AN5020
Application note

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).

Nowadays, high quality, ease-of-use, power efficiency, high level of integration, fast time-to-market and cost effectiveness are required in imaging applications.

To meet these requirements, STM32 MCUs embed a digital camera interface (DCMI), allowing connection to efficient parallel camera modules.

In addition, 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 (SPI, UART, I2C, SDIO, USB, ETHERNET, I2S...), a rich graphical portfolio (LTDC, QSPI, DMA2D,...) and an industry-leading development environment ensuring sophisticated applications and connectivity solutions (IOT).

This application note gives STM32 users a grasp of 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.

Reference documents

This application note should be read in conjunction with the reference manuals of the STM32F2, STM32F4, STM32F7 Series and STM32L4x6, STM32H7x3 lines:

- *STM32F205xx, STM32F207xx, STM32F215xx and STM32F217xx advanced ARM®-based 32-bit MCUs* (RM0033)
- *STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM®-based 32-bit MCUs* (RM0090)
- *STM32F446xx advanced ARM®-based 32-bit MCUs* (RM0390)
- *STM32F469xx and STM32F479xx advanced ARM®-based 32-bit MCUs* (RM0386)
- *STM32F75xxx and STM32F74xxx advanced ARM®-based 32-bit MCUs* (RM0385)
- *STM32F76xxx and STM32F77xxx advanced ARM®-based 32-bit MCUs* (RM0410)
- *STM32L4x5 and STM32L4x6 advanced ARM®-based 32-bit MCUs* (RM0351)
- *STM32H7x3 advanced ARM®-based 32-bit MCUs* (RM0433)

Table 1. Applicable products

Type	STM32 lines
STM32F2 Series	STM32F2x7
STM32F4 Series	STM32F407/417, STM32F427/437, STM32F429/439, STM32F446, STM32F469/479

Table 1. Applicable products (continued)

Type	STM32 lines
STM32F7 Series	STM32F7x5, STM32F7x6, STM32F7x7, STM32F7x8, STM32F7x9
STM32L4 Series	STM32L4x6
STM32H7 Series	STM32H7x3

Contents

List of tables

List of figures

1 Overview: 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.

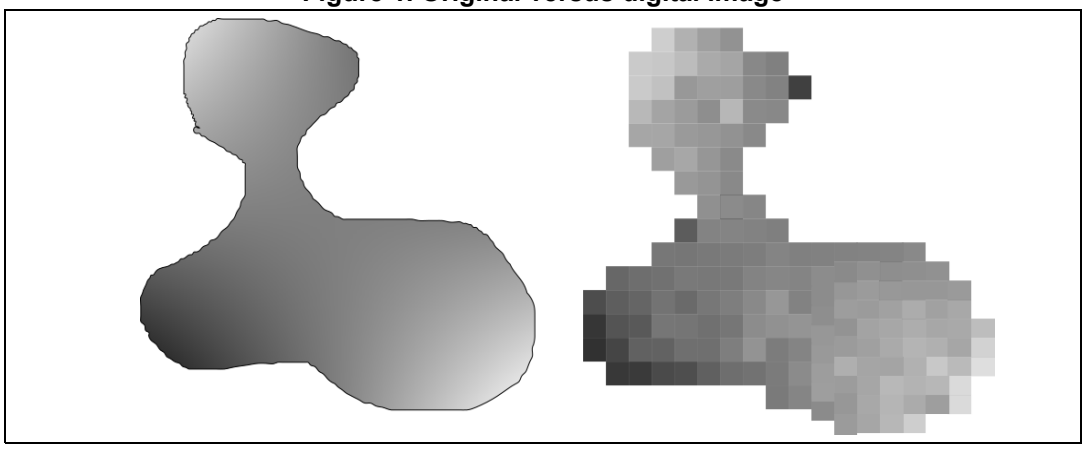
1.1 Imaging basic concepts

This section provides a small introduction to 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. It can also be referred by bit per pixel (bpp).

  Examples:

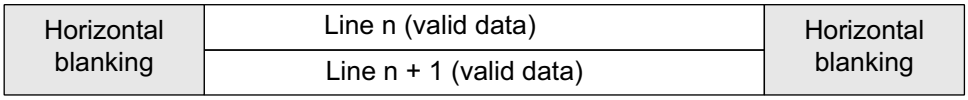
  – For bitonal image, each pixel comprises one bit. Each pixel is either black or white (0 or 1).

