Digital camera interface (DCMI) for STM32 MCUs

Introduction

As the demand for better and better image quality increases, the imaging domain continually evolves giving rise to a variety of technologies (such as 3D, computational, motion, and infrared).

Imaging applications require high-quality, ease-of-use, power efficiency, high level of integration, fast time-to-market, and cost effectiveness. To meet these requirements, the STM32 MCUs embed a digital camera interface (DCMI), which allows connection to efficient parallel camera modules.

The STM32 MCUs provide many performance levels (CPU, MCU subsystem, DSP, and FPU). They also provide various power modes, an extensive set of peripheral and interface combinations (for example, SPI, UART, I2C, SDIO, USB, ETHERNET, or I2S), a rich graphical portfolio (such as LTDC, QUADSPI, or DMA2D), and an industry-leading development environment ensuring sophisticated applications and connectivity solutions (IOT).

This application note gives to the STM32 users some basic concepts, with easy-to-understand explanations of the features, architecture, and configuration of the DCMI. It is supported by an extensive set of detailed examples.

Refer to the device reference manual and datasheet for more details.

Table 1. Applicable products

Type	STM32 lines
Microcontroller	STM32F2x7
	STM32F407/417, STM32F427/437, STM32F429/439, STM32F446, STM32F469/479
	STM32F7x0 Value line(1), STM32F7x5, STM32F7x6, STM32F7x7, STM32F7x8, STM32F7x9
	STM32H723/733, STM32H725/735, STM32H730 Value line, STM32H742, STM32H743/753, STM32H745/755, STM32H747/757, STM32H750 Value line, STM32H7A3/B3
	STM32L4x6
	STM32L4P5/Q5, STM32L4R5/S5, STM32L4R7/S7, STM32L4R9/S9
	STM32U575/585

1. *Only STM32F750xx devices.*

1 General information

This application note applies to the STM32 Series microcontrollers that are Arm® Cortex® core-based devices.

Note: Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 Camera modules and basic concepts

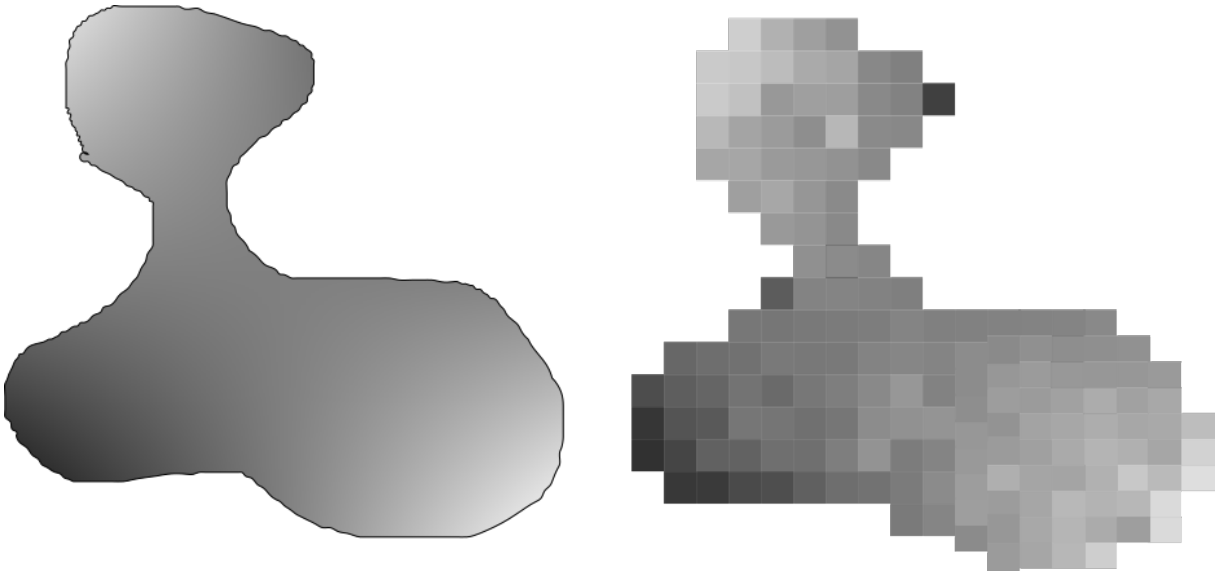
This section provides a summarized description of camera modules and their main components. It also highlights the external interface focusing on parallel camera modules.

2.1 Imaging basic concepts

This section introduces the imaging field, and gives an overview of the basic concepts and fundamentals, such as pixel, resolution, color depth, and blanking.

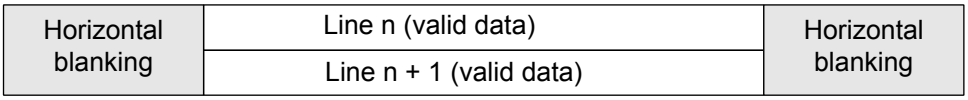
- pixel: each point of an image represents a color for color images, or a gray scale for black-and-white photos
  A digital approximation is reconstructed to be the final image. This digital image is a two-dimensional array composed of physical points. Each point is called a pixel (invented from picture elements). In other words, a pixel is the smallest controllable element of a picture. Each pixel is addressable. Figure 1 illustrates the difference between the original image and the digital approximation.

Figure 1. Original versus digital image



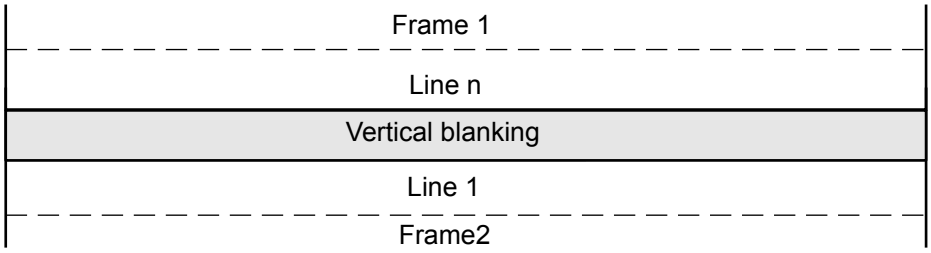
- resolution: number of pixels in the image
  The more the pixel size increases, the more the image size increases. For the same image size, the higher the number of pixels is, the more details the image contains.
- color depth (bit depth): number of bits used to indicate the color of a pixel (is also given in bit per pixel, bpp)
  Examples:
  - For a bitonal image, each pixel comprises one bit. Each pixel is either black or white (0 or 1).
  - For a gray scale, the image is most of the time composed of 2 bpp (each pixel can have one of four gray levels) to 8 bpp (each pixel can have one of 256 gray levels).
  - For color images, the number of bits per pixel varies from 8 to 24 (each pixel can have up to 16777216 possible colors).
- frame rate (for video): number of frames (or images) transferred each second, expressed in frame per second (FPS)
- horizontal blanking: ignored rows between the end of one line, and the beginning of the next one

Figure 2. Horizontal blanking illustration



- vertical blanking: ignored lines between the end of the last line of a frame, and the beginning of the first line in the next frame

Figure 3. Vertical blanking illustration

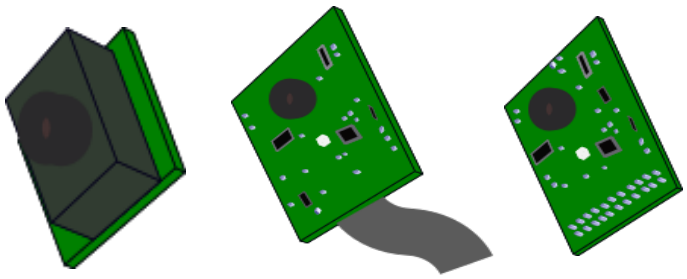


- progressive scan: a manner of dealing with moving images
  The lines can be drawn one after the other in sequence, without separating the odd lines from the even ones, as for interlaced scan. To construct the image:
  - in progressive scan, the first line is drawn, then the second, then the third
  - in interlaced scan, each frame is divided in two fields (odd and even lines), which are displayed alternatively

2.2 Camera module

A camera module consists of four parts: image sensor, lens, printed circuit board (PCB), and interface. Figure 4 shows some common camera modules examples.

Figure 4. Camera module examples



2.2.1 Camera module components

The four components of a camera module are described below.

Image sensor

It is an analog device used to convert the received light into electronic signals. These signals convey the information that constitutes the digital image.

There are two types of sensors that can be used in digital cameras:

- CCD (charge-coupled device) sensors
- CMOS (complementary metal-oxide semiconductor) sensors

Both types convert the light into electronic signals, but each has its own conversion method. As their performance continually evolves and their cost decreases, CMOS imagers now dominate the digital photography landscape.

Lens

It is an optic, which allows the reproduction of the real image captured rigorously on the image sensor. Picking the proper lens is part of the user creativity, and affects considerably the image quality.

Printed circuit board (PCB)

It is a board that comprises electronic components to ensure the good polarization and the protection of the image sensor. The PCB also supports all the other parts of the camera module.

Camera module interconnect

The camera interface is a kind of bridge that allows the image sensor to connect to an embedded system, and to send/receive signals. The following signals are transferred between a camera and an embedded system:

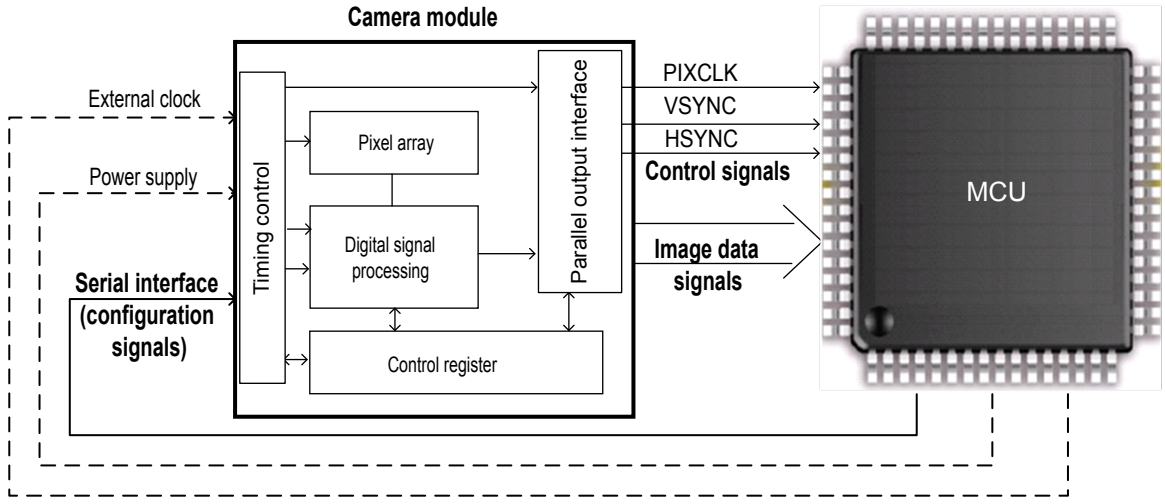
- control signals
- image data signals
- power supply signals
- camera configuration signals

The camera interfaces are divided in two types: parallel and serial interfaces, depending on the method to transfer data signals.

2.2.2 Camera module interconnect (parallel interface)

As mentioned above, a camera module requires four main types of signals to transmit image data properly: control signals, image data signals, power supply signals, and camera configuration signals. Figure 5 illustrates a typical block diagram of a CMOS sensor, and the interconnection with an MCU.

Figure 5. Interfacing a camera module with an STM32 MCU



- control signals: used for clock generation and data transfer synchronization
  The camera clock must be provided according to the camera specification. The camera also provides two data synchronization signals: HSYNC (for horizontal/line synchronization) and VSYNC (for vertical/frame synchronization).
- image data signals: each of them transmits a bit of the image data.
  Their width represents the number of bits to be transferred at each pixel clock. This number depends on the parallel interface of the camera module, and on the embedded system interface.
- power supply signals:
  As any embedded electronic system, the camera module needs to have a power supply. The operating voltage of the camera module is specified in its datasheet.
- configuration signals: used for the following:
  - to configure the appropriate image features such as resolution, format, and frame rate
  - to configure the contrast and the brightness
  - to select the type of interface (a camera module can support more than one interface: a parallel and a serial interface. The user must then choose the most convenient one for the application.)

Most camera modules are parametrized through an I^2C communication bus.

3 STM32 DCMI overview

This section gives a general preview of the DCMI availability across the various STM32 devices, and gives an easy-to-understand explanation on the DCMI integration in the STM32 MCUs architecture.

The DCMI is a synchronous parallel data bus, which is used for an easy integration and easy adaptation to specific application requirements. The DCMI connects with 8-, 10-, 12-, and 14-bit CMOS camera modules, and supports a multitude of data formats.

3.1 DCMI availability and features across STM32 MCUs

Table 2 summarizes STM32 devices embedding the DCMI, and highlights the availability of other hardware resources that facilitate the DCMI operation, or that can be used with the DCMI in the same application.

The DCMI applications need a frame buffer to store the captured images. It is then necessary to use a memory destination that varies depending on the image size and the transfer speed.

In some applications, it is necessary to interface with external memories that offer big sizes for data storage. The Quad-SPI can be used in this case. For more details, refer to the application note *Quad-SPI interface on STM32 microcontrollers and microprocessors* (AN4760).

The DMA2D (Chrom-ART Accelerator controller) is useful for color space transformation (such as RGB565 to ARGB8888), or for data transfer from one memory to another.

The JPEG codec allows data compression (JPEG encoding) or decompression (JPEG decoding).

Table 2. DCMI and related resources availability

STM32 line	Max FLASH size (bytes)	On-chip SRAM (Kbytes)	QUAD SPI	Max FMC(1) SRAM and SDRAM frequency (MHz)	Max DCMI pixel clock input (MHz)(2)	JPEG codec	DMA2D	LCD_ TFT controller (3)	LCD parallel interface	MIPI DSI host(4)	Max AHB frequency (MHz)
STM32F2x7	1 M	128	No	60	48	No	No	No	No	No	120
STM32F407 STM32F417	1 M	192	No	60	54	No	No	No	No	No	168
STM32F427 STM32F437	2 M	256	No	90	54	No	Yes	No	No	No	180
STM32F429 STM32F439	2 M	256	No	90	54	No	Yes	Yes	No	No	180
STM32F446	512 K	128	Yes	90	54	No	No	No	No	No	180
STM32F469 STM32F479	2 M	384	Yes	90	54	No	Yes	Yes	No	Yes	180
STM32F7x0	64 K	320	Yes	100	54	No	Yes	Yes	No	No	216
STM32F7x5	2 M	512	Yes	100	54	No	Yes	No	No	No	216
STM32F7x6	1 M	320	Yes	100	54	No	Yes	Yes	No	No	216
STM32F7x7	2 M	512	Yes	100	54	Yes	Yes	Yes	No	No	216
STM32F7x8 STM32F7x9	2 M	512	Yes	100	54	Yes	Yes	Yes	No	Yes	216
STM32H7x3	2 M	1000	Yes	133	80	Yes	Yes	Yes	No	No	240
STM32H725 STM32H735	1 M	564	Yes	137	110	No	Yes	Yes	No	No	275
STM32H742	2 M	864	Yes	125	80	Yes	Yes	Yes	No	No	240
STM32H745 STM32H755	2 M	864	Yes	125	80	Yes	Yes	Yes	No	No	240

STM32 line	Max FLASH size (bytes)	On-chip SRAM (Kbytes)	QUAD SPI	Max FMC[1] SRAM and SDRAM frequency (MHz)	Max DCMI pixel clock input (MHz)[2]	JPEG codec	DMA2D	LCD_ TFT controller [3]	LCD parallel interface	MIPI DSI host[4]	Max AHB frequency (MHz)
STM32H747 STM32H757	2 M	864	Yes	110	80	Yes	Yes	Yes	No	No	240
STM32H730	128 K	564	No	137	110	No	Yes	Yes	No	No	275
STM32H750	128 K	864	Yes	100	80	Yes	Yes	Yes	No	No	240
STM32H7A3 STM32H7B3	2 M	1180	No	100	80	Yes	Yes	Yes	No	No	280
STM32L4x6	1 M	320	Yes	40	32	No	Yes	No	No	No	80
STM32L4R9 STM32L4S9 STM32L4R7 STM32L4S7 STM32L4R5 STM32L4S5	2 M	640	No	60	48	No	Yes	Yes	No	Yes	120
STM32L4P5 STM32L4Q5	1 M	320	No	60	48	No	Yes	Yes	No	No	120
STM32U575 STM32U585	2 M	786	Yes	80	64	No	Yes	No	Yes	No	160

1. *FSMC for STM32F2x7, STM32F407/417, STM32L4+, and STM32U575/585 devices.*
2. *Refer to the datasheet for the pixel clock frequency (DCMI_PIXCLK).*
3. *See the application note AN4861 for more details on the STM32 LTDC peripheral.*
4. *Refer to the application note AN4860 for more details on the STM32 MIPI-DSI host.*

3.2 DCMI in a smart architecture

The DCMI is connected to the AHB bus matrix through the AHB2 peripheral bus. It is accessed by the DMA to transfer the received image data. The destination of the received data depends on the application.

The smart architecture of STM32 MCUs allows the following:

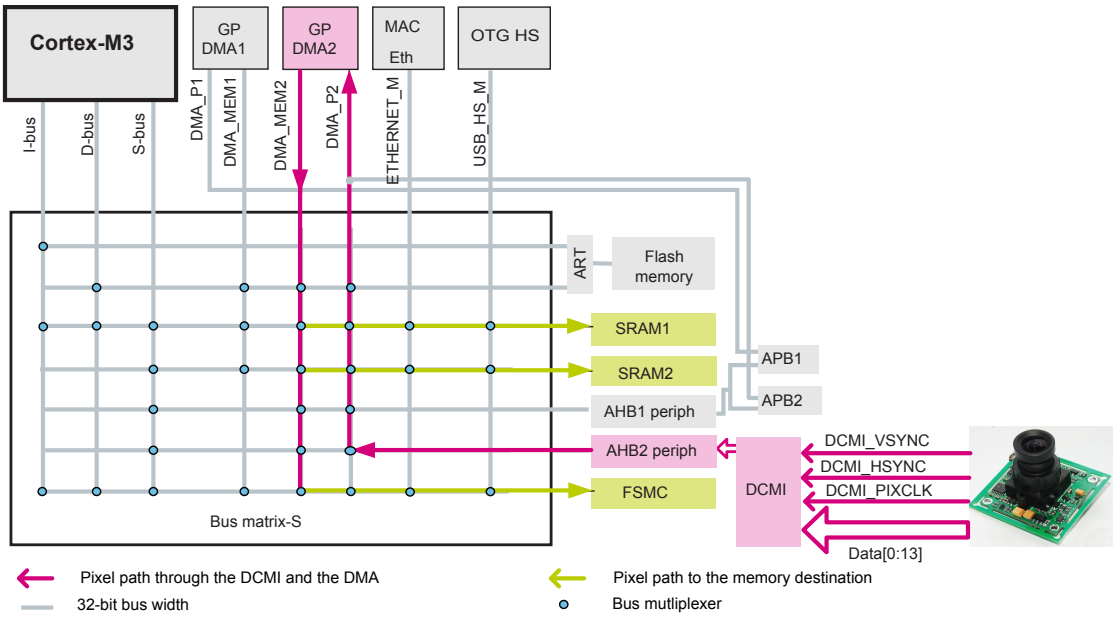
- The DMA, as an AHB master, autonomously accesses the AHB2 peripherals, and transfers the received data (image number n+1) to the memory, while the CPU processes the previously captured image (image number n).
- The DMA2D, as an AHB master, is used to transfer or modify the received data (CPU resources are kept for other tasks).
- The memories throughput and the performance are improved thanks to the multi-layer bus matrix.

3.2.1 STM32F2 system architecture

The STM32F2x7 devices are based on a 32-bit multi-layer bus matrix, used to interconnect eight masters and seven slaves. The DCMI is a slave AHB2 peripheral. The DMA2 performs the data transfer from the DCMI to internal SRAMs or external memories through the FSMC.

Figure 6 shows the DCMI interconnection and the data path in STM32F2x7 devices.

Figure 6. DCMI slave AHB2 peripheral in STM32F2x7



3.2.2 STM32F4 system architecture

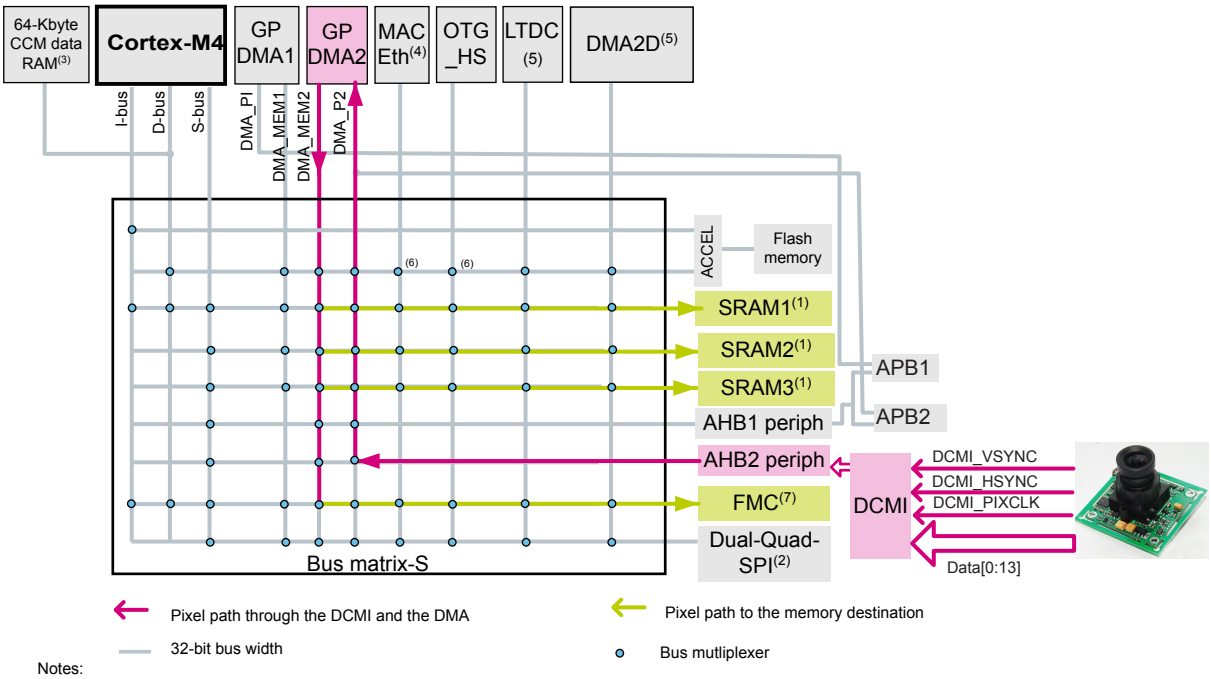
The devices of STM32F407/417, STM32F427/437, STM32F429/439, STM32F446, and STM32F469/479 lines, are based on a 32-bit multi-layer bus matrix, allowing the interconnection between:

- ten masters and eight slaves for STM32F429/439 devices
- ten masters and nine slaves for STM32F469/479 devices
- seven masters and seven slaves for STM32F446 devices
- eight masters and seven slaves for STM32F407/417 devices
- eight masters and eight slaves for STM32F427/437 devices

The DCMI is a slave AHB2 peripheral. The DMA2 performs the data transfer from the DCMI to internal SRAMs or external memories through the FMC (FSMC for STM32F407/417 line).

Figure 7 shows the DCMI interconnection and the data path in these devices.

Figure 7. DCMI slave AHB2 peripheral in STM32F4



Notes:
1. See the table below for details on the SRAMs.
2. The dual-Quad-SPI interface is available only in STM32F469/479 and STM32F446 devices.
3. The 64-Kbyte CCM data RAM is not available in STM32F446xx devices.
4. The Ethernet MAC interface is not available in STM32F446xx devices.
5. The LTDC and DMA2D are only available in STM32F429/439 and STM32F469/479 devices.
6. In STM32F407/417 devices, there is no interconnection between:
   - the Ethernet master and the DCode bus of the flash memory
   - the USB master and the DCode bus of the flash memory.

   In STM32F446 devices, there is no interconnection between the USB master and the DCODE bus of the flash memory.
7. FMSC for STM32F407/417 devices.

Table 3. SRAM availability in STM32F4 Series

STM32 line	SRAM1 (Kbytes)	SRAM2 (Kbytes)	SRAM3 (Kbytes)
STM32F407/417	112	16	N/A
STM32F427/437, STM32F429/439			64
STM32F446			N/A
STM32F469/479	160	32	128

3.2.3 STM32F7 system architecture

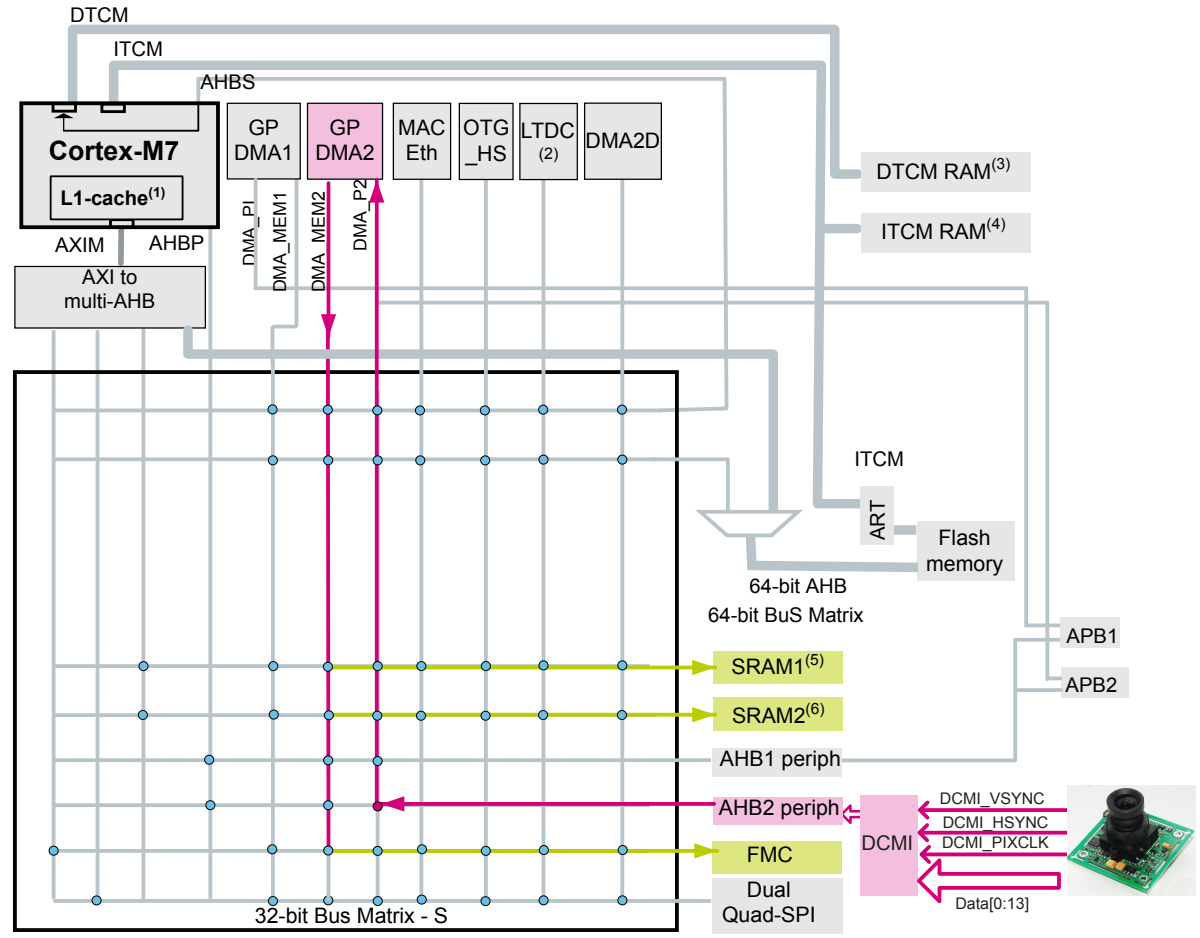
The devices of STM32F7x5, STM32F7x6, STM32F7x7, STM32F7x8, STM32F7x9 lines, and STM32F750 devices in STM32F7x0 Value line, are based on a 32-bit multilayer bus matrix, allowing the interconnection between:

- twelve masters and eight slaves for STM32F7x6, STM32F7x7, STM32F7x8, STM32F7x9, and STM32F750 devices
- eleven masters and eight slaves for STM32F7x5 devices

The DCMI is a slave AHB2 peripheral. The DMA2 performs the data transfer from the DCMI to internal SRAM or external memories through the FMC.

