Contents

Chapter 1
Direct memory access

CHAPTER GOALS

Have you ever wondered how information travels between input-output devices and main memory in a computer system? In this chapter, we provide a detailed explanation of the Direct Memory access (DMA) I/O technique used in modern computer systems, including those using the Intel and ARM family of microprocessors. This chapter also aims to demystify the DMA controller internals and its programming with various peripherals. Upon completion of this chapter, you will be able to:

- Distinguish between programmed IO, interrupt-driven IO, and DMA transfers.
- Explain the operation of the signals used in direct memory access controllers.
- Explain the function of the Intel 8237 DMA controller when used for DMA transfers.
- Explain the function of the DMA controller used in STM Cortex-M based systems.
- Explain the function of bus-mastering (also referred to as first-party DMA).

1.1 Introduction

1.2 Programmed Input/Output

The programmed I/O was the most straightforward type of I/O technique for the exchanges of data between I/O devices and memory. This data transfer method requires the least amount of hardware. With programmed I/O, data transfers between

I/O devices and memory are accomplished by the central processing unit (CPU). In the case of programmed I/O, *the I/O device does not have direct access to the main memory*. The I/O devices have memory-mapped registers. This means that the CPU accesses the I/O device's registers using LOAD/STORE instructions.

A transfer from an I/O device to the main memory (or vice versa) requires the execution of several instructions by the CPU. This includes a LOAD instruction to transfer the data from the I/O device's data register(s) to the CPU and STORE instruction to transfer the data from CPU to the main memory. Besides, the CPU must continuously sense the I/O device's status. When the CPU issues a command to the I/O device, it must wait until the I/O operation is complete or new data is available. For example, before reading the data from the I/O device with the LOAD instruction, the CPU must first read the status register (also with a LOAD instruction) of the I/O device to check if the I/O device has new data. Similarly, before writing the data to the I/O device, the CPU must first read the status register (also with a LOAD instruction) of the I/O device to check if the I/O device is prepared to accept new data. As the CPU is faster than the I/O module, the problem with programmed I/O is that the CPU has to wait a long time for the I/O device to be ready for either reception or transmission of data. The CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This process of waiting and checking the status of the I/O device is known as **polling or busy waiting**. As a result, polling severely degrades the level of the performance of the entire system. This situation can be avoided by using an interrupt-driven I/O, which we discuss in the next section.

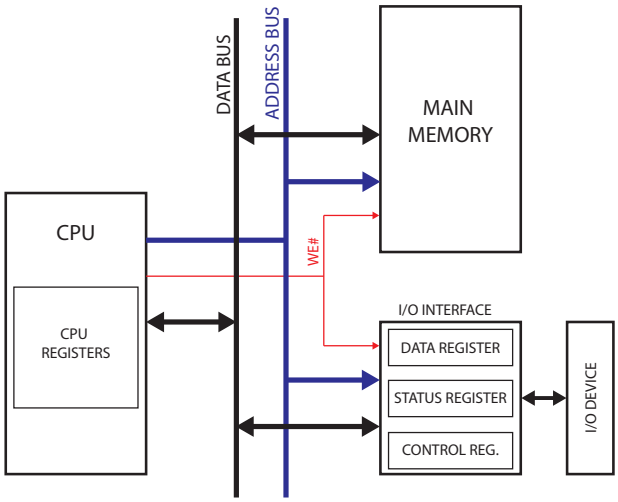

Fig. 1.1: A simplified block diagram of a computer system with programmed I/O.

Let's take a look at how programmed I/O would work if you were copying information from an I/O device to the main memory. Figure 1.1 illustrates a simplified block diagram of a computer system using programmed I/O. The I/O device shares the data, address, and control bus with the main memory. Although modern computer systems have more buses organized hierarchically, we can still simplify this discussion by assuming that there is only one bus in the system. The I/O devices in modern computer systems are memory-mapped, meaning that the CPU accesses these devices through a well-defined I/O interface. The I/O interface of an I/O device contains a set of registers, each of them having its unique address from the global address space. The CPU reads and writes to these I/O registers in the same way as it reads or writes to the main memory: using the LOAD and STORE instructions.

The I/O device in Figure 1.1 has three memory-mapped registers: a control register, a status register, and a data register. The control register is used to program the I/O device, e.g., set the data rate, parity check, etc. The status register reflects the status of the I/O device, e.g., the I/O device is ready to accept new data, or the I/O device has new data, etc. The data register is used to transfer data to/from the I/O device. In programmed I/O mode, the CPU would constantly check the status register to see if new data is available. Thus, the CPU would read the status register with the LOAD instruction and check a particular bit, which flags that the I/O device has new data. The CPU would perform the polling operation inside a program loop. In the case new data is available, the CPU would first transfer data from the data register into an internal register with the LOAD instruction. Then, the CPU would transfer data from the internal register into the memory with a STORE instruction. Listing 1.1 illustrates the programmed I/O transfer from the I/O device to the main memory:

```
1
2                   ; wait for new data
3   busy_wait:      lw r1, status_reg;
4                   beq r1, r0, busy_wait;
5
6                   ; CPU transfers data
7   transfer:       lw r2, data_reg
8                   sw rw, mem_addr
9
10
```

Listing 1.1: Programmed I/O data transfer

While not in use anymore, programmed I/O mode transfers were used in older hard drives back a few decades ago when so-called DMA transfers didn't exist. For example, programmed I/O was used by the Western Digital WD1003, the hard disk controller used by the first PCs. Programmed I/O is still used now in some low-end and embedded computer systems. Also, the Intel 80286, 80386, and 80486 microprocessors used in personal computers were well suited to programmed I0 since they can move blocks of data with a single *String Move* instruction. This data move instruction allowed programmed I/O transfers to reach speeds of about 2.5 Mbytes/s. In an embedded system where the CPU has nothing else to do, busy waiting is rea-

sonable. However, in a more sophisticated computer system where the CPU has to do other things, polling is inefficient, and a better I/O transfer method is needed.