  – For gray scale, the image is most of the time composed of 2 bpp (each pixel can have one of four gray levels) to 8 bbp (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.

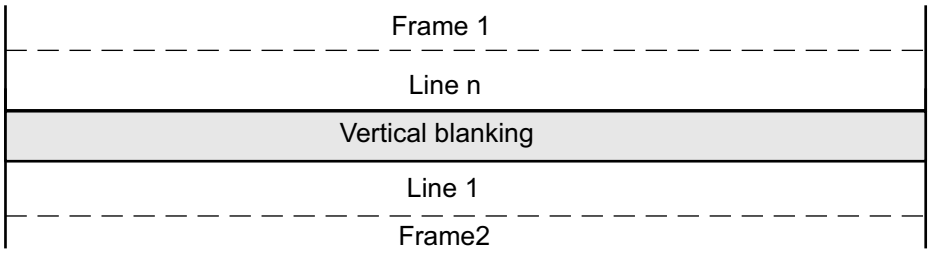
Figure 2. Horizontal blanking illustration



MSv46626V1

- **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



MSv46627V1

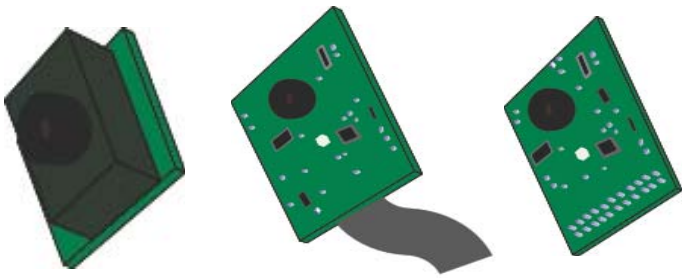
- **Progressive scan**: It is a manner of dealing with moving images. It allows to draw the lines one after the other in sequence, without separating the odd lines from the even ones as for interlaced scan. To construct the image:
  - in progressive scan, the first line is drawn, then the second, then the third.
  - In interlaced scan, each frame is divided into two fields, odd and even lines. The two fields are displayed alternatively.

1.2 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 modules



MSv46622V1

1.2.1 Camera module components

The four components of a camera module are described below:

Image sensor

It is an analog device allowing 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 convert light into electronic signals but each has its own method of conversion. As their performance continually evolves and their cost decreases, CMOS imagers have come to dominate the digital photography landscape.

Lens

The lens is an optic allowing reproduction of the real image captured rigorously on the image sensor. Picking the proper lens is a part of the user creativity and affects considerably the image quality.

Printed circuit board (PCB)

The PCB is a board that comprises electronic components to ensure the good polarization and the protection of the image sensor.

The PCB provides also a support for all the other parts of the camera module.

Camera module interconnect

The camera interface is a kind of bridge allowing the image sensor to connect to an embedded system and send or receive signals. The signals transferred between a camera and an embedded system are mainly:

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

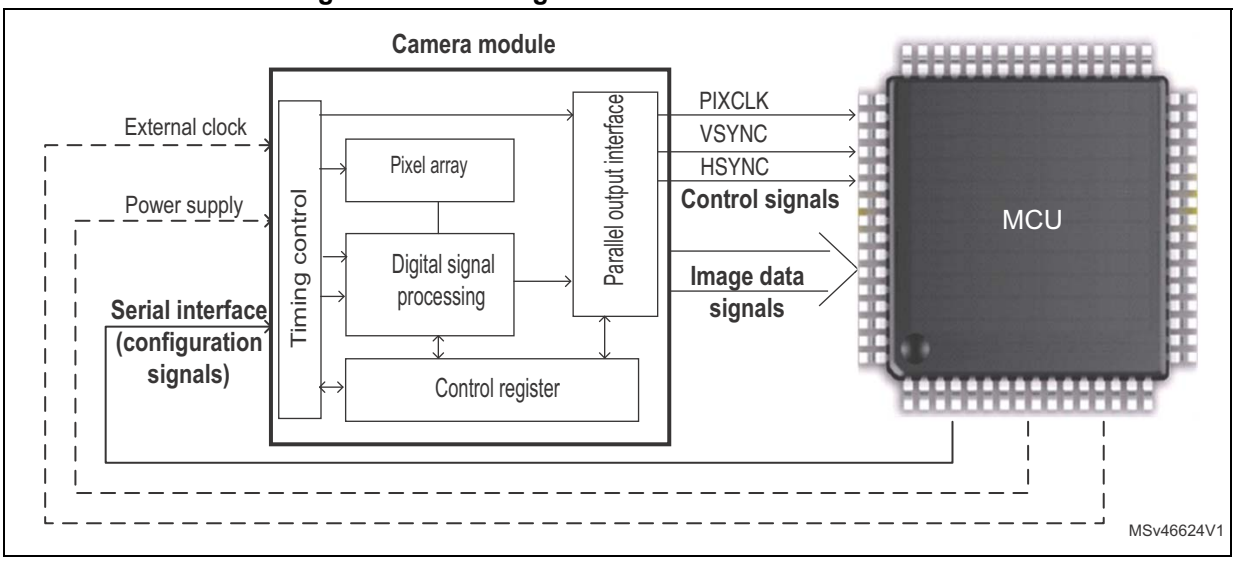
Depending on the manner of transferring data signals, camera interfaces are divided into two types: **parallel** and **serial interfaces**.

