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:

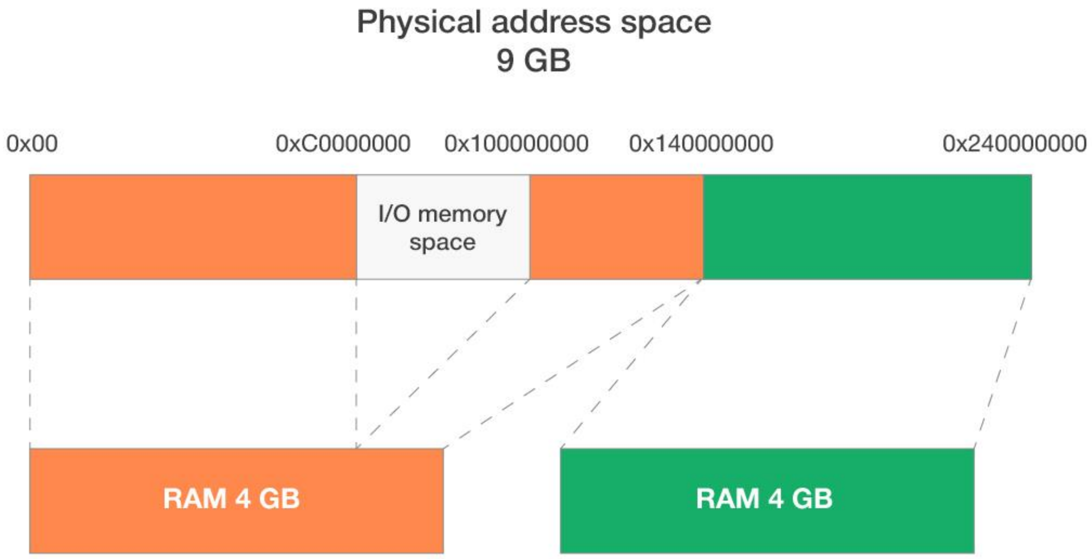
1. I/O address space
2. Interrupts

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

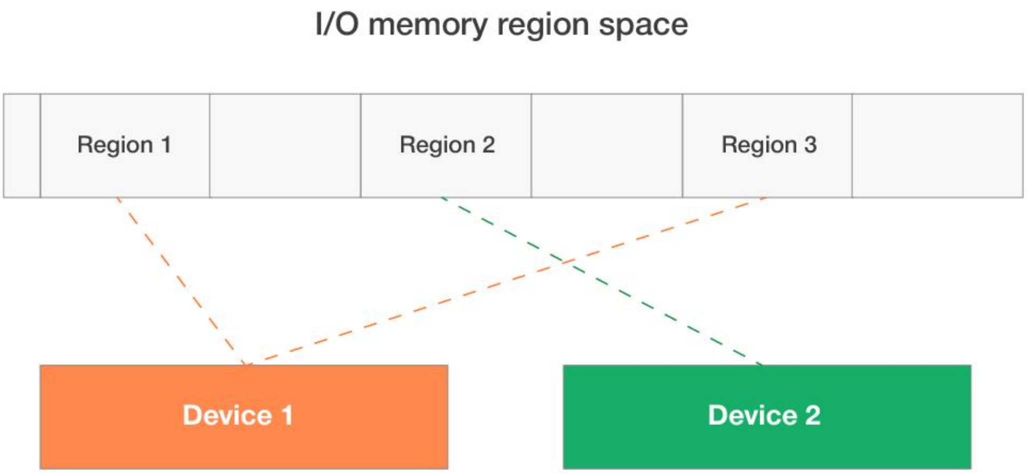
I/O address space is a region of addresses in a device (not necessarily on the physical memory of the device, but simply a region of the address space). When the operating system accesses these addresses, it generates a data access request (to read or write data) and sends it to the device. The device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



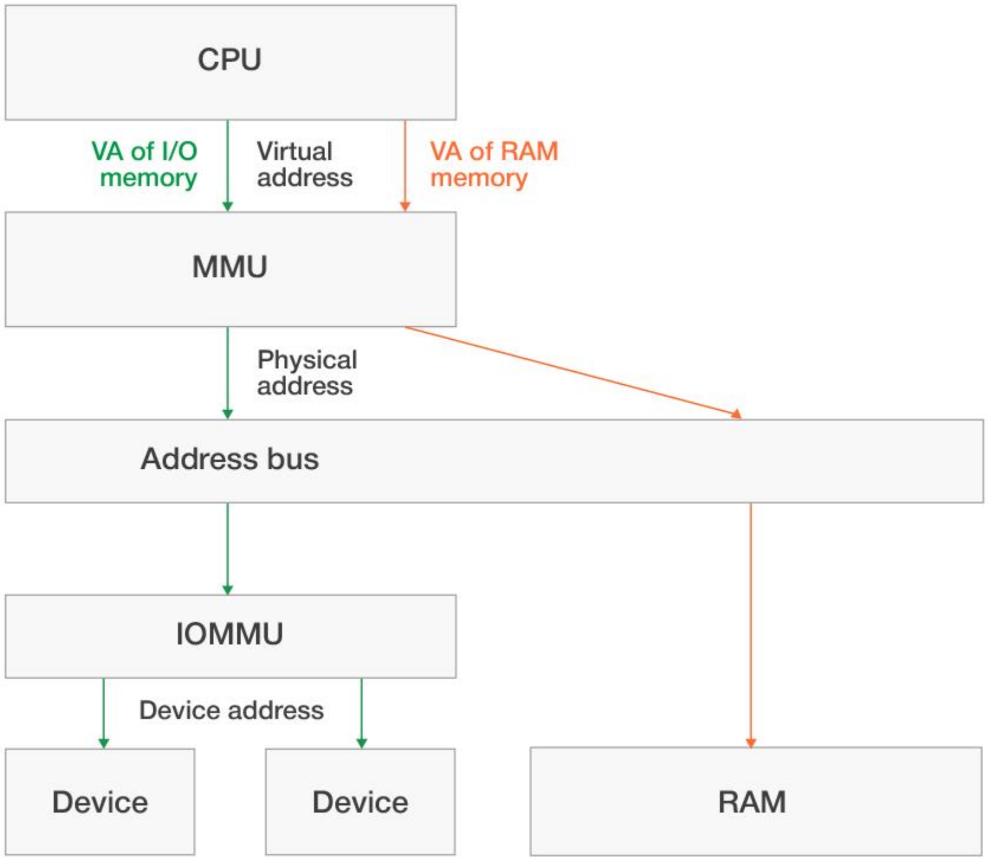
Physical address space: 0x00000000 – 0x23FFFFFFF
I/O address space: 0xC0000000 – 0xFFFFFFFF

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

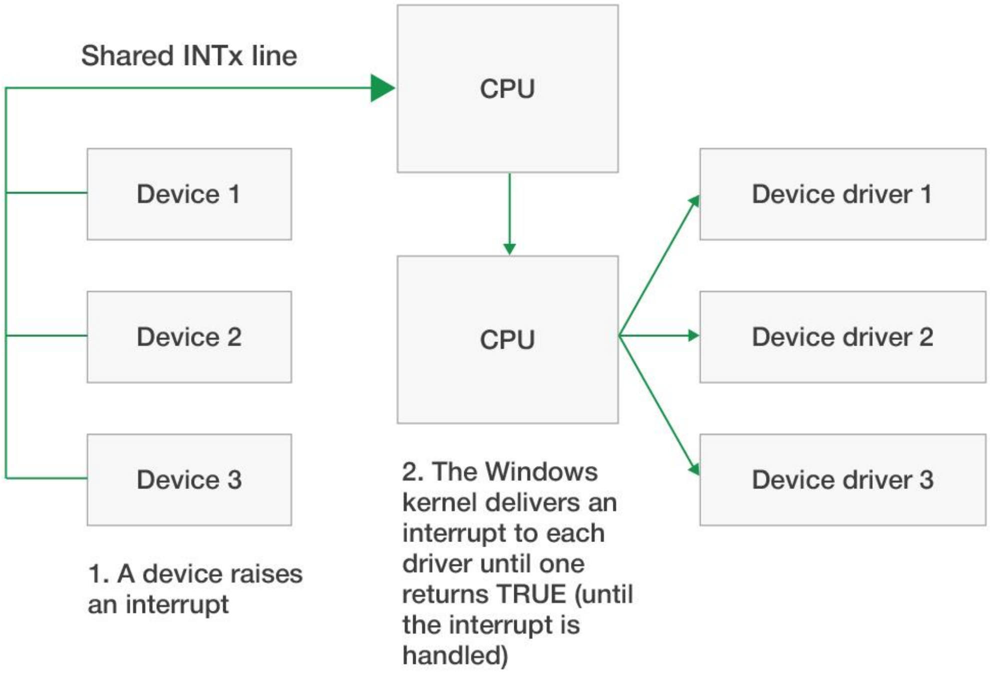
Interrupts are a special hardware mechanism with which a PCI device sends messages to the operating system when it requires the driver’s attention or wants to report an event.
A device’s ability to work with interrupts is indicated in the PCI configuration space.
There are three types of interrupts:

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.

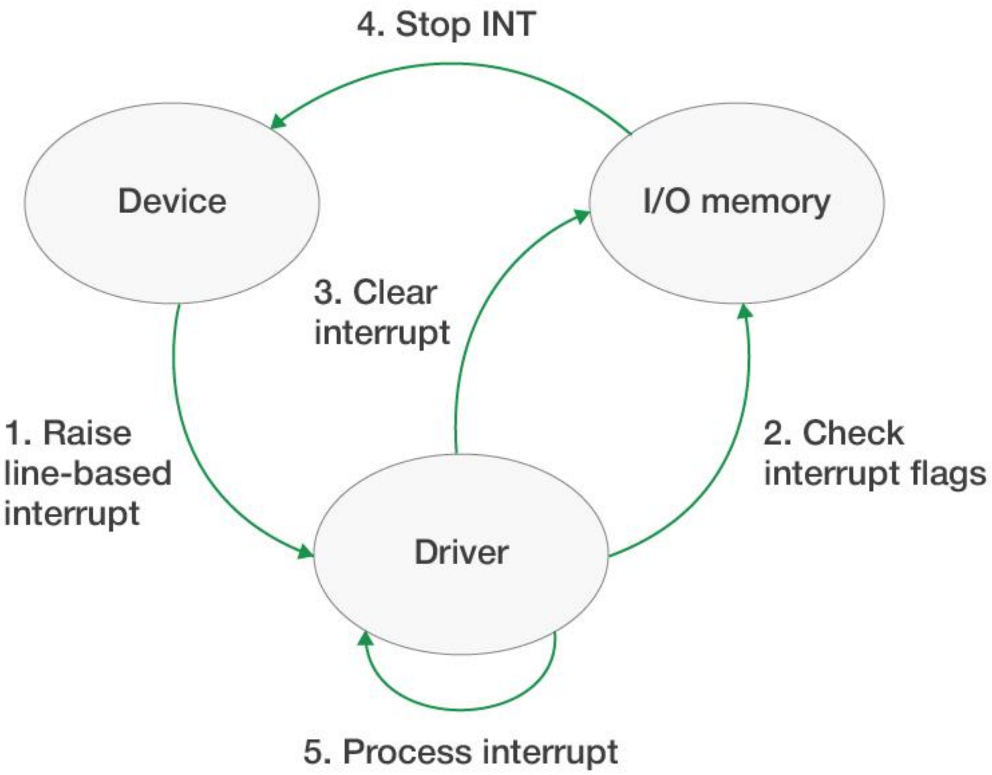

Shared INTx line
CPU
Device 1
Device 2
Device 3
CPU
Device driver 1
Device driver 2
Device driver 3
1. A device raises an interrupt
2. The Windows kernel delivers an interrupt to each driver until one returns TRUE (until the interrupt is handled)

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

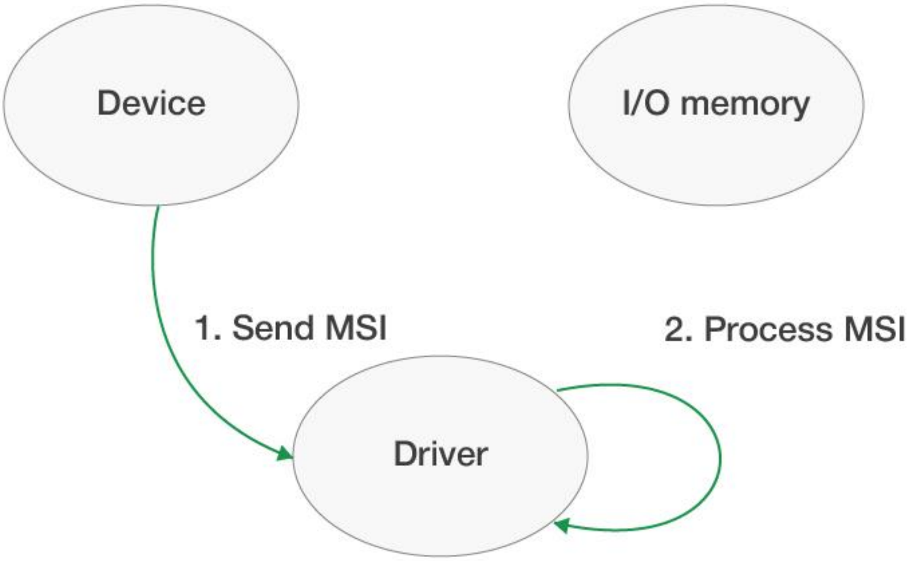
Message-signaled interrupts, or MSIs, are based on messages recorded by the device at a specific address. In other words, instead of maintaining the voltage on the interrupt line, the interrupt is sent simply by writing a few bytes to special memory. MSIs have many advantages compared to line-based interrupts. 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.

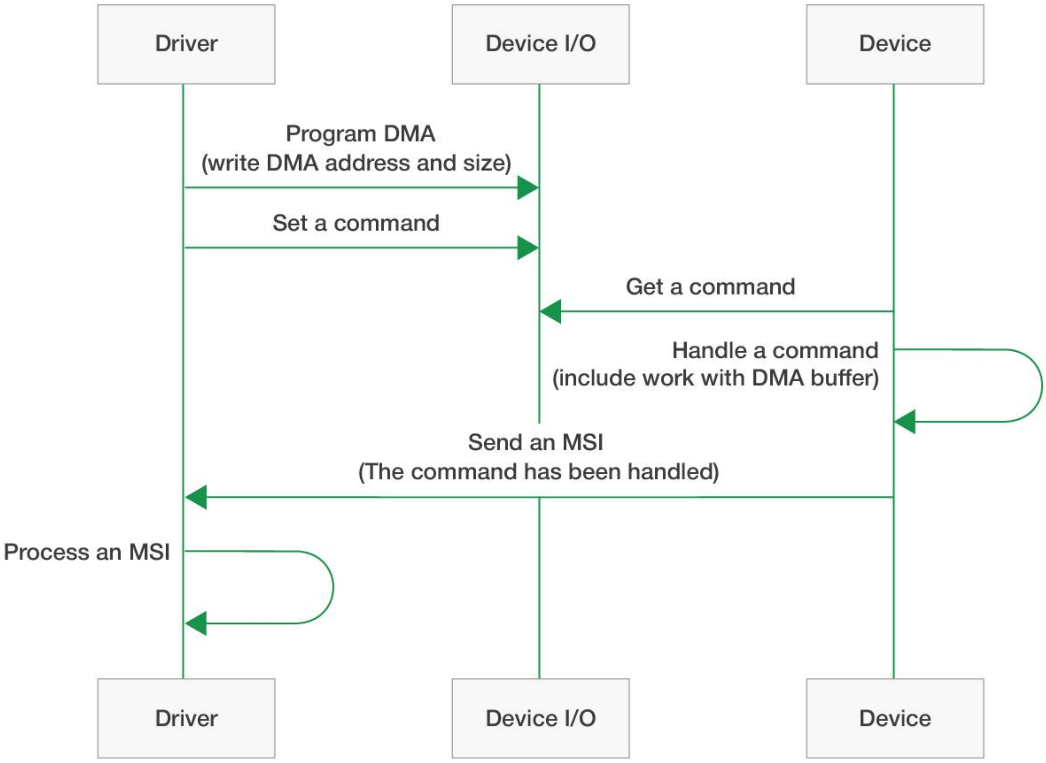
Working with DMA memory

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

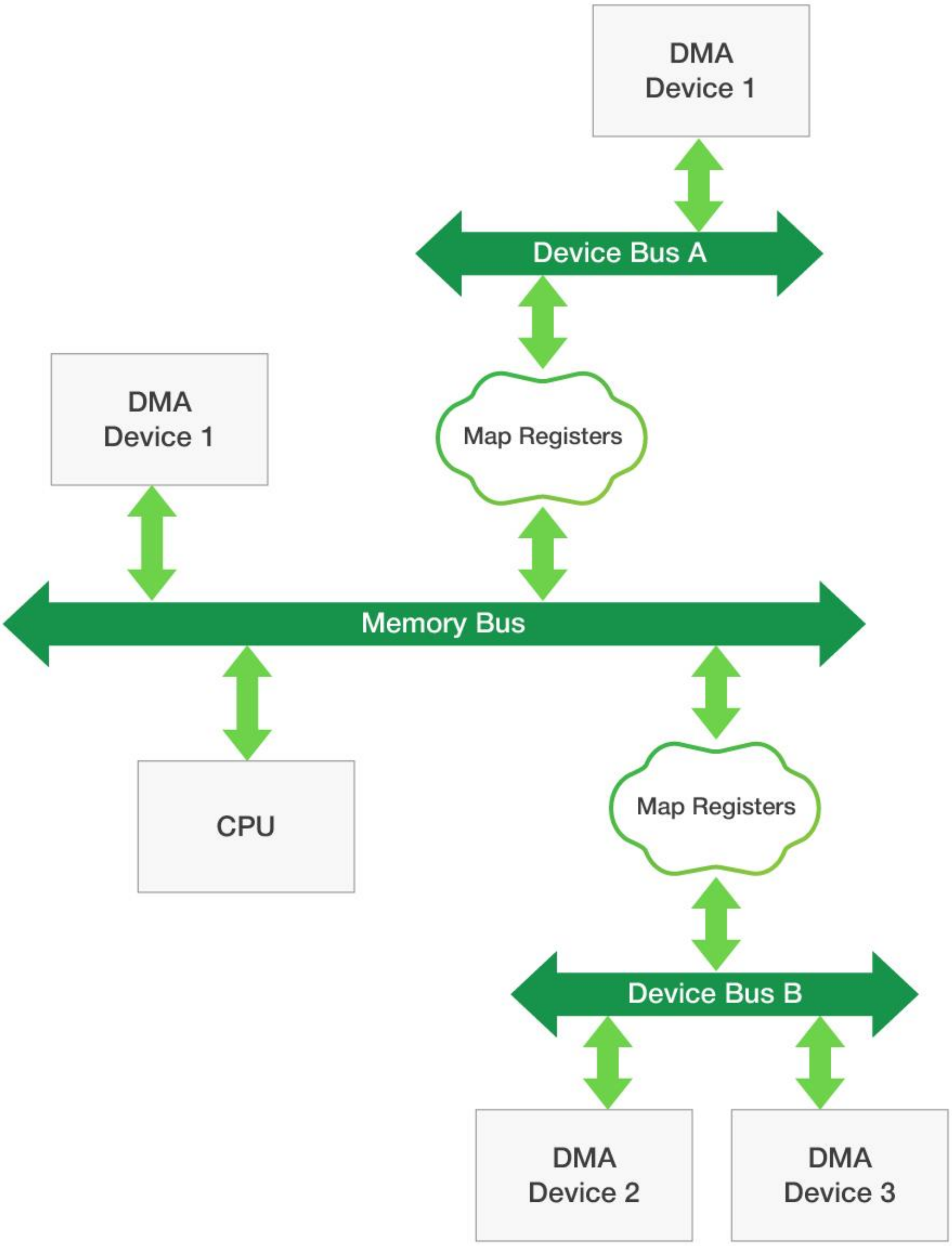
Devices with bus mastering support writing to physical memory (RAM) without using a processor. In this case, the device itself locks the address bus, sets the desired memory address, and writes or reads data. Using this mode, it's sufficient for the driver to transfer the DMA memory buffer address to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

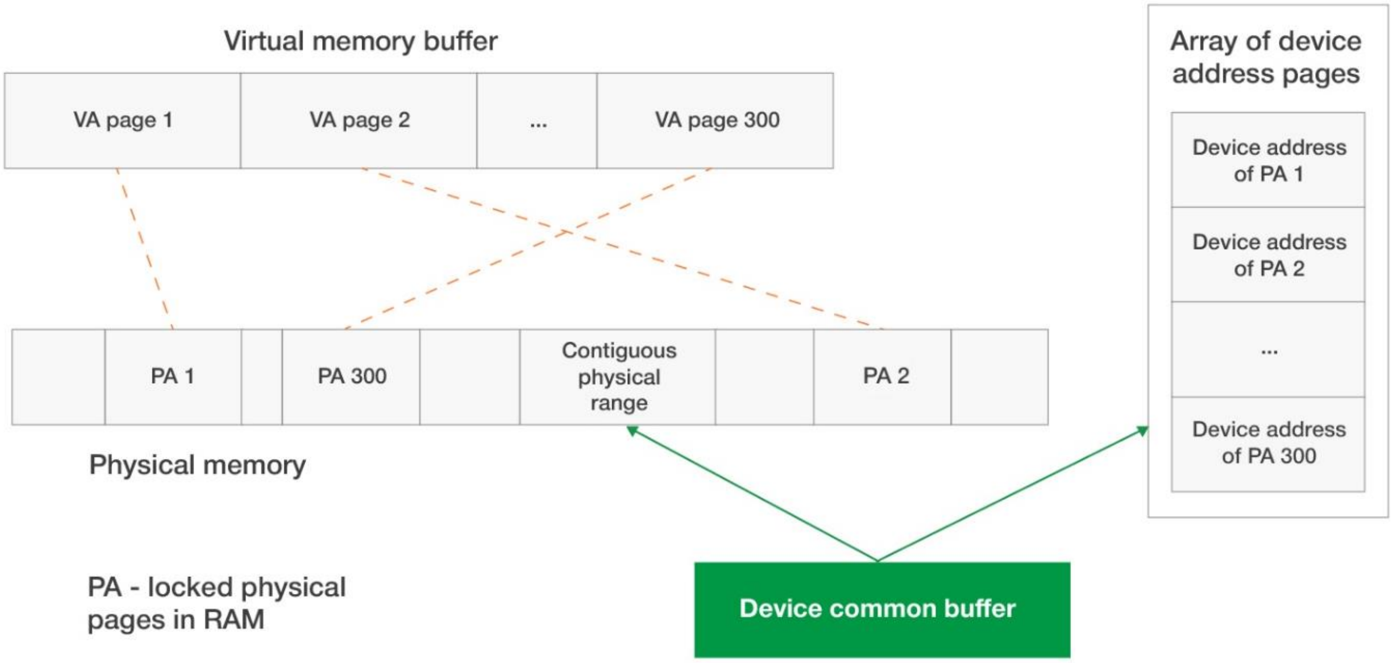
Note that the physical address for 32-bit Windows doesn't equal 32 bits because there's a Physical Address Extension (PAE), and Windows 64-bit uses only 44 bits for the physical address, which allows addressing 2^{44} = 16TB of physical memory. At the same time, the first 12 bits describe the offset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

Our test device will be very simple, with two hardware features:

1. Encryption and decryption of memory transmitted by DMA using the AES CBC algorithm.
2. Calculating the SHA256 hash of the transmitted DMA memory.

The device will support:

1. 64-bit DMA bus mastering
2. INTx and MSIs
3. Processing data and requests in one thread (it can process only one request/command at a time)

Though all features are available, the device won't support the following to make its implementation simpler:

1. Data streaming
2. Sessions (for example, it's not possible to continue calculating the SHA256 hash for the next DMA request)
3. Low power states

The key for AES will be considered hardware-based, in other words integrated into the device hardware.

The device resources are the following:

- 4KB I/O memory
- MSIs

Structure of the device I/O memory

Name	Offset	Size	Device access	Driver access	Value
ErrorCode	0x00	1 byte	Write Only	Read Only	0 – No error (default) 1 – DMA error 2 – Reset error 3 – I/O logic error 4 – Internal error

Name	Offset	Size	Device access	Driver access	Value
State	0x01	1 byte	Write Only	Read Only	0 – Device is ready (default) 1 – Working on reset command 2 – Working on AES CBC command 3 – Working on SHA-2 command
Command	0x02	1 byte	Read Only	Write Only	0 – Idle (default) 1 – Perform a reset 2 – Perform AES CBC mode encryption 3 – Perform AES CBC mode decryption 4 – Calculate SHA-2 hash
Interrupt Flag	0x03	1 byte	Read Only	Write Only	0x00 – Disable interrupts (initial value) 0xFF – Enable interrupts
DMA Buf IN Address	0x04	4 bytes	Read Only	Write Only	Address of contiguous physical memory with array of addresses of physical pages
DMA Buf IN Page Count	0x08	4 bytes	Read Only	Write Only	Number of pages in the DMA buffer IN address
DMA Buf IN Size in Bytes	0x0C	4 bytes	Read Only	Write Only	Size of the IN buffer in bytes
DMA Buf OUT Address	0x10	4 bytes	Read Only	Write Only	Address of contiguous physical memory with array of addresses of physical pages
DMA Buf OUT Page Count	0x14	4 bytes	Read Only	Write Only	Number of pages in the DMA buffer OUT address
DMA Buf OUT Size in Bytes	0x18	4 bytes	Read Only	Write Only	Size of the OUT buffer in bytes

Name	Offset	Size	Device access	Driver access	Value
MSI #1 Error	0x1C	1 byte	Write Only	Read and Write	Interrupt flag for Error event
MSI #2 Ready	0x1D	1 byte	WO	RW	Interrupt flag for Ready event
MSI #3 Reset	0x1E	1 byte	WO	RW	Interrupt flag for Reset event
Unused	0x1F	1 byte	-	-	Reserved for alignment

1. ErrorCode

This flag indicates the internal state of the device. Currently, the device supports four errors and can potentially report 255 errors while using a single MSI interrupt.

0 – The device is fully functional; there are no errors.
1 – There's a logical error while working with DMA memory. This only occurs with logical errors on the driver's side, which will help us debug and prevent damage to the physical operating system memory.
2 – The device can't perform a reset operation.
3 – An invalid request has been sent to the I/O memory. This is a logical error on the driver's side.
4 – There's an internal error in the device.

2. State

The current state of the device. This field is used to debug and obtain information about the state of the device.

0 – The device is idle and ready to process new requests.
1 – The device is processing a Reset request.
2 – The device is processing an AES CBC request in the process of data encryption or decryption.
3 – The device is processing an SHA-2 request by calculating the hash.

3. Command

This field is used for commands sent to the device. In our case, we support 255 commands, of which only three are used.

Perform a reset – The device should interrupt all its current operations, stop working with the DMA memory, and bring the internal state to the default. This command is available at any time and ensures that the device will no longer use the DMA memory upon successful execution of the command. After executing this command, the device should immediately increase the MSI #3 Reset counter and then generate MSI #3.

Perform AES CBC encryption – The device encrypts the DMA Buf IN memory with the DMA Size IN size and places the result in the DMA Buf OUT, taking into account the size of DMA Size OUT.