1.3 Interrupt-driven I/O

A disadvantage of polling is that the CPU must continuously sense the I/O device's status a loop. The waiting may significantly slow down the system capability of executing other instructions and processing other data. The so-called interrupt-driven I/O could be more efficient. In interrupt-driven I/O, the I/O device, when ready for a new transfer, initiates the data transfer by interrupting the CPU. The CPU then executes the interrupt service program that transfers the data. Similarly, as in programmed I/O, the transfer from an I/O device to the main memory (or vice versa) requires the execution of a LOAD instruction to transfer the data from the I/O device's data register(s) to the CPU and STORE instruction to transfer the data from CPU to the main memory. But now, there is no need to wait in a loop and read the I/O device's status. The interrupt-driven I/O technique requires more complex hardware but makes far more efficient use of CPU time and capacities.

For the transfer from an I/O device to memory, the device interrupts the CPU when new data has arrived and is ready to be retrieved by the system processor. As most of the I/O devices have memory-mapped registers, the interrupt service program will then read the device's data register into a CPU register and store the data from the CPU register to the memory location.

For the transfer from a memory location to an I/O device, the device delivers an interrupt either when it is ready to accept new data or to acknowledge a successful previous data transfer. The interrupt service program will then read the memory location into a CPU register and store the data from the CPU register to the device's data register.

Hence, in the interrupt-driven I/O, the CPU continuously works on given tasks. When the I/O device is ready for the data transfer, such as when someone types a key on the keyboard or a serial communication interface is ready to transmit a new byte, it interrupts the CPU from its work to take care of the data transfer. The CPU can work continuously on a task without checking the input devices, allowing the devices themselves to interrupt it as necessary.

The interrupt-driven I/O is adequate for simple computer systems, but there are situations in modern computing that complicate the picture. For example, what if the CPU is executing som critical task that should not be interrupted? What if the CPU executes an interrupt service program corresponding to an interrupt with the priority higher than the priority of the current interrupt request? In such a case, the handling of the current interrupt request from an I/O device should be deferred.

1.4 Direct Memory Access

We have seen two different methods used to transfer data between I/O devices and the main memory: polling and interrupt-driven I/O. Both techniques work well with low-bandwidth devices and some low-end computer systems, and both methods use the CPU to move data. While moving data, the CPU can not perform any other operations, making both methods inappropriate for modern, high-speed computer systems.

An alternative mechanism is to offload the CPU and to have **another device transfer data directly to or from the main memory** - without involving the CPU at all. This mechanism is called **direct memory access (DMA)**. DMA is a feature that allows systems to access the main memory without any help from the processor. The special device that performs the DMA transfer is a **DMA controller**. A DMA controller offloads the CPU tremendously as it fulfills a memory transfer without intervention from the processor. When the transfer is finished, it signals the CPU with an interrupt.

The DMA controller transfers data between the main memory and an I/O device independent of the CPU. The CPU only initializes the DMA controller. A DMA transfer is fulfilled in the following steps:

1. The CPU initializes the DMA controller: it provides the source and destination addresses of the data to be transferred, the number of bytes to be transferred, and the type of transfer to perform (we will discuss these types later).
2. When data is available, **the I/O device requests the DMA transfer** from the DMA controller. The DMA controller then requests the bus from the CPU and becomes the master of the bus and starts the transfer. During the transfer, the DMA controller supplies the memory addresses and the control signals needed to complete the transfer. If the request form the I/O device requires more than one transfer, the DMA controller will automatically generate the next memory address(es) and will complete the entire DMA transfer of hundreds of thousands of bytes without involving the CPU. The modern DMA controllers usually contain FIFO buffers that help them deal with different timings and delays during a transfer.
3. When the DMA transfer is complete, the DMA controller interrupts the CPU. The CPU can then decide if more transfers are required and reinitialize the DMA controller for new DMA transfers.

I'm sure you are now wondering how the CPU accesses the main memory during a DMA transfer. Well, (usually) it does not. But wait, how does the CPU fetch instructions and data from the main memory if the DMA controller occupies the memory bus? Remember, that CPU never directly access the main memory - it always accesses the L1, L2, and L3 caches first, and only if there is a miss in the L3 cache, the memory controller transfers the cache line to/from the main memory. Thus, again we rely on the temporal and spatial data locality and assume that there is a very high probability that instructions and operands, needed by the CPU, are already in the cache(s). So, the DMA transfer usually does not prevent the CPU

from fetching instructions and data. By using caches, the CPU leaves most of the memory bandwidth free for use by a DMA controller. In the case of the cache miss, the modern systems rely on multitasking: in that case, the OS would perform a task switch another (ready) task.

Summary: Direct Memory Access

Direct memory access (DMA) is a mechanism that allows us to offload the CPU and to have a DMA controller transfer data directly between a peripheral device and the main memory.

A DMA transfer starts with a peripheral device placing a DMA request to the DMA controller. The DMA controller then requests the bus from the CPU and starts the transfer. When the DMA transfer is complete, the DMA controller interrupts the CPU.

Because of the use of cache and memory hierarchy in modern computer systems, a DMA transfer does not prevent the CPU from fetching instructions and data.

Let's take a look at how a DMA controller transfers data between the main memory and an I/O device. Figure 1.2 illustrates a simplified block diagram of a system with a DMA controller. The DMA controller is connected to the data, address, and control buses. It also has six control signals: DREQ, DACK, HOLD, HLDA, END, and WE#. Two control signals, DREQ (DMA request) and DACK (DMA acknowledge), namely, are used between the DMA controller and an I/O device to request and acknowledge a DMA transfer. Another two control signals, HOLD (Hold request) and HLDA (Hold acknowledge), are used between the DMA controller and the CPU to request and acknowledge a DMA transfer. The WE# signal selects between the memory read or memory write operations, and the END signals the CPE that the DMA transfer is finished and data is ready for further processing. The DMA controller has two registers: the **address register** and the **count register**. The address register holds the address of the memory location from/to which the data is to be transferred. The count register holds the number of data words to be transferred. Both registers are memory-mapped, and the CPU is responsible for their initialization.