Figure 8 shows the DCMI interconnection and the data path in these devices.

Figure 8. DCMI slave AHB2 peripheral in STM32F7



Notes:
1. The I/D cache size is:
   - 4 Kbytes for STM32F7x5/F7x6 and STM32F750 devices.
   - 16 Kbytes for STM32F7x7/F7x8/F7x9 devices
2. The LTDC is only available in STM32F7x6/F7x7/F7x8/F7x9 and STM32F750 devices.
3. The DTCM RAM size is:
   - 64 Kbytes for STM32F7x5/F7x6 and STM32F750 devices.
   - 128 Kbytes for STM32F7x7/F7x8/F7x9 devices
4. The ITCM RAM size is 16 Kbytes for STM32F7x6/F7x7/F7x8/F7x9 and STM32F750 devices.
5. The SRAM1 size is:
   - 240 Kbytes for STM32F7x5/F7x6 and STM32F750 devices.
   - 368 Kbytes for STM32F7x7/F7x8/F7x9 devices
6. The SRAM2 size is 16 Kbytes for STM32F7x6/F7x7/F7x8/F7x9 and STM32F750 devices.

3.2.4 STM32H7 system architecture

The devices of STM32H723/733, STM32H743/753, STM32H7A3/B3, STM32H747/757, STM32H745/755, STM32H742, STM32H725/735, STM32H750, and STM32H730 lines, are based on an AXI bus matrix, two AHB bus matrices, and bus bridges allowing the interconnection between:

- 23 masters and 18 slaves for STM32H745/755 and STM32H747/757 devices
- 18 masters and 18 slaves for STM32H723/733, STM32H725/735, STM32H753, and STM32H730 devices
- 19 masters and 18 slaves for STM32H742 and STM32H43 devices
- 19 masters and 20 slaves for STM32H7A3/B3 devices
- 19 masters and 17 slaves for STM32H750 Value line devices

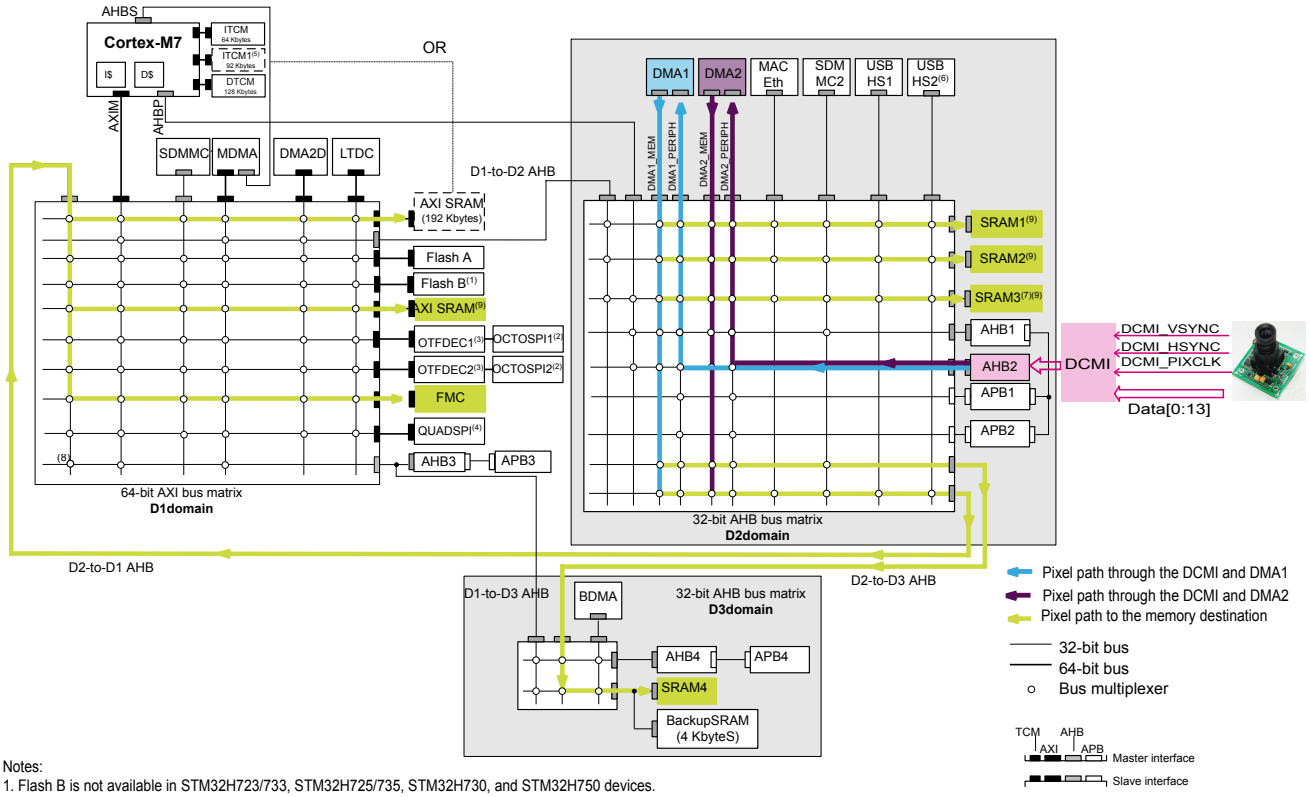
3.2.4.1 *STM32H7x3, STM32H742, STM32H725/735, STM32H730, and STM32H750 devices*

The DCMI is a slave AHB2 peripheral. The DMA1/2 perform the data transfer from the DCMI to internal SRAMs or external memories through the FMC.

The DMA1/2 are located in the D2 domain. They are able to access the slaves in the D1 and D3 domains. As a result, the DMA1/2 can transfer the data received by the DCMI (located in D2) to memories located in D1 or D3 domain.

Figure 9 shows the DCMI interconnection and the data path in these devices.

Figure 9. DCMI slave AHB2 peripheral in STM32H723/733, STM32H743/753, STM32H742, STM32H725/735, STM32H730, and STM32H750 devices



Notes:
1. Flash B is not available in STM32H723/733, STM32H725/735, STM32H730, and STM32H750 devices.
2. OCTOSPI1/2 are not available in STM32H743/753, STM32H742, and STM32H750 devices.
3. OTFDEC1/2 are only available in STM32H723/733, STM32H725/735, and STM32H730 devices.
4. The QUADSPI is only available in STM32H743/753, STM32H742, and STM32H750 devices.
5. The 192-Kbyte AXI SRAM and the 92-Kbyte ITCM are only available inSTM32H723/733, STM32H725/735, and STM32H730 devices.
6. The USBHS2 is only available in STM32H743/753, STM32H742, and STM32H750 devices.
7. The SRAM3 is only available in STM32H743/753, STM32H74,2 and STM32H750 devices.
8. There is no connection between the APB3 and the D2 domain in STM32H723/733, STM32H725/735, STM32H730, and STM32H750 devices.
9. See the table below for more details on the SRAM1/2/3 and the AXI SRAM.

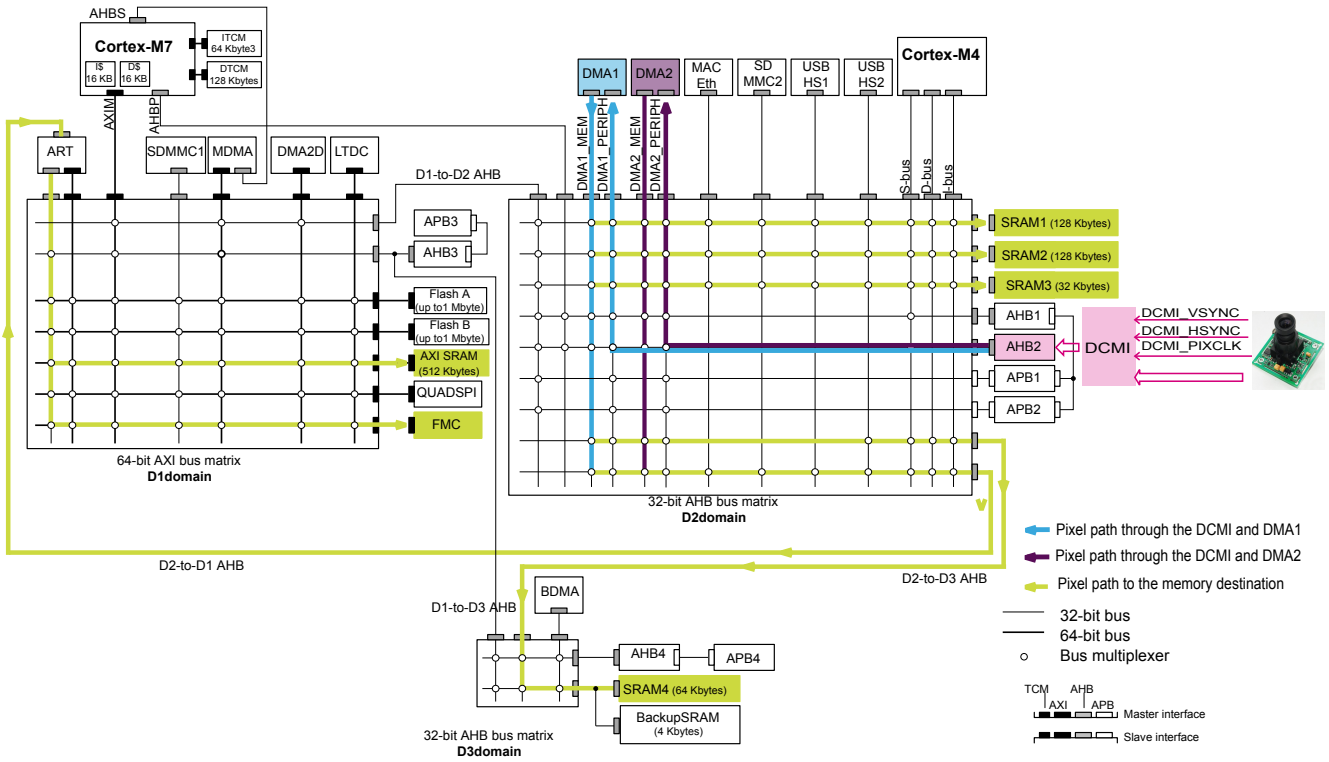
Table 4. SRAM availability in STM32H723/733, STM32H743/753, STM32H742, STM32H725/735, STM32H730, and STM32H750 devices

STM32 line	SRAM1 (Kbytes)	SRAM2 (Kbytes)	SRAM3 (Kbytes)	AXI SRAM (Kbytes)
STM32H723/733	16	16	X	128
STM32H725/735	16	16	X	128
STM32H730	16	16	X	128
STM32H743/753	128	128	32	512
STM32H742	128	128	32	512
STM32H750	128	128	32	512

3.2.4.2 *STM32H745/755 and STM32H747/757 devices*

The DMA1/2 are in the D2 domain. They are able to access slaves in the D1 and D3 domains. As a result, the DMA1/2 can transfer the data received by the DCMI (located in D2) to memories located in D1 or D3 domain. Figure 10 shows the DCMI interconnection and the data path in these devices.

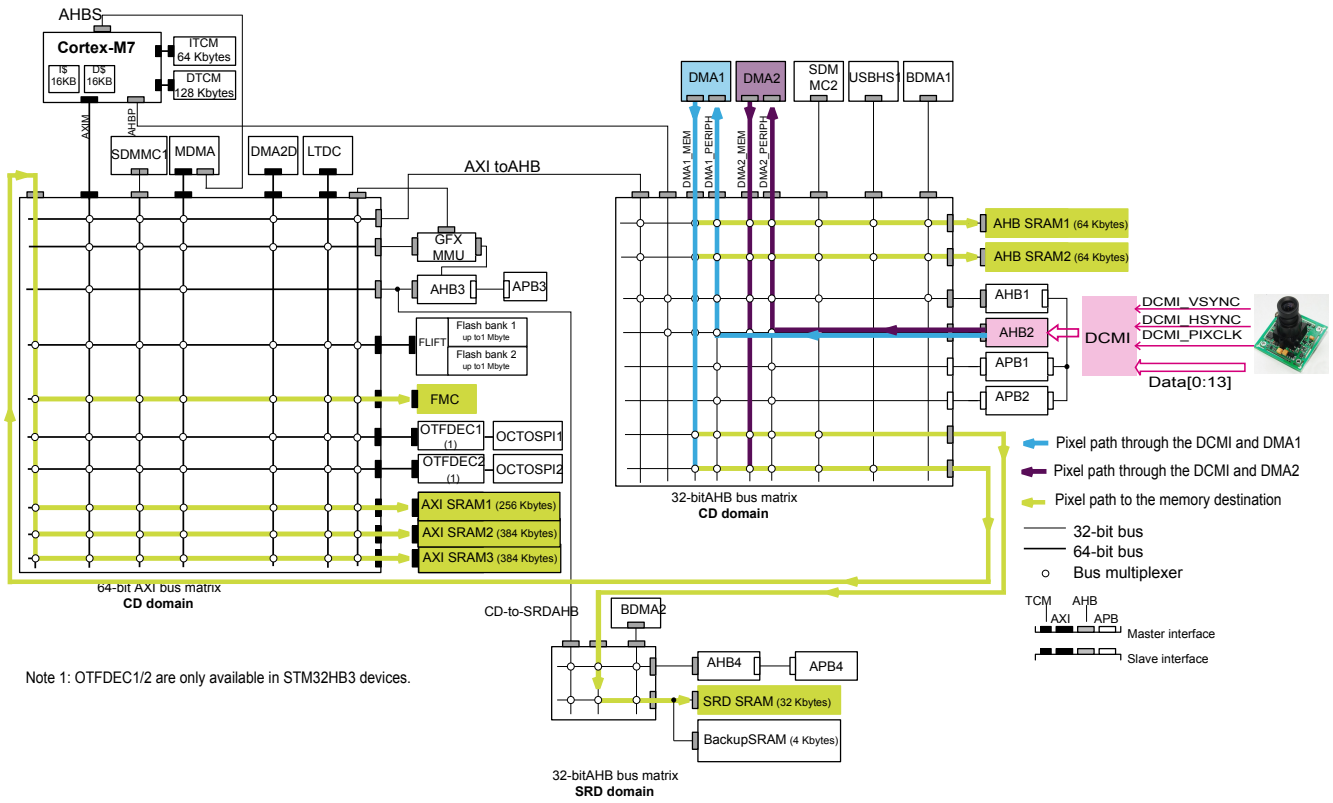
Figure 10. DCMI slave AHB2 peripheral in STM32H745/755 and STM32H747/757 devices



3.2.4.3 *STM32H7A3/7B3 devices*

The DMA1/2 are in the CD domain. They are able to access slaves in CD and SRD domains. As a result, the DMA1/2 can transfer the data received by the DCMI (located in the CD domain) to memories located in CD or SRD domain. Figure 11 shows the DCMI interconnection and the data path in these devices.

Figure 11. DCMI slave AHB2 peripheral in STM32H7A3/B3



3.2.5 STM32L4 system architecture

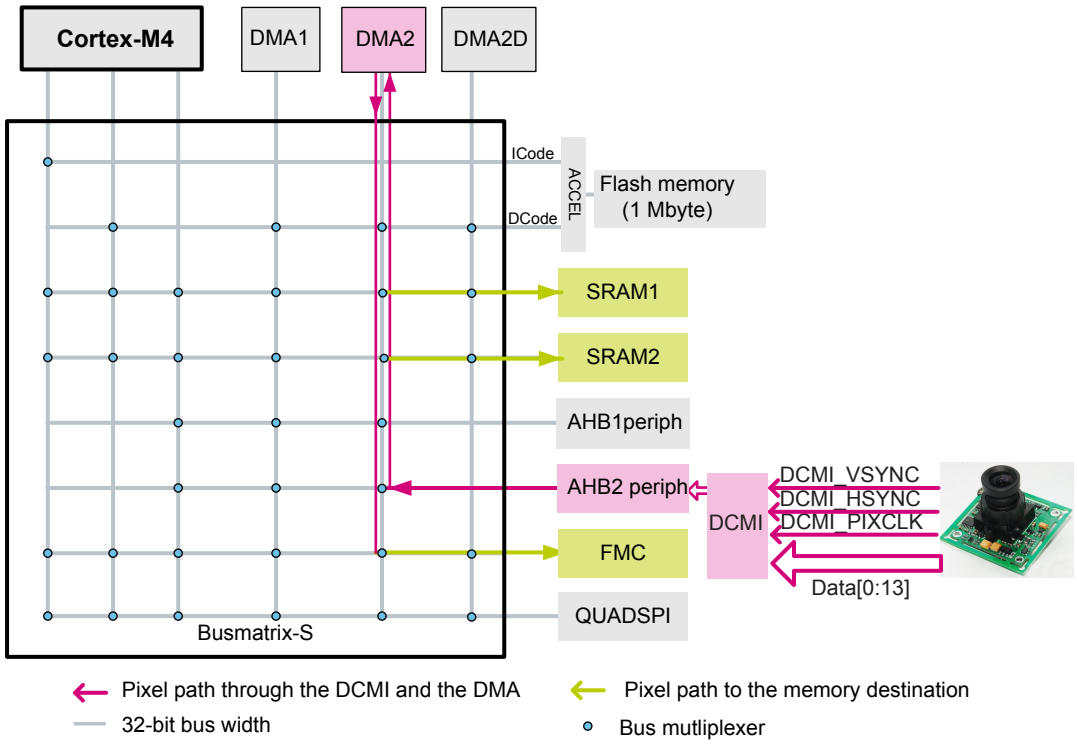
The STM32L496xx and STM32L4A6xx devices are based on a 32-bit multilayer bus matrix, allowing the interconnection between six masters and eight slaves.

The DCMI is a slave AHB2 peripheral. The DMA2 performs the data transfer from the DCMI to internal SRAMs or external memories through the FMC.

The DMA has only one port (not like STM32F2/F4/F7 and STM32H7 devices where the peripheral port is separated from the memory port), but it supports circular-buffer management, peripheral-to-memory, memory-to-peripheral, and peripheral-to-peripheral transfers.

Figure 12 shows the DCMI interconnection and the data path in these devices.

Figure 12. DCMI slave AHB2 peripheral in STM32L496/4A6



3.2.6 STM32L4+ system architecture

The devices of STM32L4R9/S9, STM32L4R7/S7, STM32L4R5/S5, and STM32L4P5/Q5 lines, are based on a 32-bit multilayer bus matrix, allowing the interconnection between:

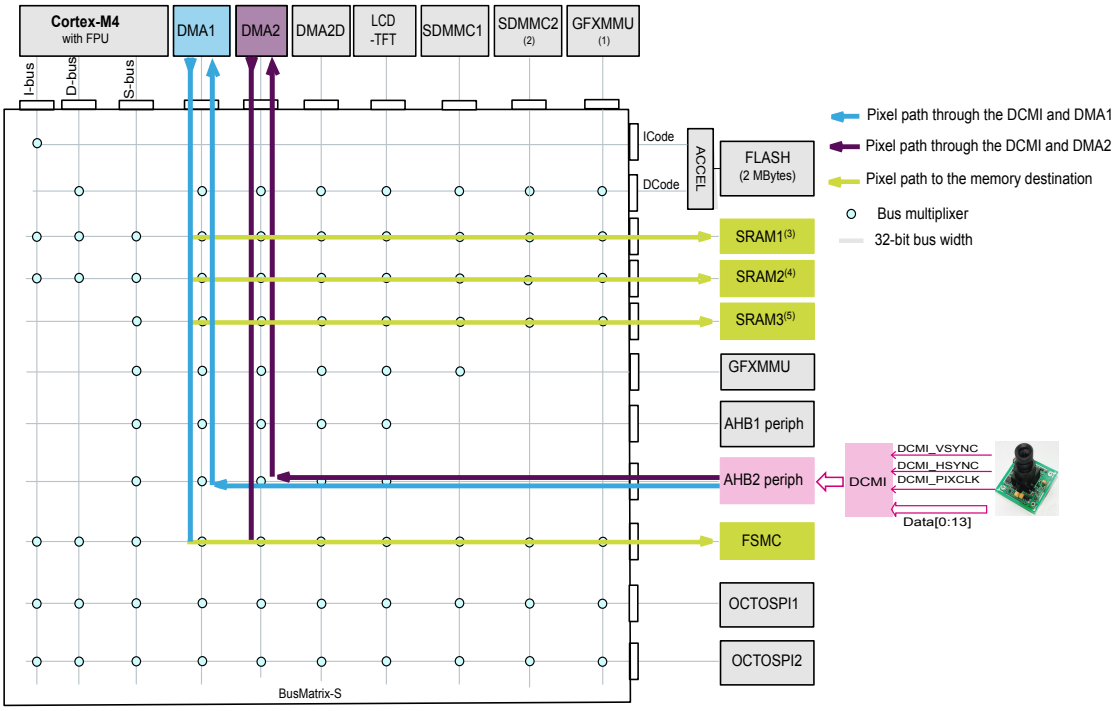
- nine masters and ten slaves for STM32L4P5/Q5 devices
- nine masters and eleven slaves for STM32L4R5/S5, STM32L4R7/S7, and STM32L4R9/S9 devices

The DCMI is a slave AHB2 peripheral. The DMA1/2 perform the data transfer from the DCMI to internal SRAMs or external memories through the FSMC.

The DMA has only one port (not like STM32F2/F4/F7 and STM32H7 devices where the peripheral port is separated from the memory port), but it supports circular-buffer management, memory-to-memory, peripheral-to-memory, memory-to-peripheral, and peripheral-to-peripheral transfers.

Figure 13 shows the DCMI interconnection and the data path in these devices.

Figure 13. DCMI slave AHB2 peripheral in STM32L4+



Notes:
1. the GFXMMU is only available in STM32L4R5/4R7/4R9/4S5/4S7/4S9 devices.
2. The SDMMC1 is only available in STM32L4P5/4Q5 devices.
3. The SRAM1 size is:
   - 128 Kbytes for STM32L4P5/4Q5 devices
   - 192 Kbytes for STM32L4R5/4R7/4R9/4S5/4S7/4S9 devices
4. The SRAM2 size is 64 Kbytes for STM32L4P5/4Q5/R5/4R7/4R9/4S5/4S7/4S9 devices.
5.The SRAM3 size is
   - 128 Kbytes for STM32L4P5/4Q5 devices
   - 384 Kbytes for STM32L4R5/4R7/4R9/4S5/4S7/4S9 devices

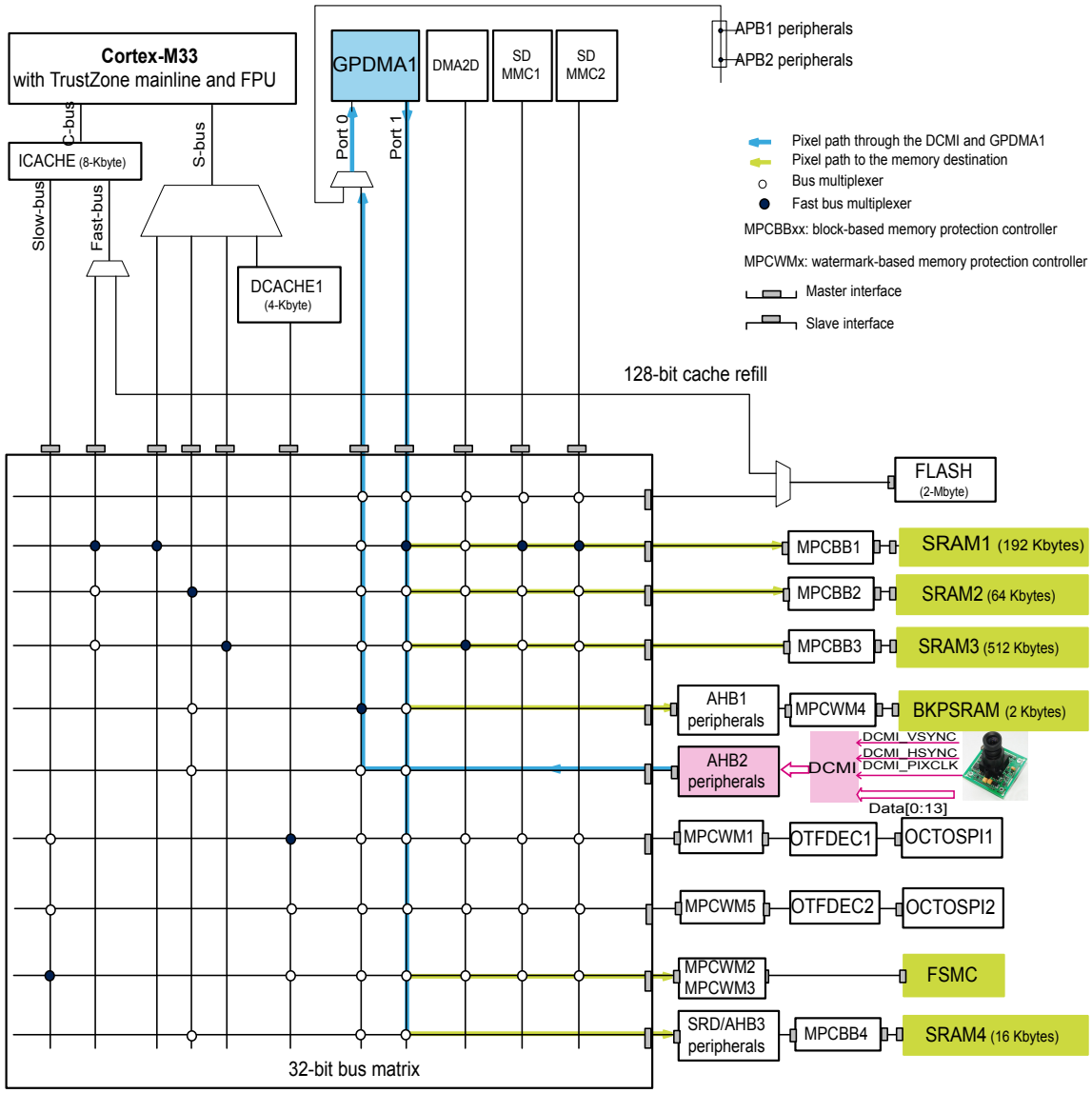
3.2.7 STM32U5 system architecture

The STM32U575/585 devices are based on a 32-bit multilayer AHB bus matrix, allowing the interconnection between elven masters and ten slaves.