Perform AES CBC decryption – The device decodes the memory from the DMA Buf IN with the size of DMA Size IN and places the result in the DMA Buf OUT, taking into account the size of DMA Buf OUT.

Calculate SHA-2 hash – The device calculates the SHA256 memory from the DMA Buf IN with the size of DMA Size IN and places the result in DMA Buf OUT, taking into account the size of DMA Buf OUT.

If any of these operations is successfully completed, the device will increase the MSI #2 Ready counter and then generate MSI #2.

In case of an error, the device will set the value in the ErrorCode, increase the MSI #1 Error counter and then generate MSI #1.

The driver's steps to perform a command are the following:

a. Setting all the necessary command parameters in the I/O memory
b. Setting the command number in the Command field in the I/O memory
c. Waiting for one of the MSI #1 or MSI #2 interrupts

4. Interrupt Flag

This field is responsible for generating interrupts by the device.
If the field contains the value 0x00, the device shouldn't generate any interrupts.
If the field contains the value 0xFF, the device is allowed to generate any number of interrupts.

5. DMA Buf IN Address

This is a pointer to an array of addresses for physical pages that describe the incoming data buffer for the device.
This address points to the contiguous physical or device memory, due to which the device can work with this address as a pointer to an array of elements.
The size of the field is 4 bytes because the contiguous physical or device address will be wrapped using the following formula:

DMA Buf IN Address = 64-bit physical address << 12

This kind of address wrapping is possible because the last 3 bytes of the address will always contain zeros (it’s guaranteed by the driver and doesn’t depend on the data alignment in the DMA buffer) since the operating system usually operates with 4KB memory pages. The device will only read this memory and never change it.
The format of the data in the memory is an array of ULONG64 values, where each value describes one 4KB physical page.
The number of elements in the array is specified in the DMA Buf IN Page Count field.

6. DMA Buf IN Page Count

Sets the number of elements in the array for DMA Buf IN Address.

7. DMA Buf IN Size in Bytes

Sets the buffer size in bytes, while the buffer is described by an array of addresses in the DMA Buf IN Address.

8. DMA Buf OUT Address

This is a pointer to an array of physical page addresses that describes the output data buffer for the device.
The data format is similar to the DMA Buf IN Address field. The device will only write this memory and never read it.

9. DMA Buf OUT Page Count

Sets the number of elements in the array for the DMA Buf OUT Address.

10. DMA Buf OUT Size in Bytes

Sets the buffer size in bytes, while the buffer is described by an array of addresses in the DMA Buf OUT Address.

11. MSI #1 Error

This field contains a flag indicating an active Error interrupt on the device if its value is not 0. This field is necessary for two cases:

- when line-based interrupt is used
- when MSI #0 case is used (it happens when the operating system is unable to allocate the necessary number of MSIs for the device, the operating system can allocate and assign only one MSI for a device)

Using this flag, we can always determine which type of event is generated by the device.

12. MSI #2 Ready

This field contains a flag that indicates an active Ready interrupt on the device if its value is not 0.
The rest of the description is identical to the MSI #1 Error field.

13. MSI #2 Reset

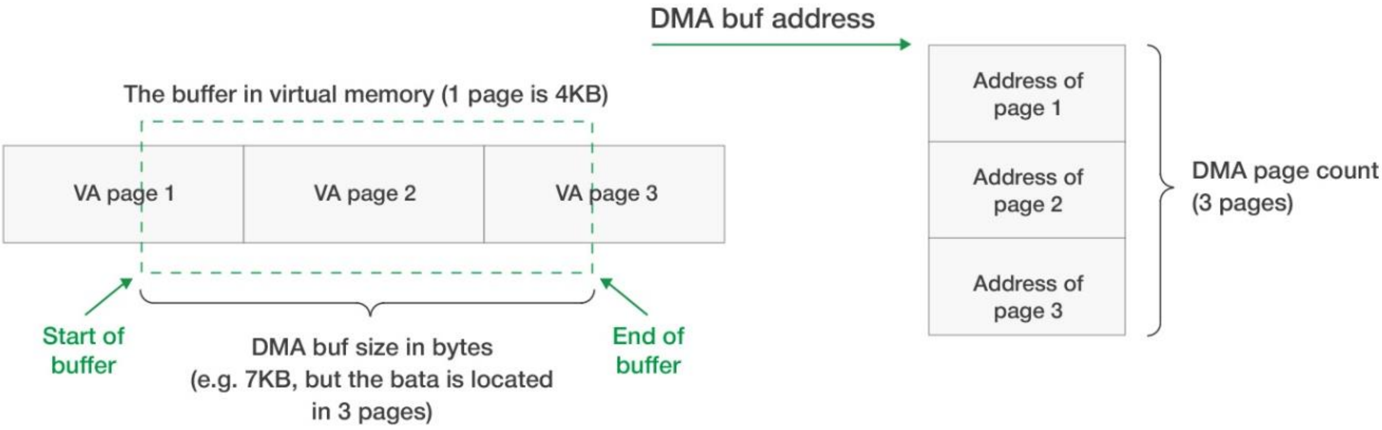
This field contains a flag that indicates an active reset interrupt on the device if its value is not 0.
The rest of the description is similar to the MSI #1 Error field.

14. Unused

Not used; this field is added for alignment.

Why don't we use the same field for State and Command?
The reason why two separate fields are used is to avoid parallel changes in the field from the device and driver side. Each field can be changed only by one side, the driver or the device. In this case, we avoid situations where the field is recorded simultaneously by both sides, resulting in an inconsistent state.

You may also ask why the IN and OUT buffers are described by three parameters (DMA Buf Address, DMA Buf Size in Bytes, and DMA Page Count) if it's enough to have two fields, Address and Size.
Since we're dealing with two different buffers (an array of addresses and a data buffer), different fields are used to set the size of each.



By using the DMA page count, we reduce the number of DMA read operations and simplify the device implementation.

Interrupts

MSI number	Description	Conditions
0	Common MSI The device reports about one of the following events: 1. Error INT 2. Ready INT 3. Reset INT	I/O: MSI #1/2/3 Error != 0
1	Error INT Request processing the error.	I/O: ErrorCode != 0 I/O: MSI #1 Error != 0
2	Ready INT The device has successfully completed a request. All DMA operations are completed. The device is no longer using previously set addresses for DMA operations.	I/O: ErrorCode == 0 I/O: State == 0 (ReadyState) I/O: Command == 0 (Idle) I/O: MSI #2 Error != 0
3	Reset INT The device has completed the reset. All DMA operations have been completed. The device is no longer using previously set addresses for DMA operations.	I/O: ErrorCode == 0 I/O: State == 0 (ReadyState) I/O: Command == 0 (Idle) I/O: DMA Buf (all values) == 0 I/O: MSI #3 Error != 0

Let’s consider why we used three MSI types instead of just one.
Firstly, the different MSI types are used to simplify logic and accelerate performance of the device.
Secondly, three different types of MSIs are used to demonstrate how to work with various MSIs (even though one would be enough for this device).

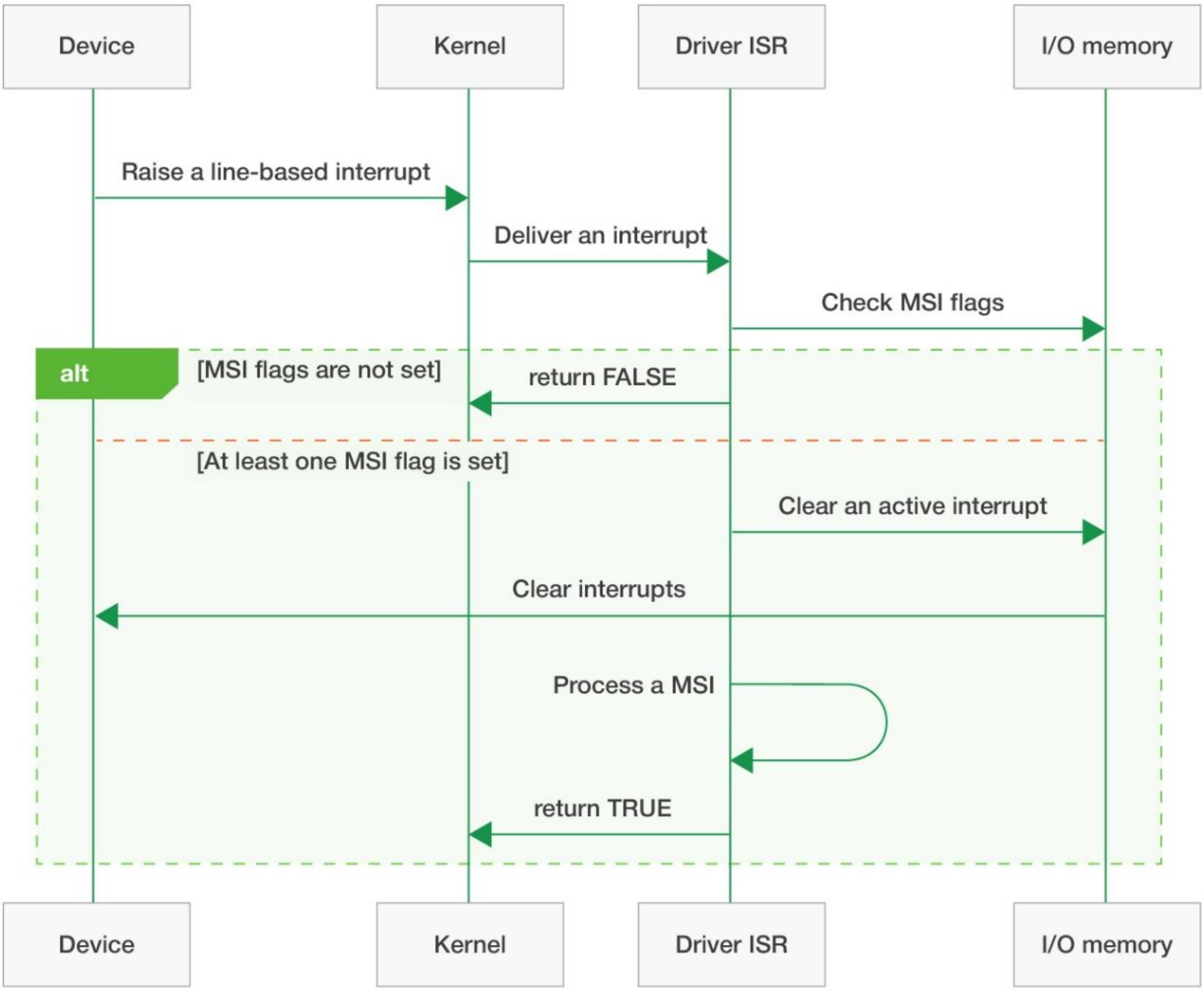
In this test device, we have three modes of operating interrupts:

1. If the operating system doesn’t support MSI, we’ll use line-based interrupts.
2. If the operating system is unable to allocate the necessary number of MSIs, we’ll use only MSI #0.
3. If the operating system has allocated all the necessary MSIs, we’ll use all the MSIs.

Why is there a separate MSI #0 for operation mode 2?
Only one of these three modes can be active at a time. Since both the driver and the device always know which mode is used, MSI #0 could be used as the only interrupt for the second case or as an MSI Error for the third case. However, we use different numbers to simplify the understanding of the test device and driver code.

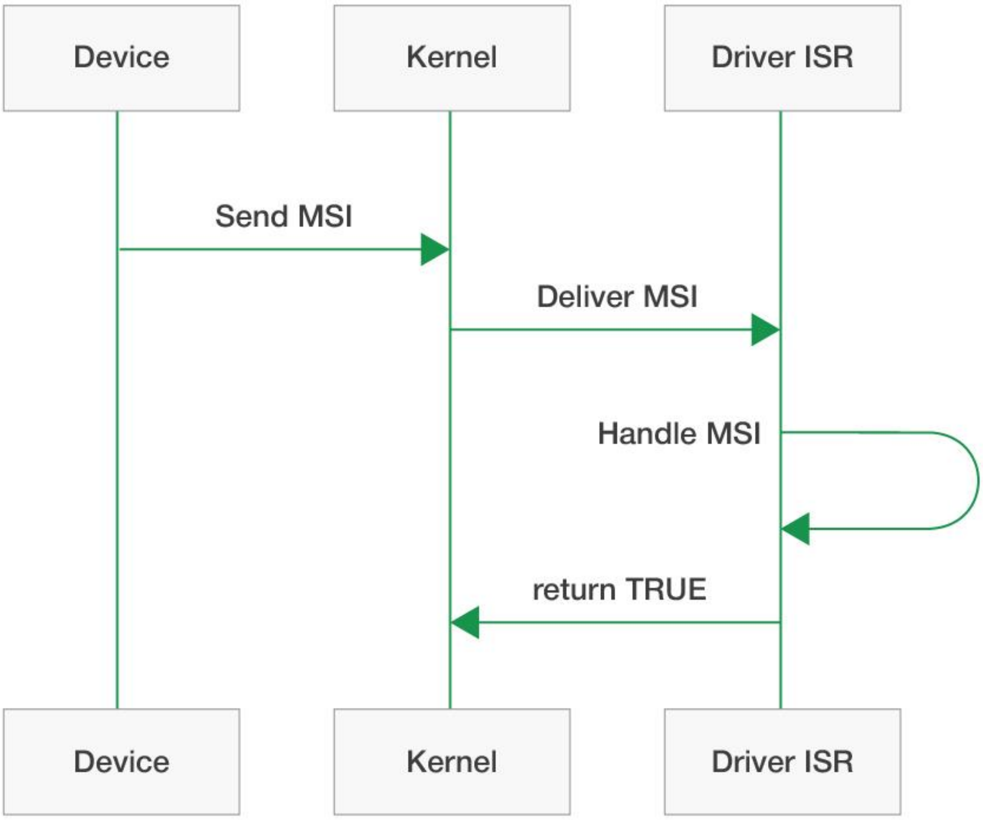
For the driver, the code for modes 1 and 2 look identical, so the work scheme is the following:

1. The driver receives an interrupt message.
2. The driver checks the flags for MSI #1 Error and MSI #2 Ready in the I/O memory.
3. If none of the flags are set, the driver skips the interrupt and informs the system that the interrupt wasn’t processed.
4. If one or several flags are set, the driver clears it or them (in this case, the device should reset the interrupt if a line-based interrupt was used) and reprocess the events. If this is the case, the driver reports to the operating system that the interrupt has been processed.



For the driver, the work scheme with mode 3 is as follows:

1. The driver receives interrupt message number N.
2. The driver processes the interrupt according to the MSI number. In this case, there's no need to check the interrupt flags in the I/O memory.



In the above situations, we need to be sure that the device won't generate additional interrupts until the driver processes the current one. We'll add this restriction to our simple device. This scheme of work with interruptions is applicable if a device processes data in batches. In the case of streaming data processing, the scheme with MSI flags isn't suitable.

For the device, the work scheme for mode 1 (line-based interrupt mode) is the following:

1. The device sets the MSI flag to the I/O memory.
2. The device raises the interrupt.
3. The device monitors the reset of the MSI flag to the I/O memory and then removes the interrupt.

For the device, the work scheme for mode 2 (MSI #0) is as follows:

1. The device sets the MSI flag to the I/O memory.
2. The device generates an MSI #0 interrupt. In this case, you don't need to reset the interrupt because MSI doesn't require it.

For the device, the work scheme for mode 3 (MSI normal mode) is the following:

1. The device generates an MSI interrupt with the appropriate number.

This is the end of the device communication protocol description. Now, you can implement a QEMU virtual device, a real device, and a Windows device driver simultaneously because everything that's needed to communicate between these components has already been specified.

The description of the data structures above is located in the general header file named *CryptoDeviceProtocol.h*. The structure of the I/O memory looks like this:

```
typedef struct tagCryptoDeviceIo
{
   /*0x00*/ uint8_t  ErrorCode;
   /*0x01*/ uint8_t  State;
   /*0x02*/ uint8_t  Command;
   /*0x03*/ uint8_t  InterruptFlag;
   /*0x04*/ uint32_t DmaInAddress;
   /*0x08*/ uint32_t DmaInPagesCount;
   /*0x0C*/ uint32_t DmaInSizeInBytes;
   /*0x10*/ uint32_t DmaOutAddress;
   /*0x14*/ uint32_t DmaOutPagesCount;
   /*0x18*/ uint32_t DmaOutSizeInBytes;
   /*0x1C*/ uint8_t  MsiErrorFlag;
   /*0x1D*/ uint8_t  MsiReadyFlag;
   /*0x1E*/ uint8_t  MsiResetFlag;
   /*0x1F*/ uint8_t  Unused;
} CryptoDeviceIo;
```

QEMU virtual device

Let's see how you can implement the described test device with the help of QEMU.
Here's the environment that we used:

1. QEMU sources from the stable-2.11 branch
2. Ubuntu 18.04 x64 for the source build and to launch QEMU
3. Windows 10 x64 as the QEMU guest operating system

To build QEMU sources, run the following commands:

```
1.  cd qemu # folder with qemu sources
2.  ./configure \
     --target-list=x86_64-softmmu \
     --enable-sdl \
     --enable-debug \ # debug build
     --extra-ldflags="`pkg-config --libs openssl`"
3.  make
```

Here’s how to launch QEMU without a test device but with the Windows 10 guest operating system installed:

```
1.  cd qemu # folder with qemu sources
2.  ./qemu/x86_64-softmmu/qemu-system-x86_64 \
    -enable-kvm \
    -m 4G \
    -cpu host \
    -smp cpus=4,cores=4,threads=1,sockets=1 \
    -hda /<path>/windows10.x64.img \
    -net nic -net user \
    -snapshot
```

QEMU has a large variety of virtual devices for both standard Windows drivers and custom drivers. The QEMU sources contain the file **qemu/hw/misc/edu.c** as an example of a simple device implementation. We’ll use this as a template for our test device. All sources for our CryptoDevice test device are located in one file, *qemu/hw/misc/crypto.c.* The test device is implemented in C (but you can also use C++).

To add the sources of the CryptoDevice test device to QEMU stable-2.11, use the *qemu-stable-2.11-crypto-device.patch* patch, which contains everything you need.

Device description in QEMU

To create a new type of device in QEMU, you first need to describe and register it:

```
static void pci_crypto_register_types(void)
{
    static InterfaceInfo interfaces[] = {
        { INTERFACE_CONVENTIONAL_PCI_DEVICE },
        { },
    };
    static const TypeInfo pci_crypto_info = {
        .name          = TYPE_PCI_CRYPTO_DEV,
        .parent        = TYPE_PCI_DEVICE,
        .instance_size = sizeof(PCICryptoState),
        .instance_init = pci_crypto_instance_init,
        .class_ini     = pci_crypto_class_init,
        .interfaces    = interfaces,
    };

    type_register_static(&pci_crypto_info);
}

type_init(pci_crypto_register_types)
```

Here, we indicate two things:

1. The device type is PCI (INTERFACE_CONVENTIONAL_PCI_DEVICE).
2. The name of our device is TYPE_PCI_CRYPTO_DEV. This name is used when launching QEMU.

```
#define TYPE_PCI_CRYPTO_DEV "pci-crypto"
```

The internal device context is described in the PCICryptoState structure:

```
typedef struct PCICryptoState
{
    /*< private >*/
    PCIDevice parent_obj;

    /*< public >*/
    MemoryRegion memio;
    CryptoDeviceIo * io;
    unsigned char memio_data[4096];
    unsigned char aes_cbc_key[32]; // 256bit

    QemuMutex io_mutex;
    QemuThread thread;
    QemuCond thread_cond;
    bool thread_running;

} PCICryptoState;
```

This structure contains the following:

1. the variables *io_mutex*, *thread*, *thread_cond* and *thread_running* for the device's worker thread. This thread will process all commands (*Reset*, *Encryption*, *Decryption*, and *SHA256* calculations);

2. *memio*, *io*, and *memio_data*, which describe the I/O space of the device;
3. *aes_cbc_key*, which is our hardware key that we use in the AES CBC.

The *callback pci_crypto_instance_init* function is called once for each device instance.

```
static void pci_crypto_instance_init(Object *obj)
{
    PCICryptoState *dev = PCI_CRYPTO_DEV(obj);
    PRINT("pci_crypto_instance_init\n");

    memset(dev->aes_cbc_key, 0, sizeof(dev->aes_cbc_key));
    object_property_add_str(obj, "aes_cbc_256",
                                            NULL,
                                            crypto_set_aes_cbc_key_256,
                                            NULL);
}
```

This callback says that the device can receive one string parameter named *aes_cbc_256* via the QEMU command line. If such a parameter is set, then QEMU will call the crypto_set_aes_cbc_key_256 callback and pass the value of this parameter to it.

```
static void crypto_set_aes_cbc_key_256(Object *obj,
                                       const char * value,
                                       Error **errp)
{
    PCICryptoState *dev = PCI_CRYPTO_DEV(obj);

    // calc sha256 from the user string => it's our 256 bit key for AES CBC
    SHA256((const unsigned char*)value, strlen(value),  dev->aes_cbc_key);
}
```

This parameter sets the data for the hardware key. Each device can have its own unique hardware key, which the user sets using the command line. To convert the user-defined string to an AES key, we use an SHA256 hash, the size of which is equal to the size of the AES key at 256 bits.

The *pci_crypto_class_init* callback in our PCI device is called to initialize the parameters of the PCI header (the initialization of the parameters for the current class of the device):

```
static void pci_crypto_class_init(ObjectClass *klass, void *data)
{
     DeviceClass *dc = DEVICE_CLASS(klass);
     PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);
     PRINT("pci_crypto_class_init\n");

     k->is_express = false;
```

```
        k->realize = pci_crypto_realize;
        k->exit = pci_crypto_uninit;
        k->vendor_id = 0x1111;
        k->device_id = 0x2222;
        k->revision = 0x00;
        k->class_id = PCI_CLASS_OTHERS;
        dc->desc = "PCI Crypto Device";
        set_bit(DEVICE_CATEGORY_MISC, dc->categories);
        dc->reset = pci_crypto_reset;
        dc->hotpluggable = false;
    }
```

In this callback, we set the Vendor Identifier (VID) and Product Identifier (PID) of the device as well as a number of other parameters. The VID and PID will be used later when installing the driver in Windows. In addition, we set another three callbacks that QEMU will use for communicating with the device.

At this stage, the device is added to QEMU. To run QEMU with this crypto device, we need to add one more line to the QEMU parameters:

-device pci-crypto,aes_cbc_256=our_secret_string

Initializing the device in QEMU

There are three callbacks in QEMU that are responsible for initializing and deinitializing a device instance:

1. The *realize* callback (the *pci_crypto_realize* function) is used to initialize the PCI resources and internal state of the device.

```
static void pci_crypto_realize(PCIDevice * pci_dev, Error **errp)
{
     PCICryptoState *dev = PCI_CRYPTO_DEV(pci_dev);
     PRINT("pci_crypto_realize\n");

     memory_region_init_io(&dev->memio,
                           OBJECT(dev),
                           &pci_crypto_memio_ops,
                           dev, // context for read/write callbacks
                           "pci-crypto-mmio",
                           sizeof(dev->memio_data));

     pci_register_bar(pci_dev,
                      0,
                      PCI_BASE_ADDRESS_SPACE_MEMORY,
                      &dev->memio);

     pci_config_set_interrupt_pin(pci_dev->config, 1);

     if (msi_init(pci_dev, 0, CryptoDevice_MsiMax, true, false, errp)) {
         PRINT("Cannot init MSI\n");
     }

     dev->thread_running = true;
     dev->io = (CryptoDeviceIo*)dev->memio_data;
     memset(dev->memio_data, 0, sizeof(dev->memio_data));

     qemu_mutex_init(&dev->io_mutex);
     qemu_cond_init(&dev->thread_cond);
     qemu_thread_create(&dev->thread,
                                        "crypto-device-worker",
                                        worker_thread,
                                        dev,
                                        QEMU_THREAD_JOINABLE);
}
```

First, the function initializes work with the I/O memory space (the same I/O memory region described in the communication protocol). Initialization is performed by two functions: *memory_region_init_io* and *pci_register_bar*.

- *memory_region_init_io*

The first function is *memory_region_init_io*. It's responsible for initializing the *memio* variable. Using this function, we set the I/O memory size, specify a user-friendly name of the region, and pass the completed *MemoryRegionOps* structure describing the properties of this memory.

The memory properties are described as follows:

```
static const MemoryRegionOps pci_crypto_memio_ops = {
         .read = pci_crypto_memio_read,
         .write = pci_crypto_memio_write,
         .endianness = DEVICE_LITTLE_ENDIAN,
         .impl = {
                 .min_access_size = 1,
                 .max_access_size = 4,
       },
         .valid = {
                .min_access_size = 1,
                .max_access_size = 4,
         },
};
```

This structure indicates that the device supports access to memory of 1, 2, or 4 bytes. If the guest operating system driver tries to read 8 bytes of memory, the request will be automatically split into two separate requests of 4 bytes each.

We also indicate that the device uses the little endian byte order and set callback functions for *read* and *write* operations. QEMU will call these functions when the guest driver reads or writes a callback to memory.

On the physical level, the I/O memory of a test device is a 4KB array located in the device context structure.

```
typedef struct PCICryptoState
{
     ....
     unsigned char memio_data[4096];
```

You can allocate such memory in any common way. You can also choose not to allocate it at all if the device doesn’t need this memory for handling read and write requests or if the I/O memory fields are stored separately from each other or calculated during request execution.

- *pci_register_bar* function

Now the I/O memory variable is initialized. The next step is registering the PCI resource through the *pci_register_bar* function. When calling this function, we transfer the sequence number of the region (in our case, 0). While there can be several regions of such memory, here we use only one. We specify the type of the resource: PCI_BASE_ADDRESS_SPACE_MEMORY.

At this point, the I/O memory space is initialized. Next, we initialize the interrupt resource. The *pci_config_set_interrupt_pin* function indicates that the device uses interrupts, and the *msi_init* function indicates that the device can work with MSIs and sets the supported MSI characteristics.

The PCI resources are now initialized, and the functions described above fill the PCI configuration space of the device. The guest operating system will later use this PCI configuration space to work with the device.

In the final stage, we initialize the device context variables and start the device worker thread, which will execute the requests.

2. The *uninti* callback (the *pci_crypto_uninit* function) is supposed to release the resources allocated in *realize*.

```
static void pci_crypto_uninit(PCIDevice * pci_dev)
{
        PCICryptoState *dev = PCI_CRYPTO_DEV(pci_dev);
        PRINT("pci_crypto_uninit\n");

        qemu_mutex_lock(&dev->io_mutex);
        dev->thread_running = false;
        qemu_mutex_unlock(&dev->io_mutex);
        qemu_cond_signal(&dev->thread_cond);
        qemu_thread_join(&dev->thread);

        qemu_cond_destroy(&dev->thread_cond);
        qemu_mutex_destroy(&dev->io_mutex);
}
```

In our case, we can simply finish the device worker thread and free the resources.