Before any data transfer takes place, the CPU should initialize the DMA controller's address and count registers. The CPU writes the address of the memory location to/from which the data is to be transferred into the address register, and the number of data words to be transferred into the count register. During the transfer, the DMA controller will decrement the count register after each data word is transferred. It will also increment or decrement the address register, depending on the mode of operation, and automatically store/read the data to/from consecutive memory locations. When the count register signals that there is no more data to be transferred, the DMA controller will activate the END signal. This signal is usu-

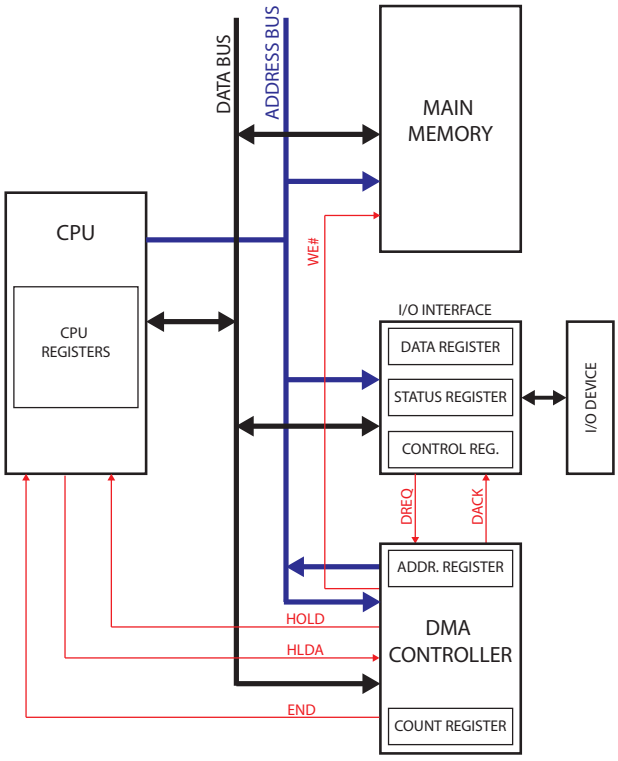

Fig. 1.2: A simplified block diagram of a computer system with a DMA controller.

ally connected to a CPU interrupt input and rises an interrupt when a transfer is completed.

Let's suppose the data transfer of one data word from the main memory to the I/O device. Firstly, the CPU writes the address of the memory location that holds the data into the address register, and the value of 1 into the count register, indicating that only one data word is to be transferred. The following steps are then required to accomplish the DMA transfer (Figure 1.3):

1. The I/O device is ready to receive data, so it asserts the DREQ signal.
2. The DMA controller requests the bus (requests the DMA transfer) from the CPU by asserting the HOLD signal.
3. The CPU relinquishes the control of the main memory. It voluntarily places all its bus signals at a high-impedance state and asserts the HLDA signal to indicate the bus is granted.
4. The DMA controller places the memory address from the address register on the address bus and puts the WE# signal into the high state to indicate the read access.
5. The main memory places the requested data onto the data bus.

6. The DMA controller asserts the DACK signal. This is to indicate that the I/O device can fetch data from the data bus. The DMA controller also decrements the count register.
7. The I/O device latches data from the data bus into its data register.
8. As the count register now indicates that there is no data left to transfer, the DMA controller de-asserts the HOLD signal to return the control over the bus to the CPU and activates the END signal to raise a CPU interrupt.
9. The CPU de-asserts the HLDA signal and eventually starts to service the interrupt request.

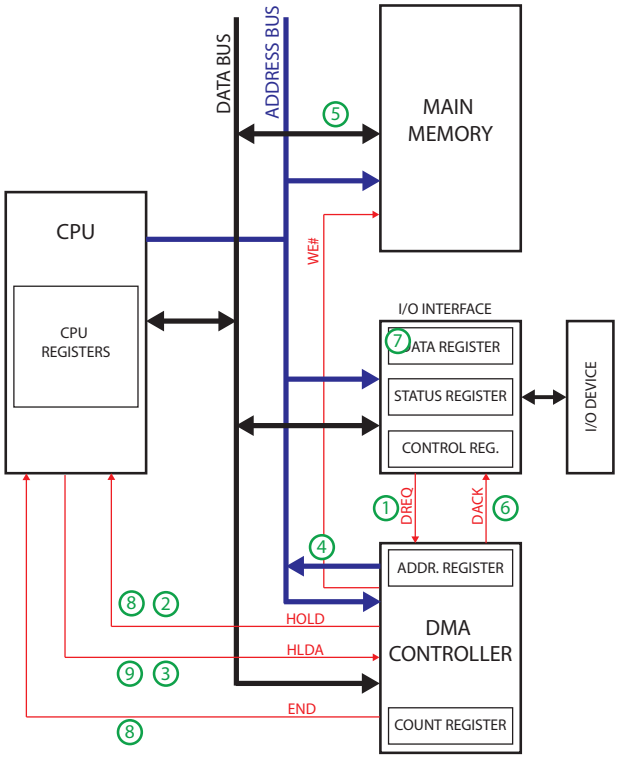

Fig. 1.3: A DMA transfer.

There is usually only one DMA controller in the computer system that is used for DMA transfers to/from several I/O devices. In that case, **the DMA controller has a separate pair of DREQ and DACK signals for each I/O device**. This separate pair (DREQ, DACK) is called **DMA channel**.

The DMA transfer described above is referred to as **"Fly-by" DMA**. This means that the data, which is transferred between an I/O device and memory, does not pass

through the DMA controller. "Fly-by" DMA refers to the DMA transfer between an I/O device and memory in which the data flows into/out of the memory using address lines for only one side of the transfer (the memory side). The other side (the I/O device) is "addressed" by the DACK signal, i.e., the DACK signal selects the I/O device involved in the DMA transfer, which should then latch data from the bus or place data onto the bus.

The way that the DMA function is implemented varies between computer architectures, and there is also another type of DMA transfer referred to as **"Fly-through" DMA**. In "Fly-through" DMA, both source and destination address need to be specified. The data flows through the DMA controller, which now has a FIFO buffer to store the data temporarily. The "Fly-through" DMA controller first places the source address onto the address bus, reads the data from the source into its internal FIFO, then places the destination address onto the address bus and writes the data from its FIFO into the destination.