The DCMI is a slave AHB2 peripheral. The GPDMA1 performs the data transfer from the DCMI to internal SRAMs or external memories through the FSMC.

Figure 14 shows the DCMI interconnection and the data path in these devices.

Figure 14. DCMI slave AHB peripheral in STM32U575/585



4 Reference boards with DCMI and/or camera modules

Many STM32 reference boards are available. Most of them embed the DCMI, and some of them have an onboard camera module. The board selection depends on the application and the hardware resources. The table below summarizes the DCMI, the camera modules, and the memories availability across various STM32 boards.

Table 5. DCMI and camera modules on STM32 boards

STM32 line	Board	Camera module	CMOS sensor	Internal SRAM (Kbytes)	External SDRAM bus width (bits)	External SRAM bus width (bits)
STM32F2x7	STM3220G-EVAL	Yes(1)	OV2640 or OV9655	132	N/A	
	STM3221G-EVAL					
STM32F407/417	STM32F4DISCOVERY	N/A(2)(3)	OV9655	196		
	STM3240G-EVAL	Yes(1)				
	STM3241G-EVAL					
STM32F429/439	32F429IDISCOVERY	N/A(3)	N/A	256	16	N/A
	STM32429I-EVAL	Yes(4)	OV2640 or OV9655		32	16
	STM32439I-EVAL					
STM32F446	STM32446E-EVAL	Yes(5)	S5k5CAGA	128	16	N/A
STM32F469/479	32F469IDISCOVERY	N/A(3)	N/A	388	32	N/A
	STM32469I-EVAL	Yes(5)	S5k5CAGA			16
	STM32479I-EVAL					
STM32F7x0	STM32F7508DISCOVERY	N/A(3)	OV9655	340	32	N/A
STM32F7x6	32F746GDISCOVERY	Yes(6)	OV9655	320	16	N/A
	STM32746G-EVAL	Yes(5)	S5k5CAGA		32	16
	STM32756G-EVAL					
STM32F7x9	32F769IDISCOVERY	N/A(3)	N/A	512	32	N/A
	STM32F769I-EVAL	Yes(5)	S5k5CAGA			16
	STM32F779I-EVAL					
STM32H7x3	STM32H743I-EVAL STM32H753I-EVAL	N/A(3)	N/A	864	32	16
STM32H747/757	STM32H747DISCOVERY	Yes(7)	OV5640 or OV9655	868	32	N/A
STM32H7A3/B3	STM32H7B3I-EVAL	Yes(8)	QSXGA	1600	32	N/A
STM32L4x6	32L496GDISCOVERY	Yes(6)	OV9655	320	N/A	
STM32L4+	32L4R9IDISCOVERY	Yes(8)	OV9655	640		
STM32U575/585	STM32U575I-EVAL	Yes(8)	QSXGA	786		

1. *Possible cameras to be connected: module CN01302H1045-C (CMOS sensor OV9655, 1.3 Mpixel) and module CN020VAH2554-C (CMOS sensor OV2640, 2 Mpixel)*
2. *N/A means not available. The application must use the desired camera module compatible with the DCMI interface.*
3. *The camera module can be connected to the DCMI through the GPIO pins.*
4. *The camera module daughterboard MB1066 is connected.*
5. *The camera module daughterboard MB1183 is connected.*
6. *The camera module can be connected to the DCMI through an FFC (flexible flat cable): the STM32F4DIS-CAM can be connected directly.*
7. *The camera module can be connected with caution before powering the Discovery board.*
8. *The camera module daughter board MB1379 is connected .*

5 DCMI description

This section details the DCMI, and its manner of dealing with the image data and the synchronization signals.

Note: The DCMI supports only the slave input mode.

5.1 Hardware interface

The DCMI consists of:

- up to 14 data lines (D13-D0)
- the pixel clock line DCMI_PIXCLK
- the DCMI_HSYNC line (horizontal synchronization)
- the DCMI_VSYNC line (vertical synchronization).

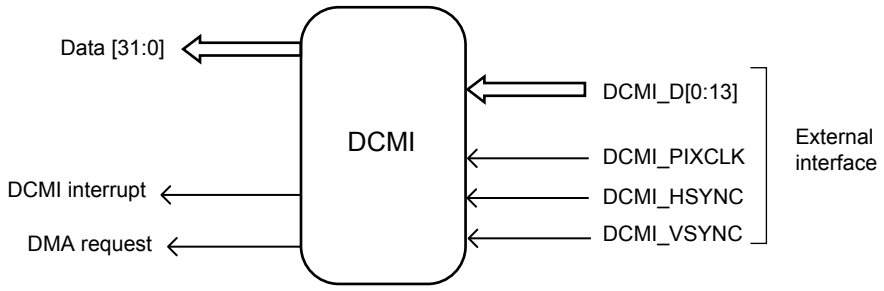
The DCMI comprises up to 17 inputs. Depending on the number of data lines enabled by the user (8, 10, 12, or 14), the number of the DCMI inputs varies (11, 13, 15, or 17 signals).

If less than 14-bit data width is used, the unused pins must not be assigned to the DCMI through GPIO alternate functions. The unused input pins can be assigned to other peripherals.

In case of embedded synchronization, the DCMI needs only nine inputs (eight data lines and DCMI_PIXCLK) to operate properly. The eight unused pins can be used for GPIO or other functions.

Figure 15 shows the DCMI signals.

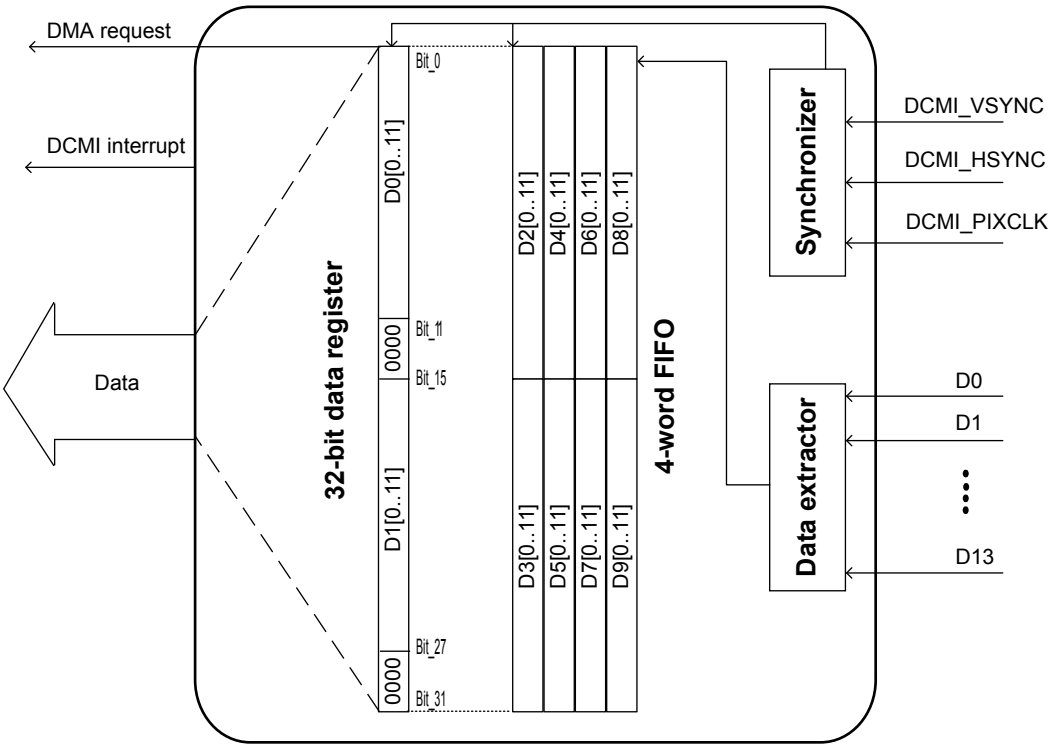
Figure 15. DCMI signals



If x-bit data width is chosen (x data lines are enabled, x = 8, 10, 12, or 14), x bits of image (or video) data are transferred each DCMI_PIXCLK cycle, and packed into a 32-bit register.

Figure 16 shows the four main DCMI components.

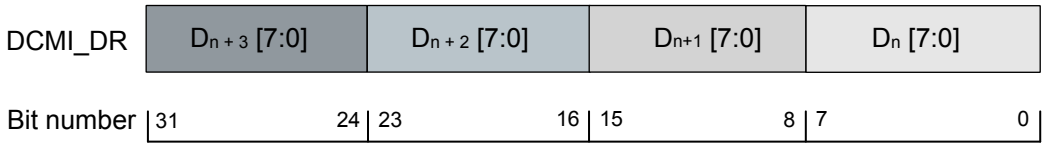
Figure 16. DCMI block diagram



- The DCMI synchronizer ensures the control of the ordered sequencing of the data flow through the DCMI. It controls the data extractor, the FIFO and the 32-bit register.
- The data extractor ensures the extraction of the data received by the DCMI.
- The 4-word FIFO is implemented to adapt the data rate transfers to the AHB. There is no overrun protection to prevent data from being overwritten if the AHB does not sustain the data transfer rate. In case of overrun or errors in the synchronization signals, the FIFO is reset, and the DCMI waits for a new start of frame.

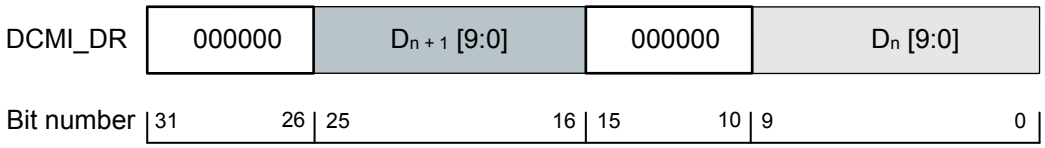
- A 32-bit data register where the data bits are packed to be transferred through a general-purpose DMA channel. The placement of the captured data in 32-bit register depends on the data width:
  - For an 8-bit data width, the DCMI captures the eight LSBs (the six other inputs D[13:8] are ignored). The first captured data byte is placed in the LSB position the 32-bit word, and the fourth captured data byte is placed in the MSB position. In this case, a 32-bit data word is made up every four pixel clock cycles. For more details, see Section 5.6 .

Figure 17. Data register filled for 8-bit data width



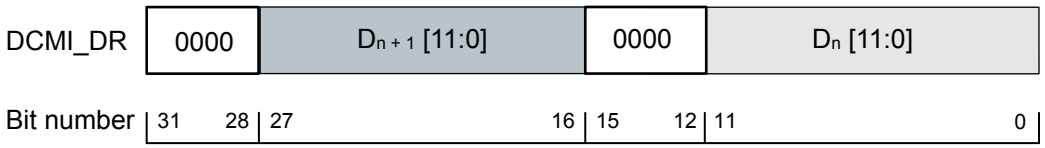
  - For a 10-bit data width, the DCMI captures the 10 LSBs (the four other inputs D[13:10] are ignored). The first 10 bits captured are placed as the 10 LSBs of a 16-bit word. The remaining MSBs in the 16-bit word of the DCMI_DR register (bits 10 to 15) are cleared. In this case, a 32-bit data word is made up every two pixel clock cycles.

Figure 18. Data register filled for 10-bit data width



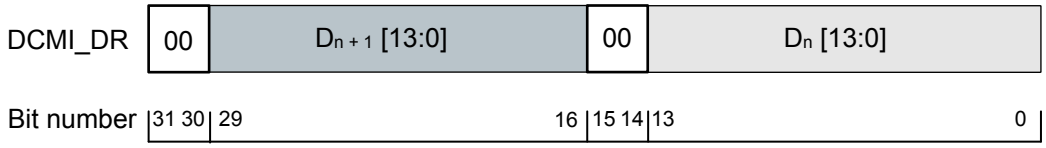
  - For a 12-bit data width, the DCMI captures the 12-bit LSBs (the two other inputs D[13:12] are ignored). The first 12 bits captured are placed as the 12 LSBs of a 16-bit word. The remaining MSBs in the 16-bit word of the DCMI_DR register (bits 12 to 15) are cleared. In this case, a 32-bit data word is made up every two pixel clock cycles.

Figure 19. Data register filled for 12-bit data width



  - For a 14-bit data width, the DCMI captures all the received bits. The first 14 bits captured are placed as the 14 LSBs of a 16-bit word. The remaining MSBs in the 16-bit word of the DCMI_DR register (bits 14 and 15) are cleared. In this case, a 32-bit data word is made up every two pixel clock cycles.

Figure 20. Data register filled for 14-bit data width

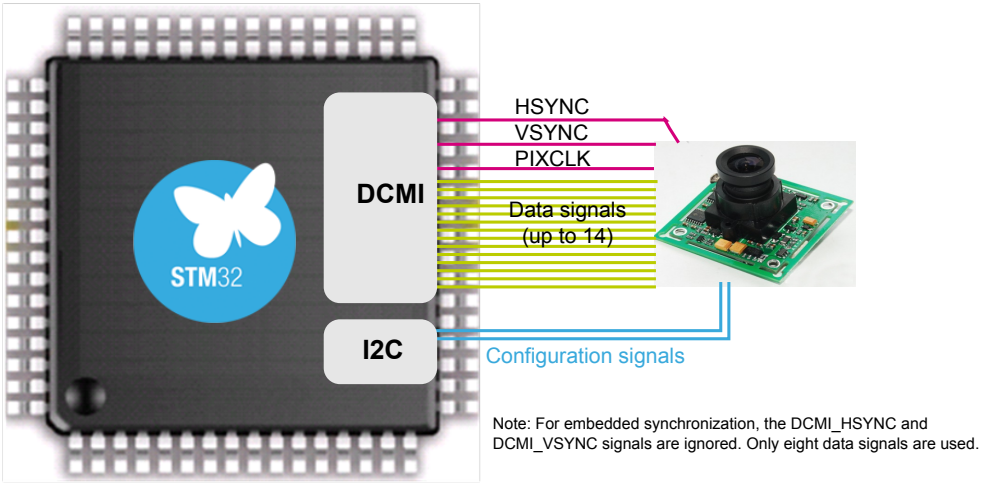


5.2 Camera module and DCMI interconnection

As mentioned in Section 2.2.2 , the camera module is connected to the DCMI through the following signal types:

- DCMI clock and data signals
- I2C configuration signals

Figure 21. STM32 MCU and camera module interconnection



5.3 DCMI functional description

The following steps summarize the internal DCMI component operation, and give an example of data flow through the system bus matrix:

1. After receiving the different signals, the synchronizer controls the data flow through the different DCMI components (data extractor, FIFO, and 32-bit data register).
2. Being extracted by the extractor, data are packed in the 4-word FIFO, then ordered in the 32-bit register.
3. Once the 32-bit data block is packed in the register, a DMA request is generated.
4. The DMA transfers the data to the corresponding memory destination.
5. Depending on the application, data stored in the memory can be processed differently.

Note: It is assumed that all image preprocessing is performed in the camera module.

5.4 Data synchronization

The camera interface has a configurable parallel data interface from 8 to 14 data lines, together with:

- a pixel clock line, DCMI_PIXCLK (rising/falling edge configuration)
- an horizontal synchronization line, DCMI_HSYNC
- a vertical synchronization line, DCMI_VSYNC, with a programmable polarity

The DCMI_PIXCLK and AHB clocks must respect the minimum ratio AHB / DCMI_PIXCLK of 2.5.

Some camera modules support the two types of synchronization, while others support either the hardware or the embedded synchronization.

5.4.1 Hardware (or external) synchronization

In this mode, the DCMI_VSYNC and DCMI_HSYNC signals are used for the synchronization:

- The line synchronization is always referred to as DCMI_HSYNC (also known as LINE VALID).
- The frame synchronization is always referred to as DCMI_VSYNC (also known as FRAME VALID).

The polarities of the DCMI_PIXCLK and the synchronization signals (DCMI_HSYNC and DCMI_VSYNC) are programmable.

Data are synchronized with DCMI_PIXCLK, and change on the rising or falling edge of the pixel clock, depending on the configured polarity.

If the DCMI_VSYNC and DCMI_HSYNC signals are programmed active level (active high or active low), the data is not valid in the parallel interface when VSYNC or HSYNC is at that level (high or low).

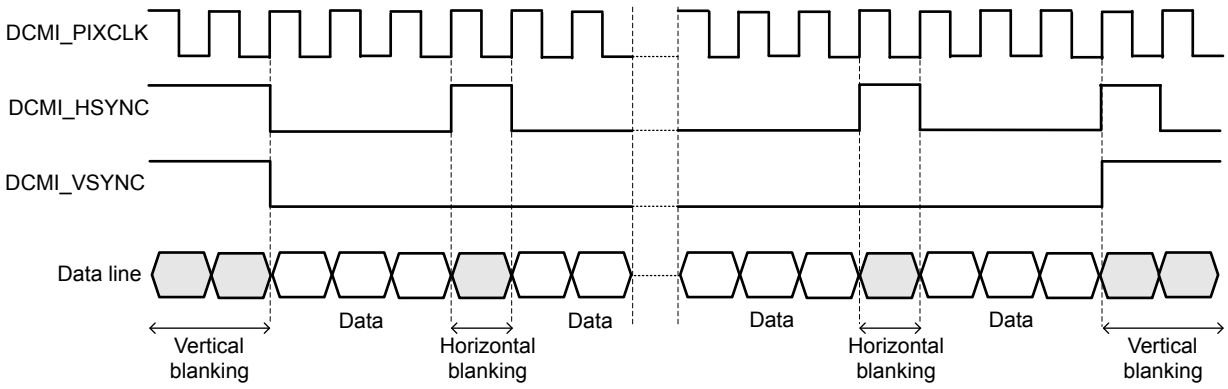
For example, if VSYNC is programmed active high:

- When VSYNC is low, the data is valid.
- When VSYNC is high, the data is not valid (vertical blanking).

The DCMI_HSYNC and DCMI_VSYNC signals act like blanking signals, since all data received during DCMI_HSYNC/DCMI_VSYNC active periods are ignored.

Figure 22 shows an example of data transfer when DCMI_VSYNC and DCMI_HSYNC are active high, and when the capture edge for DCMI_PIXCLK is the rising edge.

Figure 22. Frame structure in hardware synchronization mode



Compressed data synchronization

For compressed data (JPEG), the DCMI supports only the hardware synchronization. Each JPEG stream is divided into packets, which have programmable size. The packets dispatching depends on the image content, and results in a variable blanking duration between two packets.

DCMI_HSYNC is used to signal the start/end of a packet. DCMI_VSYNC is used to signal the start/end of the stream.

If the full data stream finishes and the detection of an end-of-stream does not occur (DCMI_VSYNC does not change), the DCMI pads out the end-of-frame by inserting zeros.

5.4.2 Embedded (or internal) synchronization

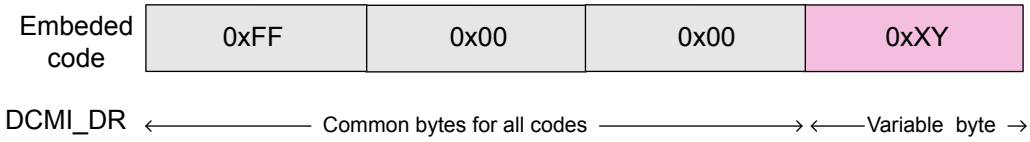
In this case, delimiter codes are used for synchronization. These codes are embedded within the data flow to indicate the start/end of line or the start/end of frame.

Note: These codes are supported only for 8-bit parallel data interface width. For other data widths, this mode generates unpredictable results, and must not be used.

The codes eliminate the need for DCMI_HSYNC and DCMI_VSYNC to signal the end/start of the line or the frame. When this synchronization mode is used, there are two values that must not be used for data: 0 and 255 (0x00 and 0xFF). These two values are reserved for data identification purposes. It is up to the camera module to control the data values. Image data can then have only 254 possible values (0x00 < image data value < 0xFF).

Each synchronization code consists of 4-byte sequence 0xFF 00 00 XY (as shown in Figure 23), where all delimiter codes have the same first 3-byte sequence 0xFF 00 00. Only the final one 0xXY is programmed to indicate the corresponding event.

Figure 23. Embedded code bytes



5.4.2.1 *Mode 1*

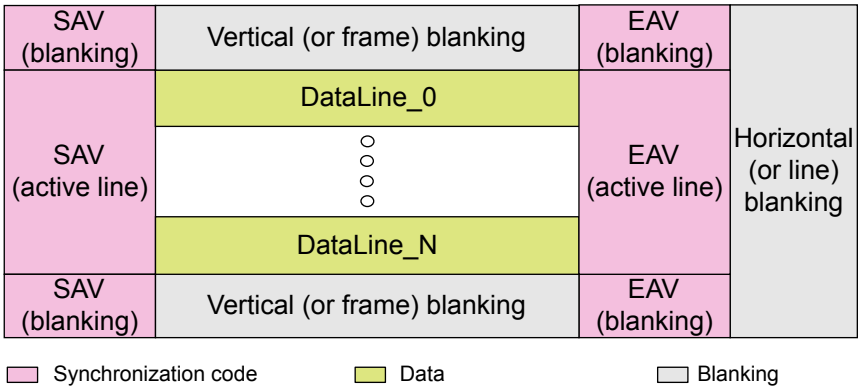
This mode is ITU656 compatible (ITU656 is the digital video protocol ITU-R BT.656).

The following reference codes indicate a set of four events:

- SAV (active line): line-start
- EAV (active line): line-end
- SAV (blanking): line-start during inter-frame blanking period
- EAV (blanking): line-end during inter-frame blanking period

Figure 24 illustrates the frame structure using this mode.

Figure 24. Frame structure in embedded synchronization mode 1



5.4.2.2 *Mode 2*

The embedded synchronization codes signal another set of events:

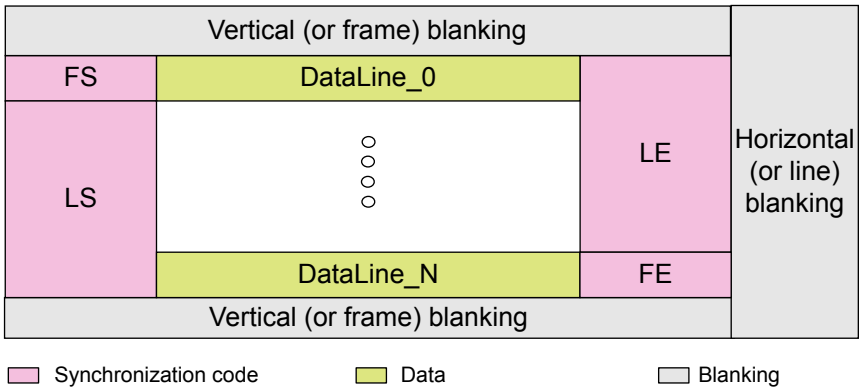
- frame-start (FS)
- frame-end (FE)
- line-start (LS)
- line-end (LE)

A 0xFF value programmed as a frame-end (FE) means that all the unused codes (the possible values of codes other than FS, LS, LE) are interpreted as valid FE codes.

In this mode, once the camera interface has been enabled, the frame capture starts after the first occurrence of an FE code followed by an FS code.

Figure 25 illustrates the frame structure when using this mode.

Figure 25. Frame structure in embedded synchronization mode 2



Note: The camera modules can have up to eight synchronization codes in interleaved mode. This mode is then not supported by the camera interface (otherwise, every other half frame is discarded). When using the embedded synchronization mode, the DCMI does not support the compressed data (JPEG) and the crop feature.

5.4.2.3 *Embedded unmask codes*

These codes are also used to signal start/end of a line or a frame. Thanks to these codes, instead of comparing all the received code with the programmed one to set the corresponding event, the user can select only some unmasked bits to compare with the bits of the programmed code having the same position.

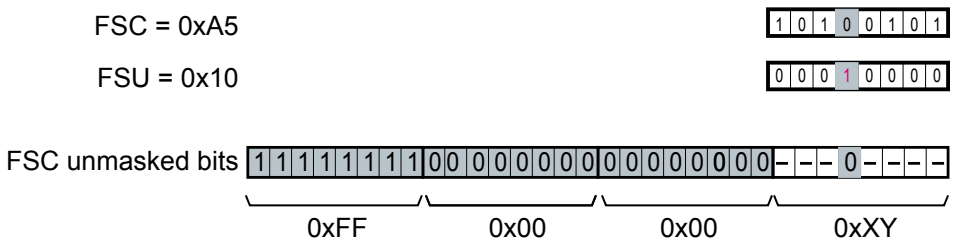
The user applies a mask to the corresponding code by configuring the DCMI embedded synchronization unmask register (DCMI_ESUR). Each byte in this register is an unmask code, corresponding to an embedded synchronization code:

- The most significant byte is the frame-end delimiter unmask (FEU): each bit set to 1 implies that this bit, in the frame-end-code, must be compared with the received data to know if it is a frame-end event or not.
- The second byte is the line-end delimiter unmask (LEU): each bit set to 1 implies that this bit, in the line-end-code, must be compared with the received data to know if it is a line-end event or not.
- The third byte is the line-start delimiter unmask (LSU): each bit set to 1 implies that this bit, in the line-start-code, must be compared with the received data to know if it is a line-start event or not.
- The less significant byte is the frame-start delimiter unmask (FSU): each bit set to 1 implies that this bit, in the frame-start-code, must be compared with the received data to know if it is a frame-start event or not.

There can be different codes for each event (line-start, line-end, frame-start, or frame-end) but all of them (the different codes corresponding to one event) have the unmasked bits in the same position (same unmask code).

Example: FSC = 0xA5 and unmask code FSU = 0x10 (as shown in Figure 26). In this case the frame-start information is embedded in the bit number 4 of the FS code. The user must compare only the bit number 4 of the received code with the bit number 4 of the programmed code, to know if it is a frame-start event or not.

Figure 26. Embedded code unmasking



Note: Make sure that each synchronization code has different unmask code to avoid synchronization errors.

5.5 Capture modes

The DCMI supports two types of capture: snapshot (a single frame) and continuous grab (a sequence of frames).

