AN4104
Application note

Using the STM32F0xx DMA controller

Introduction

This application note describes how to use the STM32F0xx direct memory access (DMA) controller. The STM32F0xx DMA controller, the Cortex™-M0 core, the advanced microcontroller bus architecture (AMBA) bus and the memory system contribute to provide a high data bandwidth and to develop very-low latency response time software.

This application note also describes how to take full advantage of these features and ensure correct response times for different peripherals and subsystems.

Note: *To ensure a quick start, application cases presented in this document are implemented in C language and are available in Project\STM32F0xx_StdPeriph_Examples within the STM32F0xx_StdPeriph_Lib package.*

Contents

1 DMA controller description

Direct memory access (DMA) is used in order to provide high-speed data transfer betweenperipherals and memory as well as memory to memory. Data can be quickly moved by DMA without any CPU actions. This keeps CPU resources free for other operations.

The DMA allows data transfers to take place in the background, without the intervention of the Cortex-M0 processor. During this operation, the main processor can execute other tasks and it is only interrupted when a whole data block is available for processing. Large amounts of data can be transferred with no major impact on the system performance.

The DMA is mainly used to implement central data buffer storage (usually in system SRAM) for different peripheral modules. This solution is less expensive in terms of silicon and power consumption compared to a distributed solution where each peripheral needs to implement its own local data storage.

Depending on the sales type used, one or two DMA controllers are implemented.

The STM32F0xx DMA controller has 5 channels for DMA1 in total, each dedicated to manage memory access requests from one or more peripherals. It has an arbiter for handling the priority between DMA requests.

1.1 DMA Overview

The DMA(s) offer(s):

- independently configurable channels (requests)
- Each channels are connected to dedicated hardware DMA requests, software trigger is also supported on each channel
- Priorities between requests from channels of one DMA are software programmable (4 levels: very high, high, medium, low) or hardware in case of equality (request 1 has priority over request 2, etc.)
- Independent source and destination transfer size (byte, half word, word), emulating packing and unpacking. The source and the address must have the same data size (to be aligned on the data size).
- Support for circular buffer management
- 3 event flags (DMA Half Transfer, DMA Transfer complete and DMA Transfer Error) logically ORed together in a single interrupt request for each channel
- Memory-to-memory transfer
- Peripheral-to-memory and memory-to-peripheral as well as peripheral-to-peripheral transfers
- Access to Flash, SRAM, APB and AHB peripherals as source and destination
- Programmable number of data to be transferred: up to 65536

The DMA aims to offer a relatively large data buffer to all peripherals. This buffer is usually located in system SRAM.

Each channel is assigned to a unique peripheral (data channel) at a given time. Peripherals connected to the same DMA channel (CH1 to CH5 in *Table 1* for STM32F0xx devices) cannot be used simultaneously with active DMA (DMA function active in the peripheral register).

The different peripherals supporting DMA transfers in STM32F0xx devices are shown in *Table 1*.

Table 1. Peripherals served by DMA1 and channel allocation

Peripherals		CH1	CH2	CH3	CH4	CH5
ADC	**ADC1**	ADC1	ADC1			
SPI	**SPI1**		SPI1_RX	SPI1_TX		
	SPI2				SPI2_RX	SPI2_TX
USART	**USART1**		USART1_TX	USART1_RX	USART1_TX	USART1_RX
	USART2				USART2_TX	USART2_RX
I^{2}C	**I^{2}C1**		I2C1_TX	I2C1_RX		
	I^{2}C2				I2C2_TX	I2C2_RX
TIM	**TIM1**		TIM1_CH1	TIM1_CH2	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP TIM1_CH3
	TIM2	TIM2_CH3	TIM2_UP	TIM2_CH2	TIM2_CH4	TIM2_CH1
	TIM3		TIM3_CH3	TIM3_CH4 TIM3_UP	TIM3_CH1 TIM3_TRIG	
	TIM6/DAC			TIM6_UP DAC		
	TIM15					TIM15_CH1 TIM15_UP TIM15_TRIG TIM15_COM
	TIM16			TIM16_CH1 TIM16_UP	TIM16_CH1 TIM16_UP	
	TIM17	TIM17_CH1 TIM17_UP	TIM17_CH1 TIM17_UP			

Note: For more details,refer to RM0091 DMA section for STM32F0xx devices.