"Fly-by" DMA is much faster because "Fly-through" DMA results in two bus transfers: one from the source to the internal FIFO and the other from the internal FIFO to the destination. But on the other hand, "Fly-through" DMA enables the memory-to-memory DMA transfers, which are not possible with "Fly-by" DMA controllers.

Summary: DMA controllers

Each DMA transfer is driven by at least the DMA controller's internal two registers: the address register and the count register.

A DMA channel is a pair of two control signals between a peripheral device and the DMA controller: DMA request (DREQ) and DMA acknowledge (DACK).

In "Fly-by" DMA transfers, the data, which is transferred between an I/O device and memory, does not pass through the DMA controller. Only the memory address needs to be specified, while the peripheral device is selected by the DACK signal. Only one memory transaction is needed to accomplish a DMA transfer.

In "Fly-through" DMA, both source and destination address need to be specified. The data flows through the DMA controller, which has a FIFO buffer to store the data temporarily. The "Fly-through" DMA controller first places the source address onto the address bus, reads the data from the source into its internal FIFO, then places the destination address onto the address bus and writes the data from its FIFO into the destination. Two memory transactions are required to accomplish one DMA transfer.

1.5 Real-world DMA Controllers

So far, we have learned that a DMA controller is a special device used to transfer data between an I/O device and the main memory without involving the CPU. Modern DMA controllers also support memory-to-memory DMA transfers, thus allowing efficient data transfers between two memory regions. For example, on most modern computer systems, the C library function `memcpy()` is implemented using the DMA transfer. In this section, we are going to describe two real-world DMA controllers and their functionality: the Intel 8237A DMA "fly-by" controller used in the older Intel PCs, and the "fly-through" DMA controller used in modern ARM Cortex-M based systems.

1.5.1 Intel 8237A DMA controller

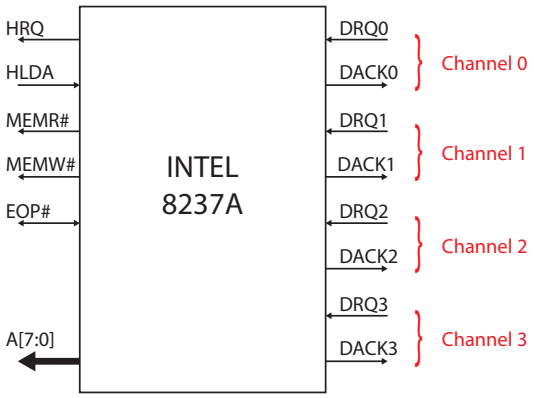

Fig. 1.4: Intel 8237A DMA controller. The signals used to initialize the DMA controller are not shown.

The Intel PC DMA subsystem is based on the Intel 8237A DMA controller. The Intel 8237A contains four DMA channels that can be programmed independently, and any of the channels may be active at any moment. These channels are numbered as 0, 1, 2, and 3. The Intel 8237A DMA controller moves one byte in each transfer and is very similar to the DMA controller described in Figure 1.2.

The Intel 8237A is depicted in Figure 1.4. It has two electrical signals for each channel, named DRQ (DMA request) and DACK (DMA acknowledge). There are additional signals with the names HRQ (Hold Request), HLDA (Hold Acknowledge), EOP# (End of Process), and the bus control signals MEMR# (Memory Read), and MEMW# (Memory Write). Table 1.1 provides full 8237A signals description.

Table 1.1: Intel 8237A signals description.

Signal name	Signal description
HRQ	Hold request is an output used to request the bus.
HLDA	Hold acknowladge is an input that signals that the CPU has granted the bus.
DREQ[3:0]	DMA request inputs are used to request a DMA transfer for each of the four DMA channels.
DACK[3:0]	DMA acknowledge outputs acknowledge the a channel DMA request and select the I/O device during the DMA transfer.
A[7:0]	These pins are outputs and are used to provide the DMA transfer memory address.
EOP#	End-of-process is a bidirectional active-low signal used used as an input to terminate a DMA transfer or as an output to signal the end of a DMA transfer.
MEMR#	Memory read is an active-low output used to read data from the selected memory location during a DMA transfer
MEMW#	Memory write is an active-low output used to write data to the selected memory location during a DMA transfer

The Intel 8237A DMA controller is a "fly-by" DMA controller. Subsequently, the DMA can only transfer data between an I/O device and a memory, but not between two I/O devices or two memory locations. Actually, the Intel 8237A controller does allow two channels to be connected to allow memory-to-memory DMA operations, but nobody in the PC industry used this DMA controller this way since it is faster to move data between memory locations using the CPU. Each DMA channel is activated only when an I/O device connected to that DMA channel requests a transfer by asserting the DRQ line.

Each channel in The 8237A DMA controller has two internal registers that control the transfer: the **count register** and the **address register**. Both registers are programmable by the CPU. The count register holds the number of bytes to be transferred, while the address register holds the (initial) memory address. When a byte of data is transferred, the address register is decremented or incremented, depending on how it is programmed. The count register is decremented after each transfer. When the value in the count register goes from zero to 0xFFFF, the EOP# output signal is activated.

The 8237A is designed to operate in two major cycles. These are called Idle and Active cycles. When no channel is requesting DMA transfer, the 8237A controller enters the Idle cycle. In this cycle, the 8237A samples the DREQ lines every clock cycle to determine if any channel is requesting a DMA transfer. When a channel requests a DMA service by asserting its DREQ signal, the 8237A asserts the HRQ signal to the microprocessor requesting the bus and enters the Active cycle. It is in the Active cycle that the DMA transfer will take place.

There are three modes of operation: single-mode, block mode, and demand mode. In single-mode, the device is programmed to make one transfer only. Single-

mode transfer releases the HOLD signal after each byte is transferred. In this mode, DRQ must be held active until DACK becomes active. If the DRQ is held active, the 8237A again requests the bus with the HOLD signal. Upon receipt of a new HLDA, another single transfer will be performed.