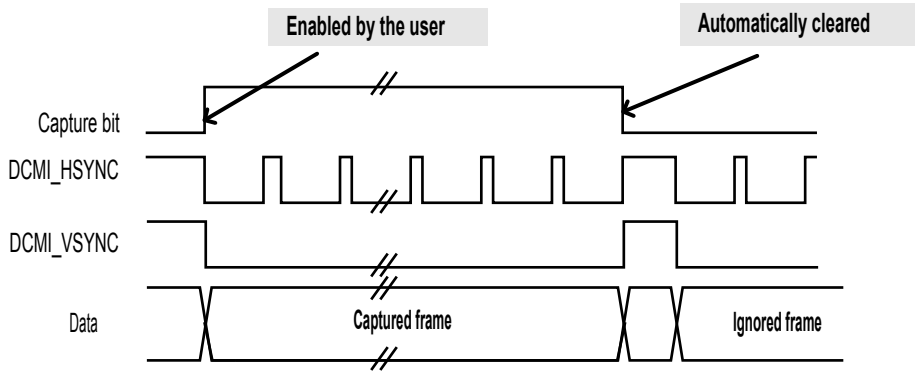
The user can control the capture rate by selecting the bytes, lines, and frames to capture in the DCMI_CR register. These features are used to convert the color format of the image, and/or to reduce the image resolution (by capturing one line out of two, the vertical resolution is divided by 2). For more details, refer to Section 5.8 .

5.5.1 Snapshot mode

In this mode, a single frame is captured. After the capture is enabled by setting the CAPTURE bit in DCMI_CR, the interface waits for the detection of a start of frame (the next DCMI_VSYNC or the next embedded frame-start code, depending on the synchronization mode) before sampling the data.

Once the first complete frame is received, the DCMI is automatically disabled (CAPTURE bit automatically cleared), and all the other frames are ignored. In case of an overrun, the frame is lost and the camera interface is disabled.

Figure 27. Frame reception in snapshot mode

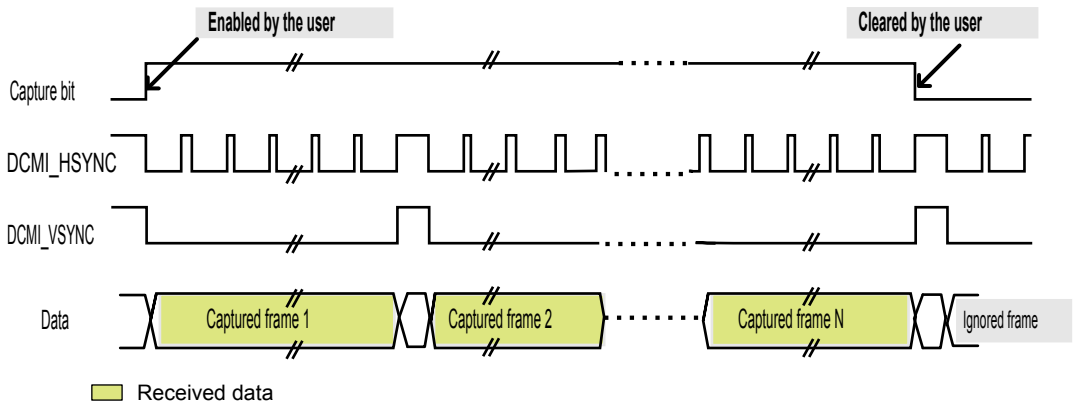


5.5.2 Continuous grab mode

Once this mode is selected and the capture is enabled (CAPTURE = 1), the interface waits for the detection of a start of frame (the next DCMI_VSYNC or the next embedded frame-start code, depending on the synchronization mode) before sampling the data.

In this mode, the DCMI can be configured to capture all the frames, every alternate frame (50% bandwidth reduction), or one frame out of four (75% bandwidth reduction). The camera interface is not automatically disabled but the user must disable it by setting CAPTURE = 0 . After being disabled by the user, the DCMI continues to grab data until the end of the current frame.

Figure 28. Frame reception in continuous grab mode



5.6 Data formats and storage

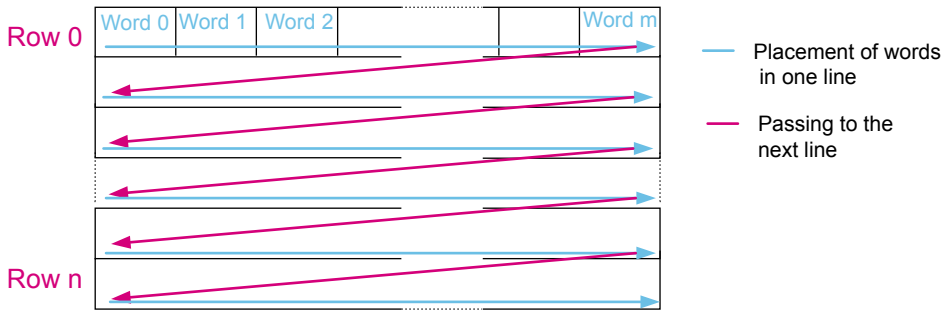
The DCMI supports the following data formats:

- 8-bit progressive video: either monochrome or raw Bayer
- YCbCr 4:2:2 progressive video
- RGB565 progressive video
- compressed data (JPEG)

For monochrome, RGB or YCbCr data, the maximum input size is 2048 * 2048 pixels, and the frame buffer is stored in raster mode. There is no size limitation for JPEG compressed data.

For monochrome, RGB and YCbCr, the frame buffer is stored in raster mode as shown in Figure 29.

Figure 29. Pixel raster scan order



Note: Only 32-bit words are used, and only the little-endian format is supported (the least significant byte is stored in the smallest address).

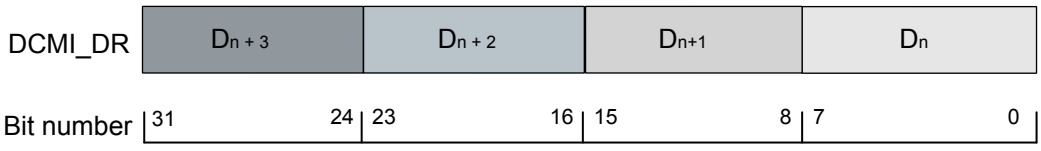
Data received from the camera can be organized in lines, frames (raw YUV/RGB/Bayer modes), or can be a sequence of JPEG images.

The number of bytes in a line may not be a multiple of four. The user must therefore be careful when handling this case since a DMA request is generated each time a complete 32-bit word has been constructed from the captured data. When an end of frame is detected and the 32-bit word to be transferred has not been completely received, the remaining data are padded with zeros, and a DMA request is generated.

5.6.1 Monochrome

The DCMI supports the monochrome format 8 bpp. In the case an 8-bit data width is selected when configuring the DCMI, the data register has the structure shown in Figure 30.

Figure 30. DCMI data register filled with monochrome data



5.6.2 RGB565

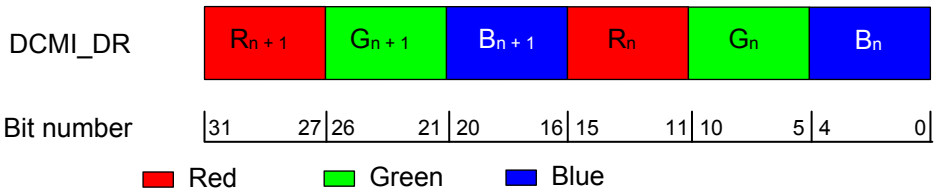
RGB refers to Red, Green, and Blue, which represent the three hues of light. Any color is obtained by mixing these three colors.

565 is used to indicate that each pixel consists of 16 bits divided as follows:

- 5 bits for encoding the red value (the most significant 5 bits)
- 6 bits for encoding the green value
- 5 bits for encoding the blue value (the less significant 5 bits)

Each component has the same spatial resolution (4:4:4 format): each sample has a red (R), a green (G) and a blue (B) component. Figure 31 shows the DCMI data register containing RGB data, when an 8-bit data width is selected.

Figure 31. DCMI data register filled with RGB data



5.6.3 YCbCr

YCbCr is a family of color spaces that separates the luminance or luma (brightness) from the chrominance or chroma (color differences).

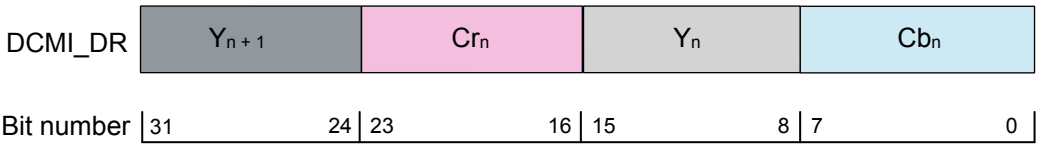
YCbCr consists of three components:

- Y refers to the luminance or luma (black and white).
- Cb refers to the blue difference chroma.
- Cr refers to the red difference chroma.

YCbCr 4:2:2 is a subsampling scheme, which requires a half resolution in horizontal direction: for every two horizontal Y samples, there is one Cb or Cr sample.

Each component (Y, Cb, and Cr) is encoded in 8 bits. Figure 32 shows the DCMI data register containing YCbCr data when an 8-bit data width is selected.

Figure 32. DCMI data register filled with YCbCr data

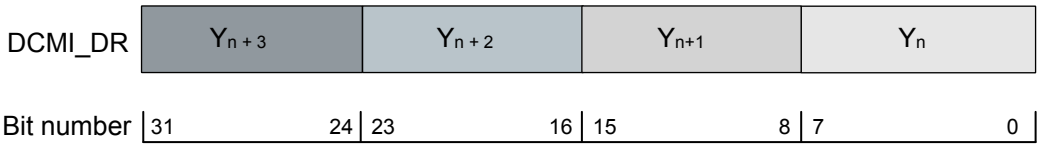


5.6.4 YCbCr, Y only

Note: This data format is only available for STM32F446, STM32F469/479, STM32F7, STM32H7, STM32L496xx, STM32L4A6, STM32L4+, and STM32U5 devices listed in Table 1.

The buffer contains only the Y information, monochrome image. The chroma information is dropped. Only luma component of each pixel, encoded in 8 bits, is stored. The result is a monochrome image having the half-horizontal resolution of the original image (YCbCr data). Figure 33 shows the DCMI register when an 8-bit data width is selected.

Figure 33. DCMI data register filled with Y only data



5.6.5 JPEG

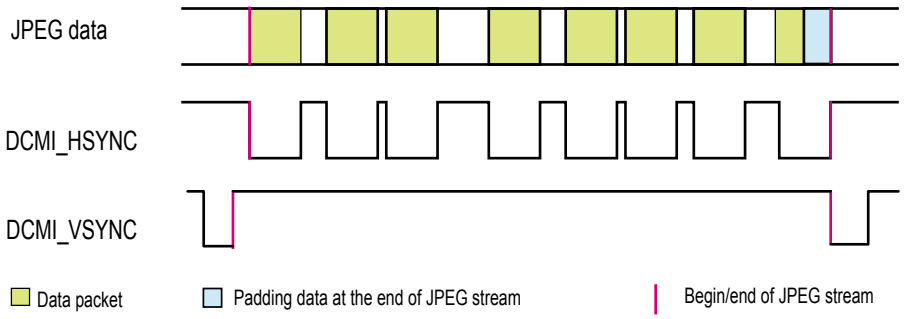
For compressed data (JPEG), the DCMI supports only the hardware synchronization, and the input size is not limited. Each JPEG stream is divided into packets, which have programmable size. The packet dispatching depends on the image content, and results in a variable blanking duration between two packets.

To allow JPEG image reception, the JPEG bit must be set to one in the DCMI_CR register. The JPEG images are not stored as lines and frames. The DCMI_VSYNC signal is used to start the capture while DCMI_HSYNC serves as a data enable signal.

If the full data stream finishes and the detection of an end of stream does not occur (DCMI_VSYNC does not change), the DCMI pads out the end of the frame by inserting zeros: if the stream size is not a multiple of four, at the end of the stream, the DCMI pads the remaining data with zeros.

Note: The crop feature and embedded synchronization mode cannot be used in the JPEG format.

Figure 34. JPEG data reception



5.7 Crop feature

With the crop feature, the camera interface selects a rectangular window from the received image. The start coordinates (upper-left corner) are specified in the 32-bit DCMI_CWSTRT register.

The window size is specified in number of pixel clocks (horizontal dimension), and in number of lines (vertical dimension) in the DCMI_CWSIZE register.

5.8 Image resizing (resolution modification)

Note: This feature is only available for STM32F446, STM32F469/479, STM32F7x5/6/7/8/9, STM32F750, STM32H7, STM32L496xx, STM32L4A6, STM32L4+, and STM32U575/585 devices listed in Table 1.

As described in Section 5.5 , the DCMI capture features are set through the DCMI_CR register.

The DCMI captures all received lines, or one line out of two (the user can choose to capture the odd or even lines).

This feature affects the vertical resolution that can be received by the DCMI as sent from the camera module or divided by two (only the odd or the even lines are received).

This interface allows also the capture of:

- all received data
- every other byte from the received data (one byte out of two, only the odd or the even bytes are received)
- one byte out of four
- two bytes out of four

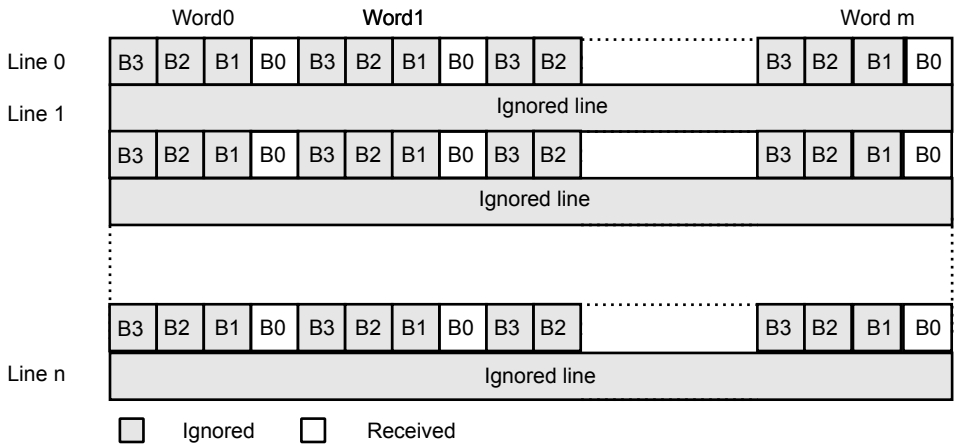
This feature affects the horizontal resolution allowing the user to select one of the following resolutions:

- the full horizontal resolution
- the half of the horizontal resolution
- the quarter of the horizontal resolution (available only for 8 bpp data formats)

Caution: For some data formats (color spaces), the modification of the horizontal resolution allows a change of the data format. For example, when the data format is YCbCr, the data is received interleaved (CbYCrYCbYCr). When the user chooses to receive every other byte, the DCMI receives only the Y component of each sample, means converting YCbCr data into Y-only data. This conversion affects both the horizontal resolution (only half of the image is received), and the data format.

Figure 35 shows one frame when receiving only one byte out of four and one line out of two.

Figure 35. Frame resolution modification



5.9 DCMI interrupts

The following interrupts can be generated:

- IT_LINE indicates the end of line.
- IT_FRAME indicates the end of frame capture.
- IT_OVR indicates the overrun of data reception.
- IT_VSYNC indicates the synchronization frame.
- IT_ERR indicates the detection of an error in the embedded synchronization code order (only in embedded synchronization mode).

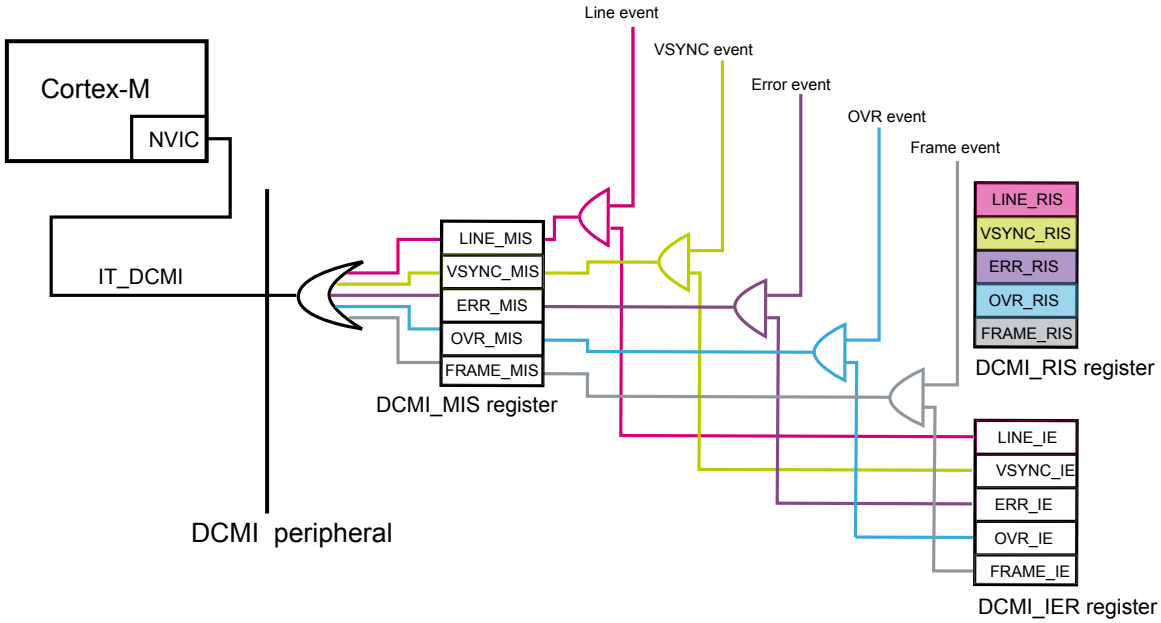
All interrupts can be masked by software. The global interrupt dcmi_it is the logic OR of all the individual interrupts.

The DCMI interrupts are handled through the following registers:

- DCMI_IER: read/write register allowing the interrupts to be generated when the corresponding event occurs
- DCMI_RIS: read-only register giving the current status of the corresponding interrupt, before masking this interrupt with DCMI_IER (each bit gives the status of the interrupt that can be enabled or disabled in DCMI_IER).
- DCMI_MIS: read-only register providing the current masked status of the corresponding interrupt, depending on DCMI_IER and DCMI_RIS.

If an event occurs and the corresponding interrupt is enabled, the DCMI global interrupt is generated.

Figure 36. **DCMI interrupts and registers**



5.10 Low-power modes

The STM32 power mode has a direct effect on the DCMI, which operates as follows over the different power modes:

- In Run mode, the DCMI and all peripherals operate normally.
- In Sleep mode, the DCMI and all peripherals work normally, and generate interrupts to wake up the CPU.
- In Stop and Standby modes, the DCMI does not work.

For some STM32 devices, there are other low-power modes where the state of the DCMI varies from one to the other:

- Low-power Run mode
- Low-power Sleep mode: interrupts from peripherals cause the device to exit this mode.
- Stop 0, Stop 1, Stop 2, Stop 3 modes: the content of peripheral registers is kept.
- Shutdown mode: the peripheral must be reinitialized when exiting Shutdown mode.

The table below summarizes the DCMI operation in the different modes.

Table 6. DCMI operation in low-power modes

Mode	DCMI operation
Run	Active
Low-power Run(1)	
Sleep	
Low-power Sleep(1)	
Stop	Frozen
Stop 0(2)	
Stop 1(2)	
Stop 2(2)	
Stop 3(3)	
Standby	Powered down
Shutdown(2)	

1. *Only for STM32L496xx, STM32L4A6xx, and STM32L4+ devices.*
2. *Only for STM32L496xx, STM32L4A6xx, STM32L4+, and STM32U575/585 devices.*
3. *Only on STM32U575/585 devices.*

6 DCMI configuration

When selecting a camera module to interface with STM32 MCUs, the user must consider some parameters such as the pixel clock, the supported data format, and the resolutions.

To correctly implement the application, the user needs to perform the following configurations:

- Configure the GPIOs.
- Configure the timings and the clocks.
- Configure the DCMI peripheral.
- Configure the DMA.
- Configure the camera module:
  - Configure the I2C to allow the camera module configuration and control.
  - Set parameters such as contrast, brightness, color effect, polarities, and data format.

Note: It is recommended to reset the DCMI and the camera module before starting the configuration. The DCMI can be reset by setting the corresponding bit in the RCC_AHB2RSTR register, which resets the clock domains.

6.1 GPIO configuration

To easily configure the DCMI GPIOs (such as data pins, control signals pins, camera configuration pins), and to avoid any pin conflicts, it is recommended to use the STM32CubeMX configuration and initialization code generator.

Thanks to the STM32CubeMX, the user generates a project with all the needed peripherals preconfigured.

Depending on the extended data mode chosen by configuring EDM bits in DCMI_CR register, the DCMI receives a 8-, 10-, 12-, or 14-bpp clock (DCMI_PIXCLK).

The user needs to configure:

- 11, 13, 15, or 17 GPIOs for the DCMI for the hardware synchronization
- only nine GPIOs (eight pins for data and one pin for DCMI_PIXCLK) for the embedded synchronization

The user needs to configure also the I2C, and in some cases the camera power supply pin (if the camera power supply source is the STM32 MCU).

Enable interrupts

To be able to use the DCMI interrupts, the user must enable the DCMI global interrupts on the NVIC side. Each interrupt is then enabled separately by enabling its corresponding enable bit in the DCMI_IER register:

- Only four interrupts (IT_LINE, IT_FRAME, IT_OVR, and IT_DCMI_VSYNC) can be used in hardware synchronization mode.
- The five interrupts can be used in embedded synchronization mode.

The software allows the user to check whether the specified DCMI interrupt has occurred or not, by checking the state of the flags.

6.2 Clock and timing configuration

6.2.1 System clock configuration (HCLK)

It is recommended to use the highest system clock to get the best performances. This recommendation applies also for the frame buffer of the external memory: if an external memory is used for the frame buffer, the clock must be set at the highest allowed speed to get the best memory bandwidth.

Examples:

- STM32F4x9xx devices: the maximum system speed is 180 MHz. If an external SDRAM is connected to the FMC, the maximum SDRAM clock is 90 MHz (HCLK/2).
- STM32F7 devices: the maximum system speed is 216 MHz. With this speed and HCLK/2 prescaler, the SDRAM speed exceeds the maximum allowed speed (see datasheets for more details). To get the maximum SDRAM, it is recommended to configure HCLK @ 200 MHz, then the SDRAM speed is set at 100 MHz.

The clock configurations providing the highest performances are the following:

- for STM32F2x7 devices, HCLK @ 120 MHz and SRAM @ 60 MHz
- for STM32F407/417 devices, HCLK @ 168 MHz and SRAM @ 60 MHz
- for STM32L4x6 devices, HCLK @ 80 MHz and SRAM @ 40 MHz

6.2.2 DCMI clock and timing configuration (DCMI_PIXCLK)

The DCMI pixel clock configuration depends on the configuration of the pixel clock of the camera module. The user must make sure that the pixel clock has the same configuration on the DCMI and the camera module sides.

DCMI_PIXCLK is an input signal for the DCMI used for input data sampling. The user selects either the rising or the falling edge for capturing data by configuring the PCKPOL bit in the DCMI_CR register.

As explained in Section 5.4 , there are two types of synchronization: embedded and hardware. To select the desired synchronization mode for the application, the user needs to configure the ESS bit in DCMI_CR.

6.2.2.1 DCMI clock configuration in hardware synchronization

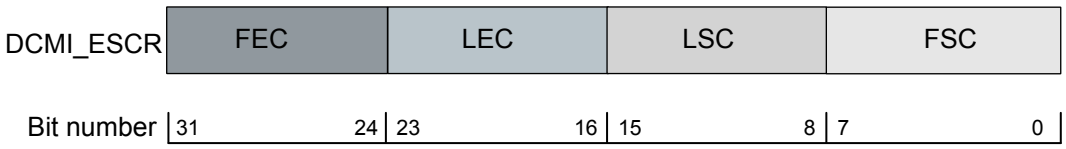
The DCMI_HSYNC and DCMI_VSYNC signals are used. The configuration of these two signals is defined by selecting each signal active level (high or low) for the VSPOL and HSPOL bits in DCMI_CR.

Note: The user must make sure that DCMI_HSYNC and DCMI_VSYNC polarities are programmed according to the camera module configuration. In the hardware synchronization mode (ESS = 0 in DCMI_CR), the IT_VSYNC interrupt is generated (if enabled), even when CAPTURE = 0 in DCMI_CR. To reduce the frame capture rate even further, the IT_VSYNC interrupt can be used to count the number of frames between two captures, in conjunction with the snapshot mode. This is not allowed by the embedded synchronization mode.

6.2.2.2 DCMI clock configuration in embedded synchronization

The line-start/line-end and frame-start/frame-end are determined by codes or markers embedded within the data flow. The embedded synchronization codes are supported only for an 8-bit parallel data interface width. The synchronization codes must be programmed in the DCMI_ESCR register as defined in Figure 37.

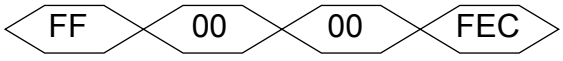
Figure 37. DCMI_ESCR register bytes



FEC (frame-end code)

The most significant byte specifies the frame-end delimiter. The camera module sends a 32-bit word containing 0xFF 00 00 XY with XY = FEC code, to signal the end of a frame. The code is received as indicated in Figure 38.

Figure 38. FEC structure



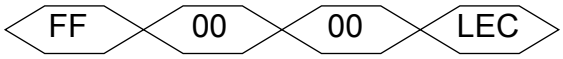
Before the reception of this FEC code, VSYNC must be set to one in DCMI_SR to indicate a valid frame. After the reception of the FEC, VSYNC must be cleared to zero to indicate that it is synchronization between frames. VSYNC must remain at zero until the reception of the next frame-start code.

If FEC = 0xFF (the camera module sends 0xFF 00 00 FF), all the unused codes are interpreted as frame-end codes. There are 253 values corresponding to the end-of-frame delimiter (0xFF0000FF and the 252 unused codes).

LEC (line-end code)

This byte specifies the line-end marker. The code received from the camera to indicate the end of line is 0xFF 00 00 XY with XY = LEC code.

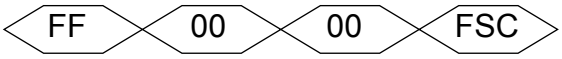
Figure 39. LEC structure



FSC (frame-start code)

This byte specifies the frame-start marker. The code received from the camera to indicate the start of new frame is 0xFF 00 00 XY with XY = FSC code.

Figure 40. FSC structure



LSC (line-start code)

This byte specifies the line-start marker. The code received from the camera to indicate the start of new line is 0xFF 00 00 XY with XY = LSC code.

If LSC = 0xFF, the camera module does not send a frame-start delimiter. The DCMI interprets the first occurrence of an LSC code after an FEC code as an FSC code occurrence.