3. The *reset* callback (the *pci_crypto_reset* function) is called when the guest operating system is loaded and must reset the state of the device to default. For CryptoDevice, setting the initial values in the I/O memory space will be enough. The I/O memory space was already initialized when the function was called.

```
static void pci_crypto_reset(DeviceState * pci_dev)
{
       PCICryptoState *dev = PCI_CRYPTO_DEV(pci_dev);
       PRINT("pci_crypto_reset\n");

       qemu_mutex_lock(&dev->io_mutex);
       dev->io->ErrorCode = CryptoDevice_NoError;
       dev->io->State = CryptoDevice_ReadyState;
       dev->io->Command = CryptoDevice_IdleCommand;
       dev->io->InterruptFlag = CryptoDevice_DisableFlag;
       dev->io->DmaInAddress = 0;
       dev->io->DmaInPagesCount = 0;
       dev->io->DmaInSizeInBytes = 0;
       dev->io->DmaOutAddress = 0;
       dev->io->DmaOutPagesCount = 0;
       dev->io->DmaOutSizeInBytes = 0;
       dev->io->MsiErrorFlag = 0;
       dev->io->MsiReadyFlag = 0;
       dev->io->MsiResetFlag = 0;
       qemu_mutex_unlock(&dev->io_mutex);
}
```

Working with the I/O memory space

As we mentioned earlier, the *MemoryRegionOps* structure (the *pci_crypto_memio_ops* variable in CryptoDevice) is filled for every I/O memory space in QEMU. In this structure, we need to specify the *read* and *write* callbacks.

Let's look at an implementation of a *read* callback:

```
static uint64_t pci_crypto_memio_read(void * opaque,
                                      hwaddr addr,
                                      unsigned size)
{
    uint64_t res = 0;
    PCICryptoState *dev = (PCICryptoState *)opaque;

    if (addr >= sizeof(dev->memio_data)) {
        PRINT("Read from unknown IO offset 0x%lx\n", addr);
        return 0;
    }

    if (addr + size >= sizeof(dev->memio_data)) {
        PRINT("Read from IO offset 0x%lx but bad size %d\n", addr, size);
        return 0;
    }

    qemu_mutex_lock(&dev->io_mutex);

    switch (size)
    {
    case sizeof(uint8_t):
        res = *(uint8_t*)&dev->memio_data[addr];
        break;
    case sizeof(uint16_t):
        res = *(uint16_t*)&dev->memio_data[addr];
        break;
    case sizeof(uint32_t):
        res = *(uint32_t*)&dev->memio_data[addr];
        break;
    }

    qemu_mutex_unlock(&dev->io_mutex);
    return res;
}
```

This function accepts three parameters:

1. **opaque** — A context pointer; this value is set by the fourth parameter in the *memory_region_init_io* function, to which we pass the device context pointer (the *PCICryptoState* structure) in this implementation.
2. **hwaddr addr** (a 64-bit unsigned value) — The offset in the I/O memory that starts from zero and is used for reading the value.
3. **unsigned size** — The size of the data to be read. In our case, this value can be 1, 2, or 4 bytes, as it's set in the *MemoryRegionOps pci_crypto_memio_ops* variable. The maximum value of this parameter is 8 bytes.

The function returns the *uint64_t* value that was read by the *addr* offset with size *size*. Since the maximum size is 8 bytes, the result can always be placed in *uint64_t*.

The implementation of the function itself is reduced to a simple switch on three possible values (1, 2, or 4 bytes) and reading from the *memio_data* variable.

Since this callback is called in the context of the QEMU thread and not of our worker thread, data access is protected with the help of a mutex.

Implementing a *write* callback is a bit more difficult because when you change some of the I/O space fields, you need to tell the device's worker thread to perform some actions. As with the *read* callback, access to *memio_data* is protected through a mutex.

```
static void pci_crypto_memio_write(void * opaque,
                                  hwaddr addr,
                                  uint64_t val,
                                  unsigned size)
{
        PCICryptoState *dev = (PCICryptoState *)opaque;

        if (addr >= sizeof(dev->memio_data)) {
                PRINT("Write to unknown IO offset 0x%lx\n", addr);
                return;
        }

        if (addr + size >= sizeof(dev->memio_data)) {
                PRINT("write to IO offset 0x%lx but bad size %d\n", addr, size);
                return;
        }

        qemu_mutex_lock(&dev->io_mutex);

#define CASE($field) \
        case offsetof(CryptoDeviceIo, $field): \
        ASSERT(size == sizeof(dev->io->$field));

        switch (addr)
        {
        CASE(ErrorCode)
                raise_error_int(dev, CryptoDevice_WriteIoError);
                break;

        CASE(State)
                raise_error_int(dev, CryptoDevice_WriteIoError);
                break;

        CASE(Command)
                dev->io->Command = (uint8_t)val;
                switch (dev->io->Command)
                {
                case CryptoDevice_ResetCommand:
                case CryptoDevice_AesCbcEncryptCommand:
                case CryptoDevice_AesCbcDecryptCommand:
                case CryptoDevice_Sha2Command:
                        qemu_cond_signal(&dev->thread_cond);
                        break;

                default:
                        ASSERT(!"Unexpected command value\n");
                        raise_error_int(dev, CryptoDevice_WriteIoError);
                }
                break;

        CASE(InterruptFlag)
                dev->io->InterruptFlag = (uint8_t)val;
                break;

        CASE(DmaInAddress)
                dev->io->DmaInAddress = (uint32_t)val;
                break;

        CASE(DmaInPagesCount)
                dev->io->DmaInPagesCount = (uint32_t)val;
                break;

        CASE(DmaInSizeInBytes)
                dev->io->DmaInSizeInBytes = (uint32_t)val;
                break;

        CASE(DmaOutAddress)
                dev->io->DmaOutAddress = (uint32_t)val;
                break;

        CASE(DmaOutPagesCount)
```

```
                dev->io->DmaOutPagesCount = (uint32_t)val;
                break;

        CASE(DmaOutSizeInBytes)
                dev->io->DmaOutSizeInBytes = (uint32_t)val;
                break;

        CASE(MsiErrorFlag)
                dev->io->MsiErrorFlag = (uint8_t)val;
                clear_interrupt(dev);
                break;

        CASE(MsiReadyFlag)
                dev->io->MsiReadyFlag = (uint8_t)val;
                clear_interrupt(dev);
                break;

        CASE(MsiResetFlag)
                dev->io->MsiResetFlag = (uint8_t)val;
                clear_interrupt(dev);
                break;
        }
#undef CASE

        qemu_mutex_unlock(&dev->io_mutex);
}
```

In this function, we use a macro:

```
#define CASE($field) \
        case offsetof(CryptoDeviceIo, $field): \
        ASSERT(size == sizeof(dev->io->$field));
```

The main task of this macro is to make sure that when accessing any of the I/O structure fields we use the data size that's equal to this field. This check is used only to track logical errors in a driver.

The *write* callback parameters are pretty similar to the ones in the *read* callback, however we add one more argument, *uint64_t val*. This argument specifies the new value of the I/O structure field. The size of the variable is 64 bits, and is equal to the maximum size allowed for working with I/O memory. The function returns nothing.

For the purposes of our discussion, we can split all operations in the current implementation of the *write* callback into four groups:

1. Changing variables that are read only on the driver's side (that have the *readonly* status according to the communication protocol). There are only two such fields:

```
CASE(ErrorCode)
        raise_error_int(dev, CryptoDevice_WriteIoError);
        break;

CASE(State)
        raise_error_int(dev, CryptoDevice_WriteIoError);
        break;
```

When attempting to write to these fields, the device will send an Error INT with the corresponding value in the ErrorCode. This is used primarily for debugging purposes. Changes to these fields aren't provided by the protocol, so the behavior of a real device may be unpredictable.

2. Changing parameters that don't require an immediate response from the device:

```
CASE(InterruptFlag)
    CASE(DmaInAddress)
    CASE(DmaInPagesCount)
    CASE(DmaInSizeInBytes)
    CASE(DmaOutAddress)
    CASE(DmaOutPagesCount)
    CASE(DmaOutSizeInBytes)
```

In such cases, you can use a simple update to the I/O memory values.

3. Changing the interrupt counters:

```
CASE(MsiErrorFlag)
        dev->io->MsiErrorFlag = (uint8_t)val;
        clear_interrupt(dev);
        break;

CASE(MsiReadyFlag)
        dev->io->MsiReadyFlag = (uint8_t)val;
        clear_interrupt(dev);
        break;

CASE(MsiResetFlag)
        dev->io->MsiResetFlag = (uint8_t)val;
        clear_interrupt(dev);
        break;
```

In this case, not only do we update the values but we also call the *clear_interrupt* function. This function is used for a line-based interrupt. We'll provide more details on its implementation and purposes later.

4. Changing the command field:

```
CASE(Command)
        dev->io->Command = (uint8_t)val;
        switch (dev->io->Command)
        {
        case CryptoDevice_ResetCommand:
        case CryptoDevice_AesCbcEncryptCommand:
        case CryptoDevice_AesCbcDecryptCommand:
        case CryptoDevice_Sha2Command:
                qemu_cond_signal(&dev->thread_cond);
                break;

        default:
                ASSERT(!"Unexpected command value\n");
                raise_error_int(dev, CryptoDevice_WriteIoError);
        }
        break;
```

When the driver changes the Command field, the device detects this change and start processing the current request. It must be processed in a separate thread because if you execute it in the current thread, a simple write operation in the I/O memory on the driver's side will be completed only after completing the entire request. Therefore, first we need to change the value of the *Command* field, then we must call *qemu_cond_signal* to send a signal and the worker thread will start processing the request. If the command value is invalid, the device will generate an error.

At this point, work with the I/O memory on the device's side is done. As you can see, the code for working with I/O memory is simple. Most of the work with request processing is performed inside QEMU, and we only need to set the request processing logic for the device.

Working with interrupts

Working with interrupts is pretty much just as easy as handling I/O requests.

For handling line-based interrupts, QEMU has a *pci_set_irq* function. The second parameter passed to the function is a flag: 1 means that the interrupt must be raised and 0 means that the interrupt must be reset.

To work with MSI on the device side, we use the next three functions from the QEMU sources:

1. **msi_enabled** — Returns *true* if MSIs were initialized
2. **msi_nr_vectors_allocated** — Returns the number of MSIs that were allocated for the device

3. **msi_notify** — Sends the MSIs (The number of interrupts is set by the second parameter.)

Here’s what the function for generating interrupts looks like:

```
static void raise_interrupt(PCICryptoState * dev, CryptoDeviceMSI msi)
{
        const uint8_t msi_flag = (1u << msi) >> 1u;
        ASSERT(msi != CryptoDevice_MsiZero);

        if (0 == (dev->io->InterruptFlag & msi_flag))
        {
                PRINT("MSI %u is disabled\n", msi);
                return;
        }

        qemu_mutex_unlock(&dev->io_mutex);

        if (msi_enabled(&dev->parent_obj))
        {
                //
                // MSI is enabled
                //
                if    (CryptoDevice_MsiMax    !=    msi_nr_vectors_allocated(&dev-
>parent_obj))
                {
                    PRINT("Send MSI 0 (origin msi =%u), allocated msi %u\n",
                                msi,
                                msi_nr_vectors_allocated(&dev->parent_obj));
                    msi = CryptoDevice_MsiZero;
                }
                else
                {
                    PRINT("Send MSI %u\n", msi);
                }
                msi_notify(&dev->parent_obj, msi);
        }
         else
         {
                //
                // Raise legacy interrupt
                //
                PRINT("Set legacy interrupt %u\n", msi);
                pci_set_irq(&dev->parent_obj, 1);
        }

        qemu_mutex_lock(&dev->io_mutex);
}
```

The function accepts a pointer to the device context and the interrupt type, as described by the following enum:

```
typedef enum tagCryptoDeviceMSI
{
        CryptoDevice_MsiZero  = 0x00,
        CryptoDevice_MsiError = 0x01,
        CryptoDevice_MsiReady = 0x02,
        CryptoDevice_MsiReset = 0x03,
        CryptoDevice_MsiMax   = 0x04
} CryptoDeviceMSI;
```

First, the function checks whether interrupts are enabled for the device:

```
 if (0 == (dev->io->InterruptFlag & msi_flag))
```

If interrupts are disabled, the function does nothing.

Next, there are several ways we can generate interrupts (all of which we described earlier):

1. If MSIs were initialized:

   ```
   if (msi_enabled(&dev->parent_obj))
   ```

   then the function checks the number of allocated MSIs through this call:

   ```
   if (CryptoDevice_MsiMax != msi_nr_vectors_allocated(&dev->parent_obj))
   {
        PRINT("Send MSI 0 (origin msi =%u), allocated msi %u\n",
              msi,
              msi_nr_vectors_allocated(&dev->parent_obj));
        msi = CryptoDevice_MsiZero;
    }
   ```

   If not all of the requested interrupts were allocated, the function replaces the current interrupt type with MSI #0, then sends the interrupt to the guest operating system:

   ```
   msi_notify(&dev->parent_obj, msi);
   ```

2. If MSIs aren't used, the function raises an INTx interrupt (a line-based interrupt):

   ```
   PRINT("Set legacy interrupt %u\n", msi);
   pci_set_irq(&dev->parent_obj, 1);
   ```

The *raise_interrupt* function must always be called with a locked *io_mutex* in order to safely access I/O values. However, this lock must be released before calling any of the QEMU functions for working with interrupts; otherwise, the interrupt won't be sent. In the end, the function acquires the lock again to save the *io_mutex* state before and the state after calling the function.

When using INTx interrupts, we need a function that will remove the interrupt after it has been processed: the *clear_interrupt* function. We've already discussed this function when talking about the I/O *write* callback. The interrupt should be removed immediately after the driver receives it. In order to remove an interrupt, the driver overwrites one of the interrupt flags and the device, in turn, calls the *clear_interrupt* function.

```
static void clear_interrupt(PCICryptoState * dev)
{
        if (!msi_enabled(&dev->parent_obj))
        {
             PRINT("Clear legacy interrupt\n");

             if (0 == dev->io->MsiErrorFlag &&
                 0 == dev->io->MsiReadyFlag &&
                 0 == dev->io->MsiResetFlag)
             {
                    pci_set_irq(&dev->parent_obj, 0);
             }
       }
}
```

The *clear_interrupt* function is needed only for INTx interrupts. First it checks the operation mode of the interrupts, and if MSI is enabled, the function does nothing.

Next, the function checks if all three interrupt flags have been removed. If they have, it removes the active interrupt by calling the *pci_set_irq* function *(& dev-> parent_obj, 0)*.

We need to check the status of all flags to make sure we won't lose the interrupt. For instance, if the device has two active interrupts (two flags are set) when we call the *clear_interrupt* function and the driver has processed only one of them and the device removes the active interrupt, the driver won't know about the second interrupt.

According to the device specification, the device must set the interrupt flag in the I/O space before generating the interrupt. For this purpose, we implement three additional functions, each of which is responsible for a separate interrupt.

```
static void raise_error_int(PCICryptoState * dev, CryptoDeviceErrorCode error)
{
        PRINT("generate error %d\n", error);
        ASSERT(error <= 0xff);

        dev->io->ErrorCode = (uint8_t)error;
        dev->io->MsiErrorFlag = 1;
        raise_interrupt(dev, CryptoDevice_MsiError);
}

static void raise_ready_int(PCICryptoState * dev)
{
        dev->io->MsiReadyFlag = 1;
        raise_interrupt(dev, CryptoDevice_MsiReady);
}

static void raise_reset_int(PCICryptoState * dev)
{
        dev->io->MsiResetFlag = 1;
        raise_interrupt(dev, CryptoDevice_MsiReset);
}
```

By design, all these functions must also be called with a locked *io_mutex*. Such small functions allow us to simplify the calling code and reduce the number of errors.

At this point, we've finished our work with the interrupts for the QEMU CryptoDevice. In the next section, we focus on handling DMA operations via the QEMU API.

Working with DMA memory

In QEMU, there are two functions for working with DMA memory (the RAM of the guest operating system):

```
             void cpu_physical_memory_read(hwaddr addr,
                                          void *buf,
                                          int len);

             void cpu_physical_memory_write(hwaddr addr,
                                           const void *buf,
                                           int len);
```

The *read* function reads the RAM, while the *write* function writes it. Let's look closer at the parameters of these functions:

1. **addr** sets the physical address of the guest operating system.

2. **buf** points to the buffer that you need to read or write.
3. **len** sets the size of the *buf* buffer in bytes.

The device can’t work directly with DMA (RAM) memory, but it can read and write such memory.

Often, you’ll have to read and write data in small portions because the device may not have enough internal memory for reading and writing all the needed memory at once.

We add two structures to store the intermediate context when working with DMA memory:

```
typedef struct DmaBuf
{
       uint64_t page_addr;   // address of the current page
       uint32_t page_offset; // offset in the current page
       uint32_t size;        // size of the remaining data
} DmaBuf;

typedef struct DmaRequest
{
       DmaBuf in;
       DmaBuf out;
} DmaRequest;
```

The *DmaRequest* structure is created every time the device starts processing a request. The next function initializes the *DmaRequest* structure:

```
static void FillDmaRequest(PCICryptoState * dev, DmaRequest * dma)
{
    dma->in.page_offset = 0;
    dma->in.page_addr = CRYPTO_DEVICE_TO_PHYS(dev->io->DmaInAddress);
    dma->in.size = dev->io->DmaInSizeInBytes;

    dma->out.page_offset = 0;
    dma->out.page_addr = CRYPTO_DEVICE_TO_PHYS(dev->io->DmaOutAddress);
    dma->out.size = dev->io->DmaOutSizeInBytes;
}
```

The CRYPTO_DEVICE_TO_PHYS macro is described in *CryptoDeviceProtocol.h*. This macro can unpack a 32-bit value to a 64-bit physical address.

The values for the *DmaRequest* structure are taken from the I/O space, which the driver fills when the request to the device is initialized. Then the *DmaRequest* structure is used as a pointer to the current position in the buffer when you read or write data to the RAM of the guest operating system.

In CryptoDevice, there’s one specific function that works with the RAM of the quest operating system:

```
static ssize_t rw_dma_data(PCICryptoState * dev,
                                            bool write,
                                            DmaBuf * dma,
                                            uint8_t * data,
                                            uint32_t size)
{
       uint32_t rw_size = 0;

       while (0 != size)
       {
            if (0 == dma->size)
            {
                        break;
            }

            uint64_t phys = 0;
            cpu_physical_memory_read(dma->page_addr, &phys, sizeof(phys));

            if (0 == phys)
            {
                       return -1;
            }

            phys += dma->page_offset;

            const uint32_t size_to_page_end = CRYPTO_DEVICE_PAGE_SIZE -
                              (phys & CRYPTO_DEVICE_PAGE_MASK);

            const uint32_t available_size_in_page = MIN(
                                        size_to_page_end,
                                        dma->size);

            const uint32_t size_to_rw = MIN(available_size_in_page, size);

            if (write)
            {
                        cpu_physical_memory_write(phys, data, size_to_rw);
            }
            else
            {
                        cpu_physical_memory_read(phys, data, size_to_rw);
            }

            data += size_to_rw;
            size -= size_to_rw;

            if (size_to_rw == size_to_page_end)
            {
```

```
                        dma->page_addr += sizeof(uint64_t);
                        dma->page_offset = 0;
            }
            else
            {
                        dma->page_offset += size_to_rw;
            }

            dma->size -= size_to_rw;
            rw_size += size_to_rw;
       }

       return rw_size;
}
```

This function accepts five parameters:

1. **dev** — a pointer to the state of the device
2. **write** — 1 for writing, 0 for reading
3. **dma** — a structure with the current position in the DMA memory
4. **data** — a pointer to the buffer for writing or reading
5. **size** — the size of the data buffer in bytes

If executed successfully, the function returns the size of the actual data that was read or written to the DMA memory. In case of an error, the function returns -1.

The function continues to work until it processes the required size of the data specified in the *size* parameter or reaches the end of the DMA buffer.

The function execution process can be divided into four stages:

1. Reading the address of the current physical page from the guest's physical memory:

```
uint64_t phys = 0;
cpu_physical_memory_read(dma->page_addr, &phys, sizeof(phys));

if (0 == phys)
{
      return -1;
}

phys += dma->page_offset;
```

2. Determining the size of the real data to be read or written to the current physical page:

```
const uint32_t size_to_page_end = CRYPTO_DEVICE_PAGE_SIZE - (phys &
CRYPTO_DEVICE_PAGE_MASK);
const uint32_t available_size_in_page = MIN(size_to_page_end, dma->size);
const uint32_t size_to_rw = MIN(available_size_in_page, size);
```

3. Writing or reading memory in the current page:

```
if (write)
{
      cpu_physical_memory_write(phys, data, size_to_rw);
}
else
{
      cpu_physical_memory_read(phys, data, size_to_rw);
}
```

4. Offsetting the current DMA memory pointer and other variables by the size of the processed data:

```
data += size_to_rw;
size -= size_to_rw;

if (size_to_rw == size_to_page_end)
{
            dma->page_addr += sizeof(uint64_t);
            dma->page_offset = 0;
}
else
{
            dma->page_offset += size_to_rw;
}

dma->size -= size_to_rw;
rw_size += size_to_rw;
```

This function is used to read and write data from the guest RAM while processing a request. On a real device, it may be possible to read and write larger volumes of data and do it more efficiently.

Processing requests

Now all the functions needed for processing device requests have been implemented. All that's left to do is implement the command logic.

First, let’s take a closer look at the main worker thread of the device, in the context of which we process the requests.

```
void* worker_thread(void * pdev)
{
    PCICryptoState * dev = (PCICryptoState*)pdev;

    qemu_mutex_lock(&dev->io_mutex);
    PRINT("worker thread started\n");

    for (;;)
    {
        while(CryptoDevice_IdleCommand == dev->io->Command
                 && dev->thread_running)
        {
            qemu_cond_wait(&dev->thread_cond, &dev->io_mutex);
        }

        if (!dev->thread_running)
        {
            PRINT("worker thread stopped\n");
            return NULL;
        }

        if (CryptoDevice_IdleCommand != dev->io->Command)
        {
            int error = 0;
            DmaRequest dma = {};
            FillDmaRequest(dev, &dma);

            switch (dev->io->Command)
            {
            case CryptoDevice_ResetCommand:
                dev->io->State = CryptoDevice_ResetState;
                DoReset(dev);
                error = CryptoDevice_DeviceHasBeenReseted;
                break;

            case CryptoDevice_AesCbcEncryptCommand:
                dev->io->State = CryptoDevice_AesCbcState;
                qemu_mutex_unlock(&dev->io_mutex);
                error = DoAesCbc(dev, &dma, true);
                qemu_mutex_lock(&dev->io_mutex);
                break;

            case CryptoDevice_AesCbcDecryptCommand:
                dev->io->State = CryptoDevice_AesCbcState;
                qemu_mutex_unlock(&dev->io_mutex);
                error = DoAesCbc(dev, &dma, false);
                qemu_mutex_lock(&dev->io_mutex);
                break;
            case CryptoDevice_Sha2Command:
                dev->io->State = CryptoDevice_Sha2State;
                qemu_mutex_unlock(&dev->io_mutex);
                error = DoSha256(dev, &dma);
                qemu_mutex_lock(&dev->io_mutex);
                break;
            }

            switch (error)
            {
            case CryptoDevice_DeviceHasBeenReseted:
                break;

            case CryptoDevice_NoError:
                raise_ready_int(dev);
                break;

            case CryptoDevice_DmaError:
            case CryptoDevice_InternalError:
                raise_error_int(dev, error);
                break;

            default:
                PRINT("Unexpected error status %d\n", error);
                raise_error_int(dev, error);
            }

            dev->io->State = CryptoDevice_ReadyState;
            dev->io->Command = CryptoDevice_IdleCommand;
        }
    }

    ASSERT(!"Never execute");
}
```

We use the condition variable for sending requests to the thread. Using this variable, the thread can learn about changes in the *Command* field from the I/O space. The thread runs in an infinite loop and the only condition for its completion is the following:

```
if (!dev->thread_running)
{
    PRINT("worker thread stopped\n");
    return NULL;
}
```

If the *Command* field is changed, the thread will first fill the structure with DMA values in the IN and OUT buffers:

```
      DmaRequest dma = {};
      FillDmaRequest(dev, &dma);
```

Then the thread will pass control to the function responsible for processing a particular request while at the same time changing the value of the *State* field in the I/O structure (the field shows the current state of the device):

```
switch (dev->io->Command)
      {
      case CryptoDevice_ResetCommand:
          dev->io->State = CryptoDevice_ResetState;
          DoReset(dev);
          error = CryptoDevice_DeviceHasBeenReseted;
          break;

      case CryptoDevice_AesCbcEncryptCommand:
          dev->io->State = CryptoDevice_AesCbcState;
          qemu_mutex_unlock(&dev->io_mutex);
          error = DoAesCbc(dev, &dma, true);
          qemu_mutex_lock(&dev->io_mutex);
          break;
     …
```

Requests that work with the DMA memory are performed without a locked *io_mutex.* These functions work only with the local parameters and don’t use the I/O space or other values from the device context.

After processing the request, the thread locks *io_mutex* again and processes the result returned by the function:

```
switch (error)
{
case CryptoDevice_DeviceHasBeenReseted:
    break;

case CryptoDevice_NoError:
    raise_ready_int(dev);
    break;

case CryptoDevice_DmaError:
case CryptoDevice_InternalError:
    raise_error_int(dev, error);
    break;

default:
    PRINT("Unexpected error status %d\n", error);
    raise_error_int(dev, error);
```

```
        }
```

If the function was executed successfully, the thread sends the *ready* interrupt. If any error occurs, the thread sends the *error* interrupt. One more possible option is for the initial request to be interrupted by a request from the driver:

```
case CryptoDevice_DeviceHasBeenReseted:
      break;
```

In this case, the thread does nothing since the *reset* interrupt has already been sent.

Finally, the thread changes the *State* and *Command* fields that indicate that the device is ready to perform the next requests and goes into standby mode:

```
dev->io->State = CryptoDevice_ReadyState;
dev->io->Command = CryptoDevice_IdleCommand;
```

Let’s look at the function that processes the SHA256 calculation request:

```
int DoSha256(PCICryptoState * dev, DmaRequest * dma)
{
    unsigned char digest[SHA256_DIGEST_LENGTH] = {};
    unsigned char page[CRYPTO_DEVICE_PAGE_SIZE] = {};
    SHA256_CTX hash = {};

    if (!dma->out.page_addr || dma->out.size < SHA256_DIGEST_LENGTH)
    {
        return CryptoDevice_DmaError;
    }
    if (!dma->in.page_addr && dma->in.size != 0)
    {
        return CryptoDevice_DmaError;
    }

    SHA256_Init(&hash);

    while (0 != dma->in.size)
    {
        ssize_t size = rw_dma_data(dev,
                                   false,
                                   &dma->in, page, sizeof(page));

        if (-1 == size)
        {
                return CryptoDevice_DmaError;
        }
        SHA256_Update(&hash, page, size);

        if (CheckStop(dev))
        {
                return CryptoDevice_DeviceHasBeenReseted;
        }
    }

    SHA256_Final(digest, &hash);

    if (sizeof(digest) != rw_dma_data(dev,
                                      true,
                                      &dma->out, digest, sizeof(digest)))
    {
          return CryptoDevice_DmaError;
    }

    return CryptoDevice_NoError;
}
```

The function accepts two parameters: the state of the device and information about the DMA memory the function will work with. First, the function checks the DMA memory values, and if there are any inconsistencies between the IN and OUT buffers, the function will return an error:

```
if (!dma->out.page_addr || dma->out.size < SHA256_DIGEST_LENGTH)
{
     return CryptoDevice_DmaError;
}

if (!dma->in.page_addr && dma->in.size != 0)
{
     return CryptoDevice_DmaError;
}
```

Next, the function reads 4KB of data from the DMA IN buffer in a loop and calculates the SHA256 hash with the help of OpenSSL. At each iteration of the loop, the function also checks if the operation has been interrupted:

```
if (CheckStop(dev))
{
      return CryptoDevice_DeviceHasBeenReseted;
}
```

The *CheckStop* function returns *true* if it’s necessary to immediately interrupt the execution and pass control back to the worker thread. In addition, if the *CheckStop* function returns *true*, it’s forbidden to read from or write to the DMA memory because the *reset* interrupt has already been sent and, according to the communication protocol, all DMA operations have been completed and are no longer scheduled for execution until the next command.