1.2.2 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 MCU



Control signals

These signals are used for clock generation and data transfer synchronization. The camera clock must be provided according to the camera specification.

The camera also provides data synchronization signals:

- **HSYNC,** used for line synchronization
- **VSYNC,** used for frame synchronization.

Image data signals

Each of these signals transmits a bit of the image data. The image data signals 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

These signals are used to:

- configure the appropriate image features such as resolution, format and frame rate
- configure the contrast and the brightness
- select the type of interface (a camera module may support more than one interface: a parallel and a serial interface. The user should then choose the most convenient one for the application.)

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

2 Overview of STM32 digital camera interface (DCMI)

This section provides a general preview of the digital camera interface (DCMI) availability across the different STM32 devices, and gives an easy-to-understand explanation on the DCMI integration in the STM32 MCUs architecture.

2.1 Digital camera interface (DCMI)

The digital camera interface (DCMI) is a synchronous parallel data bus. It allows easy integration and easy adaptation to specific requirements of an application. The DCMI connects with 8-, 10-, 12- and 14-bit CMOS camera modules and supports a multitude of data formats.

2.2 DCMI availability and features across STM32 MCUs

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

The DCMI applications need a frame buffer to store the captured image(s). 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. For this reason, the Quad-SPI can be used. For more details, refer to the application note *Quad-SPI interface on STM32 microcontrollers* (AN4760).

The DMA2D (Chrom-ART Accelerator™ controller) is useful for color spaces 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 memory size (bytes)	On-chip SRAM (bytes)	QUAD SPI	Max FMC SRAM and SDRAM frequ. (MHz)(1)	Max DCMI pixel clock input (MHz)(2)	JPEG codec	DMA2D	LCD_ TFT control-ler(3)	MIPI-DSI host(4)	Max AHB frequ. (MHz)
STM32F2x7	1 M	128	No	60	48	No	No	No	No	120
STM32F407/417	1 M	192	No	60	54	No	No	No	No	168
STM32F427/437	2 M	256	No	90	54	No	Yes	No	No	180
STM32F429/439	2 M	256	No	90	54	No	Yes	Yes	No	180
STM32F446	512 K	128	Yes	90	54	No	No	No	No	180
STM32F469/479	2 M	384	Yes	90	54	No	Yes	Yes	Yes	180
STM32F7x5	2 M	512	Yes	100	54	No	Yes	No	No	216

Table 2. DCMI and related resources availability (continued)

STM32 line	Max Flash memory size (bytes)	On-chip SRAM (bytes)	QUAD SPI	Max FMC SRAM and SDRAM frequ. (MHz)(1)	Max DCMI pixel clock input (MHz)(2)	JPEG codec	DMA2D	LCD_TFT control-ler(3)	MIPI-DSI host(4)	Max AHB frequ. (MHz)
STM32F7x6	1 M	320	Yes	100	54	No	Yes	Yes	No	216
STM32F7x7	2 M	512	Yes	100	54	Yes	Yes	Yes	No	216
STM32F7x8 STM32F7x9	2 M	512	Yes	100	54	Yes	Yes	Yes	Yes	216
STM32L4x6	1 M	320	Yes	40	32	No	Yes	No	No	80
STM32H7x3	2 M	1000	Yes	133	80	Yes	Yes	Yes	No	400

1. FSMC for STM32F2x7 and STM32F407/417 lines.
2. For the pixel clock frequency (DCMI_PIXCLK), refer to the datasheet of the corresponding device.
3. For more details on STM32 LTDC peripheral, refer to the application note AN4861.
4. For more details on STM32 MIPI-DSI host, refer to the application note AN4860.

