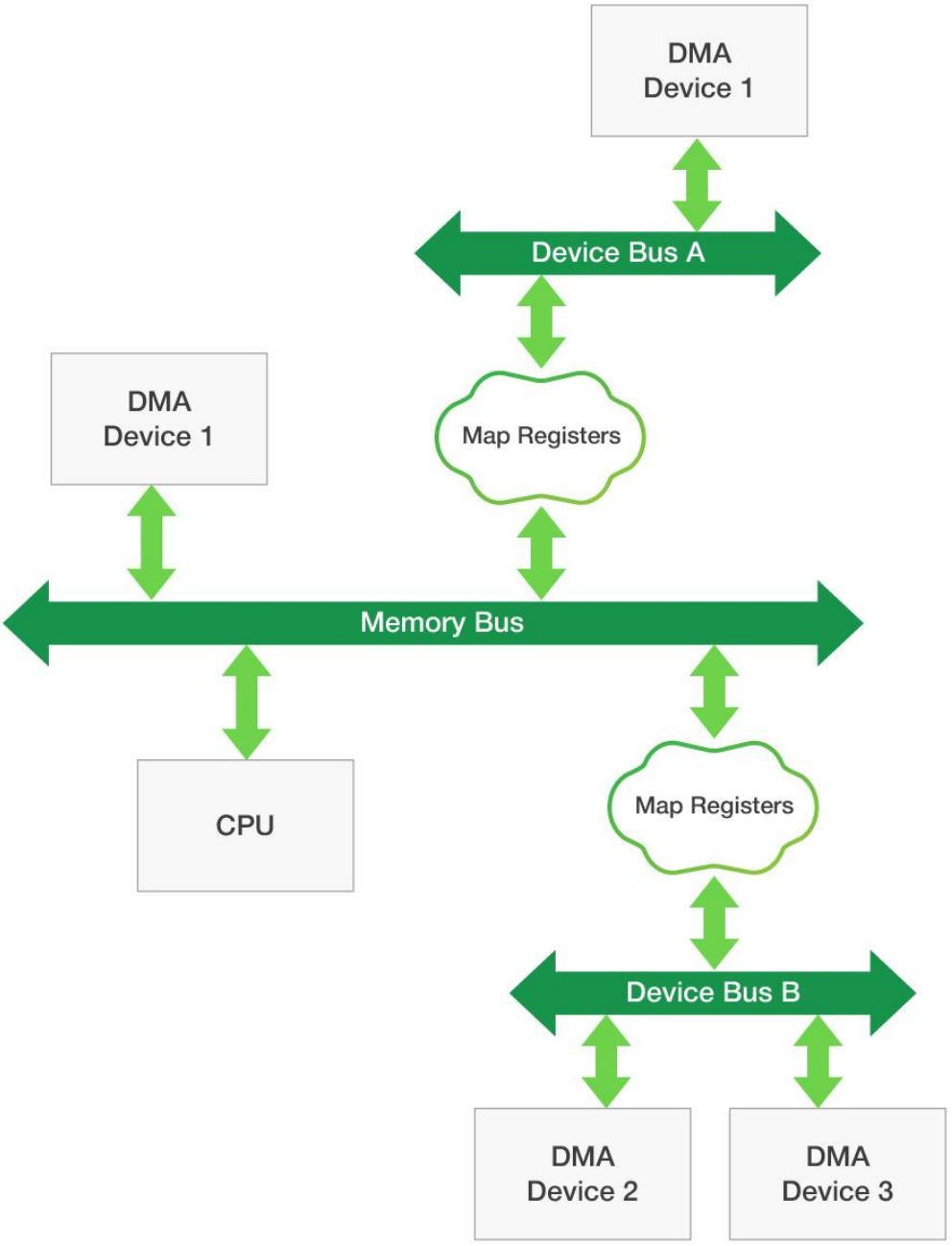
Devices with bus mastering support writing to physical memory (RAM) without using a processor. In this case, the device itself locks the address bus, sets the desired memory address, and writes or reads data. Using this mode, it's sufficient for the driver to transfer the DMA memory buffer address to the device (for example, using I/O memory) and wait for it to complete the operation (wait for the interrupt).



The actual device address should be transferred to the device instead of the virtual address that's typically used by programs and drivers. There's plenty of information about virtual, physical, and device addresses and how the operating system works with them on the internet. To work with DMA, it's enough to know the following:

1. The virtual address buffer can usually be described by two values: address and size.
2. The operating system and processor handle the memory pages rather than individual bytes, and the size of one memory page is 4KB. This has to be taken into account when working with physical pages.

3. Physical memory (RAM) can be *paged* or *non-paged*. Paged memory can be paged out to the pagefile (swap file), while non-paged memory is always located in RAM and its physical address doesn't change.
4. The physical pages of RAM for some virtual memory buffers (if the buffer wasn't allocated in a special way) aren't usually arranged one after another, meaning they aren't located in the continuous physical address space.
5. The physical RAM address and device address aren't always the same. The actual device address, which is the address accessible by the device, must be transferred to the device (we'll use the term *device address* to refer to both the device and physical address unless otherwise specified). To obtain the device address, the operating system provides a special API, while Windows Driver Frameworks uses its own API.



Considering how physical and device memory works, the driver needs to perform some additional actions to transfer DMA memory to the device. Let's take a look at how it's possible to transfer a user mode memory buffer to the device for DMA operations.