1.2 DMA Data managing

The DMA controller performs direct memory transfer by sharing the system bus with the Cortex-M0 core. When the CPU and DMA are targeting the same destination (memory or peripheral) the DMA request may stop the CPU access to the system bus for several bus cycles. The bus matrix implements round-robin scheduling, thus ensuring at least half of the system bus bandwidth (both to memory and peripheral) for the CPU.

1.2.1 Round robin priority scheme

The NVIC and Cortex-M0 processor implement a high-performance very low latency interrupt scheme. All Cortex-M0 instructions are either executed in a single cycle or are interruptible at cycle level. In order to preserve this advantage at system level, the DMA and bus matrix ensure that the DMA does not block the bus for a long time. The round-robin priority scheme ensures that the CPU can access any slave buses during every second cycle, if needed.

1.2.2 Peripheral to Memory, Memory to Peripheral and Peripheral to Peripheral DMA transactions

After an event, the peripheral sends a request signal to the DMA Controller. The DMA controller serves the request depending on the channel priorities. As soon as the DMA Controller accesses the peripheral, an Acknowledge is sent to the peripheral by the DMA Controller. The peripheral releases its request as soon as it gets the Acknowledge from the DMA Controller. Once the request is deasserted by the peripheral, the DMA Controller releases the Acknowledge. If there are more requests, the peripheral can initiate the next transaction.

Each DMA transfer consists of three operations:

- The loading of data from the peripheral data register or a location in memory addressed through an internal current peripheral/memory address register. The start address used for the first transfer is the base peripheral/memory address.
- The storage of the data loaded to the peripheral data register or a location in memory addressed through an internal current peripheral/memory address register. The start address used for the first transfer is the base peripheral/memory address.
- The post-decrementing of the DMA counter, which contains the number of transactions that still have to be performed.

1.2.3 Memory to Memory DMA transactions

The DMA channels can also work without being triggered by a request from a peripheral. This mode is called Memory to Memory mode. If the MEM2MEM bit is set, then the channel initiates transfers as soon as it is enabled by software by setting the Enable bit.

The transfer stops once the DMA counter reaches zero. Memory to Memory mode may not be used at the same time as Circular mode.

1.2.4 Choosing channel priority

In order to achieve continuous data transfers to/from a peripheral, the corresponding DMA channel must be able to sustain the peripheral data rate and ensure that the service latency is shorter than the period of time between two consecutive data.

The high speed/high bandwidth peripherals must have the highest DMA priorities. This ensures that the maximum data latency will be respected for these peripherals and over/under-run conditions will be avoided.

In case of equal bandwidth requirements, it is recommended to assign a higher priority to the peripherals working in slave mode (which have no control on the data transfer speed) compared with the ones working in master mode (which may control the data flow).

By default, the channel allocation and hardware priority (from 1 to 5 for STM32F0xx devices) are set in order to assign the fastest peripherals to the highest priority channels. However, this may not be true for some applications. In this case, the user can configure a software priority for each channel (4 levels – from Very High to Low), which takes precedence over the hardware priority.

1.3 DMA Interrupt management

An interrupt can be produced on a Half-transfer, Transfer complete or Transfer error for each DMA channel. Separate interrupt enable bits are available for flexibility.

Table 2. DMA Interrupt requests

Interrupt events	Event Flags	Control bit enable
Half Transfer	HTIF	HTIF
Transfer Complete	TCIF	TCIF
Transfer Error	TEIF	TEIF

When a DMA transfer error occurs during a DMA read or a write access, the faulty channel is automatically disabled through a hardware clear of its enable bit in the corresponding Channel configuration register (DMA_CCRx). The channel's transfer error interrupt flag (TEIF) in the DMA_IFR register is set and an interrupt is generated if the transfer error interrupt enable bit (TEIE) in the DMA_CCRx register is set.