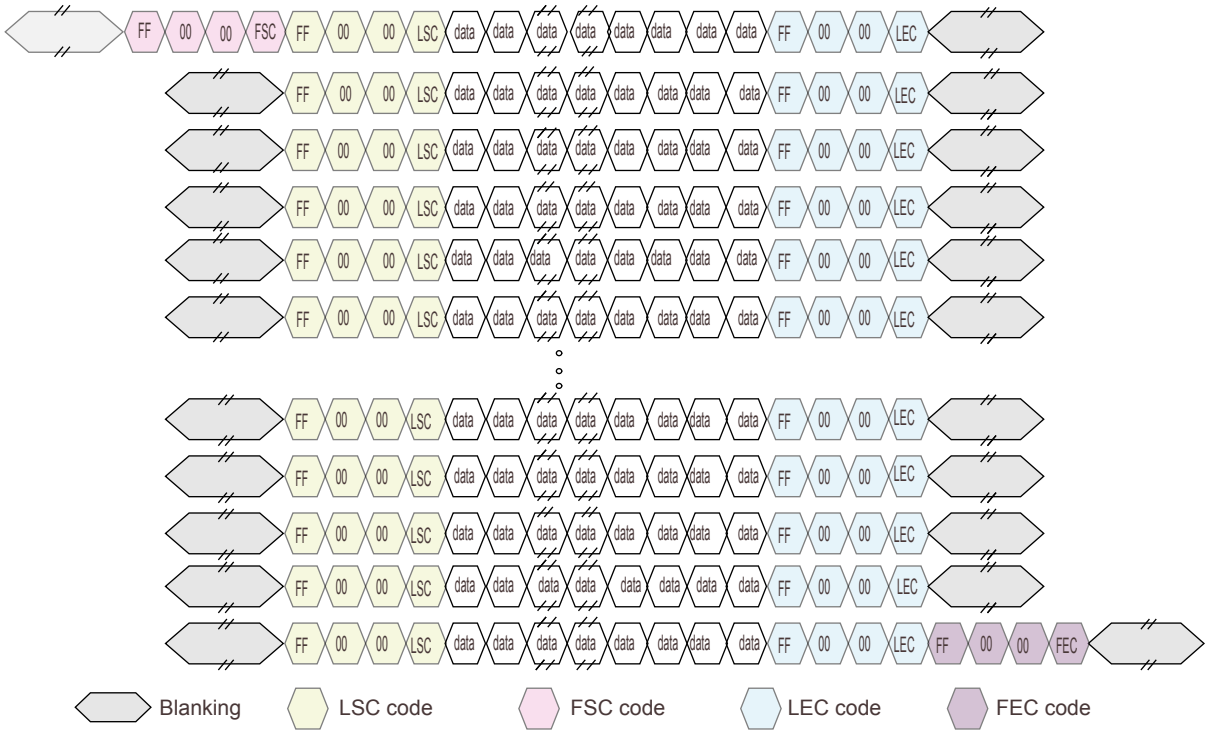
Figure 41. LSC structure

In this embedded synchronization mode, HSPOL and VSPOL bits are ignored. While the DCMI receives data (CAPTURE = 1 in DCMI_CR), the user can monitor the data flow to know if it is an active line/frame or a synchronization between lines/frames, by reading VSYNC and HSYNC in DCMI_SR.

If ERR_IE = 1 in DCMI_IER, an interrupt is generated each time an error occurs (such as embedded synchronization characters not received in the correct order).

Figure 42 shows a frame received in embedded synchronization mode.

Figure 42. Frame structure in embedded synchronization mode



6.3 DCMI configuration

The DCMI configuration allows the user to select the capture mode, the data format, the image size, and the resolution.

6.3.1 Capture mode selection

The user can capture an image or a video by selecting one of the following modes:

- the continuous grab mode to capture frames (images) continuously
- the snapshot mode to capture a single frame

The received data in snapshot or continuous grab mode are transferred to the memory frame buffer by the DMA. The buffer location and mode (linear or circular buffer) are controlled through the system DMA.

6.3.2 Data format selection

The DCMI allows the reception of compressed data (JPEG) or many uncompressed data formats (such as monochrome, RGB, or YCbCr). For more details, refer to Section 5.6 .

6.3.3 Image resolution and size

The DCMI allows the reception of a wide range of resolutions (low, medium, high) and image sizes, since the image size depends on the image resolution and data format. The DMA ensures the transfer and the placement of the received images in the memory frame buffer.

Optionally, the user can configure the byte, line, and frame select mode to modify the image resolution and size, and in some cases, the data format (see Section 5.8). The user can also configure and enable the crop feature to select a rectangular window from the received image (see Section 5.7).

Note: The DCMI configuration registers must be programmed correctly before enabling the ENABLE bit in DCMI_CR. The DMA controller and all DCMI configuration registers must be programmed correctly before enabling the CAPTURE bit in DCMI_CR.

6.4 DMA configuration

The DMA configuration is a crucial step to guarantee the application success.

As mentioned in Section 3.2 , the DMA2 ensures the transfer from the DCMI to the memory (internal SRAM or external SRAM/SDRAM) for all STM32 devices embedding the DCMI.

For STM32H7 and SMT32L4+ devices, the DMA1 can also access the AHB2 peripherals and ensure the transfer of the received data from the DCMI to the memory frame buffer.

For STM32U575/585 devices, the GPDMA1 ensures the transfer from the DCMI to the memory.

6.4.1 DMA configuration for DCMI-to-memory transfers

The transfer direction must be peripheral-to-memory by configuring:

- DIR bits in DMA_SxCR for STM32F2, STM32F4, STM32F7, and STM32H7 devices
- DIR bits in DMA_CCRx for STM32L4x6 and STM32L4+ devices
- SWREQ = 0 and REQSEL[6:0] in GPDMA_CxTR2 for STM32U575/585

The source address (DCMI data register address) must be written:

- in DMA_SxPAR for STM32F2, STM32F4, STM32F7, and STM32H7 devices
- in DMA_CPARx for STM32L4x6 and STM32L4+ devices
- in GPDMA_CxSAR for STM32U575/585 devices

The destination address (frame buffer address in internal SRAM or external SRAM/SDRAM) must be written:

- in DMA_SxMAR for STM32F2, STM32F4, STM32F7, and STM32H7 devices
- in DMA_CMARx for STM32L4x6 and STM32L4+ devices
- in GPDMA_CxDAR for STM32U575/585 devices

To ensure the data transfer from the DCMI data register, the DMA waits for the request to be generated from the DCMI. The relevant stream and channel must be configured. For more details refer to Section 6.4.3 .

Since a DMA request is generated each time the DCMI data register is filled, the data transferred from the DCMI must have a32-bit width. Data are transferred:

- to the DMA2 (or the DMA1 for STM32H7 and STM32L4+ devices) for all STM32 except for STM32U575/585
- to the GPDMA1 for STM32U575/585 devices

The peripheral data width must be 32-bit words. It is programmed:

- by PSIZE bits in DMA_SxCR for STM32F2, STM32F4, STM32F7, and STM32H7 devices
- by PSIZE bits in DMA_CCRx register for STM32L4x6 and STM32L4+ devices
- by DDW_LOG2 and DBL_1 in GPDMA_CxTR1 for STM32U575/585 devices

The DMA is the flow controller: the number of 32-bit data words to be transferred is software programmable from 1 to 65535 (see Section 6.4.4 for more details) :

- in DMA_SxNDTR for STM32F2, STM32F4, STM32F7, and STM32H7 devices
- in DMA_CNDTRx for STM32L4x6 and STM32L4+ devices
- in GPDMA_CxBr1 for STM32U575/585 devices

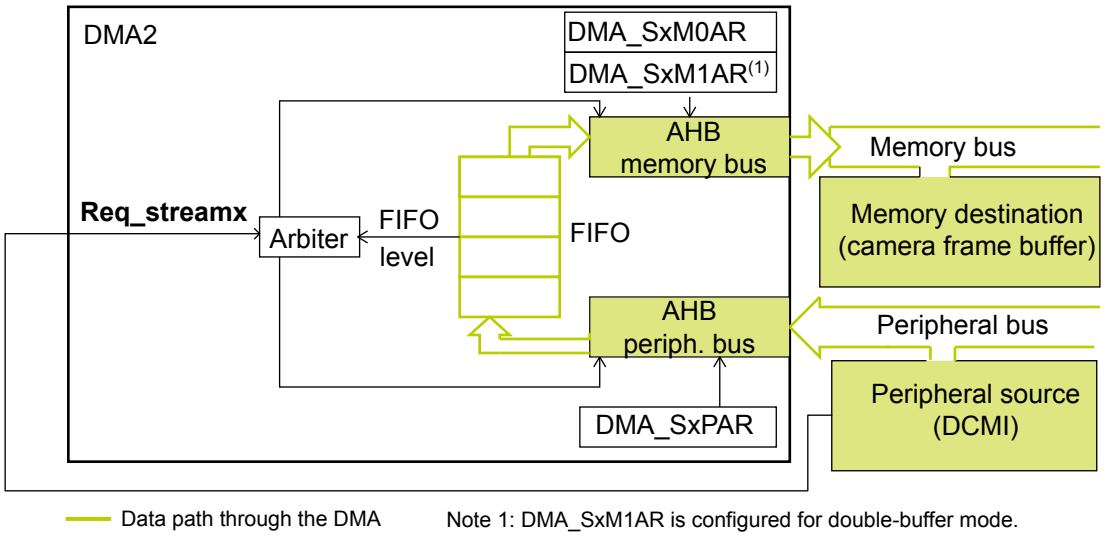
The DMA operates in one of the following modes:

- direct mode: each word received from the DCMI is transferred to the memory frame buffer.
- FIFO mode: the DMA uses its internal FIFO to ensure burst transfers (more than one word from the DMA FIFO to the memory destination).

For more details on the DMA internal FIFO, refer to Section 6.4.5 .

Figure 43 shows the DMA2 (or the DMA1 for STM32H7 and STM32L4+ devices, or the GPDMA1 for STM32U575/585 devices) operation in peripheral-to-memory mode (except for STM32L496xx and STM32L4A6xx devices because the DMA2 in these devices has only one port).

Figure 43. Data transfer through the DMA



6.4.2 DMA configuration versus image size and capture mode

The DMA must be configured according to the image size (color depth and resolution), and to the capture mode:

- In snapshot mode, the DMA must ensure the transfer of one frame (image) from the DCMI to the desired memory:
  - If the image size in words does not exceed 65535, the stream can be configured in normal mode (see Section 6.4.6).
  - If the image size in words is between 65535 and 131070, the stream can be configured in double-buffer mode (see Section 6.4.8).
  - If the image size in words exceeds 131070, the stream can be configured in double-buffer mode (see Section 6.4.9).
- In continuous mode: the DMA must ensure the transfer of successive frames (images) from the DCMI to the desired memory. Each time the DMA finishes the transfer of one frame, it starts the transfer of the next frame:
  - If one image size in words does not exceed 65535, the stream can be configured in circular mode (see Section 6.4.7).
  - If one image size in words is between 65535 and 131070, the stream can be configured in double-buffer mode (see Section 6.4.8).
  - If one image size in words exceeds 131070, the stream can be configured in double-buffer mode (see Section 6.4.9).

6.4.3 DCMI channel and stream configuration

The user must also configure the corresponding DMA2 (or the DMA1 for STM32H7 and STM32L4+ devices, or the GPDMA1 for STM32U575/585 devices) stream and channel to ensure the DMA acknowledgment each time the DCMI data register is fulfilled.

The tables below summarize the DMA stream and channels that enable the DMA request from the DCMI.

Table 7. DMA stream selection across STM32 devices

STM32	DMA stream	Channel
STM32F2	Stream 1 and Stream 7	Channel 1
STM32F4		
STM32F7		
STM32H7	Stream 0 to stream 7	Multiplexer1 request 75
STM32L4	Stream 0	Channel 6
	Stream 4	Channel 5

Table 8. DMA stream selection across STM32 devices

STM32		DMA channel	Request
STM32L4+	STM32L4Rxxx and STM32L4Sxxx	Channel 1 to channel 7	DMA request multiplexer 90
	STM32L4P5xx and STM32L4Q5xx		DMA request multiplexer 91
STM32U575/585		Channel 0 to channel 15	GPDMA1 request 86

Note: See the reference manual for a step-by-step description of the stream and channel configuration procedure.

6.4.4 DMA_SxNDTR/DMA_CNDTRx/GPDMA_CxBR1 register

The total number of words to transfer from the DCMI to the memory is programmed in this register (see Section 6.4.1).

When the DMA starts the transfer from the DCMI to the memory, the number of items decreases from the initial programmed value until the end of the transfer (reaching zero or disabling the stream by software before the number of data remaining reaches zero).

The table below gives the number of bytes corresponding to the programmed value and the peripheral data width (PSIZE bitfield).

Table 9. Maximum number of bytes transferred during one DMA transfer

Programmed value in the register	Peripheral size	Number of bytes
65535	Words	262140
0 < N < 35535		4 * N

Note: To avoid data corruption, this programmed value must be a multiple of MSIZE or PSIZE.

6.4.5 FIFO and burst transfer configuration

The DMA performs the transfer with or without enabling the 4-word FIFO. When the FIFO is enabled, the source data width (programmed in PSIZE) can differ from the destination data width (programmed in MSIZE). In this case, the user must pay attention to adapt the address to write:

- in DMA_SxPAR and DMA_SxM0AR (and DMA_SxM1AR in case of double-buffer mode configuration) to the data width programmed in PSIZE and MSIZE of DMA_SxCR for all STM32 except STM32L4/L4+ and STM32U5 devices
- in DMA_CPARx and DMA_CMARx to the data width programmed in PSIZE and MSIZE of DMA_CCRx for STM32L4 and STM32L4+ devices
- in GPDMA_CxSAR and GPDMA_CxDAR to the data width programmed with a burst length by SBL_1[5:0] (respectively DBL_1[5:0]), and with a data width defined by SDW_LOG2[1:0] (respectively DDW_LOG2[1:0]) in GPDMA_CxTR1 for STM32U575/585 devices

For a better performance, it is recommended to use the FIFO. When the FIFO mode is enabled, the user can configure the MBURST bits to make the DMA perform burst transfer (up to four words) from its internal FIFO to the destination memory, which guarantees better performance.

6.4.6 Normal mode for low resolution in snapshot capture

Low-resolution images are the ones having size (in 32-bit word) less than 65535. In snapshot mode, the normal mode can be used to ensure the transfer of low-resolution frames (see Table 9).

The maximum number of pixels depends on the bit depth of the image (number of bytes per pixel). The DCMI supports two possible bit depths:

- 1 byte per pixel in monochrome or Y only format
- 2 bytes per pixel in case of RGB565 or YCbCr format

The table below summarizes the maximum image resolution that can be transferred using the normal mode.

Table 10. Maximum image resolution in normal mode

Item	Max number of bytes	Bit depth (byte/pixel)	Max number of pixels	Max resolution
Word	262140	1	262140	720x364
		2	131070	480x272

6.4.7 Circular mode for low resolution in continuous capture

The circular mode allows the process of successive frames (continuous data flows), providing that one frame size is less than 65535. The initial size value is programmed:

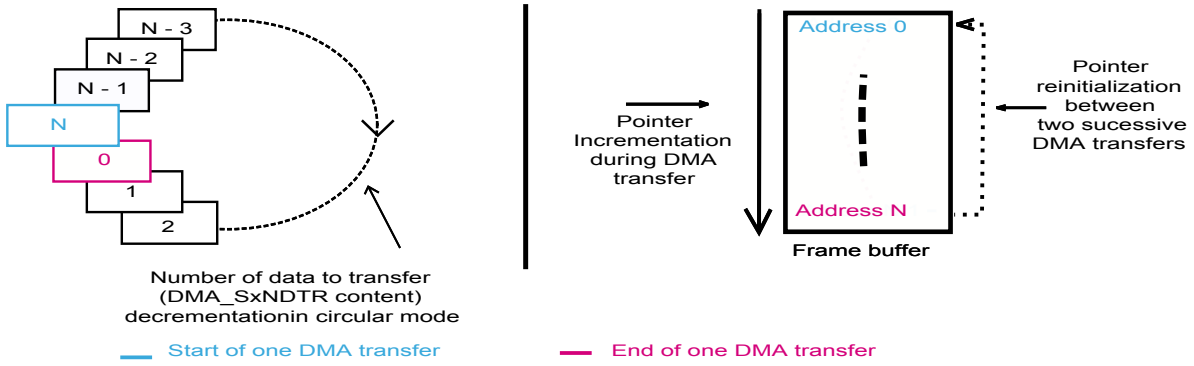
- in DMA_SxNDTR for STM32F2, STM32F4, STM32F7, and STM32H7 devices
- in DMA_CNDTRx for STM32L4x6 and STM32L4+ devices
- in GPDMA_CxBr1 for STM32U575/585 devices

Each time the number of data decrementing reaches the zero, the number of data words is automatically reloaded to the initial value. Each time the DMA pointer reaches the end of the frame buffer, it is reinitialized, and the DMA ensures the transfer of the next frame.

The resolutions listed in Table 10 are also valid for the low resolution in continuous mode.

Figure 44 shows the DMA_SxNDTR value and the frame buffer pointer modifications during a DMA transfer and between two successive DMA transfers.

Figure 44. Frame buffer and DMA_SxNDTR register in circular mode



6.4.8 Double-buffer mode for medium resolutions (snapshot or continuous capture)

Note: This mode is not available for STM32L4A6xx, STM32L496xx, and STM32U575/585 devices.

Medium resolution images are the ones having size (in 32-bit word) between 65536 and 131070. When the double-buffer mode is enabled, the circular mode is automatically enabled.

If the image size exceeds (in words) the maximum sizes mentioned in Table 10 in snapshot or continuous capture, the double-buffer mode must be used in snapshot or continuous mode. In this case, the number of pixels per frame allowed is doubled since received data are stored in two buffers: each buffer maximum size (in 32-bit words) is 65535 (the maximum frame size is 131070 words or 524280 bytes). The images sizes and resolutions allowed to be received by the DCMI and transferred by the DMA are then doubled.

Table 11. Maximum image resolution in double-buffer mode

Item	Max number of bytes	Bit depth (byte/ pixel)	programmed value in the register	Number of pixels	Max resolution
Word	524280	1	65535	524280	960x544
			0 < N < 65535	8 * N	
		2	65535	262140	720x364
			0 < N < 65535	4 * N	

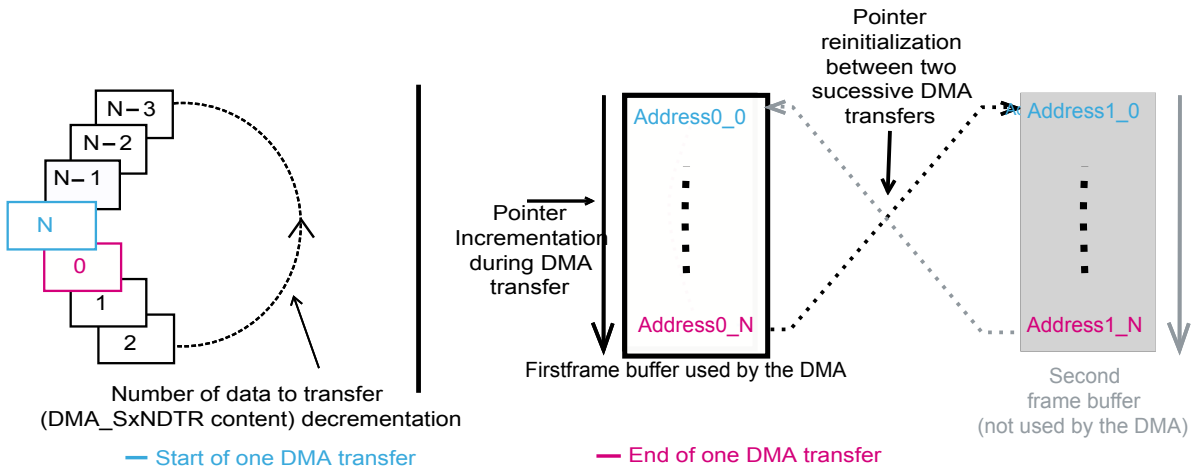
In this mode, the double-buffer stream has two pointers (two buffers for storing data), switched each end of transaction:

- In snapshot mode, the DMA controller writes the data in the first frame buffer. After this first frame buffer is fulfilled (at this level, the register is reinitialized to the programmed value, and the DMA pointer switches to the second frame buffer), data are transferred to the second buffer. The total frame size (in words) is divided by two and programmed in the register. The image is stored in two buffers having the same size.
- In continuous mode, each time one frame (image) is received and stored in the two buffers. As the circular mode is enabled, the register is reinitialized to the programmed value (total frame size divided by two), and the DMA pointer switches to the first frame buffer to receive the next frame.

The double-buffer mode is enabled by setting DBM = 1 in DMA_SxCR.

Figure 45 shows the two pointers and the DMA_SxNDTR value modifications during the DMA transfers.

Figure 45. Frame buffer and DMA_SxNDTR register in double-buffer mode



6.4.9 DMA configuration for higher resolutions

When the number of words in one frame (image) in snapshot or continuous mode exceeds 131070, and when the image resolution exceeds the indicated ones in Table 11, the DMA double-buffer mode cannot ensure the transfer of the received data.

Note: This section highlights only the DMA operation in case of high resolution. An example is developed and described using this DMA configuration in Section 8.3.6 SXGA resolution capture (YCbCr data format).

The STM32F2, STM32F4, STM32F7, STM32H7, and STM32L4+ devices embed a very important feature in double-buffer mode: the possibility to update the programmed address for the AHB memory port on-the-fly (in DMA_SxM0AR or DMA_SxM1AR) when the stream is enabled. The following conditions must be respected:

- When CT is cleared to zero in DMA_SxCR (current target memory is memory 0), the DMA_SxM1AR register can be written. Attempting to write to this register while CT = 1 generates an error flag (TEIF), and the stream is automatically disabled.
- When CT is set to one in DMA_SxCR (current target memory is memory 1), the DMA_SxM0AR register can be written. Attempting to write to this register while CT = 0 generates an error flag (TEIF), and the stream is automatically disabled.

To avoid any error condition, it is advised to change the programmed address as soon as the TCIF flag is asserted. At this point, the targeted memory must have changed from memory 0 to memory 1 (or from 1 to 0), depending on the CT bit value in DMA_SxCR.

Note: For all the other modes than the double-buffer one, the memory address registers are write-protected as soon as the stream is enabled.

The DMA allows then the management of more than two buffers:

- In the first cycle, while the DMA uses the buffer 0 addressed by pointer 0 (memory 0 address in DMA_SxM0AR), the buffer 1 is addressed by pointer 1 (memory 1 address in DMA_SxM1AR).
- In the second cycle, while DMA uses the buffer 1 addressed by pointer 1, the buffer 0 address can be changed, and the frame buffer 2 can be addressed by pointer 0.
- In the second cycle, while the DMA is using the buffer 2 addressed by pointer 0, the frame buffer 1 address can be changed, and the buffer 3 can be addressed by pointer 1.

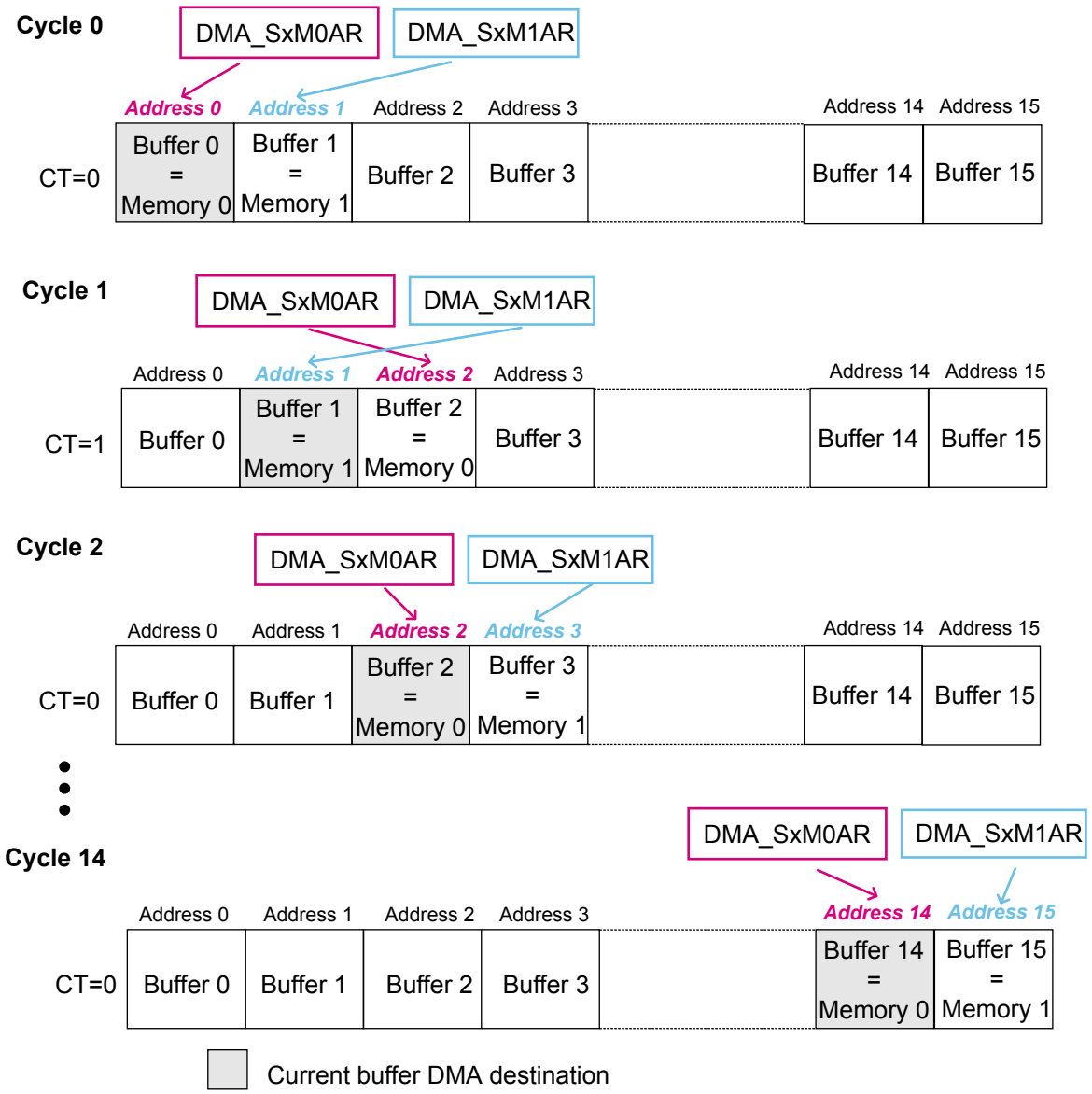
DMA_SxM0AR and DMA_SxM1AR can then be used to address many buffers, ensuring the transfer of high resolution images.

Note: To simplify the use of this specific feature, it is recommended to divide the image into equal buffers. When capturing high resolution images, the user must secure that the memory destination has a sufficient size.

Example: In case of SXGA resolution (1280x1024), the image size is 655360 words (32 bits). This size must be divided into equal buffers, with a maximum size of 65535 for each of them. To be correctly received, the image must then be divided into 16 frame buffers, with each frame buffer size equal to 40960 (lower than 65535).

Figure 46 illustrates the DMA_SxM0AR and DMA_SxM1AR update during the DMA transfer:

Figure 46. DMA operation in high resolution case



6.5 Camera module configuration

The following steps allow a correct configuration of the camera module (refer also to the camera module datasheet):

1. Configure the input/output functionalities for camera configuration pins to be able to modify its registers (serial communication, mostly I^2C).
2. Apply hardware reset on the camera module.
3. Initialize the camera module:
   - Configure the image resolution.
   - Configure the contrast and the brightness.
   - Configure the white balance of the camera (such as black and white, white negative, white normal).
   - Select the camera interface (some camera modules have serial and parallel interface).
   - Select the synchronization mode if the camera module supports more than one.
   - Configure the clock signals frequencies.
   - Select the output data format.