If the function has successfully processed the entire DMA input buffer, the SHA256 hash is written to the OUT DMA buffer:

```
if (sizeof(digest) != rw_dma_data(dev, true, &dma->out, digest, sizeof(digest)))
{
    return CryptoDevice_DmaError;
}
```

Then control is passed back to the worker thread function.

The *CheckStop* function is used to interrupt the processing of the current command in the worker thread:

```
bool CheckStop(PCICryptoState * dev)
{
    bool res = false;
    qemu_mutex_lock(&dev->io_mutex);

    if (CryptoDevice_ResetCommand == dev->io->Command || !dev->thread_running)
    {
        DoReset(dev);
        res = true;
    }

    qemu_mutex_unlock(&dev->io_mutex);
    return res;
}
```

This function checks the value of the *Command* field and resets the state of the device. Thanks to this function, the driver can interrupt the execution of any operation on the device's side and terminate the request. Each of the command handlers periodically calls the function and processes its results (in this implementation, this happens at each new iteration of the loop).

The handler for the *reset* command looks pretty simple:

```
void DoReset(PCICryptoState * dev)
{
    dev->io->ErrorCode = CryptoDevice_NoError;
    dev->io->State = CryptoDevice_ReadyState;
    dev->io->Command = CryptoDevice_IdleCommand;
    dev->io->DmaInAddress = 0;
    dev->io->DmaInPagesCount = 0;
```

```
                dev->io->DmaInSizeInBytes = 0;
                dev->io->DmaOutAddress = 0;
                dev->io->DmaOutPagesCount = 0;
                dev->io->DmaOutSizeInBytes = 0;
                raise_reset_int(dev);
            }
```

This handler is supposed to reset the values in the I/O memory and send the interrupt.

The last two commands, perform AES CBC encrypt and decrypt, are processed by the following function:

```
int DoAesCbc(PCICryptoState * dev, DmaRequest * dma, bool encrypt);
```

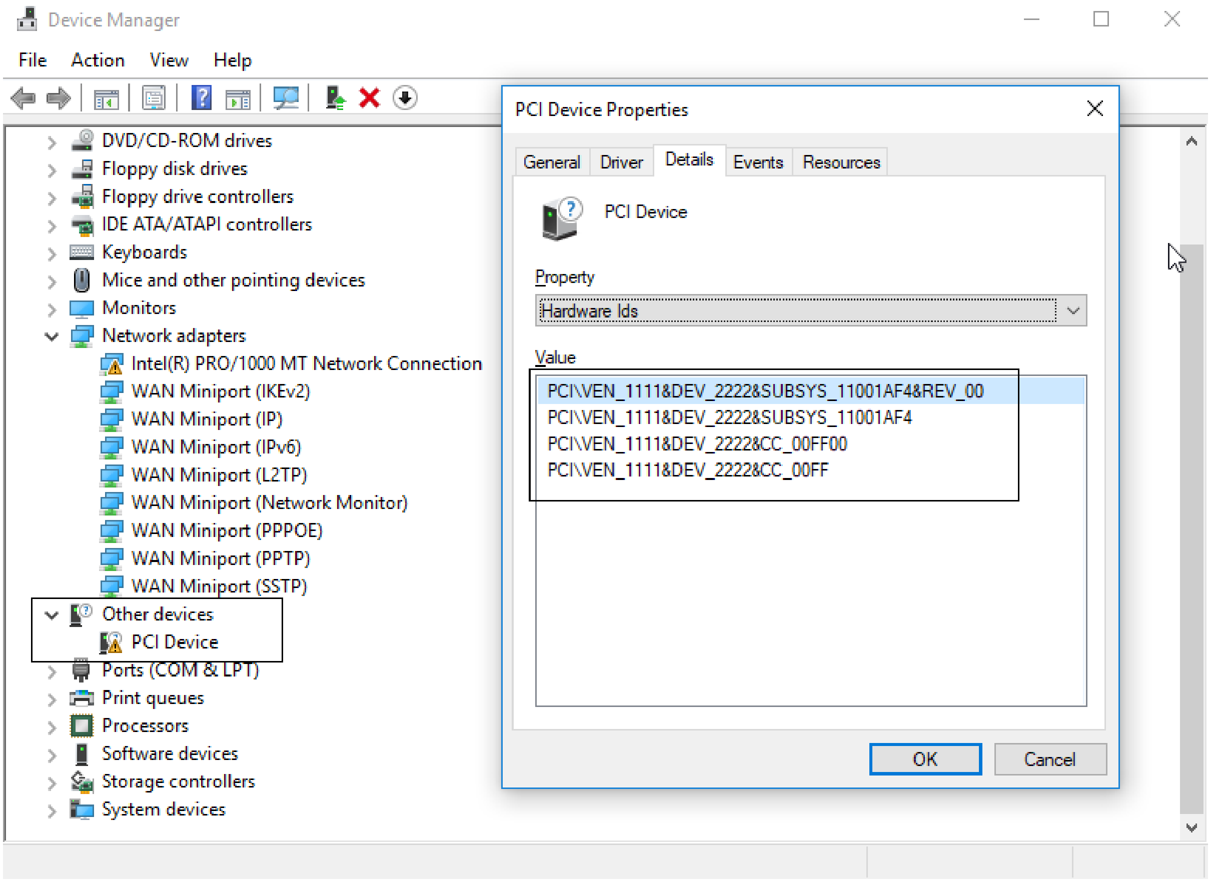
This function's code is pretty similar to the code of DoSha256 except for the fact that the results are written to the DMA OUT buffer at each iteration of the loop. The function also uses openSSL for working with AES.

QEMU device

The QEMU virtual device development is now complete. All the necessary functionality has been implemented according to the communication protocol. After building the QEMU sources, here's what the QEMU boot line with Windows 10 x64 and the crypto virtual device looks like:

```
./qemu/x86_64-softmmu/qemu-system-x86_64 \
             -enable-kvm \
             -m 4G \
             -cpu host \
             -smp cpus=4,cores=4,threads=1,sockets=1 \
             -device pci-crypto,aes_cbc_256=secret \
             -hda /<path>/windows10.x64.img \
             -net nic -net user \
       -snapshot
```

If everything was done correctly, the Windows guest operating system will detect a new unknown device with VID = 1111 and PID = 2222:



PCI Device

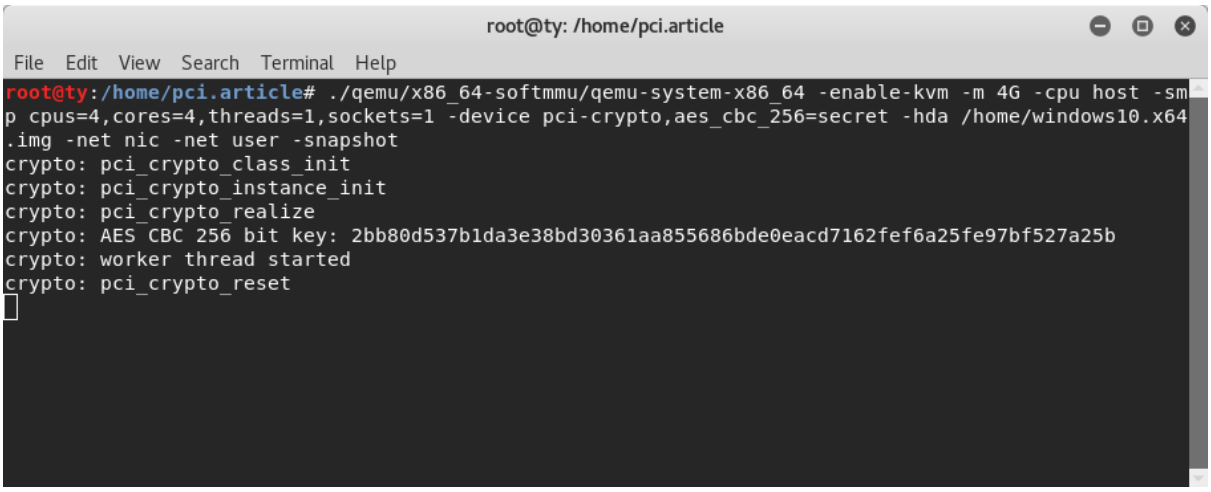
Property: Hardware Ids

Value:

```
PCI\VEN_1111&DEV_2222&SUBSYS_11001AF4&REV_00
PCI\VEN_1111&DEV_2222&SUBSYS_11001AF4
PCI\VEN_1111&DEV_2222&CC_00FF00
PCI\VEN_1111&DEV_2222&CC_00FF
```

Windows can’t recognize the device or work with it because the system can’t find a suitable driver for it, either locally in the system or on Windows servers.

Here’s what the console for starting QEMU looks like:


root@ty: /home/pci.article
File Edit View Search Terminal Help

```
root@ty:/home/pci.article# ./qemu/x86_64-softmmu/qemu-system-x86_64 -enable-kvm -m 4G -cpu host -sm
p cpus=4,cores=4,threads=1,sockets=1 -device pci-crypto,aes_cbc_256=secret -hda /home/windows10.x64
.img -net nic -net user -snapshot
crypto: pci_crypto_class_init
crypto: pci_crypto_instance_init
crypto: pci_crypto_realize
crypto: AES CBC 256 bit key: 2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b
crypto: worker thread started
crypto: pci_crypto_reset
```

You can find the complete source code for the device in the *crypto.c* file.

Implementing a WDF driver for the test device

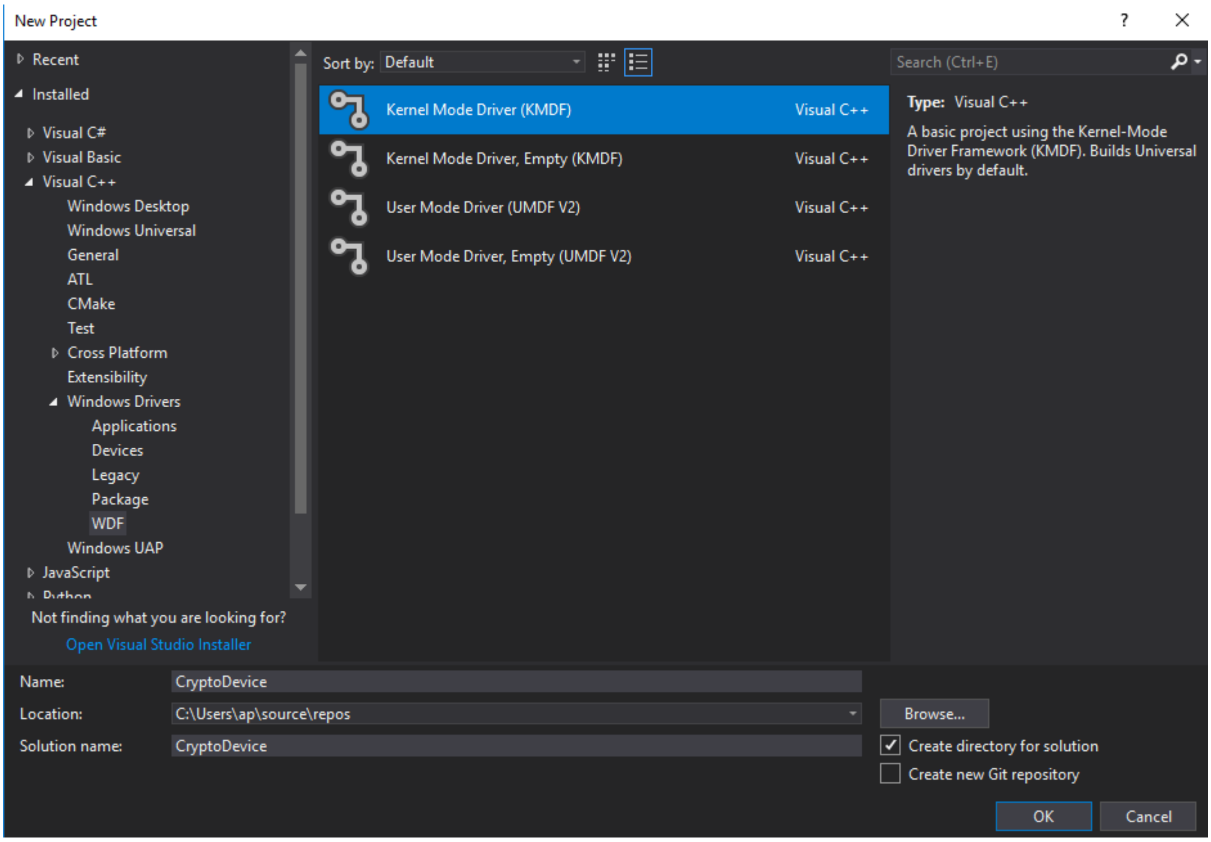
To implement the device driver, we'll use Windows Driver Frameworks (WDF).

WDF can simplify device driver development by implementing many parts of work with the device and providing an additional level of abstraction between the Windows kernel API and the driver. As a result, working with WDF is much easier than working with Windows Driver Model (WDM).

We'll use Visual Studio 2017 as our integrated development environment (IDE) and use the WDF driver template in it. We'll also need to install a WDK pack for Windows 10.

The minimum driver

To implement the minimum device driver for the crypto device, let's create a driver project:



New Project

Sort by: Default

Kernel Mode Driver (KMDF) Visual C++

Type: Visual C++

A basic project using the Kernel-Mode Driver Framework (KMDF). Builds Universal drivers by default.

Name: CryptoDevice

Location: C:\Users\ap\source\repos

Solution name: CryptoDevice

☑ Create directory for solution

□ Create new Git repository

Our driver project will contain three source files:

1. **Driver.c** — This is the entry point for the driver (the main driver function called *DriverEntry*). It's not necessary to add any functionality or change anything in this file.
2. **Queue.c** — This file contains functions needed for processing user requests (input-output control, or IOCTL). For a minimum driver, there's no need to add or change anything in this file. Later, we'll add user request handlers to the *CryptoDeviceEvtIoDeviceControl* function.
3. **Device.c** — This file is for creating the device and handling the device's WDF callbacks. Let's start with making some changes to this file.

Initializing device resources

First, we'll initialize device resources (I/O memory and interrupts). The WDF model is built in such a way that the driver sets the callback functions for the events it wants to handle and the framework calls these callbacks at the right time, in a particular context, and in a specific order.

We need to store the resource values somewhere so that we can still use these resources in future. For this purpose, each WDF device has a special device context — a data structure defined by the developer. Each device created will contain its own data set in this structure. The project template has already defined such a structure in the *Device.h* file, named DEVICE_CONTEXT. Let's change this structure by adding the necessary fields:

```
typedef struct _IO_MEMORY
{
    PVOID Memory;
    SIZE_T Size;
} IO_MEMORY;

typedef struct _DEVICE_CONTEXT
{
    IO_MEMORY IoMemoryBar0;

    WDFSPINLOCK InterruptLock;
    WDFINTERRUPT Interrupt[CryptoDevice_MsiMax];
    ULONG InterruptCount;

} DEVICE_CONTEXT, *PDEVICE_CONTEXT;
```

In this structure:

- **IoMemoryBar0** contains all the information about the I/O memory of the device.
- **InterruptLock** is a spin lock for synchronizing functions that handle interrupts.

- **Interrupt** is an array with the interrupt objects that we'll create when initializing resources. Since the maximum number of MSIs is set at four by the protocol, we need only an array of four *CryptoDevice_MsiMax* elements.
- **InterruptCount** is the actual number of interrupts that were allocated for the device. In our case, this field will contain either 1 (if using the INTx interrupts or MSI #0) or 4 (if allocating all the needed MSIs).

In order to get a pointer from a device object (WDFOBJECT) to the described structure, WDF provides a special macro:

```
WDF_DECLARE_CONTEXT_TYPE_WITH_NAME(
        DEVICE_CONTEXT,
        DeviceGetContext);
```

With the help of this macro, we call the *DeviceGetContext* function to get a pointer to DEVICE_CONTEXT.

Here's what the call to the *DeviceGetContext* function looks like:

```
PDEVICE_CONTEXT ctx = DeviceGetContext(device);
```

To receive notifications about device resources, we need to register two callback functions in the *CryptoDeviceCreateDevice* function (which creates a device) before calling the *WdfDeviceCreate* function:

```
WDF_PNPPOWER_EVENT_CALLBACKS pnpCallbacks;
WDF_PNPPOWER_EVENT_CALLBACKS_INIT(&pnpCallbacks);
pnpCallbacks.EvtDevicePrepareHardware = CryptoDeviceEvtDevicePrepareHardware;
pnpCallbacks.EvtDeviceReleaseHardware = CryptoDeviceEvtDeviceReleaseHardware;
WdfDeviceInitSetPnpPowerEventCallbacks(DeviceInit, &pnpCallbacks);
```

The first callback of the *EvtDevicePrepareHardware* function will be called when WDF is ready to handle the device sources. Here's the prototype of this function:

```
NTSTATUS CryptoDeviceEvtDevicePrepareHardware(
    _In_ WDFDEVICE Device,
    _In_ WDFCMRESLIST ResourcesRaw,
    _In_ WDFCMRESLIST ResourcesTranslated
);
```

WDF callbacks always contain the WDFDEVICE device parameter, which is why the device calls this event in the first place. Using this parameter and the *DeviceGetContext* function, we can get a pointer to the device context.

The callback for the *EvtDevicePrepareHardware* function is described in the Microsoft documentation, so we'll focus only on the code for initializing resources. Each resource has a specific type that the driver uses to recognize it. For the I/O memory, the code looks as follows:

```
PDEVICE_CONTEXT ctx = DeviceGetContext(Device);
….
switch (descriptor->Type)
{
case CmResourceTypeMemory:

    ASSERT(descriptor->u.Memory.Length == 0x1000);

    if (ctx->IoMemoryBar0.Memory)
    {
        return STATUS_DEVICE_CONFIGURATION_ERROR;
    }

    ctx->IoMemoryBar0.Memory = MmMapIoSpaceEx(
            descriptor->u.Memory.Start,
            descriptor->u.Memory.Length,
            PAGE_READWRITE | PAGE_NOCACHE);

    if (!ctx->IoMemoryBar0.Memory)
    {
            return STATUS_DEVICE_CONFIGURATION_ERROR;
    }

    ctx->IoMemoryBar0.Size = descriptor->u.Memory.Length;
    break;
```

WDF provides all necessary information about the I/O memory region:

descriptor-> u.Memory.Start — the physical address of the region
descriptor-> u.Memory.Length — the region size in bytes

To allow the driver to access this memory, we have to map it to the virtual address space of the kernel, which is exactly what the *MmMapIoSpaceEx* function does.

The virtual address pointer and the size of the memory region are stored in the device context. Using the received virtual address, the driver can communicate with the device.

The driver also checks three conditions:

1. According to the specification, the size of the I/O memory region must be equal to 4KB (0x1000).
2. There has to be only one I/O memory region. Otherwise, the driver will return an error because it doesn’t expect that there will be several I/O memory regions in the device:

    ```
    if (ctx->IoMemoryBar0.Memory) // the region has been initialized already
    {
        return STATUS_DEVICE_CONFIGURATION_ERROR;
    }
    ```

3. The *MmMapIoSpaceEx* function returns non NULL, otherwise the kernel hasn’t mapped the I/O memory to the kernel virtual address space.

Initializing interrupts looks a bit more complicated:

```
case CmResourceTypeInterrupt:

    raw = WdfCmResourceListGetDescriptor(ResourcesRaw, i);

    if (0 != ctx->InterruptCount)
    {
            return STATUS_DEVICE_CONFIGURATION_ERROR;
    }

    if (CM_RESOURCE_INTERRUPT_MESSAGE & descriptor->Flags)
    {
            ctx->InterruptCount = min(ARRAYSIZE(ctx->Interrupt),
                                     raw->u.MessageInterrupt.Raw.MessageCount);

        if (ctx->InterruptCount != ARRAYSIZE(ctx->Interrupt))
        {
                ctx->InterruptCount = 1;
        }

        for (ULONG k = 0; k < ctx->InterruptCount; ++k)
        {
                NT_CHECK(CryptoDeviceInterruptCreate(Device,
                                                    descriptor,
                                                    raw,
                                                    &ctx->Interrupt[k]));
        }
    }
    else
    {
            ctx->InterruptCount = 1;
```

```
            NT_CHECK(CryptoDeviceInterruptCreate(Device,
                                                 descriptor,
                                                 raw,
                                                 &ctx->Interrupt[0]));
        }
        break;
```

This code takes into account all three scenarios of interrupt initialization.

First, we check what type of interrupt was allocated for the device. Here's the condition responsible for this:

```
              if (CM_RESOURCE_INTERRUPT_MESSAGE & descriptor->Flags)
```

If the condition is true, we need to use MSI, meaning our next step will be getting the total number of MSIs which have been allocated for the device by an operating system or kernel.

The count of allocated MSIs is in the *raw->u.MessageInterrupt.Raw.MessageCount* unit field. Note that the system can allocate fewer MSIs than requested, so the driver should use only the number of MSIs that are actually available:

```
ctx->InterruptCount = min(ARRAYSIZE(ctx->Interrupt),
                          raw->u.MessageInterrupt.Raw.MessageCount);

if (ctx->InterruptCount != ARRAYSIZE(ctx->Interrupt))
{
            ctx->InterruptCount = 1;
}
```

Next, we create each of the allocated MSIs (we'll discuss the *CryptoDeviceInterruptCreate* function later):

```
for (ULONG k = 0; k < ctx->InterruptCount; ++k)
{
            NT_CHECK(CryptoDeviceInterruptCreate(Device,
                                                 descriptor,
                                                 raw,
                                                 &ctx->Interrupt[k]));
}
```

In this loop, either one MSI (for MSI #0) or all four MSIs can be created.
If using the INTx interrupt (the line-based interrupt), only one interrupt is created:

```
ctx->InterruptCount = 1;
    NT_CHECK(CryptoDeviceInterruptCreate(Device,
                                        descriptor,
                                        raw,
                                        &ctx->Interrupt[0]));
```

WDF helps developers neutralize the differences between handling INTx and MSIs, so we can use the same code to create and process both types of interrupts.

Finally, the *CryptoDeviceEvtDevicePrepareHardware* function verifies that all the necessary device resources have been found. If they haven't, the function returns an error:

```
if (0 == ctx->InterruptCount)
{
    return STATUS_DEVICE_INSUFFICIENT_RESOURCES;
}

if (!ctx->IoMemoryBar0.Memory)
{
    return STATUS_DEVICE_CONFIGURATION_ERROR;
}
```

At this point, we've finished initializing the device resources and the driver can use the I/O memory and receive interrupts.

In order to free the allocated device resources, WDF calls the *CryptoDeviceEvtDeviceReleaseHardware* function callback:

```
NTSTATUS CryptoDeviceEvtDeviceReleaseHardware(
    _In_ WDFDEVICE Device,
    _In_ WDFCMRESLIST ResourcesTranslated
);
```

You can find a detailed description of the parameters of this callback in the Microsoft documentation. In this function, the driver should release all resources allocated in the *CryptoDeviceEvtDevicePrepareHardware* function.

Since WDF handles the release of created interrupts, all we need to do is free the I/O memory and change the values of the variables in DEVICE_CONTEXT:

```
PDEVICE_CONTEXT ctx = DeviceGetContext(Device);

  if (ctx->IoMemoryBar0.Memory)
  {
          MmUnmapIoSpace(ctx->IoMemoryBar0.Memory,
                         ctx->IoMemoryBar0.Size);
          ctx->IoMemoryBar0.Memory = NULL;
          ctx->IoMemoryBar0.Size = 0;
  }

  ctx->InterruptCount = 0;
```

Our next step is creating the interrupt objects:

```
NTSTATUS CryptoDeviceInterruptCreate(
    _In_    WDFDEVICE Device,
    _In_    PCM_PARTIAL_RESOURCE_DESCRIPTOR InterruptTranslated,
    _In_    PCM_PARTIAL_RESOURCE_DESCRIPTOR InterruptRaw,
    _Inout_ WDFINTERRUPT *Interrupt
)
{
    PAGED_CODE();

    PDEVICE_CONTEXT devContext = DeviceGetContext(Device);

    WDF_OBJECT_ATTRIBUTES attributes;
    WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(
          &attributes,
          DEVICE_INTERRUPT_CONTEXT);

    WDF_INTERRUPT_CONFIG interruptConfig;
    WDF_INTERRUPT_CONFIG_INIT(
          &interruptConfig,
          CryptoDeviceEvtInterruptIsr,
          NULL);

    interruptConfig.EvtInterruptDpc = CryptoDeviceEvtInterruptDpc;
    interruptConfig.EvtInterruptEnable = CryptoDeviceEvtInterruptEnable;
    interruptConfig.EvtInterruptDisable = CryptoDeviceEvtInterruptDisable;

    interruptConfig.InterruptTranslated = InterruptTranslated;
    interruptConfig.InterruptRaw = InterruptRaw;
    interruptConfig.SpinLock = devContext->InterruptLock;

    NTSTATUS status = WdfInterruptCreate(
          Device,
          &interruptConfig,
          &attributes,
          Interrupt);

    return status;
}
```

This function initializes the interrupt parameters based on the parameters passed by the *CryptoDeviceEvtDevicePrepareHardware* function. Each of the interrupt objects also has its own context (the DEVICE_INTERRUPT_CONTEXT structure) that will be used for handling the interrupts.

The function specifies two important callbacks that will be used for handling the interrupts:

```
BOOLEAN CryptoDeviceEvtInterruptIsr(
    _In_ WDFINTERRUPT Interrupt,
    _In_ ULONG                 MessageID
);

VOID CryptoDeviceEvtInterruptDpc(
    _In_ WDFINTERRUPT Interrupt,
    _In_ WDFOBJECT       Device
);
```

We'll get back to the implementation of these functions later, but for now we need to focus on the *CryptoDeviceEvtInterruptIsr* function callback. This callback is called every time the device sends an interrupt. The *MessageID* parameter is the number of the MSI interrupt sent by the device. In the case of an INTx interrupt, the *MessageID* is always 0.

Now we've nearly finished creating a minimum device driver. The only thing left to do is add and process two callbacks responsible for entering and exiting the working state D0. For more information on these states, see Microsoft documentation.