2 DMA firmware driver API

This driver provides a set of firmware functions to manage the following functionalities of the DMA peripheral:

- Initialization and Configuration functions
- Data Counter functions
- Interrupts and flags management functions

For the STM32F0xx family, the DMA driver stm32f0xx_dma.c/.h can be found in the directory: STM32F0xx_StdPeriph_Lib_vX.Y.Z\Libraries\STM32F0xx_StdPeriph_Driver.

This driver provides a fully compatible API making it easy to move from one product to another.

Table 3. DMA Functions description

Groups	Function name	Description
Initialization and Configuration functions	DMA_DeInit	Deinitializes the DMAy Channelx registers to their default reset values.
	DMA_Init	Initializes the DMAy Channelx according to the specified parameters in the DMA_InitStruct.
	DMA_StructInit	Fills each DMA_InitStruct member with its default value.
	DMA_Cmd	Enables or disables the specified DMAy Channelx.
Data Counter functions	DMA_SetCurrDataCounter	Sets the number of data units in the current DMAy Channelx transfer.
	DMA_GetCurrDataCounter	Returns the number of remaining data units in the current DMAy Channelx transfer.
Interrupts and flags management functions	DMA_ITConfig	Enables or disables the specified DMAy Channelx interrupts.
	DMA_GetFlagStatus	Checks whether the specified DMAy Channelx flag is set or not..
	DMA_ClearFlag	Clears the DMAy Channelx's pending flags.
	DMA_GetITStatus	Checks whether the specified DMAy Channelx interrupt has occurred or not.
	DMA_ClearITPendingBit	Clears the DMAy Channelx's interrupt pending bit..

Several parameters such as source/destination address (Location where data is to be read or transferred) and transfer length must be specified in order for a DMA transaction to take place.

The DMA field configuration is stored in a structure as described below:

– **DMA_PeripheralBaseAddr**: Specifies the peripheral base address for DMAy Channelx.
– **DMA_MemoryBaseAddr**: Specifies the memory base address for DMAy Channelx.
– **DMA_DIR**: Specifies if the peripheral is the source or destination.
– **DMA_BufferSize**: Specifies the buffer size, in data unit, of the specified Channel. The data unit is equal to the configuration set in DMA_PeripheralDataSize or DMA_MemoryDataSize members depending in the transfer direction.
– **DMA_PeripheralInc**: Specifies whether the Peripheral address register is incremented or not after transferring each unit.
– **DMA_MemoryInc**: Specifies whether the memory address register is incremented or not after transferring each unit.
– **DMA_PeripheralDataSize**: Specifies the Peripheral data width. The size of the transfer unit can be byte, Half-Word or Word.
– **DMA_MemoryDataSize**: Specifies the Memory data width. The size of the transfer unit can be byte, Half-Word or Word.
– **DMA_Mode**: Specifies the operation mode of the DMAy Channelx (Normal or circular mode).
– **DMA_Priority**: Specifies the software priority for the DMAy Channelx.
– **DMA_M2M**: Specifies if the DMAy Channelx will be used in memory-to-memory transfer.

Note: For further details, please refer to the DMA section in the reference manual RM0091 for STM32F0x devices.

2.1 How to use DMA Driver

1. Before using the DMa driver, enable The DMA controller clock using RCC_AHBPeriphClockCmd (RCC_AHBPeriph_DMAX, ENABLE) function.
2. Enable and configure the peripheral to be connected to the DMA channel (except for internal SRAM / Flash memories: no initialization is necessary).
3. For a given channel, program the Source and Destination addresses, the transfer Direction, the Buffer Size, the Peripheral and Memory Incrementation mode and Data Size, the Circular or Normal mode, the channel transfer Priority and the Memory-to-Memory transfer mode (if needed) using the DMA_Init() function.
4. Enable the NVIC and the corresponding interrupt(s) using the function DMA_ITConfig() if you need to use DMA interrupts.
5. Enable the DMA channel using the DMA_Cmd() function.
6. Activate the needed channel request using PPP_DMACmd() function for the adequate PPP peripheral except internal SRAM and Flash (ie. SPI, USART...) The function allowing this operation is provided in each PPP peripheral driver (ie. SPI_DMACmd for SPI peripheral).
7. Optionally, configure the number of data to be transferred when the channel is disabled (ie. after each Transfer Complete event or when a Transfer Error occurs) using the function DMA_SetCurrDataCounter(). And you can get the number of remaining data to be transferred using the function DMA_GetCurrDataCounter() at run time (when the DMA channel is enabled and running).
8. To control DMA events you can use one of the following two methods:
   – Check on DMA channel flags using the function DMA_GetFlagStatus().
   – Use DMA interrupts through the function DMA_ITConfig() at initialization phase and DMA_GetITStatus() function into interrupt routines in communication phase. After checking on a flag, clear it using DMA_ClearFlag() function. And after checking on an interrupt event, clear it using DMA_ClearITPendingBit() function.