7 Power consumption and performance

7.1 Power consumption

In order to save more energy when the application is in low-power mode, it is recommended to put the camera module in low-power mode before the STM32 entry in low-power mode.

Putting camera module in low-power mode ensures a considerable gain in power consumption.

Example for OV9655 CMOS sensor:

- In active mode, the operating current is 20 mA.
- In standby mode, the current requirements drop to 1 mA in case of I2C-initiated Standby mode (the internal circuit activity is suspended but the clock is not halted), and to 10 µA in case of pin-initiated Standby mode (the internal device clock is halted and all internal counters are reset). For more details, refer to the camera datasheet.

7.2 Performance

For all STM32 MCUs, the number of bytes to be transferred at each pixel clock depends on the extended data mode:

- When the DCMI is configured to receive 8-bit data, the camera interface takes four pixel clock cycles to capture a 32-bit data word.
- When the DCMI is configured to receive 10-, 12-, or 14-bit data, the camera interface takes two pixel clock cycles to capture a 32-bit data word.

The table below summarizes the maximum data flow depending on the data width configuration.

Table 12. Maximum data flow at maximum DCMI_PIXCLK

These values are calculated for the maximum DCMI_PIXCLK given in Table 2.

STM32	Data flow (max Mbyte/s) in extended data mode			
	8-bit 1 byte per PICXCLK	10-bit 1.25 bytes per PICXCLK	12-bit 1.5 bytes per PICXCLK	14-bit 1.75 bytes per PICXCLK
STM32F2	46.875	58.594	70.312	82.031
STM32F4	52.734	65.918	79.101	92.285
STM32F7				
STM32H723/733, STM32H743/753, STM32H747/757, STM32H745/755, STM32H742, STM32H750, STM32H7A3/B3	78.125	97.656	117.187	136.718
STM32H725/735, STM32H730	107.422	134.277	161.133	187.988
STM32L4	31.25	39.062	46.875	54.687
STM32L4+	46.875	58.594	70.312	82.031
STM32U575/585	62.5	78.125	93.75	109.375

In some applications, the DMA2 (or the DMA1 for STM32H7 and STM32L4+, or the GPDMA1 for STM32U575/585) is configured to serve in parallel other requests together with the DCMI request. In this case, the user must pay attention to the stream priority configurations, and consider the performance impact when the DMA serves other streams in parallel with the DCMI.

For better performance, when using the DCMI in parallel with other peripherals having requests that can be connected to either DMA1 or DMA2, it is better to configure these streams to be served by the DMA that is not serving the DCMI.

The user must make sure the pixel clock configured on the camera module side is supported by the DCMI to avoid the overrun.

It is recommended to use the highest system speed HCLK for better performance, but the user must consider the speed of all the peripherals used (for example external memories speed) to avoid the overrun and to guarantee the application success.

The DCMI is not the only AHB2 peripheral but there are many other peripherals. The DMA is not the only master that can access the AHB2 peripherals. Using many AHB2 peripherals or other master accessing the AHB2 peripherals leads to a concurrency on the AHB2: the user must consider its impact on performance.

8 DCMI application examples

This section details how to use the DCMI, and provides step-by-step implementation examples .

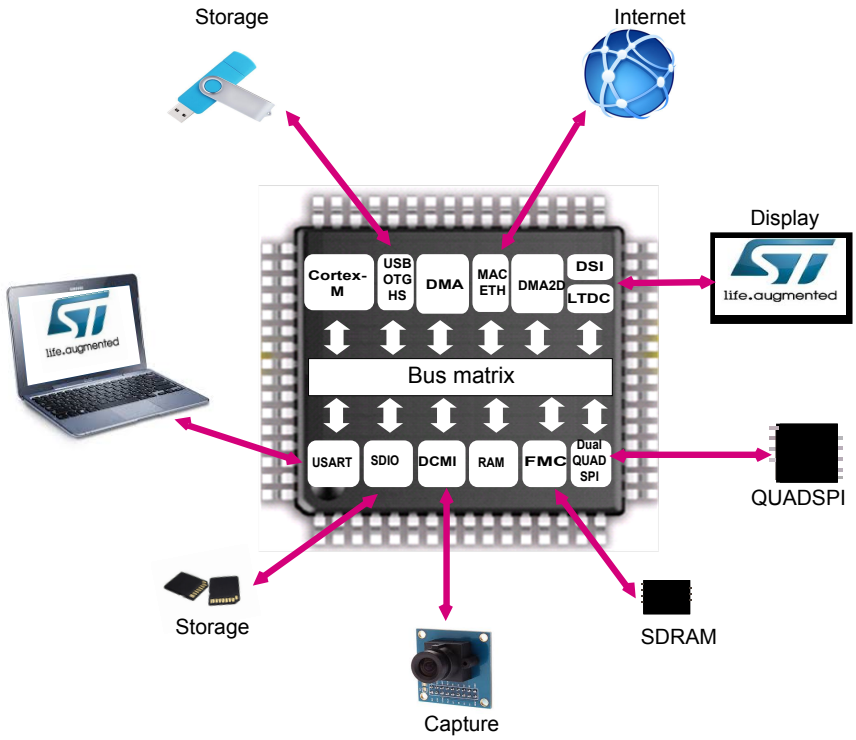
8.1 DCMI use cases

There are several imaging applications that can be implemented using the DCMI and other STM32 peripherals. Here below some applications examples:

- machine vision
- toys
- biometry
- security and video surveillance
- door phone and home automation
- industrial monitoring systems and automated inspection
- system control
- access control systems
- bar-code scanning
- video conferencing
- drones
- real-time video streaming and battery powered video camera.

Figure 47 provides application examples using an STM32 MCU that allows the user to capture data, store them in internal or external memories, display them, share them via Internet, and communicate with humans.

Figure 47. STM32 DCMI application example



8.2 STM32Cube examples

The STM32CubeF2, STM32CubeF4, STM32CubeF7, STM32CubeH7, STM32CubeL4, and STM32CubeU5 MCU Packages offer a large set of examples implemented and tested on the corresponding boards.

The table below gives an overview of the DCMI examples and applications across various STM32Cube. All these examples are developed to capture RGB data. For most of the examples, the user can select one of the following resolutions: QQVGA 160x120, QVGA 320x240, 480x272, VGA 640x480.

Table 13. STM32Cube DCMI examples

MCU Package	Project name	Board
STM32CubeF2	DCMI_CaptureMode	STM3220G-EVAL, STM3221G-EVAL
	SnapshotMode	
	Camera_To_USBDisk	
STM32CubeF4	DCMI_CaptureMode	STM32446E-EVAL, STM32429I-EVAL1, STM32469I-EVAL, STM3240G-EVAL
	SnapshotMode	
	Camera_To_USBDisk	
STM32CubeF7	DCMI_CaptureMode	STM32756G-EVAL, STM32F769I-EVAL
	SnapshotMode	
	Camera_To_USBDisk	
	Camera	STM32F7508-DISCOVERY
STM32CubeH7	DCMI_CaptureMode	STM32H747I-DISCOVERY
	SnapshotMode	
STM32CubeL4	DCMI_CaptureMode	32L496GDISCOVERY, 32L4R9IDISCOVERY
	SnapshotMode	32L496GDISCOVERY
	DCMI_Preview	
STM32CubeU5	DCMI_ContinousCap_EmbeddedSynchMode	STM32U575I-EVAL

8.3 DCMI examples based on STM32CubeMX

This section details the following typical examples of DCMI use:

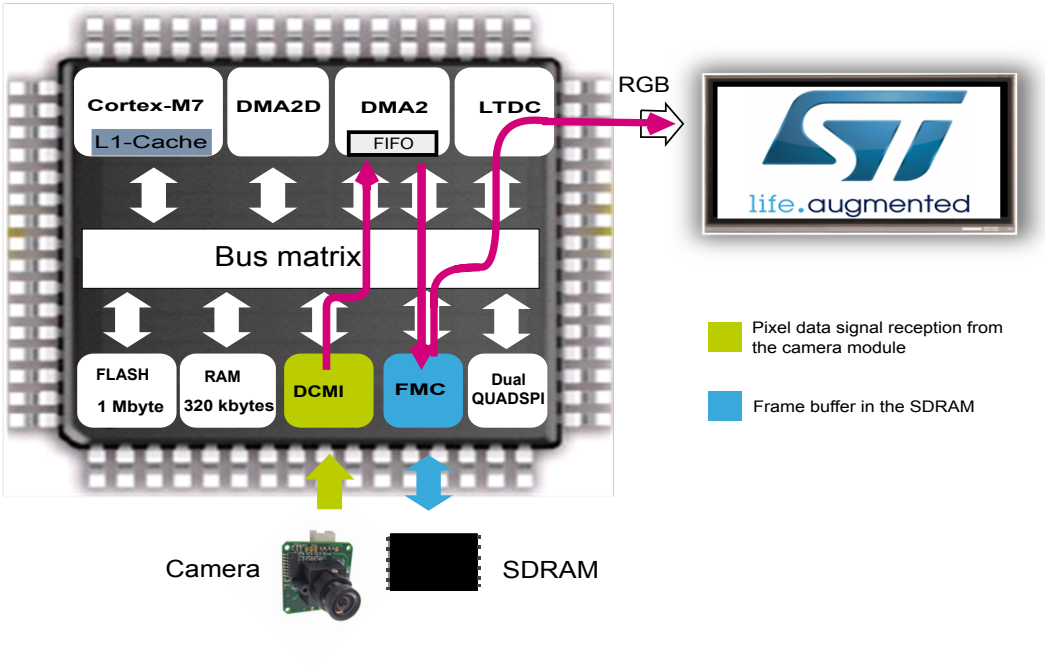
- capture and display of RGB data
  data captured in RGB565 format with QVGA (320x240) resolution, stored in the SDRAM, and displayed on the LCD-TFT
- capture of YCbCr data
  data captured in YCbCr format with QVGA (320x240) resolution and stored in the SDRAM
- capture of Y-only data
  DCMI configured to receive Y-only data to be stored in the SDRAM
- SXGA resolution capture (YCbCr data format)
  data captured in YCbCr format with SXGA (1280x1024) resolution and stored in the SDRAM
- capture of JPEG data
  data captured in JPEG format, and stored in the SDRAM

All these examples have been implemented on 32F746GDISCOVERY using STM32F4DIS-CAM (OV9655 CMOS sensor), except the capture of JPEG data that was implemented on STM324x9I-EVAL (OV2640 CMOS sensor).

As illustrated in Figure 48, the application consists of three main steps:

1. Import the received data from the DCMI to the DMA (to be stored in FIFO temporarily) through its peripheral port.
2. Transfer the data from the FIFO to the SDRAM.
3. Import data from the SDRAM to be displayed on the LCD-TFT, only for RGB data format. For YCbCr or JPEG data format, the user must convert the received data to RGB to be displayed.

Figure 48. Data path in capture and display application



For these examples, the user needs to configure the DCMI, the DMA2, the LTDC (for the RGB data capture and display example), and the SDRAM.

The five examples described in the next sections have some common configurations based on STM32CubeMX:

- GPIO configuration
- DMA configuration
- clock configuration

The following specific configurations are needed for Y-only and JPEG capture examples:

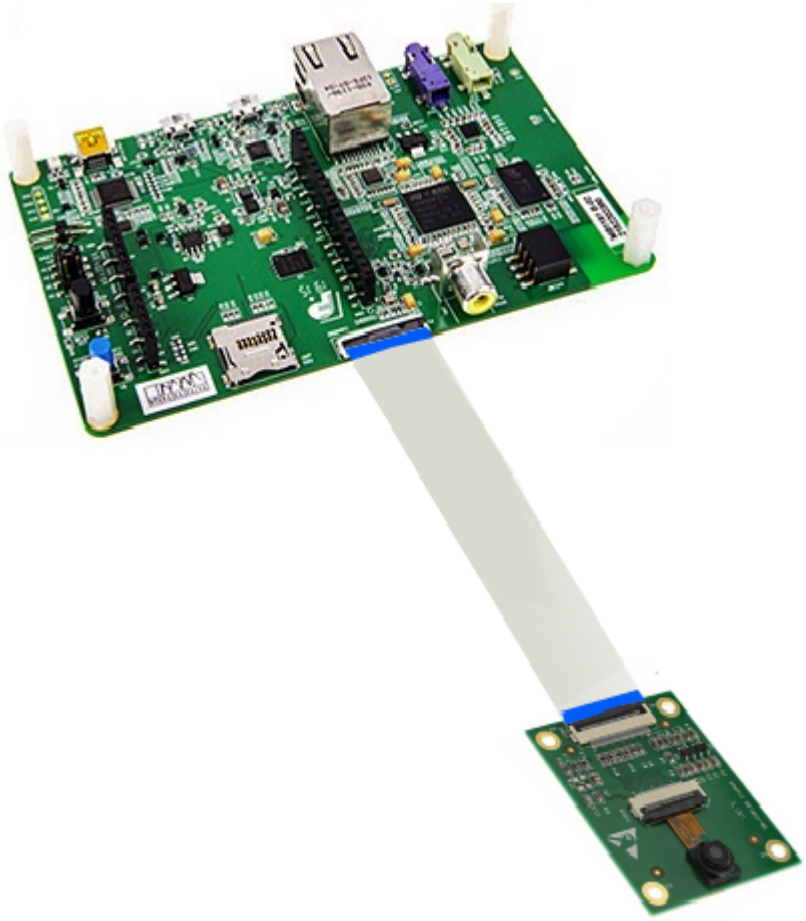
- DCMI configuration
- camera module configuration

The next sections provide the hardware description, the common configuration using STM32CubeMX, and the common modifications that have to be added to the STM32CubeMX generated project.

8.3.1 Hardware description

All examples except the JPEG capture, were implemented on 32F746GDISCOVERY using the camera board STM32F4DIS-CAM, as shown in Figure 49.

Figure 49. 32F746GDISCOVERY and STM32F4DIS-CAM interconnection



The STM32F4DIS-CAM board includes an Omnivision CMOS sensor (ov9655), 1.3 Mpixels. The resolution can reach 1280x1024. This camera module is connected to the DCMI via a 30-pin FFC.

The 32F746GDISCOVERY board features a 4.3-inch color LCD-TFT with capacitive touch screen that is used in the first example to display the captured images.

As shown in Figure 50, the camera module is connected to the STM32F7 through:

- control signals DCMI_PIXCLK, DCMI_VSYNC, DCMI_HSYNC
- image data signals DCMI_D[0..7]

Additional signals are provided to the camera module through the 30-pin FFC:

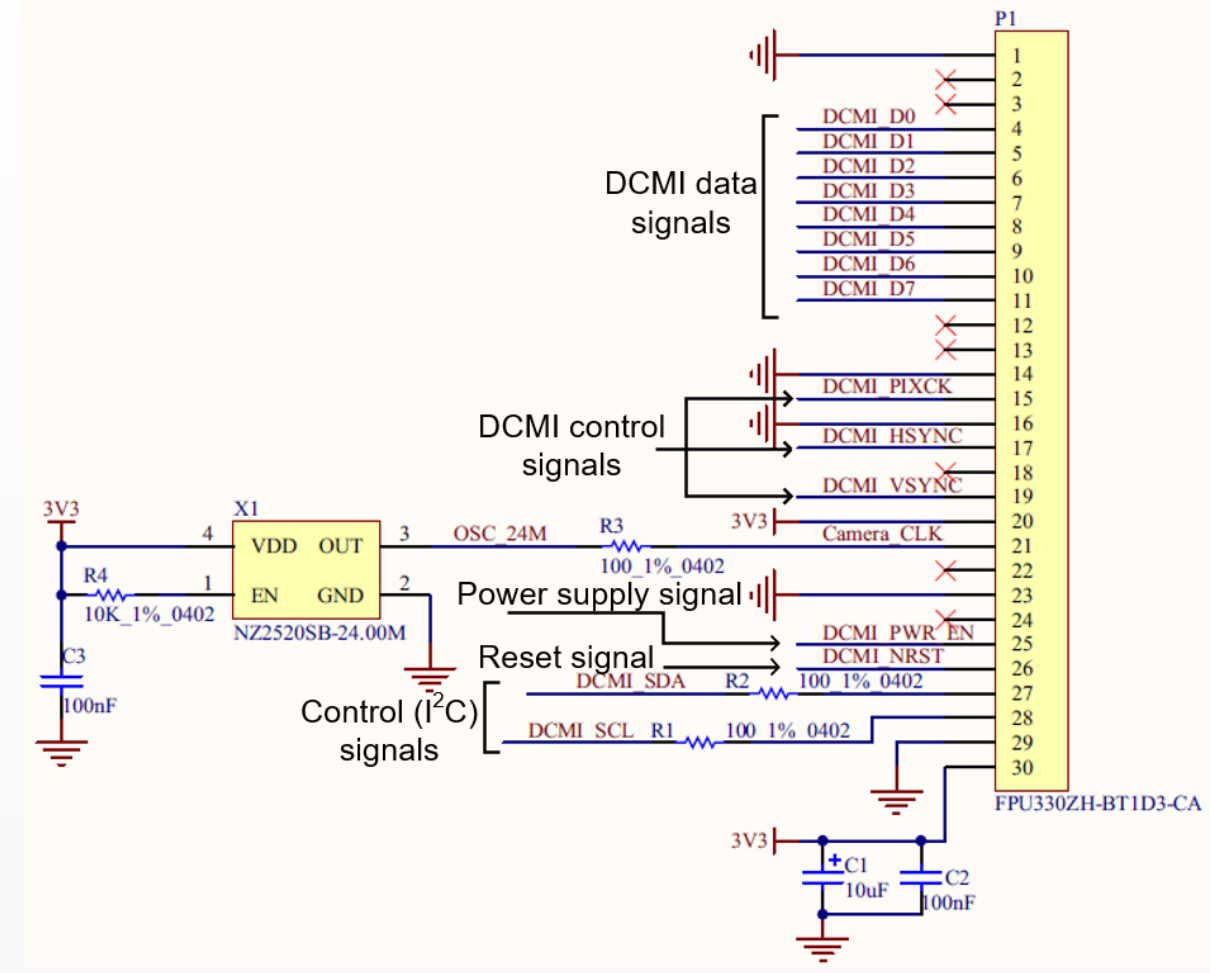
- power supply signals (DCMI_PWR_EN)
- clock for the camera module (Camera_CLK)
- configuration signals (I2C)
- reset signal (DCMI_NRST)

For more details on these signals, refer to Section 2.2.2 Camera module interconnect (parallel interface).

The camera clock is provided to the camera module through the Camera_CLK pin, by the NZ2520SB crystal clock oscillator (X1) embedded on the 32F746GDISCOVERY board. The frequency of the camera clock is equal to 24 MHz.

The DCMI reset pin (DCMI_NRST) used to reset the camera module is connected to the global MCU reset pin (NRST).

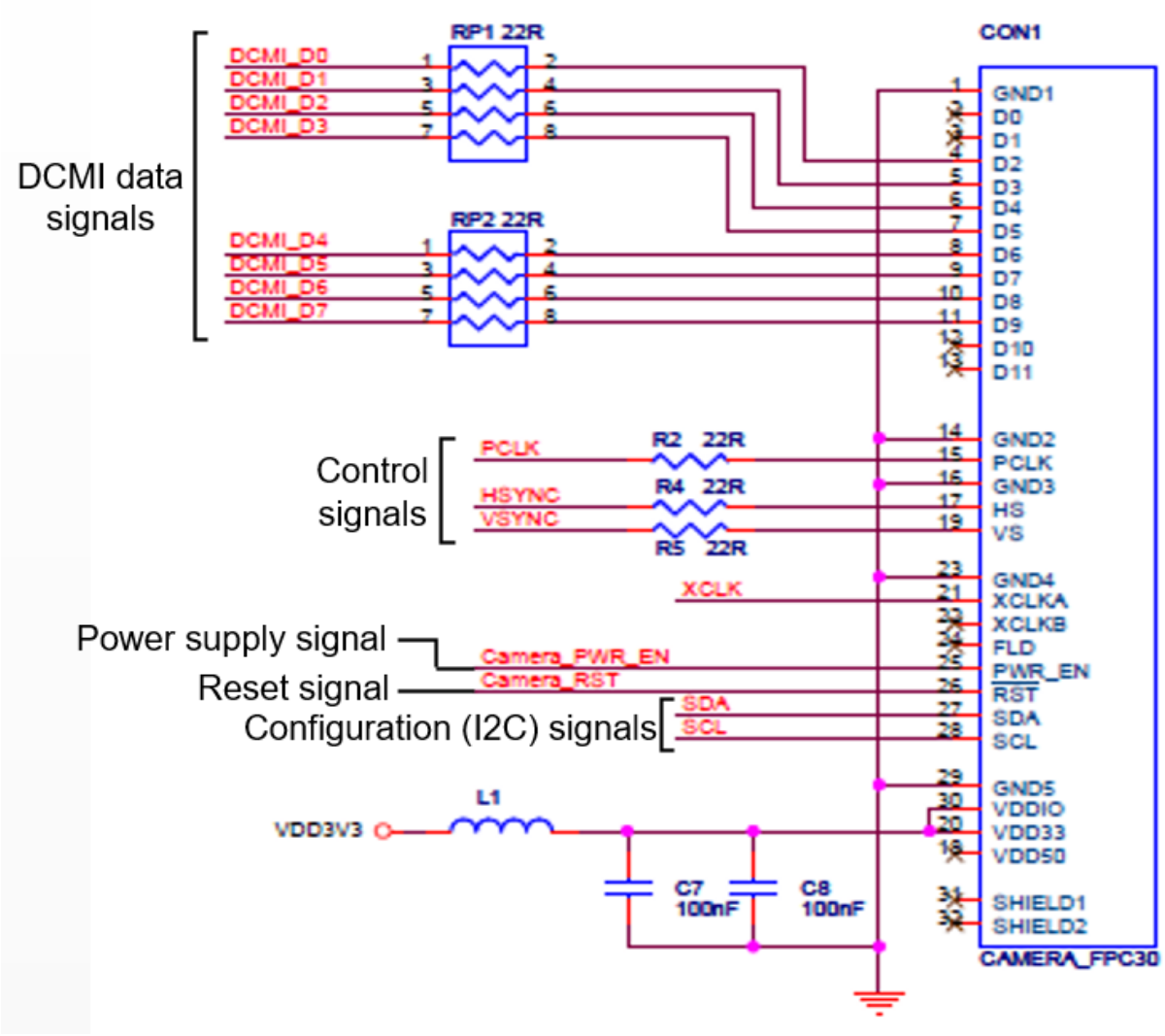
Figure 50. **Camera connector on 32F746GDISCOVERY**



For more details on the 32F746GDISCOVERY board, refer to the user manual *Discovery kit for STM32F7 Series with STM32F746NG MCU* (UM1907) available on the STMicroelectronics website.

Figure 51 details the camera module connector implemented on STM32F4DIS-CAM.

Figure 51. Camera connector on STM32F4DIS-CAM



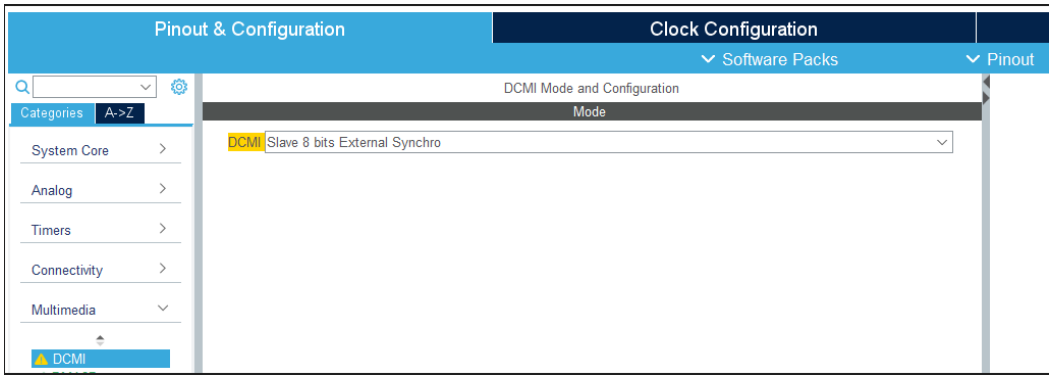
8.3.2 Configuration of common examples

When starting with STM32CubeMX, the first step is to configure the project location and the corresponding toolchain or IDE (menu *Project/Settings*).

8.3.2.1 *STM32CubeMX - Configuration of DCMI GPIOs*

1. Select the DCMI and choose "*Slave 8 bits External Synchro*" in the *Pinout & configuration* multimedia tab to configure the DCMI in slave 8-bit external (hardware) synchronization.

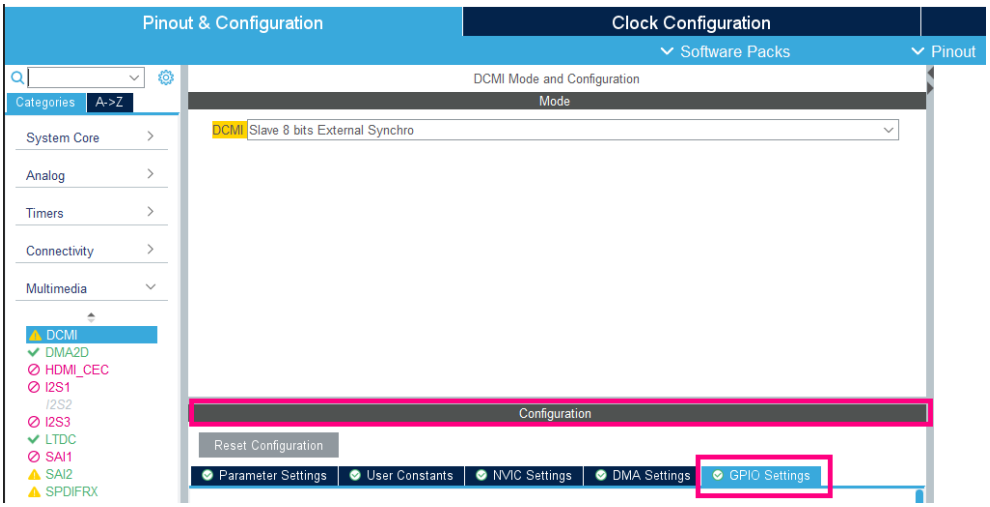
Figure 52. STM32CubeMX - DCMI synchronization mode selection



If, after selecting one hardware configuration (*Slave 8 bits External Synchro* for example), the used GPIOs do not match with the hardware, the user can change the desired GPIOs, and configure the alternate function directly on the pin.
Another method consists of configuring manually the GPIO pins by selecting the right alternate function for each of them. For more details on the GPIOs that must be configured, refer to Figure 54.
After this step, 11 pins must be highlighted in green (D[0..7], DCMI_VSYNC, DCMI_HSYNC, and DCMI_PIXCLK).

2. Select the *Configuration* tab to configure the GPIOs mode.
3. When the *DCMI configuration* window appears, select the *GPIO Settings* tab.