There are two functions used for handling these events:

1. *CryptoDeviceEvtDeviceD0EntryPostInterruptsEnabled*
2. *CryptoDeviceEvtDeviceD0ExitPreInterruptsDisabled*

Let's take a closer look at each of them.

```
NTSTATUS CryptoDeviceEvtDeviceD0EntryPostInterruptsEnabled(
    _In_ WDFDEVICE Device,
    _In_ WDF_POWER_DEVICE_STATE PreviousState
)
{
    PAGED_CODE();

    PDEVICE_CONTEXT ctx = DeviceGetContext(Device);
    CryptoDeviceInterruptEnable(&ctx->CryptoDevice);

    return STATUS_SUCCESS;
}

NTSTATUS CryptoDeviceEvtDeviceD0ExitPreInterruptsDisabled(
    _In_ WDFDEVICE Device,
    _In_ WDF_POWER_DEVICE_STATE TargetState
)
{
    PAGED_CODE();

    PDEVICE_CONTEXT ctx = DeviceGetContext(Device);
    CryptoDeviceInterruptDisable(&ctx->CryptoDevice);

    return STATUS_SUCCESS;
}
```

These functions use code we haven't described yet. However, all these functions do is change the value of the *CryptoDeviceIo :: InterruptFlag* field.

The *CryptoDeviceEvtDeviceD0EntryPostInterruptsEnabled* function sets the InterruptFlag at 0xFF, allowing the device to generate interrupts.

The *CryptoDeviceEvtDeviceD0ExitPreInterruptsDisabled* function, in turn, sets the InterruptFlag at 0x00, thus prohibiting the generation of interrupts.

Interrupts must be disabled on the device's side because WDF won't deliver interrupts to the driver after exiting the D0 state and, therefore, no function will be able to handle these interrupts.

Working with I/O memory

Working with device I/O memory looks like working with ordinary memory, only it's accessible through the READ_REGISTER_*XXX* and WRITE_REGISTER_*XXX* families of functions, taking into account the point alignment to the I/O memory.

The distinctive feature of I/O memory is that it's not only memory for data storage but is also a channel for communication with the device. This means that addressing this memory can cause some actions on the device's side. To work properly with I/O memory, you need to take into account the logic of the device performance and its specifications, like its memory structure, pre- or post-conditions, the correctness of written values, and so on. It's also necessary to ensure the synchronization of memory access:

1. Common synchronization primitives like WDFWAITLOCK are used for synchronization of driver-to-driver access (simultaneous access from the driver).
2. Device specifications and preconditions are used for synchronization of driver-to-device access (simultaneous access from the driver and device).

The *CryptoDevice.c* file contains all logic of working with the I/O memory. Only the functions from this file work with the device I/O memory and ensure the validation of the values written in the I/O memory. Basically, all functions from the *CryptoDevice.c* just read or write values to the memory.

For instance:

```
VOID CryptoDeviceProgramDmaIn(
    _In_ PCRYPTO_DEVICE Device,
    _In_ ULONG32 DmaAddress,
    _In_ ULONG32 DmaPagesCount,
    _In_ ULONG32 DmaSizeInBytes
)
{
    WRITE_REGISTER_ULONG(&Device->Io->DmaInAddress, DmaAddress);
    WRITE_REGISTER_ULONG(&Device->Io->DmaInPagesCount, DmaPagesCount);
    WRITE_REGISTER_ULONG(&Device->Io->DmaInSizeInBytes, DmaSizeInBytes);
}

VOID CryptoDeviceSetCommand(
    _In_ PCRYPTO_DEVICE Device,
    _In_ CryptoDeviceCommand Command
)
{
      ASSERT(CryptoDevice_AesCbcEncryptCommand == Command
          || CryptoDevice_AesCbcDecryptCommand == Command
          || CryptoDevice_Sha2Command == Command);

      KeClearEvent(&Device->ErrorEvent);
      KeClearEvent(&Device->ReadyEvent);
      KeClearEvent(&Device->CancelEvent);
      WRITE_REGISTER_UCHAR(&Device->Io->Command, (UINT8)Command);
}
```

The first function, *CryptoDeviceProgramDmaIn*, writes data about the DMA IN buffer to the I/O memory, while the second function, *CryptoDeviceSetCommand*, issues a command that's executed by the device immediately.

Interrupt handling

Interrupt handling usually includes several stages. WDF significantly simplifies interrupt handling by providing several scenarios for interrupt processing. In our driver, we use a classic scenario called Interrupt Service Routine (ISR) - Deferred Procedure Calls (DPC).

To start, let’s specify that a good device doesn’t generate interrupts for no reason. An interrupt is a mechanism for a device to notify a driver about an event. For instance, an interrupt can notify when writing to DMA memory has finished, reading from DMA memory has finished, an error has occurred, and so on.

In our test driver, interrupt handling includes the following stages:

1. The operating system delivers an interrupt to the driver’s ISR callback.
2. The ISR callback defines the type of interrupt and puts the call to the DPC callback in the queue.
3. The DPC callback sets an event (KEVENT) that’s responsible for processing this type of interrupt.
4. The thread that waits for interrupt events (KEVENTs) receives a notification about an incoming interrupt (or the operation execution).

When creating interrupt objects in the *CryptoDeviceInterruptCreate* function, we specify two callbacks:

```
BOOLEAN CryptoDeviceEvtInterruptIsr(
    _In_ WDFINTERRUPT Interrupt,
    _In_ ULONG        MessageID
);

VOID CryptoDeviceEvtInterruptDpc(
    _In_ WDFINTERRUPT Interrupt,
    _In_ WDFOBJECT    Device
);
```

CryptoDeviceEvtInterruptIsr is the first driver function that’s invoked by WDF for providing interrupts. Its first parameter is the interrupt object through which you can receive the WDFDEVICE object and its context (the DEVICE_CONTEXT structure).

The second parameter is the number of MSI interrupts sent by the device (if the operating function has issued all necessary interrupts). In the case of a single MSI or INTx interrupt, the MessageID is equal to 0.

There's no need to provide a separate number for MSI #0 for cases when the operating system can allocate only one interrupt instead of all requested interrupts. The driver always knows its interrupt mode (INTx, one MSI, all required MSIs), which is why MSI #0 can be used in any mode. In our case, MSI #0 is used only for INTx or for one MSI, while MSI #1, #2, and #3 are used when all required MSIs are available. This is done to simplify the logic of the test driver.

Each WDFINTERRUPT object has its context, which is described by the following structure:

```
typedef struct _DEVICE_INTERRUPT_CONTEXT
{
    MSI_FLAGS Msi;
} DEVICE_INTERRUPT_CONTEXT, *PDEVICE_INTERRUPT_CONTEXT;
```

This context is specified when creating WDFINTERRUPT in the *CryptoDeviceInterruptCreate* function:

```
WDF_OBJECT_ATTRIBUTES attributes;
WDF_OBJECT_ATTRIBUTES_INIT_CONTEXT_TYPE(
        &attributes,
        DEVICE_INTERRUPT_CONTEXT);
….
NTSTATUS status = WdfInterruptCreate(
        Device,
       &interruptConfig,
       &attributes,
       Interrupt);
```

Here there's only one field with interrupt flags. While the ISR callback is invoked on a high interrupt request level (IRQL), the capabilities of *CryptoDeviceEvtInterruptIsr* are very limited and include only filling the MSI_FLAGS field and DPC planning. The ISR callback code looks like this:

```
BOOLEAN CryptoDeviceEvtInterruptIsr(
    _In_ WDFINTERRUPT Interrupt,
    _In_ ULONG        MessageID
)
{
    PDEVICE_INTERRUPT_CONTEXT interruptContext = GetInterruptContext(Interrupt);
    PDEVICE_CONTEXT ctx = DeviceGetContext(WdfInterruptGetDevice(Interrupt));

    switch (MessageID)
    {
    case CryptoDevice_MsiZero:
        if (!CryptoDeviceInerruptGetFlags(&ctx->CryptoDevice, &interruptContext-
>Msi))
        {
            return FALSE;
        }
        break;

    case CryptoDevice_MsiError:
    case CryptoDevice_MsiReady:
    case CryptoDevice_MsiReset:
        ASSERT(MessageID < ARRAYSIZE(interruptContext->Msi.Flags));
        interruptContext->Msi.Flags[MessageID] = TRUE;
        break;
    }

    WdfInterruptQueueDpcForIsr(Interrupt);
    return TRUE;
}
```

The first two lines of the function include the interrupt context and the device context. The switch processes three possible modes of interrupt handling:

1. The *CryptoDevice_MsiZero* case is executed if INTx or single MSI#0 is used, it's necessary to determine which of three possible events (Error, Ready, Reset) the device is notifying about. To do this, it's necessary to read the corresponding fields in the device I/O memory space. According to the specifications, the device settles one of the flags in the I/O memory before generating an interrupt. If any of the flags weren't settled (which is possible with INTx, when one interrupt line is simultaneously used by several devices), it's necessary to immediately return FALSE from the ISR callback.
2. The next 3 cases (CryptoDevice_MsiError, CryptoDevice_MsiReady and CryptoDevice_MsiReset) are executed if all required MSIs have been allocated and assigned to the device. The MessageID mentions a certain device event, so there's no need to check the device I/O memory.

The function plans an invocation to the DPC callback for the interrupt object and returns TRUE if the interrupt was processed by the driver. WDF plans the DPC callback, which was transferred when creating *WDFINTERRUPT: CryptoDeviceEvtInterruptDpc*, providing that the interrupt object's context includes all information necessary for the device event to proceed.

Next, WDF invokes *CryptoDeviceEvtInterruptDpc*:

```
VOID CryptoDeviceEvtInterruptDpc(
    _In_ WDFINTERRUPT Interrupt,
    _In_ WDFOBJECT    Device
)
```

```
    {
            PDEVICE_INTERRUPT_CONTEXT interruptContext =
                        GetInterruptContext(Interrupt);

            WdfInterruptAcquireLock(Interrupt);

            MSI_FLAGS msi = interruptContext->Msi;
            RtlZeroMemory(&interruptContext->Msi, sizeof(interruptContext->Msi));

            WdfInterruptReleaseLock(Interrupt);

            PDEVICE_CONTEXT device = DeviceGetContext(Device);
            CryptoDeviceInterruptHandler(&device->CryptoDevice, &msi);
    }
```

First of all, this function copies the interrupt context into the local variable and resets all flags in this context. All work with the interrupt context in the DPC callback should be synchronized with the ISR callback using a special mechanism.

Different MSIs can be processed with different processor cores simultaneously. The DPC callback can also be processed in parallel with the ISR callback on different processor cores. In order to synchronize the work of ISR and DPC callbacks, the *CryptoDeviceInterruptCreate* function assigns the same WDFSPINLOCK object to all initiated interrupts:

```
interruptConfig.SpinLock = devContext->InterruptLock;
```

WDF will automatically choose the highest IRQL among all interrupts and use it for synchronizing work with them. Using this spin lock, WDF will increase the IRQL to the maximum chosen level each time when calling ISR. In this way, the callback will be executed synchronously and all device interrupts will be processed sequentially.

The DPC callback uses the following invocation:

```
WdfInterruptAcquireLock(Interrupt);
```

This call acquires the spin lock in the same way on the same device IRQL as it's done for ISR. In this way, we get exclusive access to the interrupt context, taking into account the ISR and multiprocessing. While a spin lock uses Device IRQL (increasing the level to the device IRQL), its code capabilities are limited to copying MSI_FLAGS.

Then, the DPC callback invokes the *CryptoDeviceInterruptHandler* function, which is executed at the DISPATCH LEVEL, and sets the corresponding driver events:

```
VOID CryptoDeviceInterruptHandler(
    _In_ PCRYPTO_DEVICE Device,
    _In_ PMSI_FLAGS Msi
)
{
      if (Msi->Flags[CryptoDevice_MsiError])
      {
            KeSetEvent(&Device->ErrorEvent, IO_NO_INCREMENT, FALSE);
      }

      if (Msi->Flags[CryptoDevice_MsiReady])
      {
            KeSetEvent(&Device->ReadyEvent, IO_NO_INCREMENT, FALSE);
      }

      if (Msi->Flags[CryptoDevice_MsiReset])
      {
            KeSetEvent(&Device->ResetEvent, IO_NO_INCREMENT, FALSE);
      }
}
```

The interrupt handling is now finished. Any function that is interested in the device MSIs must use those event objects. Those additional events are necessary because the ISR and DPC callbacks can be executed in any threads. That's why if any driver function wants to wait for the device interrupt, it should use additional mechanisms for accepting such events.

Working with DMA

The *CryptoDeviceMemory.c* file contains all functions for the driver's work with DMA. The WDFDMAENABLER object is used to allocate the DMA memory, which also describes the driver's capabilities:

```
WDF_DMA_ENABLER_CONFIG dmaConfig;
WDF_DMA_ENABLER_CONFIG_INIT(
      &dmaConfig,
      WdfDmaProfileScatterGather64Duplex,
      MAXSIZE_T);
dmaConfig.WdmDmaVersionOverride = 3;
```

Here's what Microsoft says about *WdfDmaProfileScatterGather64Duplex*:

> The device supports packet-based, scatter/gather DMA operations, using 64-bit addressing. The device also supports duplex operation.

The device driver uses a common buffer (memory is contiguous for the device), the address and size of which is transferred to the device through the I/O memory space. The driver fills this contiguous buffer with an array of device logical bus addresses. This array describes the user buffer for input or output data. All work for allocating DMA memory is performed within this function:

```
NTSTATUS MemCreateDmaForUserBuffer(
    _In_ PVOID UserBuffer,
    _In_ ULONG UserBufferSize,
    _In_ WDFDMAENABLER DmaEnabler,
    _In_ BOOLEAN WriteToDevice,
    _Out_ PDMA_USER_MEMORY Dma
);
```

UserBuffer and *UserBufferSize* contain data that should be transferred to the device. *UserBuffer* points to the user mode address for the context of the current process. There are no additional requirements for this buffer, as the memory can be allocated in any possible way and there are no limitations for alignment or size (the buffer's size is limited only by system resources). *DmaEnabler* describes the device capabilities for working with DMA and is used as a parameter for the WDF functions. *WriteToDevice* is TRUE if the device will write in this memory and FALSE if the device will only read it. On the basis of this parameter, WDF will process the processor cache either before or after handling the DMA transaction. *Dma* is an output structure with all allocated and filled memory buffers.

The execution of the *MemCreateDmaForUserBuffer* function can be divided into three stages:

1. Creating MDL for the user buffer:

```
//
// Create MDL and validate the memory range
//
NT_CHECK_GOTO_CLEAN(MemCreateUserBufferMdl(
            UserBuffer,
            UserBufferSize,
            WriteToDevice ? IoReadAccess : IoWriteAccess,
            &Dma->UserBufferMdl));
```

At this stage, the user buffer is validated and, if validation is successful, MDL is created. MDL describes the locked user mode memory pages.

2. Allocating the common buffer:

```
WDF_COMMON_BUFFER_CONFIG dmaBufConfig;
WDF_COMMON_BUFFER_CONFIG_INIT(
            &dmaBufConfig,
            CRYPTO_DEVICE_PAGE_MASK);

 NT_CHECK_GOTO_CLEAN(WdfCommonBufferCreateWithConfig(
          DmaEnabler,
          Dma->DmaBufferSize,
          &dmaBufConfig,
          WDF_NO_OBJECT_ATTRIBUTES,
          &Dma->DmaBuffer));

dmaBufVa = WdfCommonBufferGetAlignedVirtualAddress(Dma->DmaBuffer);
dmaBufPa = WdfCommonBufferGetAlignedLogicalAddress(Dma->DmaBuffer);
RtlZeroMemory(dmaBufVa, Dma->DmaBufferSize);

Dma->DmaAddress = CRYPTO_DEVICE_TO_DMA(dmaBufPa.QuadPart);
```

The *WdfCommonBufferCreateWithConfig* function can allocate the contiguous memory for the device side on the basis of the WDFDMAENABLER object. The size of this buffer is calculated on the basis of *UserBuffer* and *UserBufferSize*, as there should be enough memory to save the address of each separate page from the *UserBuffer* and *UserBufferSize* regions.

The *WdfCommonBufferGetAlignedVirtualAddress* function returns the virtual address of the allocated buffer through which we can work with the memory from the driver.

The *WdfCommonBufferGetAlignedLogicalAddress* function returns the DMA memory address which should be transferred to the device. This address is not necessarily equal to its physical address in RAM, but it usually is. Before transferring this address to the device, the driver wraps it from a 64-bit value to a 32-bit value. The allocated common buffer address should be 4KB, which is specified in these lines:

```
WDF_COMMON_BUFFER_CONFIG dmaBufConfig;
WDF_COMMON_BUFFER_CONFIG_INIT(
            &dmaBufConfig,
            CRYPTO_DEVICE_PAGE_MASK); // 4KB alignment
```

That’s why a DMA address less than 12 bits will always be equal to zero.

3. At the last stage, the common buffer is filled with the UserBuffer (MDL) pages. All this is done through the WDF functions, which perform additional work such as resetting the processor cache and allocating mapped registers. For the driver, it’s enough to create, initialize, and perform the DMA transaction. WDF will do the rest:

```
//
  // Create DMA transaction
  //
  NT_CHECK_GOTO_CLEAN(WdfDmaTransactionCreate(
        DmaEnabler,
        WDF_NO_OBJECT_ATTRIBUTES,
        &Dma->DmaTransaction));

  PVOID va = MmGetMdlVirtualAddress(Dma->UserBufferMdl);
  ULONG length = MmGetMdlByteCount(Dma->UserBufferMdl);

  ASSERT(va == UserBuffer);
  ASSERT(length == UserBufferSize);

  if (0 == length)
  {
        NT_CHECK_GOTO_CLEAN(STATUS_UNSUCCESSFUL);
  }

  WDF_DMA_DIRECTION dmaDirection = WriteToDevice
           ? WdfDmaDirectionWriteToDevice
           : WdfDmaDirectionReadFromDevice;

  NT_CHECK_GOTO_CLEAN(WdfDmaTransactionInitialize(
         Dma->DmaTransaction,
         MemEvtProgramDma,
         dmaDirection,
         Dma->UserBufferMdl,
         va,
         length));

  //
  // Fill out contiguous memory with SG values
  //
  NT_CHECK_GOTO_CLEAN(WdfDmaTransactionExecute(
    Dma->DmaTransaction,
    Dma));
```

Filling of the common buffer is executed in the WDF callback function, called *MemEvtProgramDma*, which WDF calls immediately after all actions for working with

MDL have been performed. WDF transfers the structure of SCATTER_GATHER_LIST to this callback. SCATTER_GATHER_LIST contains the logical device addresses for the whole user mode buffer described through MDL at the first stage. Then the callback transfers these addresses from the common buffer allocated in the second stage.

This is what the work with DMA memory looks like for a testing device that supports scatter-gather 64-bit DMA transfer. The main work is performed by the framework, which is why it's so easy to work with the DMA memory in WDF.

Sending requests to the device

When everything is ready for the device communication and control, we can unite the work with DMA, MSI, and the I/O memory into logical requests for the device. According to the device specification, the following operations are available:

1. Reset device
2. Calculate a SHA-2 hash
3. Encrypt with AES 256
4. Decrypt with AES 256
5. Get device status

You can find the functions for working with these five requests in the *CryptoDeviceLogic.c* file. The file contains kernel mode functions which are used as interface to work with the device.

Those functions also contain a synchronization logic for the device which is need because the device supports only a single thread of command execution and doesn't have an inner command queue. The driver uses WDFWAITLOCK to ensure access to the device only from one thread.

Let's look closer at these operations:

1. *Reset device* is software reset of the device, which is used to clear the error state or cancel an operation on the device's side. In the event of a successful reset, the device shouldn't call the DMA memory until the next command so the DMA memory can be cleared. Here's the function code:

```
NTSTATUS CryptoDeviceResetRequest(
    _In_ PCRYPTO_DEVICE Device
)
{
      PAGED_CODE();
```

```
        WdfWaitLockAcquire(Device->ResetLock, NULL);

        NTSTATUS status = STATUS_UNSUCCESSFUL;
        WdfWaitLockAcquire(Device->IoLock, NULL);

        if (CryptoDeviceGetState(Device) != CryptoDevice_ResetState)
        {
              CryptoDeviceReset(Device);
              status = STATUS_SUCCESS;
        }
        else
        {
              status = STATUS_DEVICE_BUSY;
        }

        WdfWaitLockRelease(Device->IoLock);

        NT_CHECK_GOTO_CLEAN(status);
        NT_CHECK_GOTO_CLEAN(CryptoDeviceWaitReset(Device));
        KeSetEvent(&Device->CancelEvent, IO_NO_INCREMENT, FALSE);

  clean:
        WdfWaitLockRelease(Device->ResetLock);
        return status;
  }
```

The *CryptoDeviceResetRequest* function uses one additional (its own) WDFWAITLOCK object, the name of the variable is ResetLock (details are in code above). ResetLock is locked for the whole time of function execution to avoid parallel queries to reset the device.

WDFWAITLOCK IoLock is used to synchronize access to the device I/O memory. This lock should use all functions that invoke the functions from *CryptoDevice.c*.

CryptoDeviceResetRequest sets the *State* field in the I/O space of *CryptoDevice_ResetCommand* and waits for the result of the Reset event, which will be set from ISR->DPC. If successful, the function sets CancelEvent, which is used by other functions in this file to determine when the request was cancelled.

2. Operations with AES and SHA256 have one implementation and only differ in terms of the command number:

```
NTSTATUS CryptoDeviceAesCbcEncryptRequest(
    _In_ PCRYPTO_DEVICE Device,
    _In_ PVOID UserBufferIn,
    _In_ ULONG UserBufferInSize,
    _In_ PVOID UserBufferOut,
    _In_ ULONG UserBufferOutSize
)
{
      PAGED_CODE();

      return CryptoDeviceCommandRequestInOut(
            Device,
            UserBufferIn,
            UserBufferInSize,
            UserBufferOut,
            UserBufferOutSize,
            CryptoDevice_AesCbcEncryptCommand);
                          // or CryptoDevice_Sha2Command
                          // or CryptoDevice_AesCbcDecryptCommand
}
```

The *CryptoDeviceCommandRequestInOut* function process any requests used by the IN and OUT user mode buffers. This function allocates DMA memory to handle a request for both the IN and OUT buffer:

```
DMA_USER_MEMORY bufferIn = { 0 };
DMA_USER_MEMORY bufferOut = { 0 };
NTSTATUS status = STATUS_UNSUCCESSFUL;

if (0 != UserBufferInSize)
{
      NT_CHECK_GOTO_CLEAN(MemCreateDmaForUserBuffer(
             UserBufferIn,
             UserBufferInSize,
             Device->DmaEnabler,
             FALSE,
            &bufferIn));
}

if (0 != UserBufferOutSize)
{
      NT_CHECK_GOTO_CLEAN(MemCreateDmaForUserBuffer(
             UserBufferOut,
             UserBufferOutSize,
             Device->DmaEnabler,
             TRUE,
             &bufferOut));
}
```

After that, the function writes the information about the DMA buffers to the I/O memory and sets ID commands for their execution (work with I/O memory is performed with acquired IoLock):

```
WdfWaitLockAcquire(Device->IoLock, NULL);

   if (CryptoDeviceGetErrorCode(Device) != CryptoDevice_NoError)
   {
         status = STATUS_DEVICE_DATA_ERROR;
   }
   else if (CryptoDeviceGetState(Device) != CryptoDevice_ReadyState)
   {
         status = STATUS_DEVICE_BUSY;
   }
   else
   {
         CryptoDeviceProgramDmaIn(Device,
                                  bufferIn.DmaAddress,
                                  bufferIn.DmaCountOfPages,
                                  UserBufferInSize);

       CryptoDeviceProgramDmaOut(Device,
                                  bufferOut.DmaAddress,
                                  bufferOut.DmaCountOfPages,
                                  UserBufferOutSize);

       CryptoDeviceSetCommand(Device, Command);
       status = STATUS_SUCCESS;
   }

   WdfWaitLockRelease(Device->IoLock);
```

Then the function waits for the end of the request processing by using the call:

```
CryptoDeviceWaitForReadyOrError(Device, NULL);
```

This function waits for one of three events:

1. ReadyEvent — The operation has completed successfully (Ready MSI).
2. ErrorEvent — An error has occured on the device’s side during execution (Error MSI).
3. CancelEvent — The operation has been cancelled from another thread.

Finally, the function releases all allocated resources and returns the status of the command execution.

3. The last command obtains the current state of the device:

    ```
    NTSTATUS CryptoDeviceStateRequest(
        _In_ PCRYPTO_DEVICE Device,
        _Out_ PDEVICE_STATE State
    )
    {
          PAGED_CODE();

          WdfWaitLockAcquire(Device->IoLock, NULL);
          State->State = CryptoDeviceGetState(Device);
          State->Error = CryptoDeviceGetErrorCode(Device);
          WdfWaitLockRelease(Device->IoLock);
          return STATUS_SUCCESS;
    }
    ```

    This call is used to monitor the status of the device and get information about an error on the device’s side.

The mentioned device logical requests can be used for processing requests from user mode applications.

Processing requests from a user mode application

The driver provides an input-output control (IOCTL) interface for executing the five mentioned commands from the application:

```
//
// Reset device software
//
// IN: None
// OUT: None
//
#define IOCTL_CRYPTO_DEVICE_RESET ...

//
// Get current device state
//
// IN: None
// OUT: CryptoDeviceStatus
//
#define IOCTL_CRYPTO_DEVICE_GET_STATUS ...
```

```
//
// Encrypt buffer with AES CBC
//
// IN: CryptoDeviceBufferInOut
// OUT: None
//
#define IOCTL_CRYPTO_DEVICE_AES_CBC_ENCRYPT ...

//
// Decrypt buffer with AES CBC
//
// IN: CryptoDeviceBufferInOut
// OUT: None
//
#define IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT ...

//
// Calculate SHA256 for the buffer
//
// IN: CryptoDeviceBufferInOut
// OUT: None
//
#define IOCTL_CRYPTO_DEVICE_SHA256 ...
```

The interface for working with the device driver from the user mode application is described in the *Public.h* file, which contains IOCTLs, data structure, and the GUID to access the device driver.

The user mode code in the *CryptoDeviceCtrl.cpp* file contains the *CryptoDeviceCtrl* class, which encapsulates the work with the IOCTL driver and provides a high-level interface for working with it:

```
struct DeviceStatus
{
    CryptoDeviceState State;
    CryptoDeviceErrorCode ErrorCode;
};

class CryptoDeviceCtrl
{
public:
    void ResetDevice() const;
    DeviceStatus GetDeviceStatus() const;

    void AesCbcEncrypt(const void * bufferIn,
                       size_t bufferInSize,
                       void * bufferOut,
                       size_t bufferOutSize) const;
```

```
        void AesCbcDecrypt(const void * bufferIn,
                           size_t bufferInSize,
                           void * bufferOut,
                           size_t bufferOutSize) const;

        std::vector<uint8_t> Sha256(const void * buffer,
                                    size_t bufferSize) const;

        void Sha256(const void * buffer,
                    size_t bufferSize,
                    Sha256Buffer& hash) const;
    };
```

In order to work with the driver, all you need to do is create a *CryptoDeviceCtrl* object and call one of the methods. Also, the IN and OUT buffers may point to the same memory region for the AES functions, provided that the OUT buffer size is enough for writing the aligned results. In this case, the device will put the AES request in place without additional buffers.

The following is implemented at the current stage:

1. QEMU virtual device
2. Windows PCI device driver
3. User mode interface to control the driver

Further, you need to test all these, taking the following into account:

1. Setting up Windows kernel debugger for QEMU
2. Quality assurance of the driver code
3. Writing tests for the driver
4. Testing the driver with Driver Verifier and WDF Verifier

Testing and debugging

In our case, driver testing and debugging is only possible with the QEMU virtual machine, as our device is virtual and is implemented in QEMU.