2.3 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 DMA, as an AHB master, to autonomously access AHB2 peripherals and transfer the received data (image number n+1) to the memory, while the CPU is processing the previously captured image (image number n)
- the DMA2D, as an AHB master, to be used to transfer or modify the received data and keep the CPU resources for other tasks
- the memories throughput amelioration and the performance improvement, thanks to the multi-layer bus matrix.

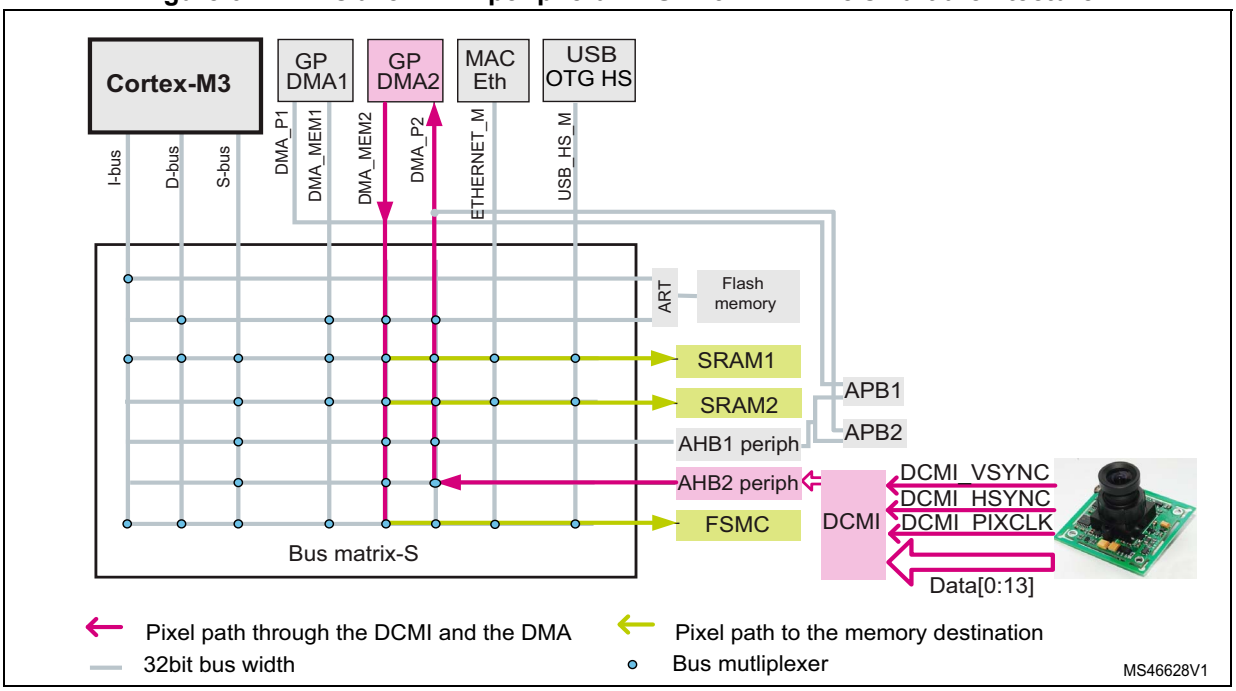
2.3.1 System architecture of STM32F2x7 line

STM32F2x7 line devices are based on a 32-bit multi-layer bus matrix, allowing the interconnection between 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 STM32F2x7xx devices.