Figure 53. STM32CubeMX - GPIO settings selection



4. Select all the DCMI pins.

Figure 54. STM32CubeMX - DCMI pin selection

Pin Name	Signal on Pin	GPIO output l...	GPIO mode	GPIO Pull-up/...	Maximum out...	User Label	Modified
PA4	DCMI_HSYNC	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_HSYNC	☑
PA6	DCMI_PIXCLK	n/a	Alternate Fun...	No pull-up an...	Low		☐
PD3	DCMI_D5	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_D5	☑
PE5	DCMI_D6	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_D6	☑
PE6	DCMI_D7	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_D7	☑
PG9	DCMI_VSYNC	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_VSYNC	☑
PH9	DCMI_D0	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_D0	☑
PH10	DCMI_D1	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_D1	☑
PH11	DCMI_D2	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_D2	☑
PH12	DCMI_D3	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_D3	☑
PH14	DCMI_D4	n/a	Alternate Fun...	No pull-up an...	Low	DCMI_D4	☑

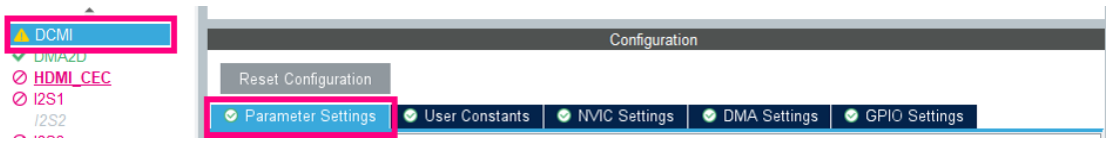
5. Set the GPIO pull-up/pull-down.

Figure 55. STM32CubeMX - GPIO no pull-up and no pull-down selection

8.3.2.2 *STM32CubeMX - Configuration of DCMI control signals and capture mode*

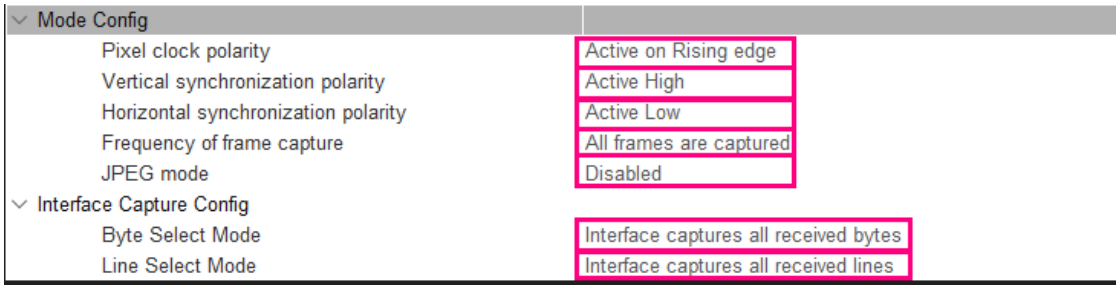
1. Click on *Parameter Settings* tab in *DCMI Configuration* window.

Figure 56. STM32CubeMX - Parameter Settings tab



2. Set the different parameters (vertical synchronization, horizontal synchronization, and pixel clock polarities) that must be programmed according to the camera module configuration.

Figure 57. STM32CubeMX - DCMI control signals and capture mode

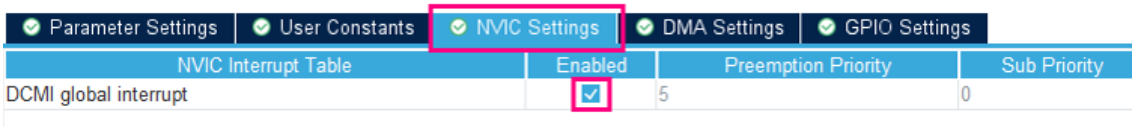


Note: The vertical synchronization polarity must be active high, and the horizontal synchronization polarity must be active low. They must not be inverted for this configuration of the camera module.

8.3.2.3 *STM32CubeMX - Enable DCMI interrupts*

1. Select *NVIC Settings* tab in *DCMI Configuration* window, and check the DCMI global interrupt.

Figure 58. STM32CubeMX - Configuration of DCMI interrupts



8.3.2.4 *STM32CubeMX - DMA configuration*

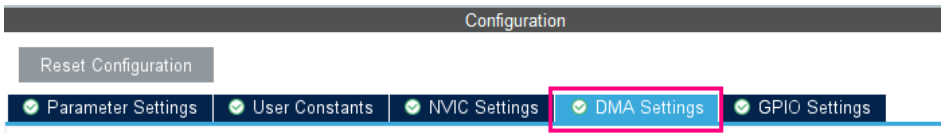
This configuration aims to receive RGB565 data (2 byte/pixel). The image resolution is QVGA (320x240). The image size is then 320 × 240 × 2 = 153600 bytes.

Since the data width sent from the DCMI is 4 bytes (32-bit words sent from the DCMI data register), the number of data items in the DMA_SxNDTR register is the number of words to transfer. The number of words is then 38400 (153600 / 4) which is less than 65535.

In snapshot mode, the user can configure the DMA in normal mode. In continuous mode, the user can configure the DMA in circular mode.

1. Select *DMA Settings* tab in *DCMI Configuration* window.

Figure 59. STM32CubeMX - DMA Settings tab

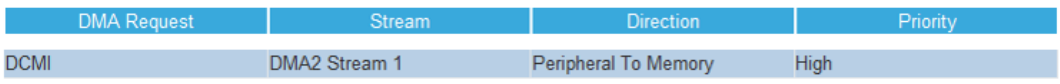


2. Click on the *Add* button.

Figure 60. STM32CubeMX - Add button

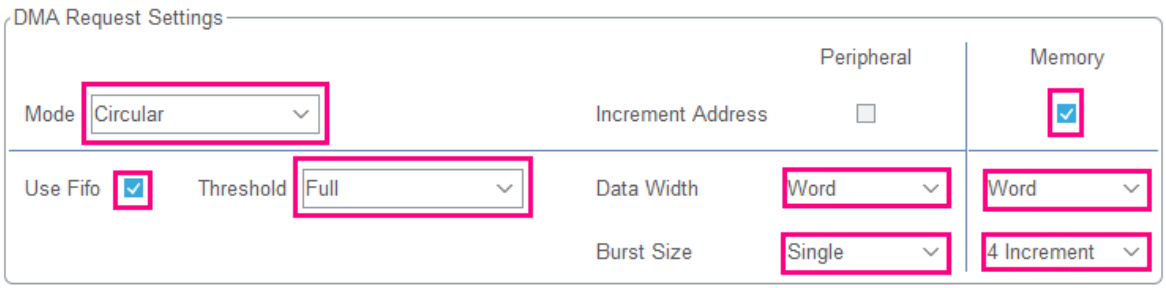
3. Click on *Select* under DMA Request, and choose DCMI. The DMA2 Stream 1 channel 1 is configured to transfer the DCMI request each time its time register is fulfilled.

Figure 61. STM32CubeMX - DMA stream configuration



4. Modify the *DMA Request Settings*.

Figure 62. STM32CubeMX - DMA configuration

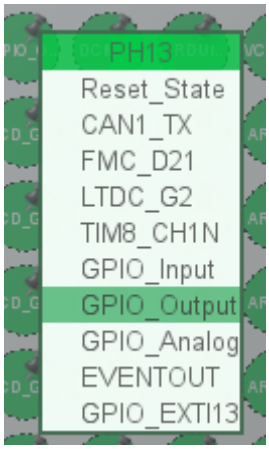


8.3.2.5 *STM32CubeMX - Camera module power-up pins*

To power up the camera module, the PH13 pin must be configured for 32F746GDISCOVERY.

1. Click on the PH13 pin and select GPIO_Output in the *Pinout* tab.

Figure 63. STM32CubeMX - PH13 pin configuration

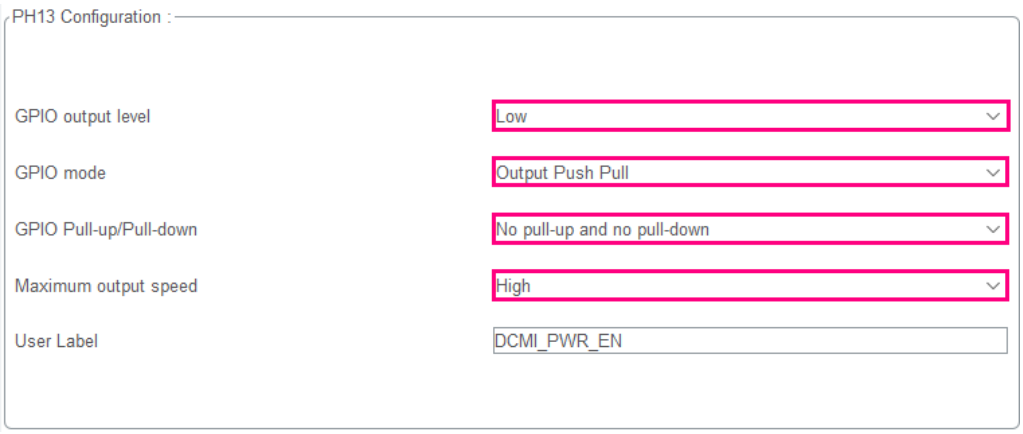


2. In *Pinout & Configuration* tab system core, click on the *GPIO* button.

Figure 64. STM32CubeMX - GPIO button

3. Set the parameters.

Figure 65. STM32CubeMX - DCMI power pin configuration



8.3.2.6 *STM32CubeMX - System clock configuration*

In this example the system clock is configured as follow:

- use of external HSE clock, where the main PLL is used as system source clock
- HCLK @ 200 MHz, so the Cortex-M7 and LTDC are both running at 200 MHz

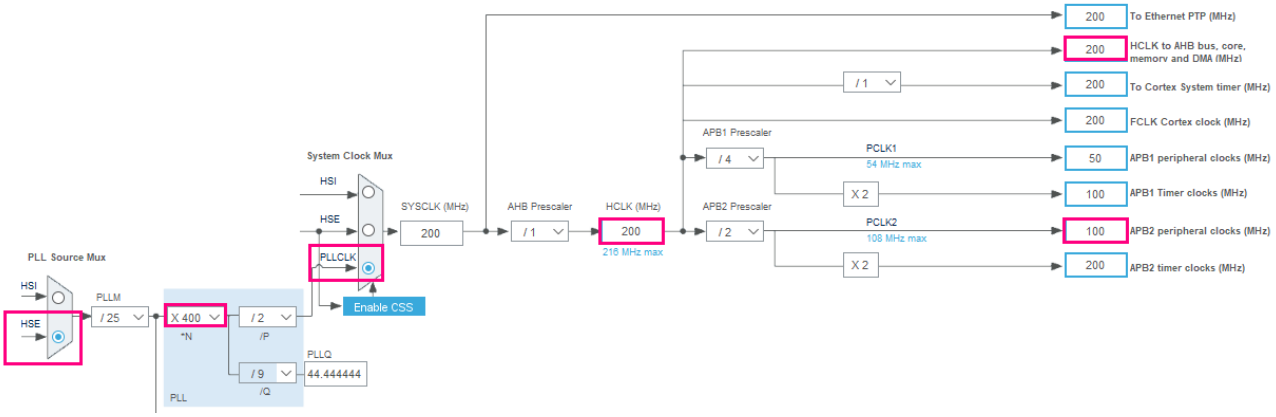
Note: HCLK is set to 200 MHz but not 216 MHz, in order to set the SDRAM_FMC at its maximum speed of 100 MHz with HCLK/2 prescaler.

1. Select the *Clock Configuration* tab.

Figure 66. STM32CubeMX - HSE configuration

2. Set the PLLs and the prescalers in the *Clock Configuration* tab, to get the system clock HCLK @ 200 MHz.

Figure 67. STM32CubeMX - Clock configuration



8.3.2.7 *Adding files to the project*

Generate the code and open the generated project using the preferred toolchain. Then follow these steps:

1. Right click on `Drivers/STM32F7xx_HAL_Driver`.
2. Choose "*Add Existing Files to group 'Drivers/STM32F7xx_HAL_Driver...*"
3. Select the following files in `Drivers/STM32F7xx_HAL_Driver/Src`:
   - `stm32f7xx_hal_dma2d.c`
   - `stm32f7xx_hal_ltdc.c`
   - `stm32f7xx_hal_ltdc_ex.c`
   - `stm32f7xx_hal_sdram.c`
   - `stm32f7xx_hal_uart.c`
   - `stm32f7xx_ll_fmc.c`
4. Create a new group called, for example, Imported_Drivers.
5. Copy the following files from the STM32746G_Discovery folder in C: directory to the Src folder of the project:
   - `stm32746g_discovery.c`
   - `stm32746g_discovery_sdram.c`
6. Copy the following files from the STM32746G_Discovery folder in C: directory to the Inc folder of the project:
   - `stm32746g_discovery.h`
   - `stm32746g_discovery_sdram.h`
7. Copy `ov9655.c` from the Components folder to the Src folder.
8. Copy `ov9655.h` from the Components folder to the Inc folder.
9. Copy `camera.h` from the Component/Common folder to the Inc folder.
10. Add the following files in the new group (called Imported_Drivers in this example):
    - `stm32746g_discovery.h`
    - `stm32746g_discovery_sdram.h`
    - `ov9655.c`
11. Allow modifications on `ov9655.h` and `camera.h` (read-only by default):
    a. Click right on the file.
    b. Uncheck read-only.
    c. Click on Apply and OK.

12. Modify `ov9655.h` by replacing `#include "../Common/camera.h"` by `#include "camera.h"`.
13. Copy the following files to the Inc folder:
    – `rk043fn48h.h` from Components folder
    – `fonts.h` from Utilities/Fonts folder
14. Copy `fonts24.c` from Utilities/Fonts folder to the Src folder.
15. Check that no problem happened by rebuilding all files. There must be no error and no warning.

8.3.2.8 *Modifications in main.c file*

1. Update `main.c` by inserting some instructions to include the needed files in the adequate space (indicated in bold below). This task provides the project modification and regeneration without losing the user code.

```
/* USER CODE BEGIN Includes */
#include "stm32746g_discovery.h"
#include "stm32746g_discovery_sdram.h"
#include "ov9655.h"
#include "rk043fn48h.h"
#include "fonts.h"
/* USER CODE END Includes */
```

Some variable declarations must then be inserted in the adequate space indicated in bold below.

```
/* USER CODE BEGIN PV */
/* Private variables ---------------------------------------------------------*/
typedef enum
{
CAMERA_OK = 0x00,
CAMERA_ERROR = 0x01,
CAMERA_TIMEOUT = 0x02,
CAMERA_NOT_DETECTED = 0x03,
CAMERA_NOT_SUPPORTED = 0x04
} Camera_StatusTypeDef;
typedef struct
{
uint32_t TextColor;
uint32_t BackColor;
sFONT *pFont;
}LCD_DrawPropTypeDef;
typedef struct
{
int16_t X;
int16_t Y;
}Point, * pPoint;
static LCD_DrawPropTypeDef DrawProp[2];
LTDC_HandleTypeDef hltdc;
LTDC_LayerCfgTypeDef layer_cfg;
static RCC_PeriphCLKInitTypeDef periph_clk_init_struct;
CAMERA_DrvTypeDef *camera_driv;
/* Camera module I2C HW address */
static uint32_t CameraHwAddress;
/* Image size */
uint32_t Im_size = 0;
/* USER CODE END PV */
```

The function prototypes must also be inserted in the adequate space indicated in bold below.

```
/* USER CODE BEGIN PFP */
/* Private function prototypes -----------------------------------------------*/
uint8_t CAMERA_Init(uint32_t );
static void LTDC_Init(uint32_t , uint16_t , uint16_t , uint16_t, uint16_t
);
void LCD_GPIO_Init(LTDC_HandleTypeDef *, void *);
/* USER CODE END PFP */
```

2. Update `main()` function by inserting some functions in the adequate space (indicated in bold below).
   - `LTDC_Init` allows the configuration and initialization of the LCD.
   - `BSP_SDRAM_Init` allows the configuration and initialization of the SDRAM.
   - `CAMERA_Init` allows the configuration of the camera module, DCMI registers, and DCMI parameters.
   - One of the two functions `HAL_DCMI_Start_DMA` allowing the DCMI configuration in snapshot or in continuous mode, must be uncommented.

```
/* USER CODE BEGIN 2 */
LTDC_Init(FRAME_BUFFER, 0, 0, 320, 240);
BSP_SDRAM_Init();
CAMERA_Init(CAMERA_R320x240);
HAL_Delay(1000); //Delay for the camera to output correct data
Im_size = 0x9600; //size=320*240*2/4
/* uncomment the following line in case of snapshot mode */
//HAL_DCMI_Start_DMA(&hdcmi, DCMI_MODE_SNAPSHOT, (uint32_t)FRAME_BUFFER, Im_size);
/* uncomment the following line in case of continuous mode */
HAL_DCMI_Start_DMA(&hdcmi, DCMI_MODE_CONTINUOUS , (uint32_t)FRAME_BUFFER, Im_size);
/* USER CODE END 2 */
```

3. Insert the implementation of the new functions (called in the `main()` function), out of the main function, in the adequate space, indicated in bold below.

```
/* USER CODE BEGIN 4 */
void LCD_GPIO_Init(LTDC_HandleTypeDef *hltdc, void *Params)
{
GPIO_InitTypeDef gpio_init_structure;
/* Enable the LTDC and DMA2D clocks */
__HAL_RCC_LTDC_CLK_ENABLE();
__HAL_RCC_DMA2D_CLK_ENABLE();
/* Enable GPIOs clock */
__HAL_RCC_GPIOE_CLK_ENABLE();
__HAL_RCC_GPIOG_CLK_ENABLE();
__HAL_RCC_GPIOI_CLK_ENABLE();
__HAL_RCC_GPIOJ_CLK_ENABLE();
__HAL_RCC_GPIOK_CLK_ENABLE();
/*** LTDC Pins configuration ***/
/* GPIOE configuration */
gpio_init_structure.Pin = GPIO_PIN_4;
gpio_init_structure.Mode = GPIO_MODE_AF_PP;
gpio_init_structure.Pull = GPIO_NOPULL;
gpio_init_structure.Speed = GPIO_SPEED_FAST;
gpio_init_structure.Alternate = GPIO_AF14_LTDC;
HAL_GPIO_Init(GPIOE, &gpio_init_structure);
/* GPIOG configuration */
gpio_init_structure.Pin = GPIO_PIN_12;
gpio_init_structure.Mode = GPIO_MODE_AF_PP;
gpio_init_structure.Alternate = GPIO_AF9_LTDC;
HAL_GPIO_Init(GPIOG, &gpio_init_structure);
/* GPIOI LTDC alternate configuration */
gpio_init_structure.Pin = GPIO_PIN_9 | GPIO_PIN_10 | GPIO_PIN_13 |
GPIO_PIN_14 | GPIO_PIN_15;
gpio_init_structure.Mode = GPIO_MODE_AF_PP;
gpio_init_structure.Alternate = GPIO_AF14_LTDC;
HAL_GPIO_Init(GPIOI, &gpio_init_structure);
/* GPIOJ configuration */
gpio_init_structure.Pin = GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 |
GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7 | GPIO_PIN_5
| GPIO_PIN_6 | GPIO_PIN_7 |GPIO_PIN_8 | GPIO_PIN_9 | GPIO_PIN_10 |
GPIO_PIN_11 | GPIO_PIN_13 | GPIO_PIN_14 | GPIO_PIN_15;
gpio_init_structure.Mode = GPIO_MODE_AF_PP;
gpio_init_structure.Alternate = GPIO_AF14_LTDC;
HAL_GPIO_Init(GPIOJ, &gpio_init_structure);
/* GPIOK configuration */
gpio_init_structure.Pin = GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 |
GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7;
gpio_init_structure.Mode = GPIO_MODE_AF_PP;
gpio_init_structure.Alternate = GPIO_AF14_LTDC;
HAL_GPIO_Init(GPIOK, &gpio_init_structure);
/* LCD_DISP GPIO configuration */
gpio_init_structure.Pin = GPIO_PIN_12; /* LCD_DISP pin has to be
manually controlled */
gpio_init_structure.Mode = GPIO_MODE_OUTPUT_PP;
HAL_GPIO_Init(GPIOI, &gpio_init_structure);
/* LCD_BL_CTRL GPIO configuration */
gpio_init_structure.Pin = GPIO_PIN_3; /* LCD_BL_CTRL pin has to be
manually controlled */
gpio_init_structure.Mode = GPIO_MODE_OUTPUT_PP;
HAL_GPIO_Init(GPIOK, &gpio_init_structure);
}
static void LTDC_Init(uint32_t FB_Address, uint16_t Xpos, uint16_t Ypos,
uint16_t Width, uint16_t Height)
{
/* Timing Configuration */
hltdc.Init.HorizontalSync = (RK043FN48H_HSYNC - 1);
hltdc.Init.VerticalSync = (RK043FN48H_VSYNC - 1);
hltdc.Init.AccumulatedHBP = (RK043FN48H_HSYNC + RK043FN48H_HBP - 1);
hltdc.Init.AccumulatedVBP = (RK043FN48H_VSYNC + RK043FN48H_VBP - 1);
hltdc.Init.AccumulatedActiveH = (RK043FN48H_HEIGHT + RK043FN48H_VSYNC +
RK043FN48H_VBP - 1);
```

```
hltdc.Init.AccumulatedActiveW = (RK043FN48H_WIDTH + RK043FN48H_HSYNC +
RK043FN48H_HBP - 1);
hltdc.Init.TotalHeigh = (RK043FN48H_HEIGHT + RK043FN48H_VSYNC +
RK043FN48H_VBP + RK043FN48H_VFP - 1);
hltdc.Init.TotalWidth = (RK043FN48H_WIDTH + RK043FN48H_HSYNC +
RK043FN48H_HBP + RK043FN48H_HFP - 1);
/* LCD clock configuration */
periph_clk_init_struct.PeriphClockSelection = RCC_PERIPHCLK_LTDC;
periph_clk_init_struct.PLLSAI.PLLSAIN = 192;
periph_clk_init_struct.PLLSAI.PLLSAIR = RK043FN48H_FREQUENCY_DIVIDER;
periph_clk_init_struct.PLLSAIDivR = RCC_PLLSAIDIVR_4;
HAL_RCCEx_PeriphCLKConfig(&periph_clk_init_struct);
/* Initialize the LCD pixel width and pixel height */
hltdc.LayerCfg->ImageWidth = RK043FN48H_WIDTH;
hltdc.LayerCfg->ImageHeight = RK043FN48H_HEIGHT;
hltdc.Init.Backcolor.Blue = 0;/* Background value */
hltdc.Init.Backcolor.Green = 0;
hltdc.Init.Backcolor.Red = 0;
/* Polarity */
hltdc.Init.HSPolarity = LTDC_HSPOLARITY_AL;
hltdc.Init.VSPolarity = LTDC_VSPOLARITY_AL;
hltdc.Init.DEPolarity = LTDC_DEPOLARITY_AL;
hltdc.Init.PCPolarity = LTDC_PCPOLARITY_IPC;
hltdc.Instance = LTDC;
if(HAL_LTDC_GetState(&hltdc) == HAL_LTDC_STATE_RESET)
{
LCD_GPIO_Init(&hltdc, NULL);
}
HAL_LTDC_Init(&hltdc);
/* Assert display enable LCD_DISP pin */
HAL_GPIO_WritePin(GPIOI, GPIO_PIN_12, GPIO_PIN_SET);
/* Assert backlight LCD_BL_CTRL pin */
HAL_GPIO_WritePin(GPIOK, GPIO_PIN_3, GPIO_PIN_SET);
DrawProp[0].pFont = &Font24 ;
/* Layer Init */
layer_cfg.WindowX0 = Xpos;
layer_cfg.WindowX1 = Width;
layer_cfg.WindowY0 = Ypos;
layer_cfg.WindowY1 = Height;
layer_cfg.PixelFormat = LTDC_PIXEL_FORMAT_RGB565;
layer_cfg.FBStartAdress = FB_Address;
layer_cfg.Alpha = 255;
layer_cfg.Alpha0 = 0;
layer_cfg.Backcolor.Blue = 0;
layer_cfg.Backcolor.Green = 0;
layer_cfg.Backcolor.Red = 0;
layer_cfg.BlendingFactor1 = LTDC_BLENDING_FACTOR1_PAxCA;
layer_cfg.BlendingFactor2 = LTDC_BLENDING_FACTOR2_PAxCA;
layer_cfg.ImageWidth = Width;
layer_cfg.ImageHeight = Height;
HAL_LTDC_ConfigLayer(&hltdc, &layer_cfg, 1);
DrawProp[1].BackColor = ((uint32_t)0xFFFFFFFF);
DrawProp[1].pFont = &Font24;
DrawProp[1].TextColor = ((uint32_t)0xFF000000);
}
uint8_t CAMERA_Init(uint32_t Resolution) /*Camera initialization*/
{
uint8_t status = CAMERA_ERROR;
/* Read ID of Camera module via I2C */
if(ov9655_ReadID(CAMERA_I2C_ADDRESS) == OV9655_ID)
{
camera_driv = &ov9655_drv;/* Initialize the camera driver structure */
CameraHwAddress = CAMERA_I2C_ADDRESS;
if (Resolution == CAMERA_R320x240)
{
camera_driv->Init(CameraHwAddress, Resolution);
HAL_DCMI_DisableCROP(&hdcmi);
}
status = CAMERA_OK; /* Return CAMERA_OK status */
}
```

```
else
{
status = CAMERA_NOT_SUPPORTED; /* Return CAMERA_NOT_SUPPORTED status */
}
return status;
}
/* USER CODE END 4 */
```

8.3.2.9 *Modifications in main.h file*

Update main.h by inserting the frame buffer address declaration in the adequate space, indicated in bold below.

```
/* USER CODE BEGIN Private defines */
#define FRAME_BUFFER 0xC0000000
/* USER CODE END Private defines */
```

At this stage, the user can build, debug, and run the project.

8.3.3 RGB data capture and display

To simplify this example, data are captured and displayed in RGB565 format (2 bpp). The image resolution is 320x240 (QVGA). The frame buffer is placed in the SDRAM. Camera and LCD data are located in the same frame buffer. The LCD displays then directly the data captured through the DCMI without any processing. The camera module is configured to output RGB565 data, QVGA (320x240).

The configuration of this example can be done by following the steps described in Section 8.3.2 .

8.3.4 YCbCr data capture

This implementation example aims to receive YCbCr data from the camera module, and to transfer them to the SDRAM.

Displaying the YCbCr received data on the LCD (configured to display RGB565 data in the previous configuration) is not correct, but can be used for verification.

To display images correctly, YCbCr data must be converted into RGB565 data (or RGB888 or ARGB8888, depending on the application needs).

All the configuration steps signaled in Section 8.3.2 must be followed.

Some instructions must be added to obtain YCbCr data. Only the camera configuration has to be updated by adding:

- a table of constants allowing the configuration of camera module registers
- a new function used to configure the camera module by sending the register configuration through the I2C.