Here's our environment for driver debugging:

1. Ubuntu 18.04 x64 for QEMU
2. Windows 10 x64 as a guest operating system

Setting up Windows kernel debugging for a Windows guest OS in QEMU

There are several ways of setting up Windows kernel debugging for a Windows guest operating system in QEMU. In our case, we use Windows Network debugging, which supports kernel debugging over a local network. This method requires the following:

1. The target and host operating system must be on the same local network.
2. The network adapter on the target operating system must be mentioned in the list of supported devices.

In order to place the QEMU guest operating system (Windows target) on the same local network as the Windows host, create a TAP device on Ubuntu — where the QEMU guest operating system will run, provided that Ubuntu and the Windows host are on the same local network — and connect it with a bridge to the interface of the local network.

The Ubuntu script (without checking the results of commands) looks like this:

```
NETWORK_INTERFACE=eth0
BRIDGE_NAME=qemu_br0
TAP_NAME=`tunctl -b`

ip link add $BRIDGE_NAME type bridge
ip addr flush dev $NETWORK_INTERFACE
ip link set $NETWORK_INTERFACE master $BRIDGE_NAME
ip link set $TAP_NAME master $BRIDGE_NAME
ip link set dev $BRIDGE_NAME up
ip link set dev $TAP_NAME up
dhclient $BRIDGE_NAME
```

After performing these commands, a TAP device will be created. This device should be used while running QEMU in the following way:

```
TAP_NAME=tap0
./qemu/x86_64-softmmu/qemu-system-x86_64 \
        -enable-kvm \
        -m 4G \
        -cpu host \
        -smp cpus=4,cores=4,threads=1,sockets=1 \
        -device pci-crypto,aes_cbc_256=secret \
        -hda /<path>/windows10.x64.img \
        -device e1000,netdev=network0 \
        -netdev tap,id=network0,ifname=$TAP_NAME,script=no,downscript=no
```

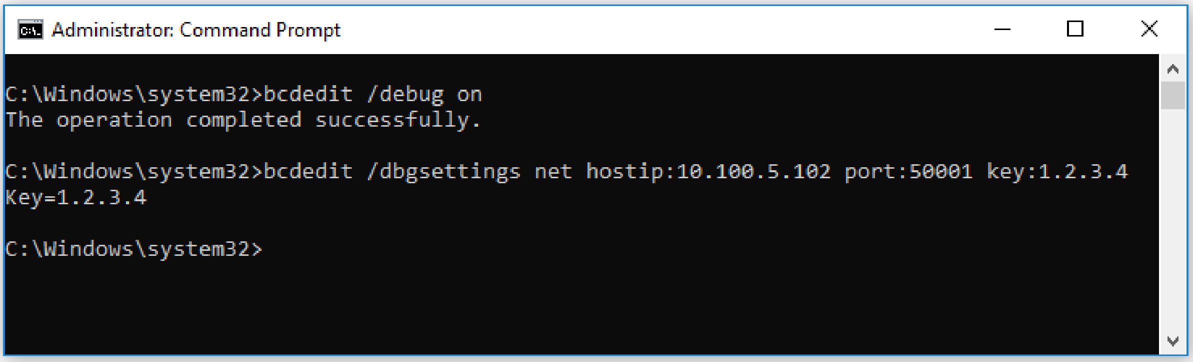
TAP_NAME=tap0 is a TAP device.
-*device e1000* makes QEMU use the network adapter E1000, which is supported by Windows for network kernel debugging.

If everything has been done correctly, the Windows 10 QEMU guest operating system will automatically get an IP address of the local network created by Ubuntu (provided that DHCP is available). Thus, all three operating systems will be available on the same local network: Ubuntu, Windows target (the Windows QEMU guest) and Windows host (with the driver source code and WinDBG).

Then, run *cmd* with admin rights on the Windows 10 guest OS and execute two commands

- *bcdedit /debug on*
- *bcdedit /dbgsettings net hostip:w.x.y.z port:50001 key:1.2.3.4*

where w.x.y.z is the IP address of the Windows host (make sure there's a connection via ping in advance). Now restart the Windows guest operating system.

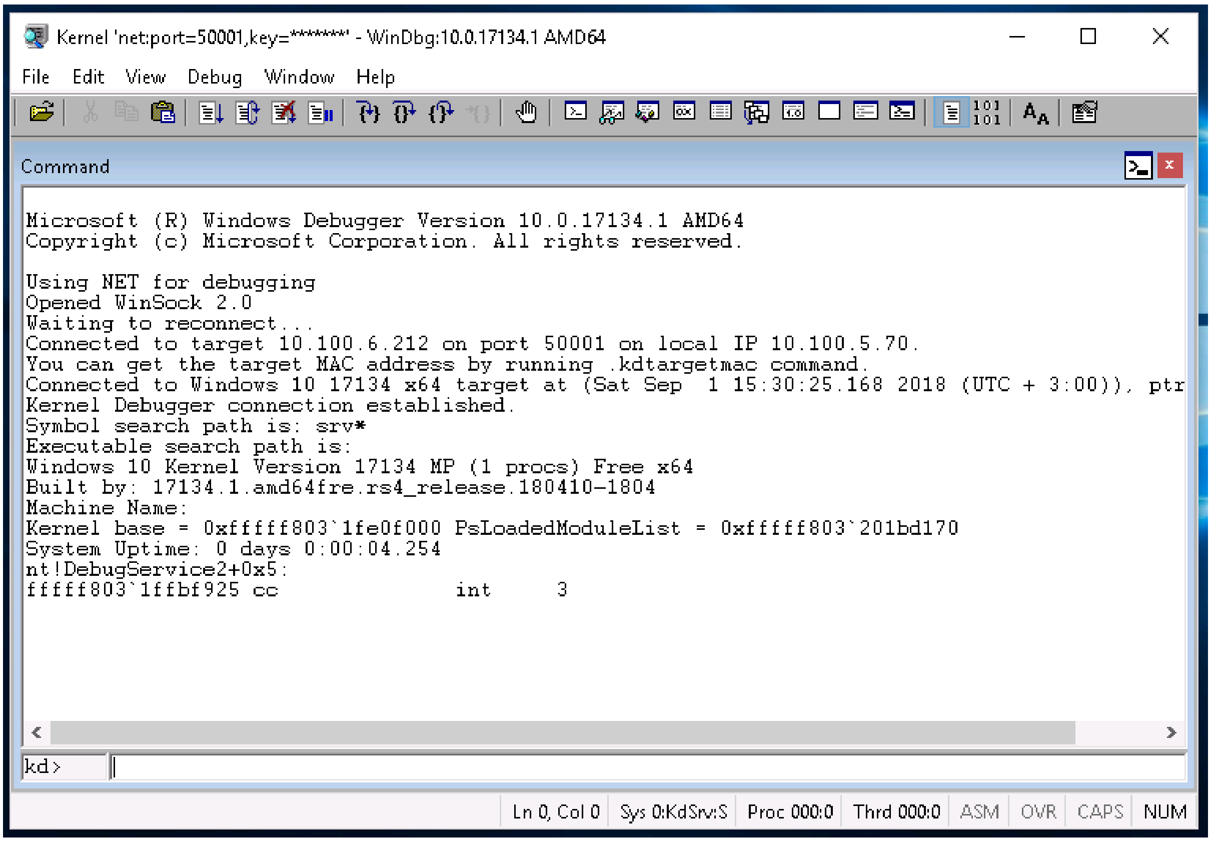


```
C:\Windows\system32>bcdedit /debug on
The operation completed successfully.

C:\Windows\system32>bcdedit /dbgsettings net hostip:10.100.5.102 port:50001 key:1.2.3.4
Key=1.2.3.4

C:\Windows\system32>
```

After that, run Windbg on the Windows host operating system:

cd "C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\"
start windbg.exe -b -k net:port=50001,key=1.2.3.4 -c "ed Kd_DEFAULT_MASK 0xf"

If you've done everything right, WinDGB should now be connected to the Windows guest operating system and then WinDGB should stop. To continue the work of the Windows guest operating system, press F9:



```
Microsoft (R) Windows Debugger Version 10.0.17134.1 AMD64
Copyright (c) Microsoft Corporation. All rights reserved.

Using NET for debugging
Opened WinSock 2.0
Waiting to reconnect...
Connected to target 10.100.6.212 on port 50001 on local IP 10.100.5.70.
You can get the target MAC address by running .kdtargetmac command.
Connected to Windows 10 17134 x64 target at (Sat Sep  1 15:30:25.168 2018 (UTC + 3:00)), ptr
Kernel Debugger connection established.
Symbol search path is: srv*
Executable search path is:
Windows 10 Kernel Version 17134 MP (1 procs) Free x64
Built by: 17134.1.amd64fre.rs4_release.180410-1804
Machine Name:
Kernel base = 0xfffff803`1fe0f000 PsLoadedModuleList = 0xfffff803`201bd170
System Uptime: 0 days 0:00:04.254
nt!DebugService2+0x5:
fffff803`1ffbf925 cc              int     3
```

When setting up the kernel debugger, it's better to shut down the Windows guest OS and run QEMU with *-snapshot*.
If the IP address of the Windows host OS changes, you'll need to re-execute the command on the Windows guest OS with the new IP address and restart the system:

```
bcdedit /dbgsettings net hostip:w.x.y.z port:50001 key:1.2.3.4
shutdown -r -t 0
```

Here's how you can turn off the Windows kernel debugger:

```
bcdedit /debug off
```

Note: If you run the operating system during driver debugging and the Blue Screen of Death (BSOD) appears, meaning that you can't run the operating system because of a driver bug, you can just delete the line with the device (in our case, *-device pci-crypto,aes_cbc_256=secret*) from the QEMU command line options. The driver won't run in this case as there won't be a device for it.

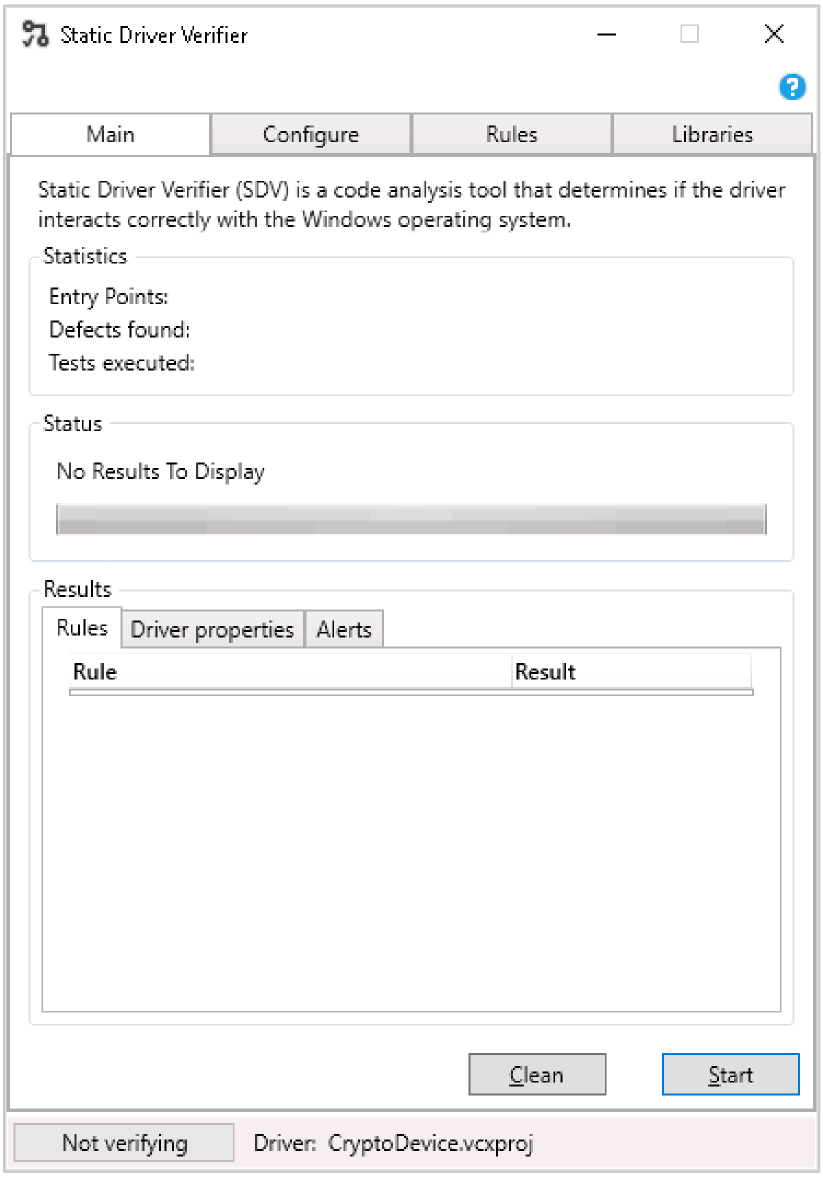
Quality control of driver code

During driver development, we recommend to do the following:

1. Use Warning Level 4
2. Set the Static Code Analyzer to Microsoft Native Recommended Rules
3. Use SAL 2.0 annotations
4. Clearly specify which functions on which IRQL will work (see PAGED_CODE() and #pragma alloc_text)

Pay attention to all compiler warnings, look for their causes, and fix sources.

Additionally, Microsoft offers Static Driver Verifier, a static verification tool that's capable of discovering defects and design issues in drivers.



Static Driver Verifier (SDV) is a code analysis tool that determines if the driver interacts correctly with the Windows operating system.

Statistics

Entry Points:
Defects found:
Tests executed:

Status

No Results To Display

Results

Rule	Result

Not verifying

Driver: CryptoDevice.vcxproj

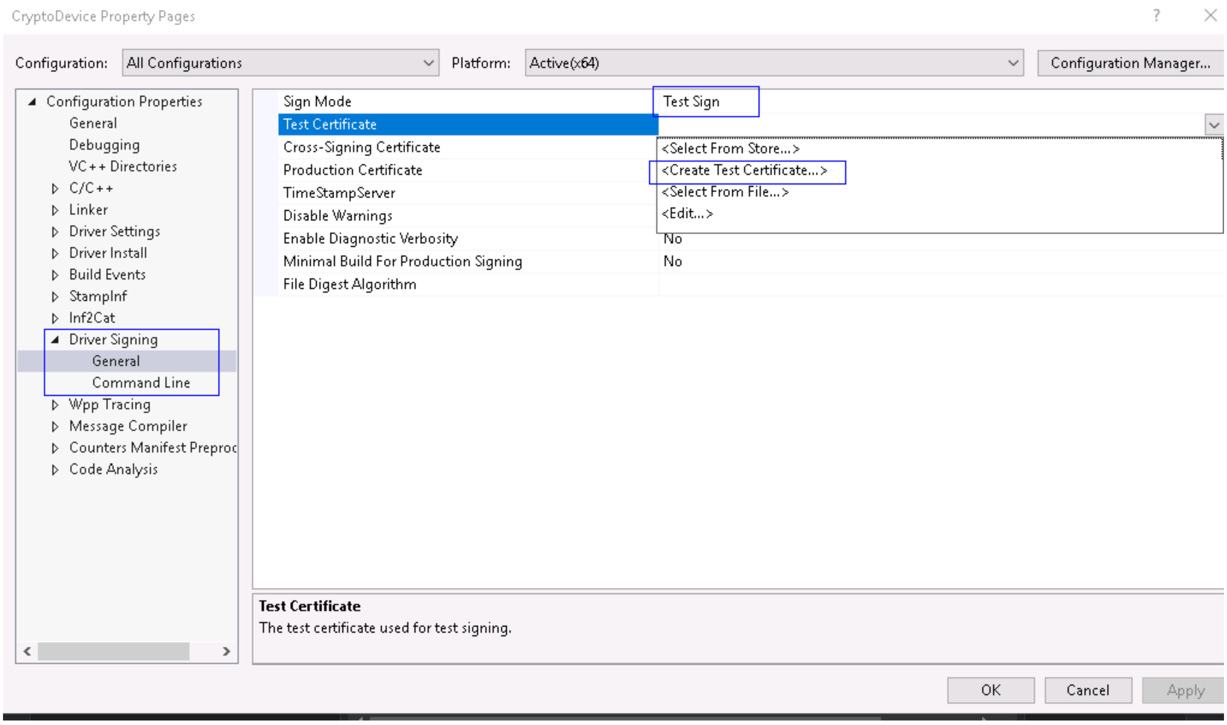
Driver installation

During driver installation, the Windows x64 kernel verifies the signature of the *.sys* file, which is why you need to prepare for loading and testing the x64 driver version in one of the following ways:

1. Set up Windows kernel debugging and run WinDBG. When the debugger is active, Windows doesn't verify driver signatures, so we can install any drivers with or without a digital signature.
2. Set up the system to work with test certificates (a special mode for Windows driver developers).

a. Turn on Test Mode by running *cmd* with admin rights and entering this command: *bcdedit.exe -set TESTSIGNING ON*

b. Create a test certificate and sign your driver with it. Visual Studio has a special add-in for driver projects:



CryptoDevice Property Pages

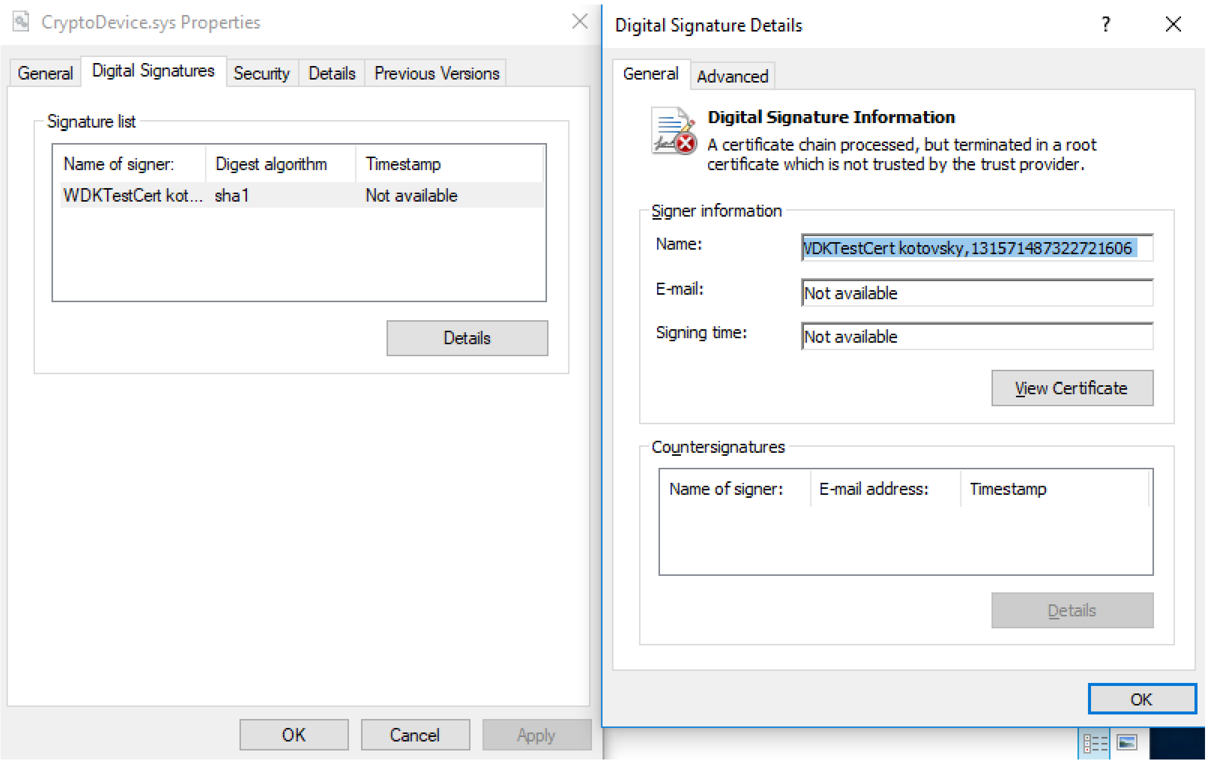
Configuration: All Configurations Platform: Active(x64)

Sign Mode: Test Sign
Test Certificate:
- <Select From Store...>
- <Create Test Certificate...>
- <Select From File...>
- <Edit...>

Cross-Signing Certificate
Production Certificate
TimeStampServer
Disable Warnings
Enable Diagnostic Verbosity: No
Minimal Build For Production Signing: No
File Digest Algorithm

Test Certificate
The test certificate used for test signing.

Visual Studio will automatically sign your driver file with a test signature during the project build. You can make sure that the driver is signed in the *.sys* file properties:



CryptoDevice.sys Properties

Signature list

Name of signer:	Digest algorithm	Timestamp
WDKTestCert kot...	sha1	Not available

Digital Signature Details

Digital Signature Information
A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider.

Signer information

Name: WDKTestCert kotovsky,131571487322721606

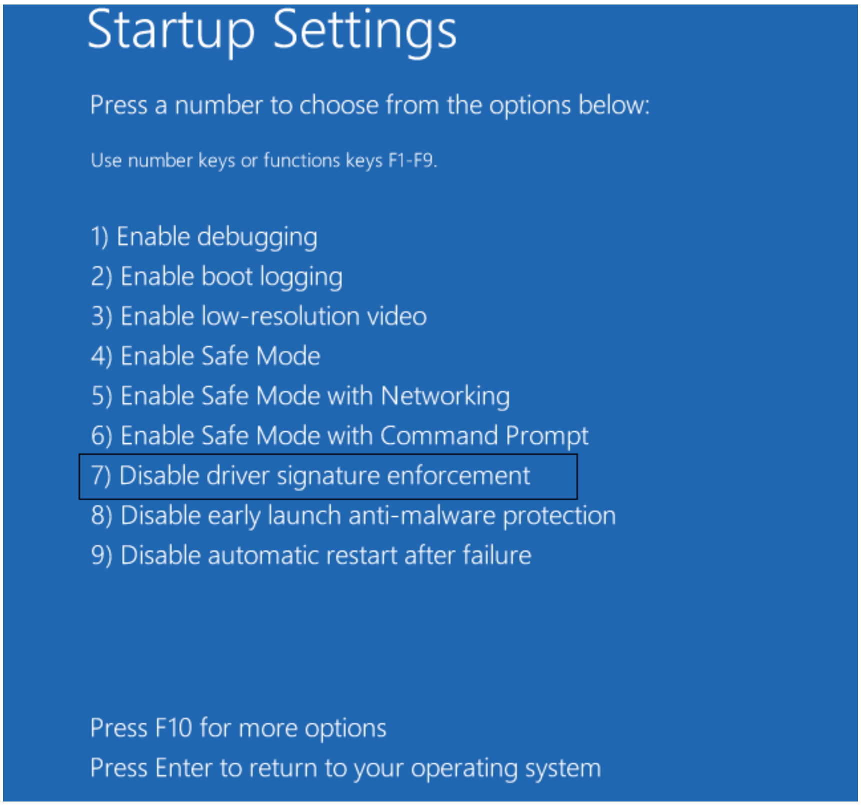
E-mail: Not available

Signing time: Not available

Countersignatures

Name of signer:	E-mail address:	Timestamp

c. Load Windows in **Disable driver signature enforcement** mode. Just press **Start** –> **Power** –> **Reboot** while holding down **Shift** and open the Windows boot menu. After that, choose the following: **Troubleshoot** –> **Advanced options** –> **Startup settings** –> **Restart**. When Windows loads the startup settings menu, press **F7**:



Startup Settings

Press a number to choose from the options below:

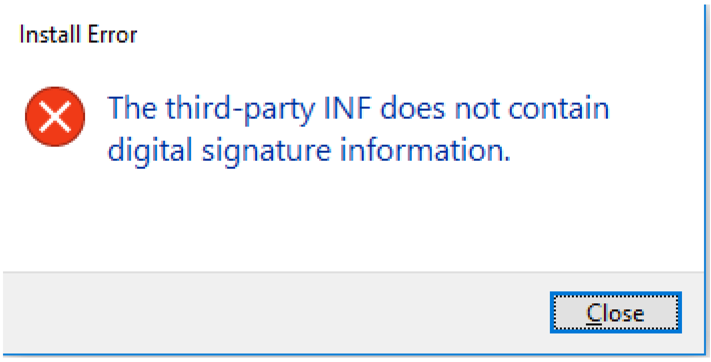
Use number keys or functions keys F1-F9.

1) Enable debugging
2) Enable boot logging
3) Enable low-resolution video
4) Enable Safe Mode
5) Enable Safe Mode with Networking
6) Enable Safe Mode with Command Prompt
7) Disable driver signature enforcement
8) Disable early launch anti-malware protection
9) Disable automatic restart after failure

Press F10 for more options
Press Enter to return to your operating system

Note: You need to choose **Disable driver signature enforcement** each time, as this option is active only for one system boot.

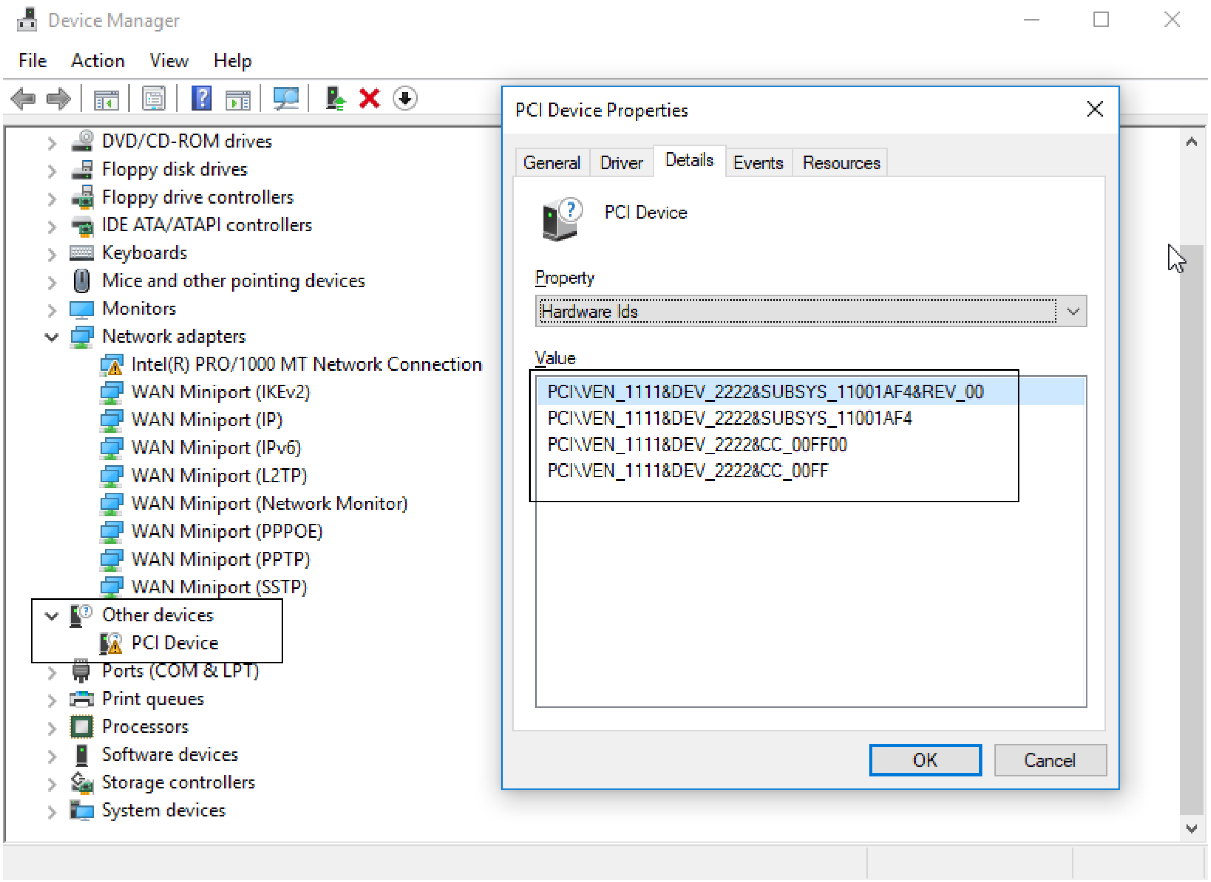
After you prepared the system using one of the suggested ways, you can start installing the test driver. If Windows is unable to verify the driver's digital signature information, the following error will be displayed:



Install Error

The third-party INF does not contain digital signature information.