Block mode automatically transfers the number of bytes indicated by the count register. The count register will be decremented, and the address register decremented or incremented following each transfer. In block mode, the DMA controller is activated by DREQ to continue making transfers until the count register goes from 0 to FFFFH, or an external EOP# is activated. DREQ need only be held active until DACK becomes active.

In demand mode, the DMA controller transfers data until an external EOP# is asserted or until DREQ goes inactive. This mode is used when there is a block of data to be transferred, but the I/O device has not a high data capacity, and the transfer should be paused until the I/O device is ready again. During the time when the transfer is paused, the CPU is allowed to use the bus, and the intermediate values of address and word count are preserved. When the I/O is ready to continue the transfer, the DMA transfer is re-established by activation of the DREQ signal.

Intel 8237A was also used for a DRAM refresh. To refresh one row in DRAM, the 8237A DMA controller reads data from memory onto the data bus. During this dummy read, sense amplifiers in the memory chips are enabled. This automatically leads to the refresh of one memory cell row. But the data is not fetched by an I/O device, as no device has issued a DRQ, and the DMA controller does not assert a DACK.

Summary: Intel 8237A DMA Controller

The Intel 8237A controller is a "fly-by" DMA controller. Subsequently, the DMA can only transfer data between an I/O device and a memory. Each DMA transfer requires only one memory transaction. It was used in Intel-based PC systems.

It contains four DMA channels, and any of the channels may be active at any moment. Each channel in The 8237A DMA controller has two internal registers that control the transfer: the count register and the address register. Both registers are programmable by the CPU.

1.5.2 STM32H7 series DMA controller

This subsection describes the direct memory access (DMA) controller available in the STM32H7 Arm Cortex-M7 core-based series of systems-on-chips. The STM32H7 series DMA controller allows data transfers to take place in the background without the intervention of the Cortex-M7 processor. During this operation,

the processor can execute other tasks, and it is only interrupted when a whole data block is transferred and available for processing. The STM32H7 series DMA controller is a fly-through DMA controller and supports the transfer of large amounts of data with no significant impact on system performance. The DMA controller can do automated memory-to-memory, peripheral-to-memory and peripheral-to-peripheral transfers.

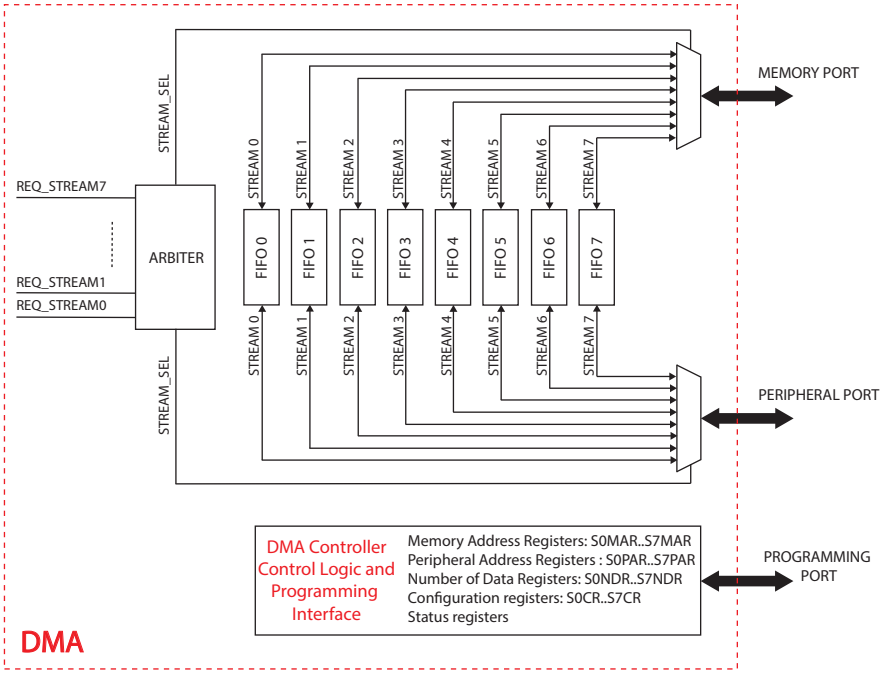

Fig. 1.5: Simplified block diagram of a STM32H7 series DMA controller.

Figure 1.5 illustrates the simplified block diagram of the STM32H7 series DMA controller. The STM32H7 series DMA controller features three ports: a programming port for DMA programming and two ports (peripheral and memory ports) that allow the DMA to initiate data transfers between different I/O devices and memory. A **port** is a connection to the data, address, and control bus. Thus, one port comprises data lines, address lines, and control signals, which are not depicted in Figure 1.5 due to simplicity.

Each STM32H7 series DMA controller supports up to eight streams. A **stream** is an active DMA transfer between a peripheral device and memory, two peripheral devices, or between two memory blocks. Each stream has eight selectable channels (requests). This selection is software-configurable and allows several peripherals to initiate DMA requests. Each channel is associated with a peripheral device that can trigger a data transfer request when ready. Thus, the DMA controller can be used

by up to 64 I/O devices (64 channels) and can manage up to eight interleaved DMA transfers (streams). More than one enabled DMA stream must not serve the same peripheral request.

The DMA controller contains an arbiter for handling the priority between DMA streams. Stream priority is software-configurable (four levels: very high, high, medium, low). The arbiter selects the stream with the highest priority. If two or more DMA streams have the same software priority level, the lowest stream number gets priority.

The programmable 8-channel DMAMUX multiplexer block (Figure 1.6) enables routing DMA request lines between the peripherals and the DMA controller. Each channel contains one 128-to-1 multiplexer and selects a unique DMA request line from peripherals. Each DMA stream is driven by one DMAMUX output channel (request). Any DMAMUX output request can be individually programmed to select the DMA request source signal from up to 128 available request input signals. The assignment of DMAMUX request multiplexer inputs to the DMA request lines from peripherals are detailed in the product reference manual.