Figure 6. DCMI slave AHB2 peripheral in STM32F2x7 line smart architecture



2.3.2 System architecture of STM32F407/417, STM32F427/437, STM32F429/439, STM32F446 and STM32F469/479 lines

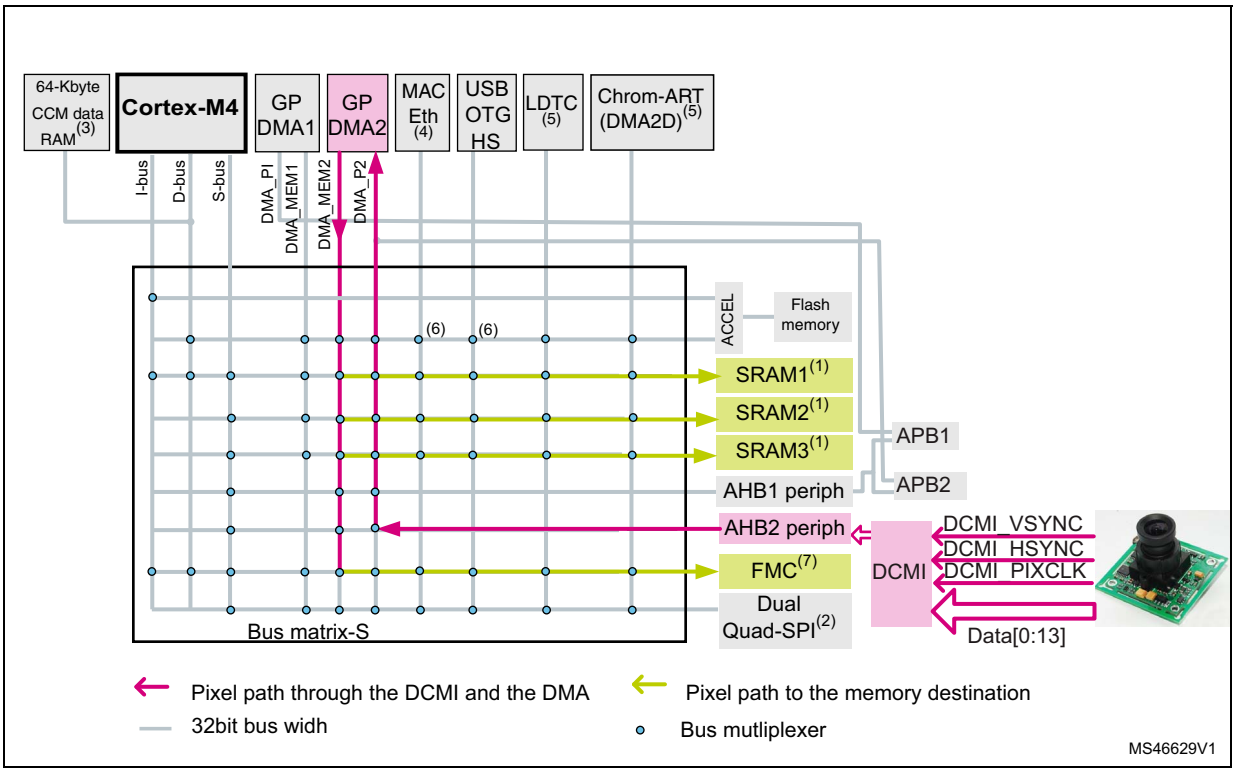
The devices of STM32F407/417, STM32F427/437, STM32F429/439, STM32F446 and STM32F469/479 lines, are based on a 32-bit multi-layer bus matrix, allowing the interconnection between:

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

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 microcontrollers of STM32F407/417, STM32F427/437, STM32F429/439, STM32F446 and STM32F469/479 lines.

Figure 7. DCMI slave AHB2 peripheral in STM32F407/417, STM32F427/437,STM32F429/439, STM32F446 and STM32F469/479 lines smart architecture



1. For more information about SRAM1, SRAM2 and SRAM3, see *Table 3*.

Table 3. SRAM availability in STM32F4 Series

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

2. Dual Quad-SPI interface is available only in STM32F469/479 and STM32F446 lines.
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 only lines embedding the LTDC and the DMA2D are STM32F429/439 and STM32F469/479.
6. For STM32F407/417 line, 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.
   For STM32F446 line, there is no interconnection between the USB master and the DCODE bus of the Flash memory.
7. FSMC for STM32F407/417 line.