3 DMA programming examples

The DMA firmware driver is provided with a set of examples, so you can quickly become familiar with the DMA peripheral. It demonstrates how to use the DMA in different modes.

Both the package and the application note are available for download from the STMicroelectronics website: http://www.st.com.

3.1 ADC DMA transfer to TIM example

This example provides a description of how to use a DMA channel to transfer continuously a data from a peripheral (ADC) to another peripheral (TIM) supporting DMA transfer. The ADC is configured to operate in Continuous Conversion mode. TIM is configured to generate a PWM signal on its output.

The dedicated DMA channel is configured to transfer in circular mode the last ADC channel converted value to the TIMER Capture/Compare register. The DMA channel request is driven by the TIM update event. The duty cycle of TIMER channel output signal is then changed each time the input voltage value on ADC channel pin is modified.

The duty cycle variation can be visualized on oscilloscope while changing the analog input on ADC channel using the potentiometer.

3.2 DMA Flash to RAM example

This example illustrates how to use the DMA to transfer data between two memory locations.

It provides a description of how to transfer a word data buffer located in Flash memory to embedded SRAM memory using DMA channel.

DMA Channel is configured to transfer the contents of a 32-word data buffer stored in Flash memory to the reception buffer declared in RAM.

The start of transfer is triggered by software. DMA Channel memory-to-memory transfer is enabled. Source and destination addresses incrementing is also enabled.

The transfer is started by setting the Channel enable bit for DMA Channel. At the end of the transfer, a Transfer Complete interrupt is generated since it is enabled. Once interrupt is generated, the remaining data to be transferred is read which must be equal to 0 (DMA Counter reaches 0 if all Data are transferred). The Transfer Complete Interrupt pending bit is then cleared.

A comparison between the source and destination buffers is done to check that all data have been correctly transferred.

3.3 DMA RAM to DAC example

This example provides a description of how to use a DMA channel to transfer data buffer from memory (RAM memory) to the peripheral DAC.

The DMA channel is configured to transfer continuously, word by word, a Half-word buffer from the RAM memory to the DAC register DAC_DHR12R. The DAC channel conversion is configured to be triggered by TIM2 TRGO triggers and without noise/triangle wave generation. 12bit right data alignment is selected since we choose to access DAC_DHR12R register.

3.4 SPI DMA example: communication between two SPIs using DMA

This example provides an SPI communications using DMA.

In master board, the SPI peripheral is configured as Master full duplex with DMA and NSS hardware mode.

The TIM2 is configured to generate 4 KHz PWM signal with 50% duty cycle on TIM2_CH2 pin (PA.01), this signal is used as DMA trigger and as NSS signal input to latch the SPI data transfers. Whereas in Slave board, the SPI peripheral is configured as Slave Full duplex with DMA and NSS hardware mode.

- The Master sends the specific command to the Slave using the TIM2_CH2 DMA request (DMA1_Channel3) (the command contains the transaction code (CMD_RIGHT, CMD_LEFT, CMD_UP, CMD_DOWN or CMD_SEL) and receives the ACK command from the Slave using SPI_Rx DMA request (DMA1_Channel2).
- The Slave receives the command using SPI_Rx DMA request (DMA1_Channel2) and sends the ACK command using the SPI_Tx DMA request (DMA1_Channel3).

3.5 USART communication boards data exchange using DMA example

This example provides a small application of USART communications using DMA.

In both boards, the data transfers is managed using the USART Tx/Rx channels DMA requests.

4 Revision history

Table 4. Document revision history

Date	Revision	Changes
02-May-2012	1	Initial release