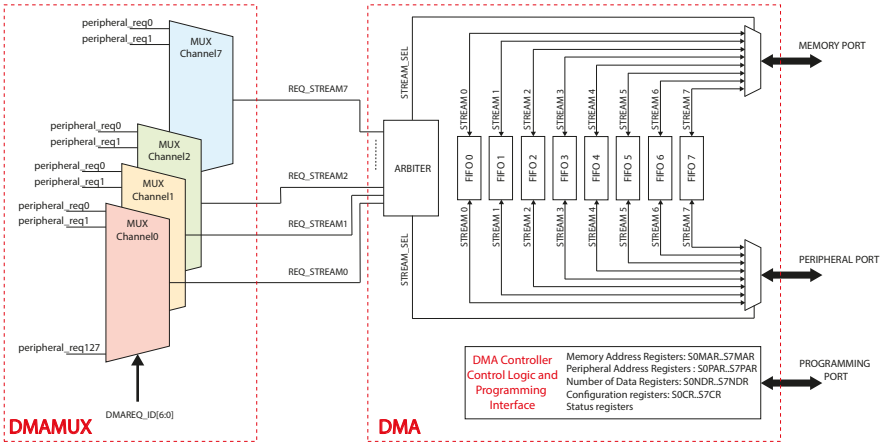

Fig. 1.6: Simplified block diagram of DMAMUX block and STM32H7 DMA controller.

When a peripheral is ready, it sends a DMA request to the DMAMUX, which routes the request to a selected DMA stream request input of the DMA controller. Depending on the stream priority, the DMA controller will then serve the DMA request. As this is a "fly-through" DMA controller, the data flows through the DMA controller, which has a FIFO buffer associated with each stream. The FIFO buffer is used to store the data temporarily and to amortize the difference in transmission speeds of two peripheral devices. Standard block transfer is accomplished by the DMA controller performing a sequence of memory transfers. Each transfer involves

a load operation from a source address into the FIFO, followed by a store operation from the FIFO to a destination address.

The DMA controller's control logic and programming interface are accessed through the programming port. The programming interface comprises a set of memory-mapped registers per stream. Each stream is characterized by four registers: Memory Address Register (SxMAR), Peripheral Address Register (SxPAR), Number of Data Register (SxNDR), and Configuration Register (SxCR). All these registers are memory-mapped and will be discussed in the following subsections.

STM32H7 devices embed two DMA controllers, offering up to 16 streams in total (eight per controller), each dedicated to managing memory access requests from one or more peripherals.

Summary: STM32H7 series DMA controller

The STM32H7 series DMA controller is a "fly-through" DMA controller used in the STM32H7 Arm Cortex-M based systems.

The STM32H7 series DMA controller features three ports: a programming port for DMA programming and initialization, and two ports (peripheral and memory ports) that allow the DMA to initiate data transfers between different I/O devices and memory.

Each STM32H7 series DMA controller supports up to eight *streams*. A stream is an active DMA transfer between a peripheral device and memory, two peripheral devices, or between two memory blocks.

Each DMA stream is driven by one DMAMUX output channel (request).

As this is a "fly-through" DMA controller, the data flows through the DMA controller, which has a FIFO buffer associated with each stream.

STM32H7 devices embed two DMA controllers.

1.5.2.1 Peripheral and memory addresses

Each DMA transfer is defined by a source address and a destination address. Both addresses should be aligned to transfer size. The transfer size value defines the volume of data to be transferred from source to destination. Each stream has a pair of registers to store these addresses: Peripheral Address Register (SxPAR - *Stream x Peripheral Address Register*) and Memory Address Register (SxMAR - *Stream x Memory Address Register*). Before each transfer, the CPU should initialize both registers with the valid addresses. It is possible to configure the DMA to automatically increment the source and/or destination address after each data transfer.

1.5.2.2 Transfer size, type and mode

Each DMA transfer is defined by the transfer size and the transfer mode. The transfer size is a value that defines the volume of data to be transferred from source to destination. This value is stored in the so-called Number of Data Register (NDR). Each stream has its Number of Data Register, labeled as SxNDTR. Each SxNDTR is a 16-bit register, and the number of data items to be transferred is software programmable from 1 to 65535. After each transfer, the value in SxNDTR is decreased by the amount of the transferred data; thus, SxNDTR contains the number of data transfers still to be performed.

The STM32H7 series DMA controller can perform two transfer types: normal type and circular type. In normal type, once the SxNDTR register reaches zero (the transfer has completed), the stream is disabled. This means that the CPU should reinitialize the DMA controller in order to activate the stream again. In circular type, the DMA controller can handle circular buffers and continuous data flow. In this type, the SxNDTR register is reloaded automatically with the previously programmed value when a transfer has completed.

Each STM32H7 series DMA controller is capable of performing three different transfer modes:

1. peripheral to memory,
2. memory to peripheral,
3. memory to memory (only the second DMA controller is able to do such transfer; in this mode, the circular type is not allowed).

1.5.2.3 FIFOs and burst transfers

Each stream has a 4x32 bits FIFO that is used to temporarily store data coming from the source before transmitting them to the destination. The DMA FIFOs help to reduce memory access and to do burst transactions which optimize the transfer bandwidth. They also allow independent source and destination transfer width (byte, half-word, word): when the data widths of the source and destination are not equal, the DMA automatically packs/unpacks the necessary transfers to optimize the bandwidth. For example, the data from the source can be transferred into FIFO as bytes or 16-bit half-words and then transferred to the destination from FIFO as bytes, 16-bit half-words, or 32-bit words.

Because of the internal FIFOs, the DMA controller is capable of burst transfers of length 4x, 8x, or 16x data units. A data unit can be a byte, a 16-bit half-word, or a 32-bit word. The burst size on the DMA peripheral port must be set according to the peripheral needs/capabilities. The size of the burst is software-configurable, usually equal to half the FIFO size of the peripheral.