1. The memory utilized in user mode usually contains paged physical pages; therefore, such memory should be fixed in RAM (to make it non-paged). This will ensure that physical pages aren't unloaded into the pagefile while the device is working with them.
2. Physical pages may be located outside the contiguous physical memory range, making it necessary to obtain a device address for each of the pages or every continuous region with region size.
3. After that, all acquired device memory addresses should be transferred to the device. In order to maintain the same address format for the memory page, we'll use a **64-bit address** for both the x86 and x64 versions of Windows.

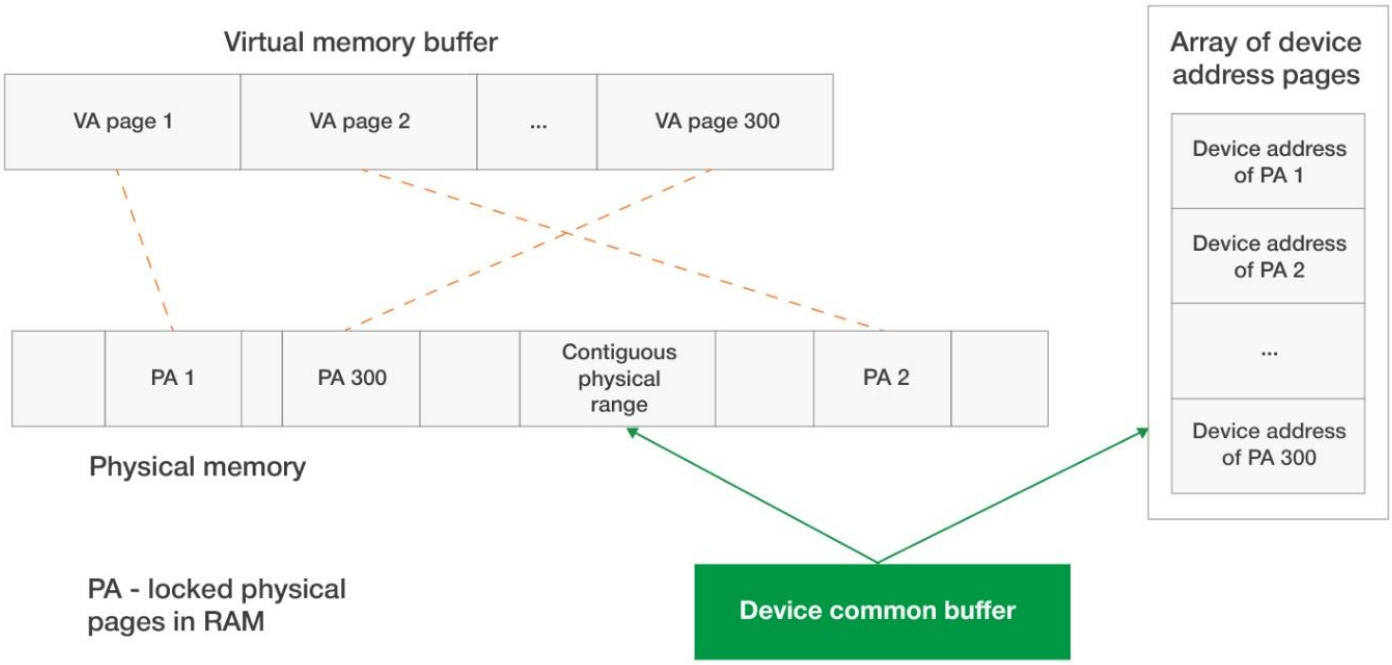
Note that the physical address for 32-bit Windows doesn't equal 32 bits because there's a Physical Address Extension (PAE), and Windows 64-bit uses only 44 bits for the physical address, which allows addressing 2^{44} = 16TB of physical memory. At the same time, the first 12 bits describe the offset in the current memory page (the address of one page of physical memory in Windows can be set by using only 44 - 12 = 32 bits).

To simplify our implementation, we won't wrap the addresses. Each memory page will be described by an address of 64 bits, both for x86 and x64 versions of the driver.

There are two ways to transfer addresses of all pages or regions to the device:

a. Using the I/O memory. In this case, the device must contain enough memory to store the entire array of addresses. The size of the I/O memory is usually fixed, adding some restrictions on the maximum size of the DMA buffer.
b. Using a common buffer as an additional memory buffer that contains page addresses. If the physical memory of this additional buffer is located in continuous physical or device memory, it will be enough to transfer just the address of the beginning of the buffer and its size to the device. The device can work with this memory as with a regular data array.

Both approaches are used, and each has its pros and cons. Let's consider approach *b*. Windows has a family of special functions used to allocate contiguous device memory (or a common buffer). Schematically, the user mode buffer transferred to the device for DMA operation looks like this:



Windows Driver Frameworks offers a family of functions for working with DMA memory, and only these particular functions should be used. This set of functions takes into account device capabilities, performs the necessary work to provide access to memory from the driver and device side, configures the mapped register, and so on.

The same memory can have three different types of addresses:

1. A virtual address for accessing the memory from the driver or a user mode process.
2. A physical address in RAM.
3. A device address (local bus address, DMA address) to access the memory from the device.

These mechanisms for communicating with the device will be enough to implement a test driver in Windows. All these mechanisms are reviewed here briefly and are described only to simplify the understanding of the device specifications listed below.

Test device specifications

Before starting to implement a QEMU virtual device or a Windows device driver, it's necessary to determine device functionality and the communication protocol between the device and its driver.