1. Add the table containing the configuration of camera module registers in `main.c`, below `/* Private variables ----------*/`.

```
const unsigned char OV9655_YUV_QVGA [ ][2]=
{ { 0x12, 0x80 },{ 0x00, 0x00 },{ 0x01, 0x80 },{ 0x02, 0x80 },{ 0x03, 0x02
},{ 0x04, 0x03 },{ 0x0e, 0x61 }, { 0x0f, 0x42 },{ 0x11, 0x01 },{ 0x12, 0x62
},{ 0x13, 0xe7 },{ 0x14, 0x3a },{ 0x16, 0x24 },{ 0x17, 0x18 }, { 0x18, 0x04
},{ 0x19, 0x01 },{ 0x1a, 0x81 } ,{ 0x1e, 0x04 },{ 0x24, 0x3c },{ 0x25, 0x36
},{ 0x26, 0x72 }, { 0x27, 0x08 },{ 0x28, 0x08 },{ 0x29, 0x15 },{ 0x2a, 0x00
},{ 0x2b, 0x00 },{ 0x2c, 0x08 },{ 0x32, 0x24 }, { 0x33, 0x00 },{ 0x34, 0x3f
},{ 0x35, 0x00 },{ 0x36, 0x3a },{ 0x38, 0x72 },{ 0x39, 0x57 } ,{ 0x3a, 0x0c
},{ 0x3b, 0x04 },{ 0x3d, 0x99 },{ 0x3e, 0x0e },{ 0x3f, 0xc1 },{ 0x40, 0xc0}
,{ 0x41, 0x01 },{ 0x42, 0xc0 },{ 0x43, 0x0a },{ 0x44, 0xf0 },{ 0x45, 0x46
},{ 0x46, 0x62} ,{ 0x47, 0x2a },{ 0x48, 0x3c },{ 0x4a, 0xfc }, { 0x4b, 0xfc
},{ 0x4c, 0x7f },{ 0x4d, 0x7f}, { 0x4e, 0x7f },{ 0x52, 0x28 },{ 0x53, 0x88
},{ 0x54, 0xb0 }, { 0x4f, 0x98 },{ 0x50, 0x98} ,{ 0x51, 0x00 },{ 0x58, 0x1a
},{ 0x59, 0x85 },{ 0x5a, 0xa9 },{ 0x5b, 0x64 } ,{ 0x5c, 0x84 },{ 0x5d, 0x53
},{ 0x5e, 0x0e },{ 0x5f, 0xf0 },{ 0x60, 0xf0 },{ 0x61, 0xf0 } ,{ 0x62, 0x00
}, { 0x63, 0x00 },{ 0x64, 0x02 },{ 0x65, 0x20 },{ 0x66, 0x00 },{ 0x69, 0x0a
},{ 0x6b, 0x5a },{ 0x6c, 0x04 }, { 0x6d, 0x55 },{ 0x6e, 0x00 },{ 0x6f, 0x9d
},{ 0x70, 0x21 },{ 0x71, 0x78 },{ 0x72, 0x11 },{ 0x73, 0x01 }, { 0x74, 0x10
},{ 0x75, 0x10 } ,{ 0x76, 0x01 },{ 0x77, 0x02 },{ 0x7a, 0x12 },{ 0x7b, 0x08
},{ 0x7c, 0x15 }, { 0x7d, 0x24 },{ 0x7e, 0x45 },{ 0x7f, 0x55 },{ 0x80, 0x6a
},{ 0x81, 0x78 },{ 0x82, 0x87 },{ 0x83, 0x96 }, { 0x84, 0xa3 },{ 0x85, 0xb4
},{ 0x86, 0xc3 },{ 0x87, 0xd6 },{ 0x88, 0xe6 } ,{ 0x89, 0xf2 },{ 0x8a, 0x24
}, { 0x8c, 0x80 },{ 0x90, 0x7d },{ 0x91, 0x7b },{ 0x9d, 0x02 } ,{ 0x9e, 0x02
},{ 0x9f, 0x7a },{ 0xa0, 0x79 }, { 0xa1, 0x40 },{ 0xa4, 0x50 },{ 0xa5, 0x68
},{ 0xa6, 0x4a },{ 0xa8, 0xc1 },{ 0xa9, 0xef },{ 0xaa, 0x92 }, { 0xab, 0x04
} ,{ 0xac, 0x80 },{ 0xad, 0x80 },{ 0xae, 0x80 },{ 0xaf, 0x80 },{ 0xb2, 0xf2
},{ 0xb3, 0x20 } ,{ 0xb4, 0x20 },{ 0xb5, 0x00 },{ 0xb6, 0xaf },{ 0xbb, 0xae
},{ 0xbc, 0x7f },{ 0xbd, 0x7f } ,{ 0xbe, 0x7f },{ 0xbf, 0x7f },{ 0xc0, 0xaa
},{ 0xc1, 0xc0 },{ 0xc2, 0x01 },{ 0xc3, 0x4e } ,{ 0xc6, 0x05 },{ 0xc7, 0x81
},{ 0xc9, 0xe0 },{ 0xca, 0xe8 },{ 0xcb, 0xf0 },{ 0xcc, 0xd8 } ,{ 0xcd, 0x93
}, { 0xcd, 0x93 },{ 0xFF, 0xFF } };
```

2. The new function prototype has to be inserted below `/* Private function prototypes ------*/`.

```
void OV9655_YUV_Init (uint16_t );
```

3. The second step of modifications in `main.c` described in Section 8.3.2.8 has to be updated. Modify the `main()` function by inserting the following functions in the adequate space, indicated in bold below. One of the two functions allowing the DCMI configuration in snapshot or in continuous mode must be uncommented.

```
/* USER CODE BEGIN 2 */
BSP_SDRAM_Init();
CAMERA_Init(CameraHwAddress);
OV9655_YUV_Init(CameraHwAddress);
HAL_Delay(1000); //Delay for the camera to output correct data
Im_size = 0x9600; //size=320*240*2/4
/* uncomment the following line in case of snapshot mode */
//HAL_DCMI_Start_DMA(&hdcmi, DCMI_MODE_SNAPSHOT, (uint32_t)FRAME_BUFFER, Im_size);
/* uncomment the following line in case of continuous mode */
HAL_DCMI_Start_DMA(&hdcmi, DCMI_MODE_CONTINUOUS , (uint32_t)FRAME_BUFFER, Im_size);
/* USER CODE END 2 */
```

4. The third step of modifications in `main.c` described in Section 8.3.2.8 has to be updated by adding the new function implementation.

```
void OV9655_YUV_Init(uint16_t DeviceAddr)
{ uint32_t index;
for(index=0; index<(sizeof(OV9655_YUV_QVGA)/2); index++)
{ CAMERA_IO_Write(DeviceAddr, OV9655_YUV_QVGA[index][0],
OV9655_YUV_QVGA[index][1]);
CAMERA_Delay(1);
} }
```

8.3.5 Y only data capture

In this example, the camera module is configured to output YCbCr data format. By using the byte select feature on the DCMI side, the chrominance components (Cb and Cr) are ignored: only the Y component is transferred to the frame buffer in the SDRAM.

All the configuration steps signaled in Section 8.3.2 must be followed.

Some instructions must be added to obtain the Y only data. Only the camera and the DCMI configuration must be updated. To simplify this task, `main.c` must be modified as described in Section 8.3.4 , but the second step of STM32CubeMX - DCMI control signals and capture mode configuration, or the static `void MX_DCMI_Init(void)` function (this function is implemented in `main.c`) must be modified..

```
hdcmi.Instance = DCMI;
hdcmi.Init.SynchroMode = DCMI_SYNCHRO_HARDWARE;
hdcmi.Init.PCKPolarity = DCMI_PCKPOLARITY_RISING;
hdcmi.Init.VSPolarity = DCMI_VSPOLARITY_HIGH;
hdcmi.Init.HSPolarity = DCMI_HSPOLARITY_LOW;
hdcmi.Init.CaptureRate = DCMI_CR_ALL_FRAME;
hdcmi.Init.ExtendedDataMode = DCMI_EXTEND_DATA_8B;
hdcmi.Init.ByteSelectMode = DCMI_BSM_OTHER;
hdcmi.Init.ByteSelectStart = DCMI_OEBS_EVEN;
hdcmi.Init.LineSelectMode = DCMI_LSM_ALL;
hdcmi.Init.LineSelectStart = DCMI_OELS_ODD;
```

8.3.6 SXGA resolution capture (YCbCr data format)

This implementation example aims to receive YCbCr data from the camera module, and to transfer them to the SDRAM. The captured image resolution is SXGA (1280x1024).

To display images correctly, YCbCr data must be converted into RGB565 data (or RGB888 or ARGB8888, depending on the application needs).

All the configuration steps details in Section 8.3.2 must be followed.

Some instructions must be added to obtain the YCbCr data. Only the DMA and the camera module configuration have to be updated.

DMA configuration

The DMA is configured as described in Section 6.4.9 DMA configuration for higher resolutions. The `HAL_DMA_START` function ensures this configuration because the image size exceeds the maximum allowed size for double-buffer mode.

Calling `HAL_DMA_START` ensures the division of the received frames to equal parts, and the placement of each part in one frame buffer. As explained, for the SXGA resolution, each frame is divided in 16 frame buffers. Each buffer size is equal to 40960 words.

For the buffers addresses, `HAL_DMA_START` ensures the placement of the 16 frame buffers in the memory. In this case, the address of the first frame buffer is 0xC0000000. The second address is then 0xC0163840 (0xC0000000 + (40960 * 4)). The 16th frame buffer address is (0xC0000000 + 16 * (40960 * 4)).

Each end of transfer, the DMA has filled one frame, an interrupt is generated, the address of the next buffer is calculated, and one pointer is modified as illustrated in Figure 46. DMA operation in high resolution case.

Camera module configuration

The new camera module configuration is done by adding:

- a table of constants allowing the configuration of camera module registers
- a new function used to configure the camera module by sending the register configuration through the I2C

To ensure that the camera module sends image having SXGA resolution and YCbCr format, the CMOS sensor registers must be configured with the following steps:

1. Add the table containing the configuration of camera module registers in `main.c` below `/* Private variables ----------*/`.

```
const unsigned char ov9655_yuv_sxga[][2]= {
{ 0x12, 0x80 },{ 0x00, 0x00 },{ 0x01, 0x80 },{ 0x02, 0x80 },{ 0x03, 0x1b },{
0x04, 0x03 }, { 0x0e, 0x61 },{ 0x0f, 0x42 },{ 0x11, 0x00 },{ 0x12, 0x02 },{
0x13, 0xe7 },{ 0x14, 0x3a },{ 0x16, 0x24 }, { 0x17, 0x1d },{ 0x18, 0xbd },{
0x19, 0x01 },{ 0x1a, 0x81 }, { 0x1e, 0x04 }, { 0x24, 0x3c }, { 0x25, 0x36
},{ 0x26, 0x72 }, { 0x27, 0x08 }, { 0x28, 0x08 },{ 0x29, 0x15 },{ 0x2a, 0x00
},{ 0x2b, 0x00 },{ 0x2c, 0x08 },{ 0x32, 0xff },{ 0x33, 0x00 },{ 0x34, 0x3d
},{ 0x35, 0x00 },{ 0x36, 0xf8 },{ 0x38, 0x72 },{ 0x39, 0x57 }, { 0x3a, 0x0c
},{ 0x3b, 0x04 },{ 0x3d, 0x99 }, { 0x3e, 0x0c },{ 0x3f, 0xc1 },{ 0x40, 0xd0
},{ 0x41, 0x00 },{ 0x42, 0xc0 },{ 0x43, 0x0a },{ 0x44, 0xf0 },{ 0x45, 0x46
},{ 0x46, 0x62 }, { 0x47, 0x2a }, { 0x48, 0x3c },{ 0x4a, 0xfc },{ 0x4b, 0xfc
},{ 0x4c, 0x7f },{ 0x4d, 0x7f },{ 0x4e, 0x7f },{ 0x52, 0x28 },{ 0x53, 0x88
},{ 0x54, 0xb0 },{ 0x4f, 0x98 },{ 0x50, 0x98 },{ 0x51, 0x00 },{ 0x58, 0x1a
},{ 0x58, 0x1a },{ 0x59, 0x85 },{ 0x5a, 0xa9 },{ 0x5b, 0x64 },{ 0x5c, 0x84
},{ 0x5d, 0x53 },{ 0x5e, 0x0e }, { 0x5f, 0xf0 }, { 0x60, 0xf0 }, { 0x61,
0xf0 },{ 0x62, 0x00 }, { 0x63, 0x00 }, { 0x64, 0x02 },{ 0x65, 0x16 },{ 0x66,
0x01 },{ 0x69, 0x02 },{ 0x6b, 0x5a }, { 0x6c, 0x04 }, { 0x6d, 0x55 }, {
0x6e, 0x00 },{ 0x6f, 0x9d },{ 0x70, 0x21 }, { 0x71, 0x78 },{ 0x72, 0x00 },{
0x73, 0x01 },{ 0x74, 0x3a },{ 0x75, 0x35 },{ 0x76, 0x01 },{ 0x77, 0x02 },{
0x7a, 0x12 },{ 0x7b, 0x08 }, { 0x7c, 0x15 }, { 0x7d, 0x24 },{ 0x7e, 0x45 },{
0x7f, 0x55 },{ 0x80, 0x6a },{ 0x81, 0x78 },{ 0x82, 0x87 },{ 0x83, 0x96 },{
0x84, 0xa3 },{ 0x85, 0xb4 }, { 0x86, 0xc3 },{ 0x87, 0xd6 },{ 0x88, 0xe6 },
{ 0x89, 0xf2 },{ 0x8a, 0x03 }, { 0x8c, 0x0d }, { 0x90, 0x7d }, { 0x91, 0x7b
}, { 0x9d, 0x03 },{ 0x9e, 0x04 }, { 0x9f, 0x7a }, { 0xa0, 0x79 }, { 0xa1,
0x40 }, { 0xa4, 0x50 },{ 0xa5, 0x68 }, { 0xa6, 0x4a }, { 0xa8, 0xc1 },{
0xa9, 0xef }, { 0xaa, 0x92 },{ 0xab, 0x04 },{ 0xac, 0x80 },{ 0xad, 0x80 },{
0xae, 0x80 },{ 0xaf, 0x80 },{ 0xb2, 0xf2 },{ 0xb3, 0x20 },{ 0xb4, 0x20 },{
0xb5, 0x00 },{ 0xb6, 0xaf },{ 0xbb, 0xae },{ 0xbc, 0x7f },{ 0xbd, 0x7f },{
0xbe, 0x7f }, { 0xbf, 0x7f },{ 0xc0, 0xe2 },{ 0xc1, 0xc0 },{ 0xc2, 0x01 },
{ 0xc3, 0x4e }, { 0xc6, 0x05 },{ 0xc7, 0x80 }, { 0xc9, 0xe0 },{ 0xca, 0xe8
}, { 0xcb, 0xf0 },{ 0xcc, 0xd8 },{0xcd, 0x93} ,{ 0xFF, 0xFF } };
```

2. The new function prototype has to be inserted below `/* Private function prototypes -----------*/` .

```
void OV9655_YUV_Init (uint16_t );
```

3. The second step of modifications in `main.c` in this example is to update the `main()` function by inserting the following functions in the adequate space (indicated in bold below). One of the two functions allowing the DCMI configuration in snapshot or in continuous mode must be uncommented.

```
/* USER CODE BEGIN 2 */
BSP_SDRAM_Init();
CAMERA_Init(CameraHwAddress);
OV9655_YUV_Init(CameraHwAddress);
HAL_Delay(1000); //Delay for the camera to output correct data
Im_size = 0xA0000; //size=1280*1024*2/4
/* uncomment the following line in case of snapshot mode */
//HAL_DCMI_Start_DMA(&hdcmi, DCMI_MODE_SNAPSHOT, (uint32_t)FRAME_BUFFER,
Im_size);
/* uncomment the following line in case of continuous mode */
HAL_DCMI_Start_DMA(&hdcmi, DCMI_MODE_CONTINUOUS , (uint32_t)FRAME_BUFFER,
Im_size);
/* USER CODE END 2 */
```

4. The third step of modifications in `main.c` described in Section 8.3.2.8 has to be updated by adding the new function implementation below `/* USER CODE BEGIN 4 */`.

```
void OV9655_YUV_Init(uint16_t DeviceAddr)
{
uint32_t index;
for(index=0; index<(sizeof(ov9655_yuv_sxga)/2); index++)
{
CAMERA_IO_Write(DeviceAddr, ov9655_yuv_sxga[index][0],
ov9655_yuv_sxga[index][1]);
CAMERA_Delay(1);
}
}
```

Note: In case of SXGA frame with RGB data format, the user can reduce the resolution to display the received images on the LCD_TFT by using the resizing feature of the DCMI.

8.3.7 JPEG data capture

The OV9655 CMOS sensor embedded in the STM32F4DIS-Cam board does not support the compressed output data. This example is then implemented using OV2640 CMOS sensor, supporting the 8-bit format compressed data.

This example is based on the STM324x9I-EVAL (REV B) board embedding the OV2640 CMOS sensor (MB1066).

The compressed data (JPEG) must be uncompressed to have YCbCr data, and converted to RGB to be displayed, but this implementation example aims only to receive JPEG data through the DCMI, and to store them in the SDRAM.

This example is developed based on the DCMI example (SnapshotMode) provided within the STM32CubeF4 firmware, located in `Projects\STM324x9I_EVAL\Examples\DCMI\DCMI_SnapshotMode`. The provided example, aims to capture one RGB frame (QVGA resolution), and to display it on the LCD-TFT, having the following configuration:

- The DCMI and I2C GPIOs are configured as described in Section 8.3.2 .
- The system clock runs at 180 MHz.
- SDRAM clock runs at 90 MHz.
- The DCMI is configured to capture 8-bit data width in hardware synchronization (uncompressed data).
- The camera module is configured to output RGB data images with QVGA resolution.

Based on this example, to be able to capture JPEG data, the user needs to modify the DCMI and the camera module configuration.

DCMI configuration

The DCMI needs to be configured to receive compressed data (JPEG) by setting the JPEG bit in DCMI_CR. To set this bit, the user must simply add the instruction written in bold below in the `stm324x9i_eval_camera.c` file in `uint8_t BSP_CAMERA_Init(uint32_t Resolution)` (this function is called in `main()` to configure the DCMI and the camera module). The DCMI previous configuration is kept.

```
phdcmi->Init.CaptureRate = DCMI_CR_ALL_FRAME;
phdcmi->Init.HSPolarity = DCMI_HSPOLARITY_LOW;
phdcmi->Init.SynchroMode = DCMI_SYNCHRO_HARDWARE;
phdcmi->Init.VSPolarity = DCMI_VSPOLARITY_LOW;
phdcmi->Init.ExtendedDataMode = DCMI_EXTEND_DATA_8B;
phdcmi->Init.PCKPolarity = DCMI_PCKPOLARITY_RISING;
phdcmi->Init.JPEGMode = DCMI_JPEG_ENABLE;
```

Camera module configuration

The configuration of CMOS sensor (ov2640) registers must be inserted in the `ov2640.c` file.

```
const unsigned char OV2640_JPEG[][2]=
{ {0xff, 0x00},{0x2c, 0xff},{0x2e, 0xdf},{0xff, 0x01},{0x12,
0x80},{0x3c, 0x32},{0x11, 0x00},{0x09,0x02},{0x04, 0x28},{0x13,
0xe5},{0x14, 0x48},{0x2c, 0x0c},{0x33, 0x78},{0x3a, 0x33},{0x3b,
0xfb},{0x3e, 0x00},{0x43, 0x11},{0x16, 0x10},{0x39, 0x02},{0x35,
0x88},{0x22, 0x0a},{0x37, 0x40},{0x23, 0x00},{0x34,
0xa0},{0x36,0x1a},{0x06, 0x02},{0x07, 0xc0}, {0x0d, 0xb7},{0x0e,
0x01},{0x4c, 0x00},{0x4a, 0x81},{0x21, 0x99},{0x24, 0x40},{0x25,
0x38},{0x26, 0x82}, {0x5c, 0x00},{0x63, 0x00},{0x46, 0x3f},{0x61,
0x70},{0x62, 0x80},{0x7c, 0x05}, {0x20, 0x80},{0x28, 0x30},{0x6c,
0x00},{0x6d, 0x80},{0x6e, 0x00},{0x70, 0x02},{0x71,0x94},{0x73,
0xc1},{0x3d, 0x34},{0x5a, 0x57},{0x4f, 0xbb},{0x50, 0x9c},{0xff,
0x00},{0xe5, 0x7f},{0xf9, 0xc0},{0x41, 0x24},{0xe0, 0x14},{0x76,
0xff},{0x33, 0xa0},{0x42, 0x20},{0x43, 0x18},{0x4c, 0x00},{0x87,
0xd0},{0x88, 0x3f},{0xd7, 0x03},{0xd9, 0x10},{0xd3, 0x82},{0xc8,
0x08},{0xc9, 0x80},{0x7c, 0x00}, {0x7d, 0x00},{0x7c, 0x03},{0x7d,
0x48},{0x7d, 0x48},{0x7c,0x08},{0x7d, 0x20},{0x7d, 0x10},{0x7d,
0x0e},{0x90, 0x00},{0x91, 0x0e},{0x91, 0x1a},{0x91, 0x31},{0x91,
0x5a},{0x91, 0x69},{0x91, 0x75},{0x91, 0x7e},{0x91, 0x88},{0x91,
0x8f},{0x91, 0x96}, {0x91, 0xa3},{0x91, 0xaf},{0x91, 0xc4},{0x91,
0xd7},{0x91, 0xe8},{0x91, 0x20},{0x92, 0x00},{0x93, 0x06},{0x93,
0xe3},{0x93, 0x05},{0x93, 0x05},{0x93, 0x00},{0x93, 0x04},{0x93,
0x00},{0x93, 0x00},{0x93, 0x00},{0x93, 0x00},{0x93, 0x00},{0x93,
0x00},{0x93, 0x00},{0x96, 0x00},{0x97, 0x08},{0x97, 0x19},{0x97,
0x02},{0x97, 0x0c},{0x97, 0x24},{0x97, 0x30},{0x97, 0x28},{0x97,
0x26},{0x97, 0x02},{0x97, 0x98},{0x97, 0x80},{0x97, 0x00},{0x97,
0x00},{0xc3, 0xed},{0xc5, 0x11},{0xc6, 0x51},{0xbf, 0x80},{0xc7,
0x00},{0xb6, 0x66},{0xb8, 0xA5},{0xb7, 0x64},{0xb9, 0x7C},{0xb3,
0xaf},{0xb4, 0x97},{0xb5, 0xFF},{0xb0, 0xC5},{0xb1, 0x94},{0xb2,
0x0f},{0xc4, 0x5c},{0xc0, 0xc8},{0xc1, 0x96},{0x86, 0x1d},{0x50,
0x00},{0x51, 0x90},{0x52, 0x18}, {0x53, 0x00},{0x54, 0x00},{0x55,
0x88},{0x57, 0x00},{0x5a, 0x90},{0x5b, 0x18}, {0x5c, 0x05},{0xc3,
0xed},{0x7f, 0x00},{0xda, 0x00},{0xe5, 0x1f},{0xe1, 0x77},{0xe0,
0x00},{0xdd, 0x7f},{0x05, 0x00},{0xFF, 0x00},{0x05, 0x00},{0xDA,
0x10},{0xD7, 0x03},{0xDF, 0x00},{0x33, 0x80},{0x3C, 0x40}, {0xe1, 0x77},
{0x00, 0x00} };
```

To modify the camera module registers, the previous table must be sent to the camera through the I2C. In `ov2640.c`, in `void ov2640_Init(uint16_t DeviceAddr, uint32_t resolution)`, replace:

```
case CAMERA_R320x240:
{
for(index=0; index<(sizeof(OV2640_QVGA)/2); index++)
{
CAMERA_IO_Write(DeviceAddr, OV2640_QVGA[index][0],
OV2640_QVGA[index][1]);
CAMERA_Delay(1);
}
break;
}
```

by

```
case CAMERA_R320x240:
{
for(index=0; index<(sizeof( OV2640_JPEG)/2); index++)
{
CAMERA_IO_Write(DeviceAddr, OV2640_JPEG[index][0],
OV2640_JPEG[index][1]);
CAMERA_Delay(1);
}
break;
}
```

9 Supported devices

To know if a CMOS sensor (a camera module) is compatible with the DCMI or not, the user must check the following points in the CMOS sensor specifications:

- parallel interface (8-, 10-, 12-, or 14-bit)
- control signals (VSYNC, HSYNC, and PIXCLK)
- supported pixel clock frequency output
- supported data output

There is a wide range of camera modules and CMOS sensors that are compatible with the STM32 DCMI. The table below lists some of them.

Table 14. Examples of supported camera modules

CMOS sensor	Camera module	Formats	Parallel interface
MT9D111	ArduCAM 2 Mpixels	RGB, YCbCr, JPEG	10-bit
MT9P111	5 Mpixels	RGB, YCbCr, JPEG	8-bit
NT99141	ArduCAM 1 Mpixel	RGB, YCbCr, JPEG	8-bit
OV2640	ArduCAM 2 Mpixels	RGB, YCbCr, JPEG	8-bit, 10-bit
OV3660	3 Mpixels	RGB, YCbCr	8-bit, 10-bit
OV5640	ArduCAM 5 Mpixels	RGB, YCbCr, JPEG	8-bit, 10-bit
OV5642	ArduCAM 5 Mpixels	RGB, YCbCr, JPEG	8-bit, 10-bit
OV9655	ArduCAM 1.3 Mpixels	RGB, YCbCr	8-bit
S5k4ECGX	5 Mpixels	RGB, JPEG	10-bit
S5k5CAGA	3 Mpixels	RGB, JPEG	10-bit

10 Conclusion

The DCMI represents an efficient interface to connect the camera modules to the STM32 MCUs supporting high speed, high resolutions, and a variety of data formats/widths.

Together with the variety of peripherals and interfaces integrated in STM32 MCUs and benefiting from the STM32 smart architecture, the DCMI can be used in large and sophisticated imaging applications.

This application note covers the DCMI across the STM32 MCUs, providing all the necessary information to correctly use the DCMI, and to succeed in implementing applications starting from the compatible camera module selection to detailed examples implementation.

Revision history

Table 15. **Document revision history**

Date	Version	Changes
3-Aug-2017	1	Initial release.
17-Oct-2022	2	Updated: • Table 1. Applicable products • Table 2. DCMI and related resources availability • Section 3.2 DCMI in a smart architecture with all new products • Table 5. DCMI and camera modules on STM32 boards • Section 5.6.4 YCbCr, Y only • Section 5.8 Image resizing (resolution modification) • Table 6. DCMI operation in low-power modes • Section 6.4.1 DMA configuration for DCMI-to-memory transfers • Table 7. DMA stream selection across STM32 devices • Section 6.4.4 DMA_SxNDTR/DMA_CNDTRx/GPDMA_CxBR1 register • Table 12. Maximum data flow at maximum DCMI_PIXCLK • Table 13. STM32Cube DCMI examples • Section 8.3.2 Configuration of common examples • Table 14. Examples of supported camera modules

Contents

List of tables

List of figures