1.5.2.4 DMA Transactions

A DMA transaction consists of a sequence of a given number of data transfers. The number of data items to be transferred and their width (8-bit, 16-bit or 32-bit) are software-programmable. Each DMA transfer consists of three operations:

1. loading from the peripheral data register or a location in memory, addressed through the DMA_SxPAR or DMA_SxM0AR register
2. storage of the data loaded to the peripheral data register or a location in memory addressed through the DMA_SxPAR or DMA_SxM0AR register
3. post-decrement of the DMA_SxNDTR register containing the number of transactions that still have to be performed

The peripheral sends a request signal to the DMA controller through the DMAMUX block. The DMA controller serves the request depending on the channel priorities. As soon as the DMA controller accesses the peripheral, an acknowledgement signal is sent to the peripheral by the DMA controller, which in turn releases its request. Once the peripheral de-asserts the request, the DMA controller releases the acknowledge signal.

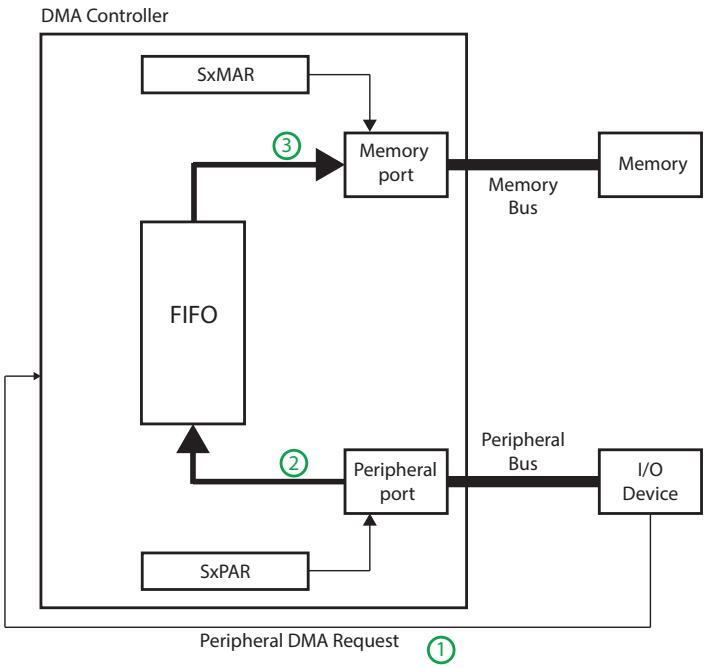

Fig. 1.7: A peripheral-to-memory DMA transaction.

Figure 1.7 illustrates a peripheral-to-memory DMA transaction. Each time a peripheral request occurs, the stream initiates a transfer from the source (address is in SxPAR) to fill the FIFO. Then, the contents of the FIFO are drained and stored in the destination (address is in the SxMAR). The transfer stops once the SxNDTR register reaches zero, or when the enable bit in the SxCR register is cleared by software (stream disabled).

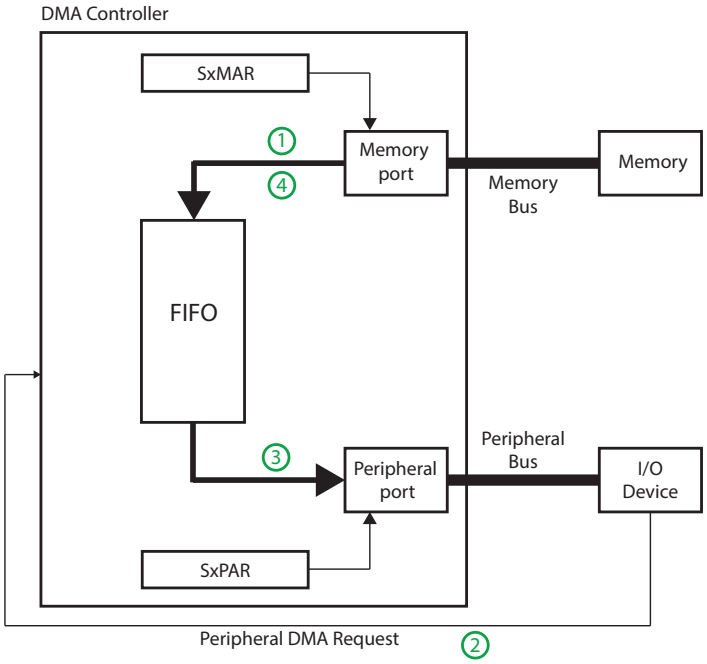

Fig. 1.8: A memory-to-peripheral DMA transaction. In this mode, the stream immediately initiates transfers from the memory to entirely fill the FIFO. When a peripheral request occurs, the contents of the FIFO are stored in the peripheral device.

Figure 1.8 illustrates a memory-to-peripheral DMA transaction. In this mode, the stream immediately initiates transfers from the source (address is in SxMAR) to entirely fill the FIFO, and the SxMAR register is incremented/decremented. The DMA controller does not wait for DMA request from a peripheral device to read from memory. When a peripheral request occurs, the contents of the FIFO are drained and stored in the destination (address is in the SxPAR). The DMA controller then reloads the empty internal FIFO again with the next data to be transferred from memory (address is in SxMAR). The transfer stops once the SxNDTR register reaches zero, or when the enable bit in the SxCR register is cleared by software (stream disabled).

1.5.2.5 Programming and using the STM32H7 series DMA controller

Programming and using the STM32H7 series DMA controller is relatively easy. Each stream is controlled using four memory-mapped registers: memory address register (SxMAR), peripheral address register (SxPAR), number of data register (SxNDTR), and configuration register (SxCR). Once set, the DMA controller handles data transfers and increments memory addresses without disturbing the CPU. To configure the DMA controller and a DMA stream, the following procedure should be applied:

1. If the stream is enabled, disable it by resetting the stream enable bit in the SxCR register, then read this bit to confirm that there is no ongoing stream operation. When the EN bit is 0, the stream is ready to be configured.
2. Set the peripheral port register address in the SxPAR register. After the peripheral DMA request, the data will be moved from/to this address to/from the peripheral port.
3. Set the memory address in the SxMAR register. After the peripheral DMA request, the data will be written to or read from this memory address.
4. Configure the total number of data items to be transferred in the SxNDTR register. After each (burst) transfer, this value is decremented accordingly.
5. Use DMAMUX to route a peripheral DMA request line to the DMA stream request signal.
6. Configure the stream priority, the data transfer direction, single or burst transactions, peripheral and memory data widths, circular/normal transfer type, and interrupts in the SxCR register.
7. Activate the stream by setting the stream enable bit in the SxCR register.