When loading QEMU with a virtual device for the first time, Windows won't be able to find the driver for this device. The Device Manager will display an unknown device with our VID PID:

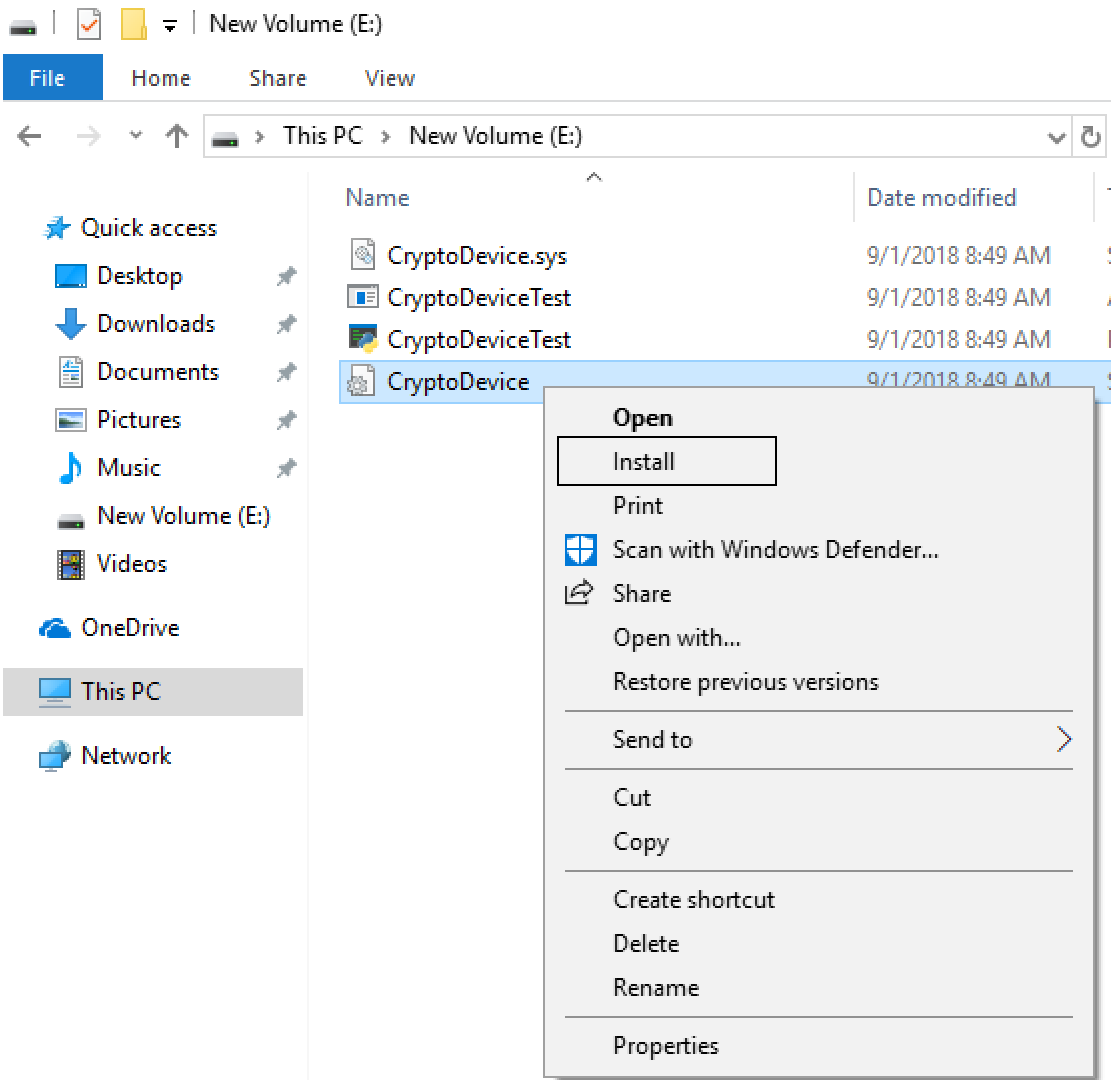


Property: Hardware Ids

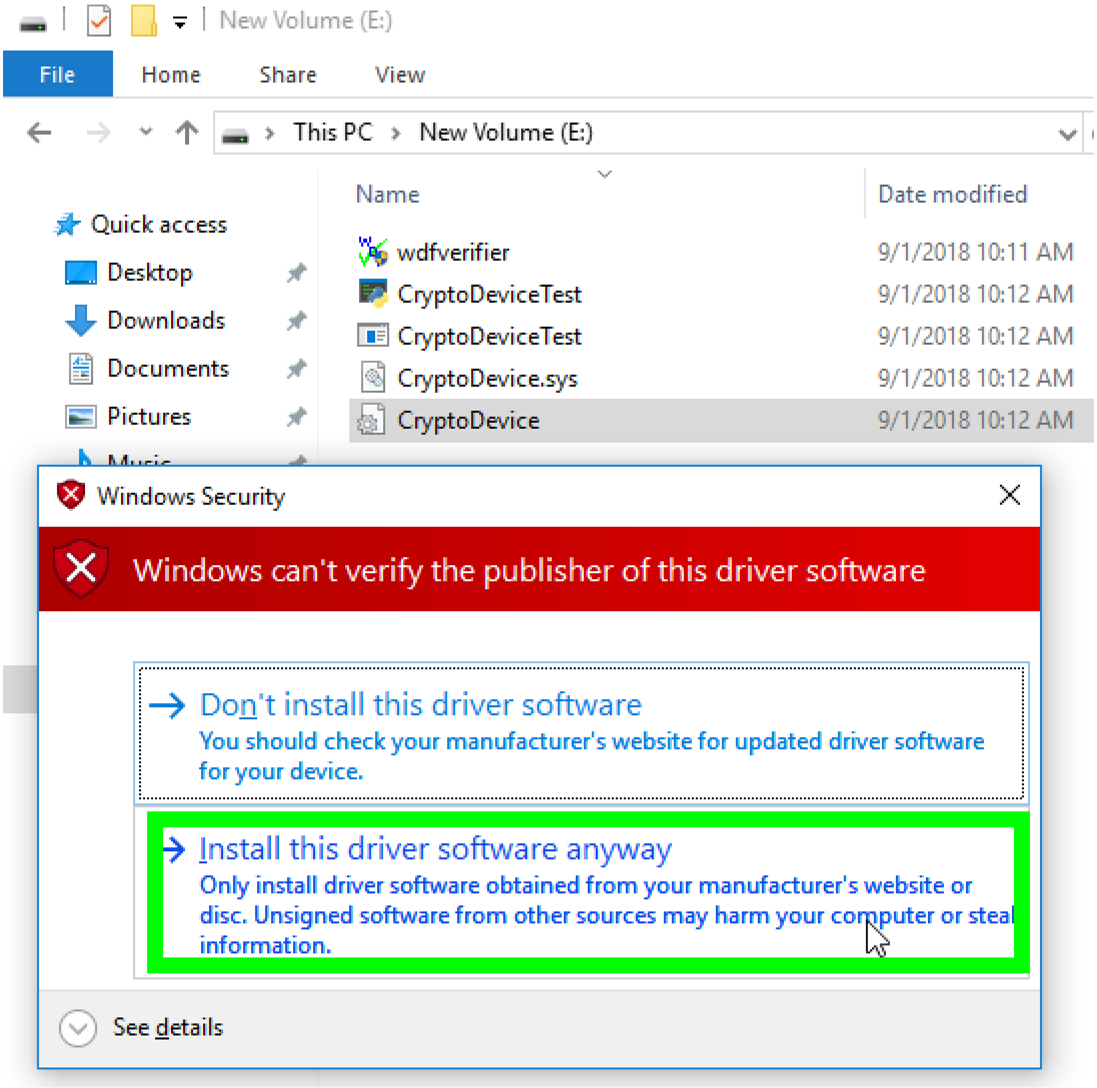
Value:

```
PCI\VEN_1111&DEV_2222&SUBSYS_11001AF4&REV_00
PCI\VEN_1111&DEV_2222&SUBSYS_11001AF4
PCI\VEN_1111&DEV_2222&CC_00FF00
PCI\VEN_1111&DEV_2222&CC_00FF
```

To install the test driver, copy the files from the *CryptoDevice/bin/x64/Release* folder to the guest operating system. All information for installation is available in the INF file, so for installing, it’s enough to right-click and choose **Install**.



Then you need to confirm the driver installation with the test certificate:



Name	Date modified
wdfverifier	9/1/2018 10:11 AM
CryptoDeviceTest	9/1/2018 10:12 AM
CryptoDeviceTest	9/1/2018 10:12 AM
CryptoDevice.sys	9/1/2018 10:12 AM
CryptoDevice	9/1/2018 10:12 AM

Windows Security

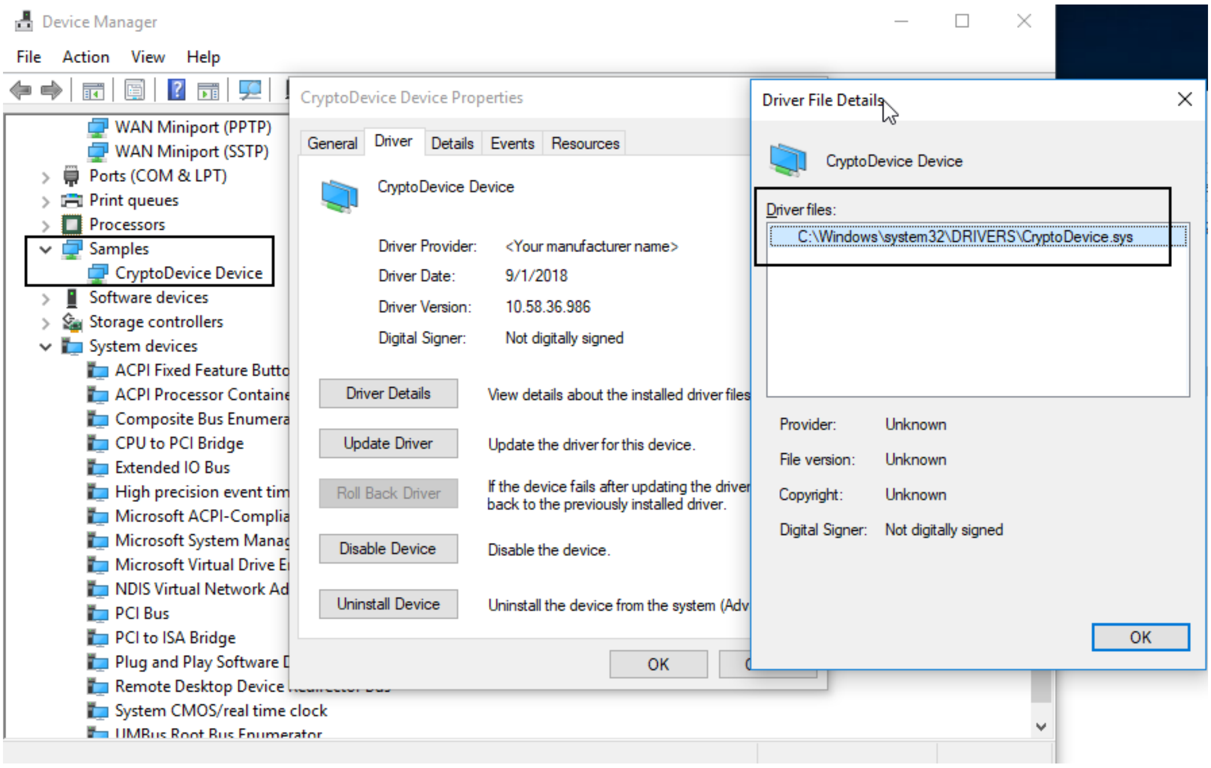
Windows can't verify the publisher of this driver software

→ Don't install this driver software
You should check your manufacturer's website for updated driver software for your device.

→ Install this driver software anyway
Only install driver software obtained from your manufacturer's website or disc. Unsigned software from other sources may harm your computer or steal information.

See details

After that, Windows will load the driver for the test device using Plug and Play, and the Device Manager will display a new device with the installed driver:



CryptoDevice Device

Driver Provider: <Your manufacturer name>
Driver Date: 9/1/2018
Driver Version: 10.58.36.986
Digital Signer: Not digitally signed

Driver Details — View details about the installed driver files

Update Driver — Update the driver for this device.

Roll Back Driver — If the device fails after updating the driver back to the previously installed driver.

Disable Device — Disable the device.

Uninstall Device — Uninstall the device from the system (Adv

Driver File Details

CryptoDevice Device

Driver files:
C:\Windows\system32\DRIVERS\CryptoDevice.sys

Provider: Unknown
File version: Unknown
Copyright: Unknown
Digital Signer: Not digitally signed

The installation is complete and the driver is ready to use. Plug and Play will load and unload the driver automatically when the device is discovered or removed.

Driver communication

The *CryptoDeviceCtrl* class is implemented for communication with the driver, and its interface fully repeats the device capabilities:

```
class CryptoDeviceCtrl
    {
    public:
        static constexpr size_t AesBlockSize = 16;
        static constexpr size_t Sha256Size = 32;
        using Sha256Buffer = std::array<uint8_t, Sha256Size>;

    public:
        explicit CryptoDeviceCtrl(const std::wstring& interfaceName);

        void ResetDevice() const;
        DeviceStatus GetDeviceStatus() const;

        void AesCbcEncrypt(const void * bufferIn
                     , size_t bufferInSize
                     , void * bufferOut
                     , size_t bufferOutSize) const;
        void AesCbcDecrypt(const void * bufferIn
                     , size_t bufferInSize
                     , void * bufferOut
                     , size_t bufferOutSize) const;

        std::vector<uint8_t> Sha256(const void * buffer, size_t bufferSize) const;
        void Sha256(const void * buffer, size_t bufferSize, Sha256Buffer& hash)
const;

        static std::vector<std::wstring> GetDevicesIds();
    };
```

To create an example of the *CryptoDeviceCtrl* class, the line with the device identifier should be sent. To obtain all available device identifiers in the system, you need to call the static *CryptoDeviceCtrl::GetDevicesIds()* function. This function returns a vector with device names. If *CryptoDevice.sys* doesn't identify any available CryptoDevice, the function will return an empty vector. If several devices are identified, the function will return all of them.

The WinAPI DeviceIoControl function is used to send requests to the driver.

Implementing driver unit tests

All unit tests are implemented in the */src/CryptoDeviceTest* project and can be executed only if the device is available in the system. All unit tests can be divided into two types:

1. Tests that verify the interface for working with the driver through the *CryptoDeviceCtrl* class. These tests are implemented in such files as *CryptoDevice_Sha256Test.cpp* and *CryptoDevice_AesCbcTest.cpp*.

   These tests:

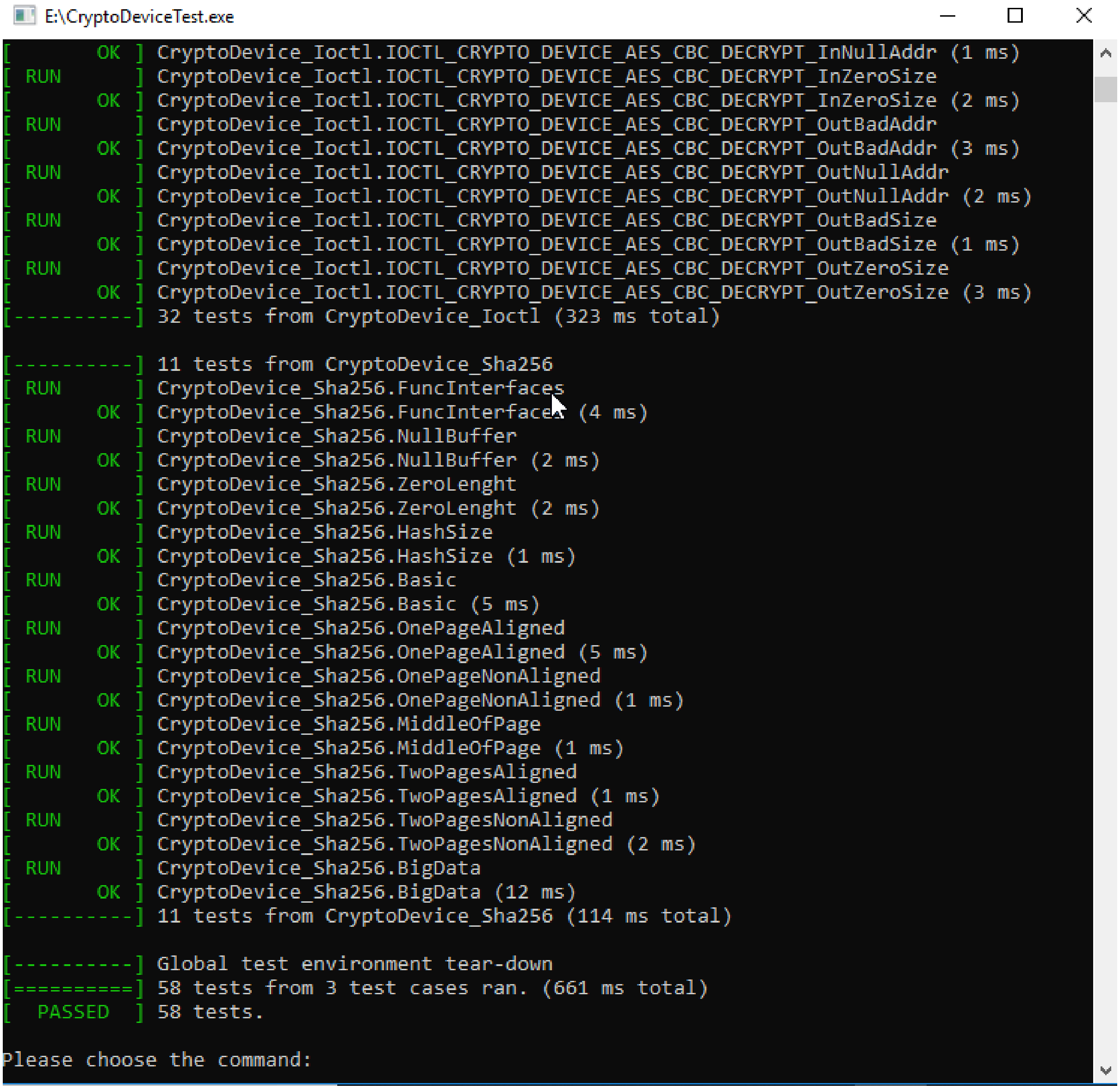
a. verify the main functionality of the device and its driver;
b. test possible cases with incorrect function parameters;
c. test possible options of structuring transferred buffers regarding virtual memory pages to verify the correctness of work with DMA.

   These tests are written to verify the performance of the driver and the device's main functionality. If they work well, then the main functionality for most typical use cases of the driver application also work.

2. Tests that verify driver IOCTL processing. These tests are implemented for each driver IOCTL and are available in the *CryptoDevice_IoctlTest.cpp* file. The main goal of these

tests is to check all possible IOCTL options for the input and output buffers. The driver should process any requests from the user mode (there should be no vulnerabilities, BSOD, etc). It's not possible to implement such tests using *CryptoDeviceCtrl*, as it hides the work with the data structures for IOCTL.

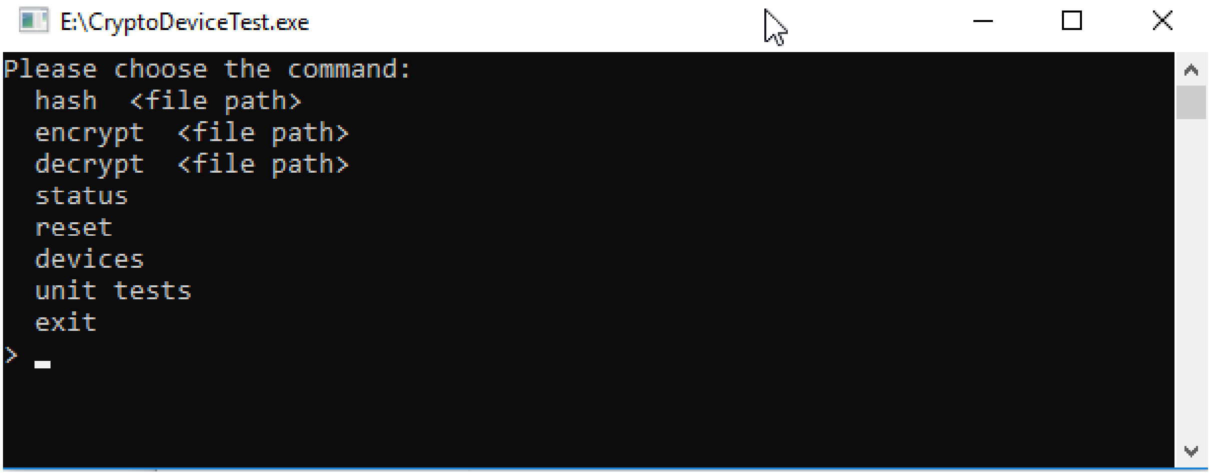
To run tests, a device and its driver should be installed on the system. The successful start of tests looks like this:



E:\CryptoDeviceTest.exe

```
[       OK ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_InNullAddr (1 ms)
[ RUN      ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_InZeroSize
[       OK ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_InZeroSize (2 ms)
[ RUN      ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_OutBadAddr
[       OK ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_OutBadAddr (3 ms)
[ RUN      ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_OutNullAddr
[       OK ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_OutNullAddr (2 ms)
[ RUN      ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_OutBadSize
[       OK ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_OutBadSize (1 ms)
[ RUN      ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_OutZeroSize
[       OK ] CryptoDevice_Ioctl.IOCTL_CRYPTO_DEVICE_AES_CBC_DECRYPT_OutZeroSize (3 ms)
[----------] 32 tests from CryptoDevice_Ioctl (323 ms total)

[----------] 11 tests from CryptoDevice_Sha256
[ RUN      ] CryptoDevice_Sha256.FuncInterfaces
[       OK ] CryptoDevice_Sha256.FuncInterfaces (4 ms)
[ RUN      ] CryptoDevice_Sha256.NullBuffer
[       OK ] CryptoDevice_Sha256.NullBuffer (2 ms)
[ RUN      ] CryptoDevice_Sha256.ZeroLenght
[       OK ] CryptoDevice_Sha256.ZeroLenght (2 ms)
[ RUN      ] CryptoDevice_Sha256.HashSize
[       OK ] CryptoDevice_Sha256.HashSize (1 ms)
[ RUN      ] CryptoDevice_Sha256.Basic
[       OK ] CryptoDevice_Sha256.Basic (5 ms)
[ RUN      ] CryptoDevice_Sha256.OnePageAligned
[       OK ] CryptoDevice_Sha256.OnePageAligned (5 ms)
[ RUN      ] CryptoDevice_Sha256.OnePageNonAligned
[       OK ] CryptoDevice_Sha256.OnePageNonAligned (1 ms)
[ RUN      ] CryptoDevice_Sha256.MiddleOfPage
[       OK ] CryptoDevice_Sha256.MiddleOfPage (1 ms)
[ RUN      ] CryptoDevice_Sha256.TwoPagesAligned
[       OK ] CryptoDevice_Sha256.TwoPagesAligned (1 ms)
[ RUN      ] CryptoDevice_Sha256.TwoPagesNonAligned
[       OK ] CryptoDevice_Sha256.TwoPagesNonAligned (2 ms)
[ RUN      ] CryptoDevice_Sha256.BigData
[       OK ] CryptoDevice_Sha256.BigData (12 ms)
[----------] 11 tests from CryptoDevice_Sha256 (114 ms total)

[----------] Global test environment tear-down
[==========] 58 tests from 3 test cases ran. (661 ms total)
[  PASSED  ] 58 tests.

Please choose the command:
```

In addition to unit tests, we also implemented an interactive interface to work with the driver. It looks like this:



```
Please choose the command:
  hash  <file path>
  encrypt  <file path>
  decrypt  <file path>
  status
  reset
  devices
  unit tests
  exit
> _
```

Using this console interface, you can check the performance of the device and its driver on any available data. Namely, you can test the following:

1. Encrypt a file with the AES algorithm
2. Decrypt a file with the AES algorithm
3. Calculate a file with SHA256
4. Receive data on the device status
5. Get a list of available devices
6. Reset the device

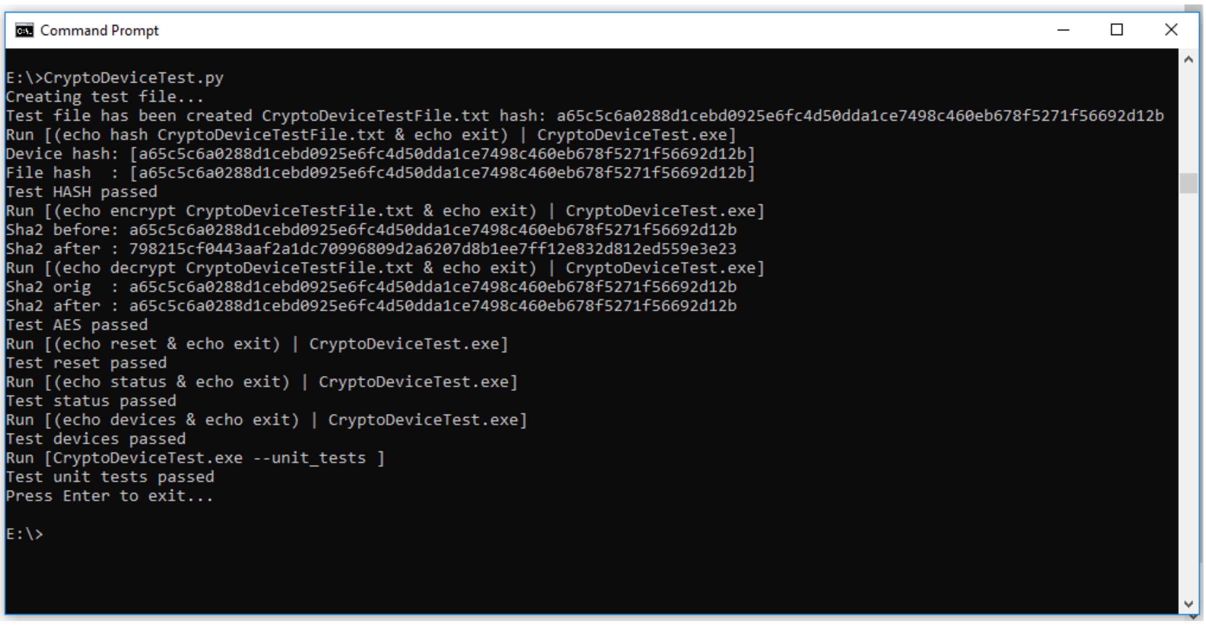
The maximum file size is limited by the available resources of the operating system (RAM) but can't be more than 4GB. Such operations as Encrypt, Decrypt, and Generate SHA256 are executed asynchronously and can be interrupted on the device's side at any time.