As soon as the stream is enabled, it can serve any DMA request from the peripheral connected to the stream and DMA transactions using the stream can be performed.

Summary: STM32H7 series DMA transfers

Each DMA transfer is defined by a source address and a destination address, and each stream has a pair of registers to store these addresses: Peripheral Address Register (SxPAR -Streamx Peripheral Address Register) and Memory Address Register (SxMAR -Stream x Memory Address Register)

Each DMA transfer is defined by the transfer size and the transfer mode. Each stream has its Number of Data Register (SxNDR), which stores the transfer size.

The STM32H7 series DMA controller can perform two transfer types: normal type and circular type.

FIFOs allow independent source and destination transfer width and burst transfers.

Each DMA transfer consists of two transactions on the bus: loading from the peripheral data register or a location in memory, and storage of the data to the peripheral data register or a location in memory.

1.6 Bus Mastering DMA

So far, we have learned that we can use a special piece of hardware, a DMA controller, namely, to transfer large amounts of data between a peripheral device and memory. This approach is sometimes referred to as **third-party DMA**. Third-party DMA requires an independent DMA controller, which is built into motherboard chipsets, to move data between a peripheral device (referred to as the *first party*) and system RAM (referred to as the *second party*). Here, the DMA controller is shared by multiple peripheral devices, which is why it is viewed as the third party DMA. As we have learned previously, each "fly-through" DMA transfer (fly-through is the type of the DMA used in the majority of today's computer systems) requires two memory transactions: one to load the data from the source, and one to store the data to the destination.

The better approach to DMA transfers would be to have only one memory transaction per DMA transfer, but still avoiding third-party "fly-by" DMA controllers. This is possible with the latest I/O devices built in the modern computer systems, where each I/O device can act as a *bus master*, i.e., each device can directly access any other I/O device or memory on the bus. Indeed, each modern I/O device now contains its own, integrated, DMA controller, which is not shared by other I/O devices. This highest performing DMA type is called first-party DMA or **Bus Mastering DMA**. Peripheral devices, which support the Bus Mastering technology, have the ability to move data to and from system memory without the intervention of the CPU or a third party DMA controller.

Bus Mastering allows data to be transferred much faster than third party DMA. This is because half as many bus cycles are needed. The third-party DMA requires the DMA controller to alternately read a segment of data from one device (this can be a peripheral device or system memory) and write it to the other device. Each data segment requires at least one bus cycle to be read and one bus cycle to be written. Bus mastering devices only require bus cycles when accessing system memory, so half as many bus cycles are needed. Because of this, devices that support Bus Mastering can move data many times faster than third party DMA. While bus mastering theoretically allows one peripheral device to directly communicate with another, in practice almost all peripherals master the bus exclusively to perform peripheral-to-memory and memory-to-peripheral transfers.

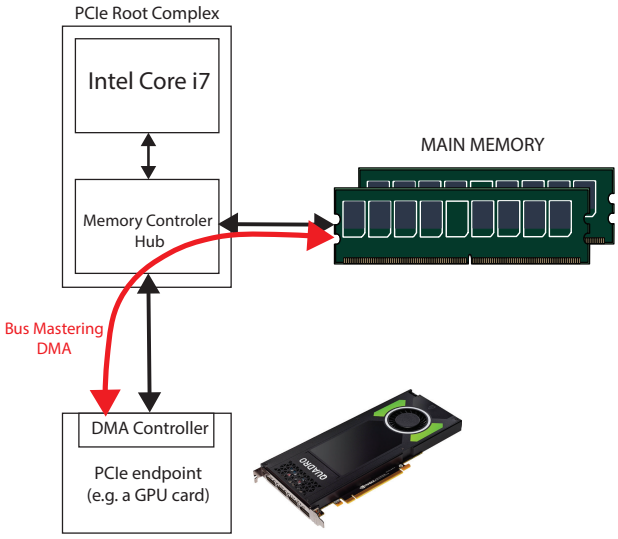

Fig. 1.9: Bus Maserting in an Intel based system. Bus Mastering is the feature integrated into PCIe endpoint devices. A DMA transfer either transfers data from an endpoint device into system memory or from system memory into the endpoint device on the PCI Express bus. The DMA request is always initiated by the integrated DMA controller in the endpoint device after being initialized from the application driver (i.e., receiving parameters that define DMA transaction and memory buffer address).

Bus Mastering is used in the computer systems with a PCI Express (PCIe) bus. A Bus Mastering DMA implementation is by far the most common type of DMA found in systems based on PCI Express and resides within the peripheral device, which is called Bus Master because it initiates the movement of data to and from system memory. Figure 1.9 shows a typical Intel system architecture. The system in-

cludes the CPU core(s) and a memory controller hub—these two form the so-called PCIe root complex. The system in Figure 1.9 also contains the main memory and one PCIe peripheral device (e.g. a GPU card). The peripheral device is connected to the PCIe bus. PCIe peripheral devices are called PCI **endpoint devices**. The memory controller hub also acts as a bridge between the PCIe bus, the CPU bus, and the main memory bus. A DMA transfer either transfers data from an endpoint device into system memory or from system memory into the endpoint device on the PCI Express bus. The DMA request is always initiated by the integrated DMA controller in the endpoint device after being initialized from the application driver (i.e., receiving parameters that define DMA transaction and memory buffer address).

Summary: Bus Mastering

Bus Mastering DMA is the highest performing DMA type. Peripheral devices, which support the Bus Mastering technology, have the ability to move data to and from system memory without the intervention of the CPU or a third party DMA controller.

Bus Mastering is used in the computer systems with a PCI Express (PCIe) bus.

PCIe peripheral devices are called PCI **endpoint devices**. Endpoint devices have their own integrated DMA controller, which is not shared by other endpoint devices.