Implementing driver autotest

Autotest is a Python script that verifies if all interactive commands of the abovementioned console utility are operable. The test is implemented in the *CryptoDeviceTest.py* file and does the following:

1. Generates a test file of a large size.
2. Performs all interactive console commands in a row.
3. Checks the results of performing all interactive commands (including starting unit tests and run-time check).

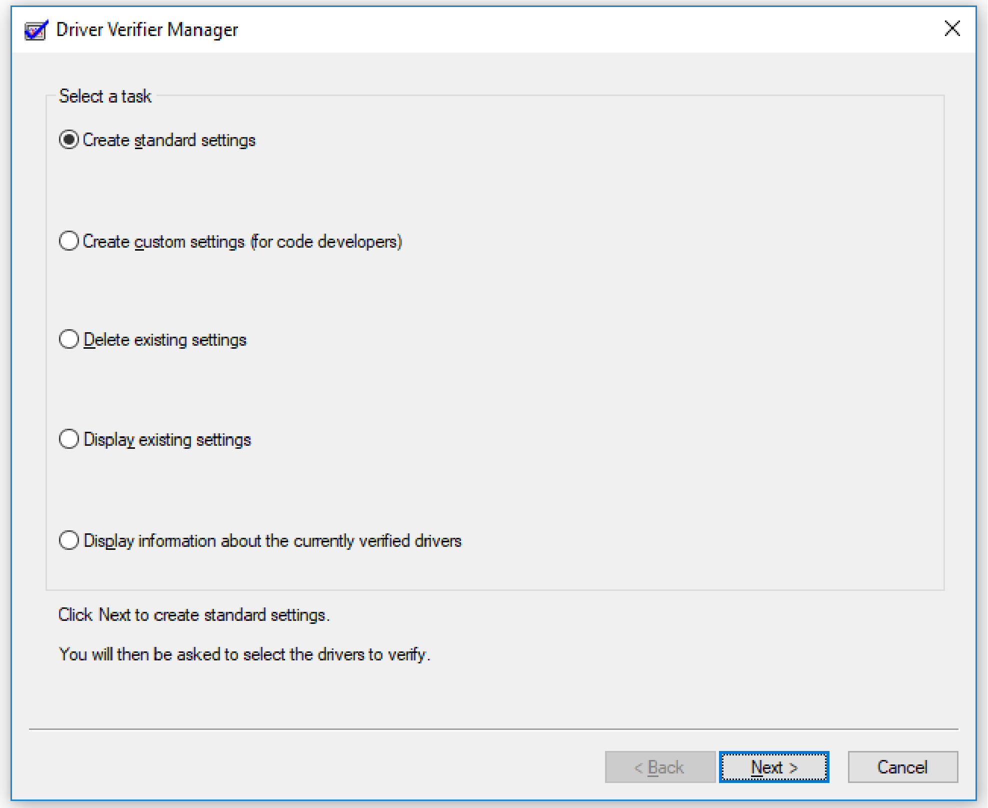
With this test, you can check the performance of the main capabilities of the console, driver, and device in one click. Additionally, this test checks are used as payload for the Driver Verifier and the WDF Verifiers. The start of the test looks like this:



```
E:\>CryptoDeviceTest.py
Creating test file...
Test file has been created CryptoDeviceTestFile.txt hash: a65c5c6a0288d1cebd0925e6fc4d50dda1ce7498c460eb678f5271f56692d12b
Run [(echo hash CryptoDeviceTestFile.txt & echo exit) | CryptoDeviceTest.exe]
Device hash: [a65c5c6a0288d1cebd0925e6fc4d50dda1ce7498c460eb678f5271f56692d12b]
File hash  : [a65c5c6a0288d1cebd0925e6fc4d50dda1ce7498c460eb678f5271f56692d12b]
Test HASH passed
Run [(echo encrypt CryptoDeviceTestFile.txt & echo exit) | CryptoDeviceTest.exe]
Sha2 before: a65c5c6a0288d1cebd0925e6fc4d50dda1ce7498c460eb678f5271f56692d12b
Sha2 after : 798215cf0443aaf2a1dc70996809d2a6207d8b1ee7ff12e832d812ed559e3e23
Run [(echo decrypt CryptoDeviceTestFile.txt & echo exit) | CryptoDeviceTest.exe]
Sha2 orig  : a65c5c6a0288d1cebd0925e6fc4d50dda1ce7498c460eb678f5271f56692d12b
Sha2 after : a65c5c6a0288d1cebd0925e6fc4d50dda1ce7498c460eb678f5271f56692d12b
Test AES passed
Run [(echo reset & echo exit) | CryptoDeviceTest.exe]
Test reset passed
Run [(echo status & echo exit) | CryptoDeviceTest.exe]
Test status passed
Run [(echo devices & echo exit) | CryptoDeviceTest.exe]
Test devices passed
Run [CryptoDeviceTest.exe --unit_tests ]
Test unit tests passed
Press Enter to exit...

E:\>
```

Driver verification with Driver Verifier and WDF Verifier

The Driver Verifier tool is a built-in Windows utility for driver verification. To run this utility, just run the *verifier.exe* command:



Select a task

● Create standard settings

○ Create custom settings (for code developers)

○ Delete existing settings

○ Display existing settings

○ Display information about the currently verified drivers

Click Next to create standard settings.

You will then be asked to select the drivers to verify.

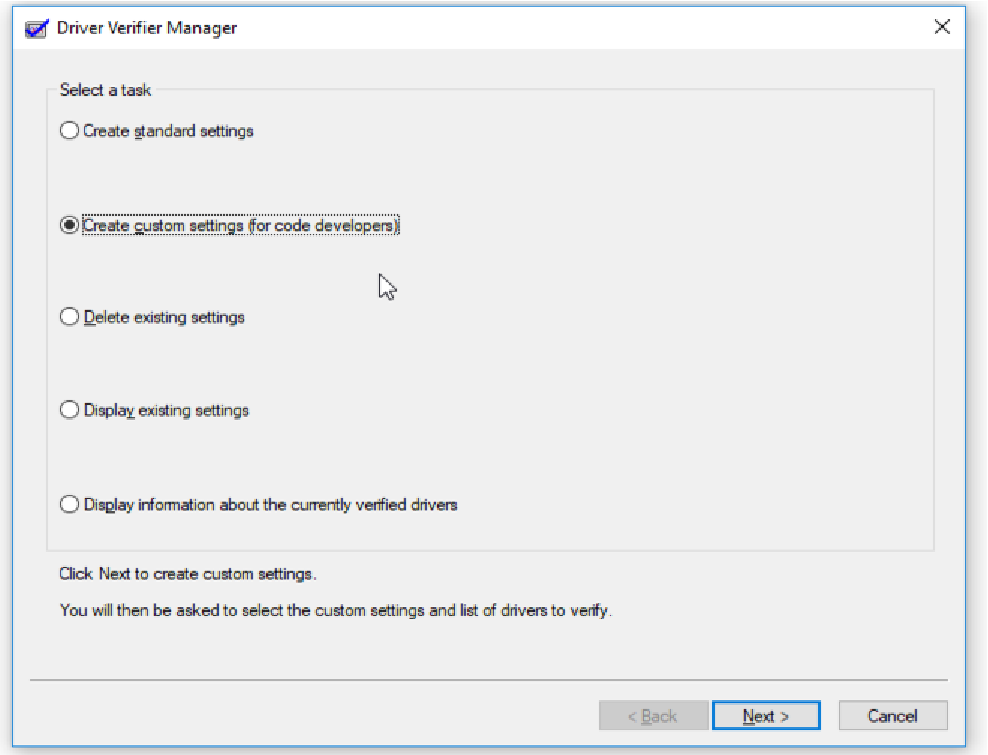
Verifying a driver with Driver Verifier happens in two stages:

1. Driver verification with all flags except:
    a. Randomized low resources simulation
    b. Systematic low resources simulation
2. Driver verification with all flags

At the first stage, we expect no BSOD and expect everything to work without failing (just like without Driver Verifier). If you get a BSOD when running Driver Verifier, then in 99.9% of cases it's a driver defect. By analyzing the **.dmp* file, you can usually understand the cause of failure and the error code. All starts of auto/unit tests should execute without failure.

At the second stage, we expect no BSOD. However, the driver may and will return failures and request processing interrupts. It's okay if there are failures with some driver functionality. In this mode, the operating system will simulate a lack of resources, memory allocation functions will occasionally return NULL, the kernel object creation functions will return errors or bad statuses, and so on. The main goal of this check is to see that the driver can work without failures. It shouldn't cause the system to freeze, give a BSOD, cause resource and memory leaks, and so on.

For driver verification, you need to set up Driver Verifier and restart the operating system:



○ Create standard settings

● Create custom settings (for code developers)

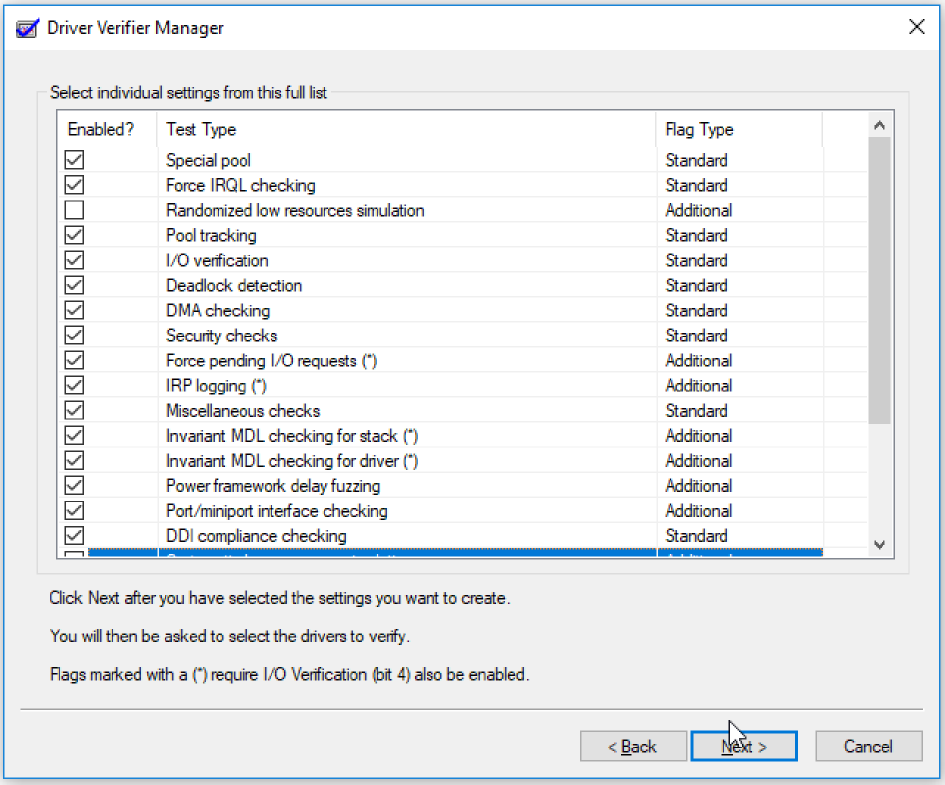
○ Delete existing settings

○ Display existing settings

○ Display information about the currently verified drivers

Click Next to create custom settings.

You will then be asked to select the custom settings and list of drivers to verify.



Driver Verifier Manager

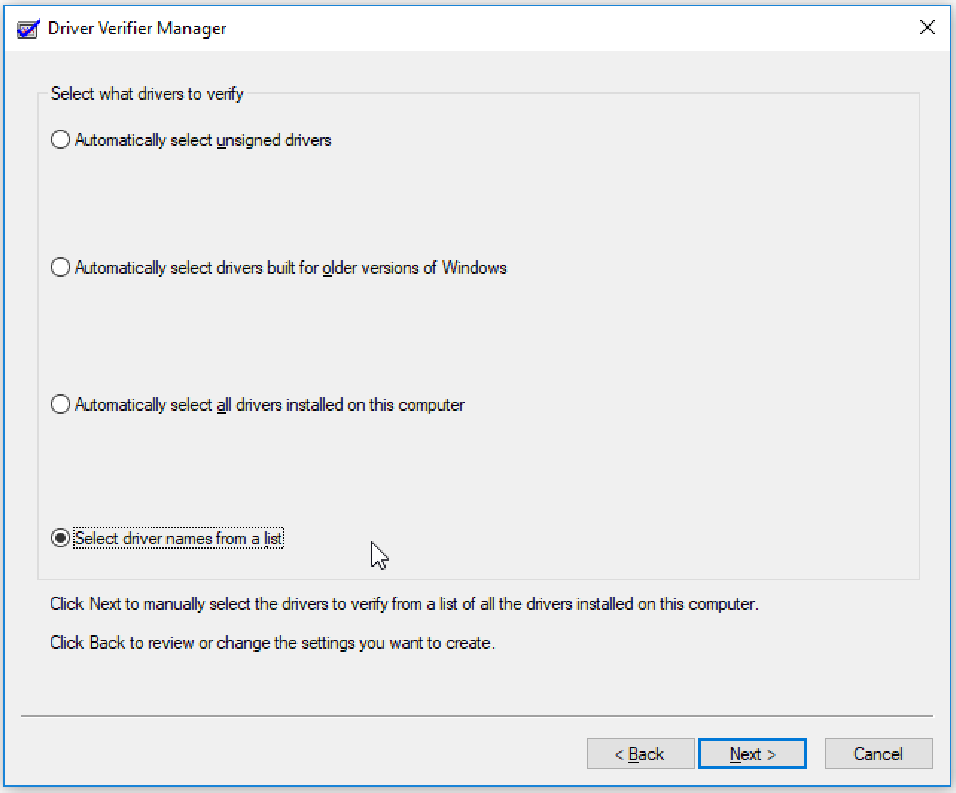
Select individual settings from this full list

Enabled?	Test Type	Flag Type
☑	Special pool	Standard
☑	Force IRQL checking	Standard
☐	Randomized low resources simulation	Additional
☑	Pool tracking	Standard
☑	I/O verification	Standard
☑	Deadlock detection	Standard
☑	DMA checking	Standard
☑	Security checks	Standard
☑	Force pending I/O requests (*)	Additional
☑	IRP logging (*)	Additional
☑	Miscellaneous checks	Standard
☑	Invariant MDL checking for stack (*)	Additional
☑	Invariant MDL checking for driver (*)	Additional
☑	Power framework delay fuzzing	Additional
☑	Port/miniport interface checking	Additional
☑	DDI compliance checking	Standard

Click Next after you have selected the settings you want to create.

You will then be asked to select the drivers to verify.

Flags marked with a (*) require I/O Verification (bit 4) also be enabled.



Driver Verifier Manager

Select what drivers to verify

○ Automatically select unsigned drivers

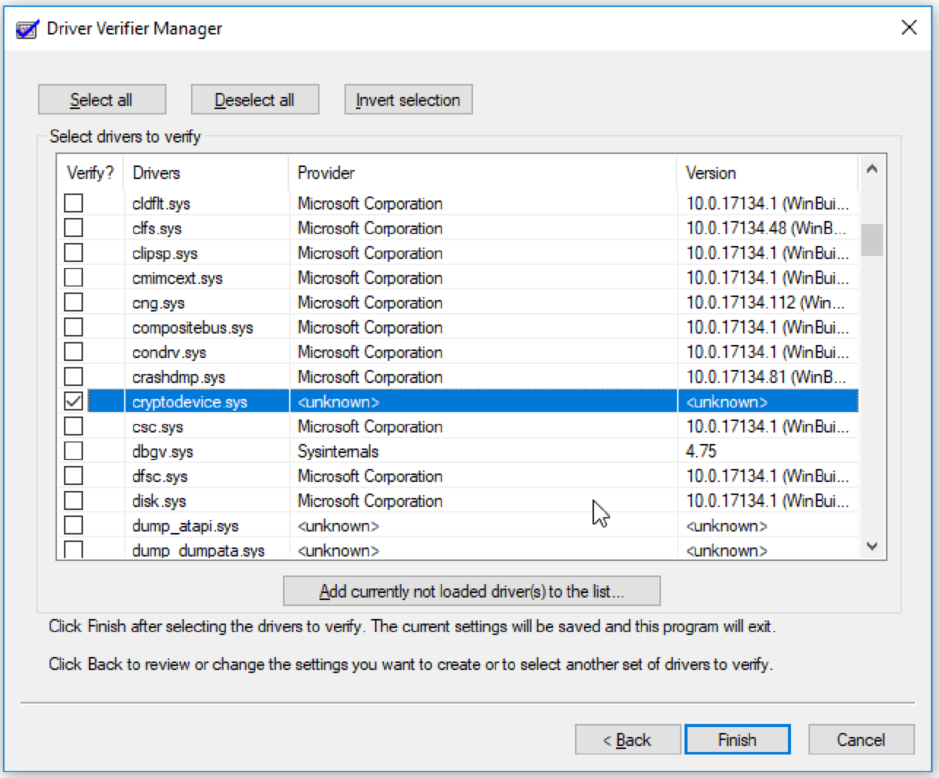
○ Automatically select drivers built for older versions of Windows

○ Automatically select all drivers installed on this computer

● Select driver names from a list

Click Next to manually select the drivers to verify from a list of all the drivers installed on this computer.

Click Back to review or change the settings you want to create.



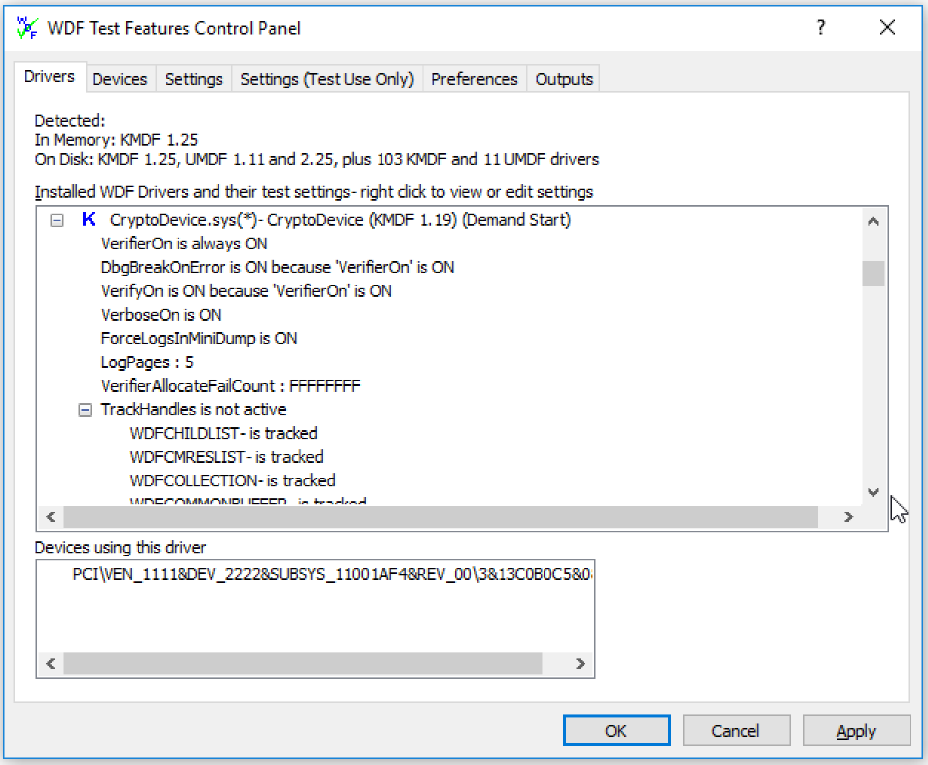
Select drivers to verify

Verify?	Drivers	Provider	Version
□	cldflt.sys	Microsoft Corporation	10.0.17134.1 (WinBui...
□	clfs.sys	Microsoft Corporation	10.0.17134.48 (WinB...
□	clipsp.sys	Microsoft Corporation	10.0.17134.1 (WinBui...
□	cmimcext.sys	Microsoft Corporation	10.0.17134.1 (WinBui...
□	cng.sys	Microsoft Corporation	10.0.17134.112 (Win...
□	compositebus.sys	Microsoft Corporation	10.0.17134.1 (WinBui...
□	condrv.sys	Microsoft Corporation	10.0.17134.1 (WinBui...
□	crashdmp.sys	Microsoft Corporation	10.0.17134.81 (WinB...
☑	cryptodevice.sys	<unknown>	<unknown>
□	csc.sys	Microsoft Corporation	10.0.17134.1 (WinBui...
□	dbgv.sys	Sysinternals	4.75
□	dfsc.sys	Microsoft Corporation	10.0.17134.1 (WinBui...
□	disk.sys	Microsoft Corporation	10.0.17134.1 (WinBui...
□	dump_atapi.sys	<unknown>	<unknown>
□	dump_dumpata.sys	<unknown>	<unknown>

Click Finish after selecting the drivers to verify. The current settings will be saved and this program will exit.

Click Back to review or change the settings you want to create or to select another set of drivers to verify.

WDF Verifier is part of the Windows WDK. It's available in *C:\Program Files (x86)\Windows Kits\10\Tools\x64\wdfverifier.exe* if you're using WDK version 10.
This tool allows you to find defects when using the WDF API. You can set up WDF Verifier in the following way:



Detected:
In Memory: KMDF 1.25
On Disk: KMDF 1.25, UMDF 1.11 and 2.25, plus 103 KMDF and 11 UMDF drivers

Installed WDF Drivers and their test settings- right click to view or edit settings

```
⊟ K CryptoDevice.sys(*)- CryptoDevice (KMDF 1.19) (Demand Start)
      VerifierOn is always ON
      DbgBreakOnError is ON because 'VerifierOn' is ON
      VerifyOn is ON because 'VerifierOn' is ON
      VerboseOn is ON
      ForceLogsInMiniDump is ON
      LogPages : 5
      VerifierAllocateFailCount : FFFFFFFF
   ⊟ TrackHandles is not active
         WDFCHILDLIST- is tracked
         WDFCMRESLIST- is tracked
         WDFCOLLECTION- is tracked
         WDFCOMMONBUFFER - is tracked
```

Devices using this driver

PCI\VEN_1111&DEV_2222&SUBSYS_11001AF4&REV_00\3&13C0B0C5&0

Here’s the common process for CryptoDevice driver verification:

1. Install the release driver version.
2. Run WDF Verifier.
3. Run Driver Verifier in mode 1.
4. Reboot the operating system.
5. Run the Python script with autotests several times in a row (all runs should end successfully).
6. Using the Device Manager, turn the CryptoDevice device on and off several times and run autotests after each on-off cycle.
7. Reboot the operating system to verify the driver unload and detect memory leaks.
8. Add all other flags to the Driver Verifier (mode 2).
9. Run the Python script with autotests several times in a row (not all runs may end successfully).
10. Using the Device Manager, turn the CryptoDevice device on and off several times and run autotests after each on-off cycle.
11. Reboot the operating system and run the autotests again.

In addition, it’s necessary to check all three possible scenarios of working with interrupts for device drivers that support work with lined-base interrupts and MSIs. To do this, you need to modify the INF file in the following way:

1. **To work with all necessary MSI messages:**

```
[CryptoDevice_Device_MSI]
HKR, Interrupt Management,, 0x00000010
HKR, Interrupt Management\MessageSignaledInterruptProperties,, 0x00000010
HKR, Interrupt Management\MessageSignaledInterruptProperties,MSISupported, 0x00010001, 1
HKR, Interrupt Management\MessageSignaledInterruptProperties,MessageNumberLimit, 0x00010001,4
```

2. **To work with one MSI message:**

```
[CryptoDevice_Device_MSI]
HKR, Interrupt Management,, 0x00000010
HKR, Interrupt Management\MessageSignaledInterruptProperties,, 0x00000010
HKR, Interrupt Management\MessageSignaledInterruptProperties,MSISupported, 0x00010001, 1
HKR, Interrupt Management\MessageSignaledInterruptProperties,MessageNumberLimit, 0x00010001,1
```

3. **To work with line-based interrupts:**

```
[CryptoDevice_Device_MSI]
HKR, Interrupt Management,, 0x00000010
HKR, Interrupt Management\MessageSignaledInterruptProperties,, 0x00000010
HKR, Interrupt Management\MessageSignaledInterruptProperties,MSISupported, 0x00010001, 0
HKR, Interrupt Management\MessageSignaledInterruptProperties,MessageNumberLimit,
0x00010001,1
```

You can install and test each configuration separately. The test QEMU virtual device will display information on the types of interrupts used in the QEMU console.

If no issues arise while performing all the steps above (no system freezes, BSOD, etc.), the driver can be considered stable and ready for testing.

It’s better to perform driver development and testing, including testing of driver version releases, with the kernel debugger active so you can analyze issues right when they appear.

References

1. http://online.osr.com/article.cfm?article=539
2. https://www.osr.com/blog/2014/04/04/dma-cache-coherent-arm/
3. https://www.xilinx.com/Attachment/Xilinx_Answer_58495_PCIe_Interrupt_Debugging_Guide.pdf
4. http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/dma.docx
5. http://download.microsoft.com/download/5/7/7/577a5684-8a83-43ae-9272-ff260a9c20e2/MSI.doc
6. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/setting-up-a-network-debugging-connection
7. https://docs.microsoft.com/en-us/visualstudio/code-quality/using-sal-annotations-to-reduce-c-cpp-code-defects
8. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier
9. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/driver-verifier
10. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/wdf-verifier-control-application
11. https://docs.microsoft.com/en-us/windows-hardware/drivers/driversecurity/driver-security-checklist
12. https://github.com/Microsoft/GSL
13. https://docs.microsoft.com/en-us/windows-hardware/drivers/install/kernel-mode-code-signing-policy--windows-vista-and-later-
14. https://docs.microsoft.com/en-us/windows-hardware/drivers/install/the-testsigning-boot-configuration-option
15. https://github.com/qemu/qemu/tree/stable-2.11

The e-book on the development of Windows driver using a QEMU virtual device is based on the experience of Apriorit team who uses QEMU for driver and kernel development along with other virtualization technologies.

This e-book is intended for information purposes only. Any trademarks and brands are property of their respective owners and used for identification purposes only.

About Apriorit Inc.

Apriorit Inc. is a software development service provider headquartered in the Dover, DE, US, with several development centers in Eastern Europe. With over 350 professionals, we bring high-quality services on software consulting, research, and development to software vendors and IT companies worldwide.

Apriorit's main specialties are cybersecurity and data management projects, where system programming, driver and kernel level development, research and reversing matter. The company has an independent web platform development department focusing on building cloud platforms for business.

Apriorit team will be glad to contribute to your software engineering projects.

Clutch
FIRMS THAT DELIVER

Find us on Clutch.co

For more information please contact:

Apriorit Inc.

Headquarters:
8 The Green
Suite #7106, Dover, DE, 19901, US
Phone:
202-780-9339

E-mail: info@apriorit.com
www.apriorit.com