2.3.3 System architecture of STM32F7x5, STM32F7x6, STM32F7x7, STM32F7x8 and STM32F7x9 lines

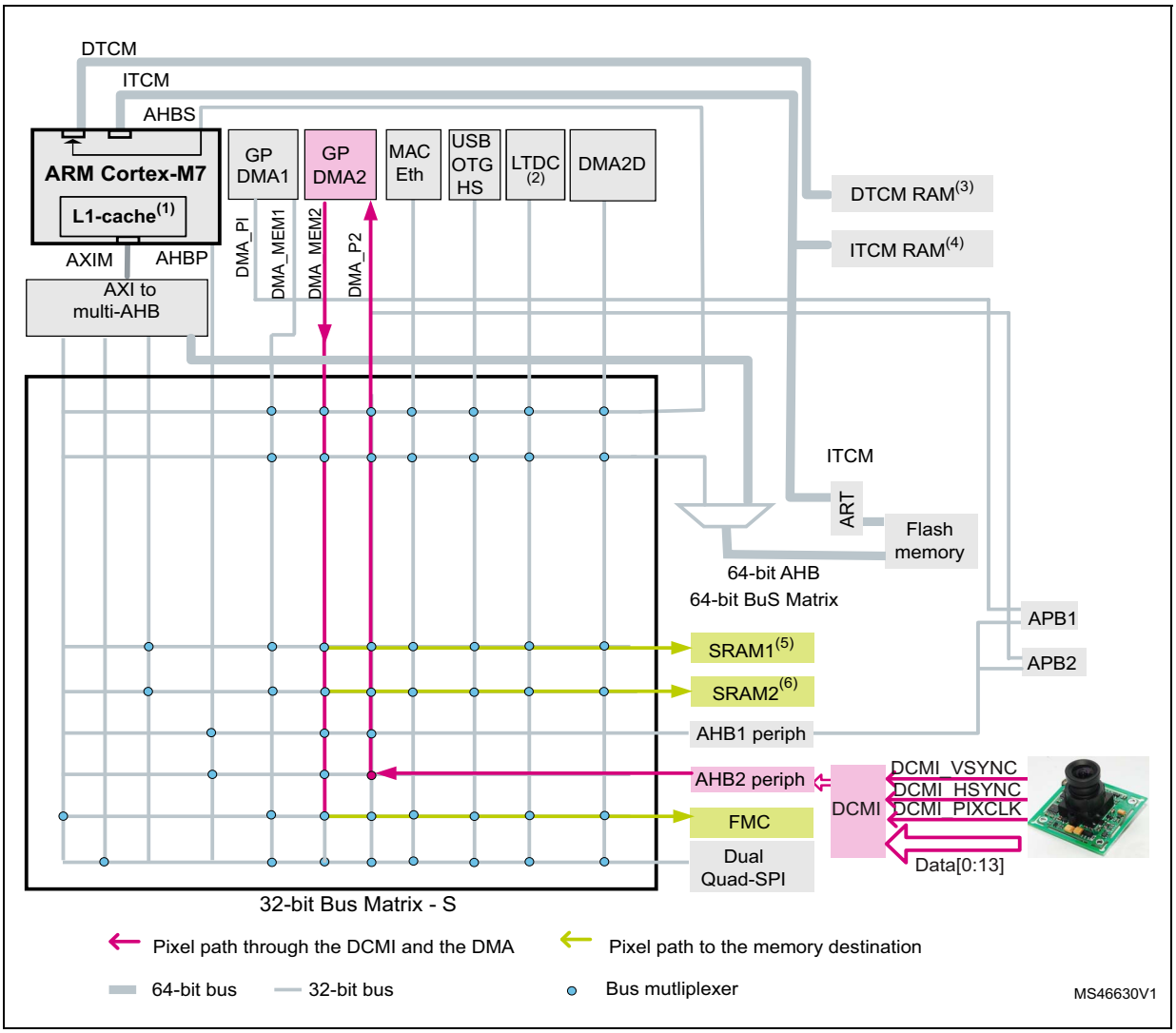
The devices of STM32F7x5, STM32F7x6, STM32F7x7, STM32F7x8 and STM32F7x9 lines are based on a 32-bit multi-layer bus matrix, allowing the interconnection between:

- twelve masters and eight slaves for STM32F7x6, STM32F7x7, STM32F7x8 and STM32F7x9 lines
- eleven masters and eight slaves for STM32F7x5 line.

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 the STM32F7x5, STM32F7x6, STM32F7x7, STM32F7x8 and STM32F7x9 line devices.

Figure 8. DCMI slave AHB2 peripheral in STM32F7x5, STM32F7x6, STM32F7x7, STM32F7x8 and STM32F7x9 lines smart architecture



1. The I/D cache size is:
   - 4 Kbytes for STM32F7x5 and STM32F7x6 lines
   - 16 Kbytes for STM32F7x7, STM32F7x8 and STM32F7x9 lines.
2. The LTDC (LCD-TFT controller) is available only in STM32F7x6, STM32F7x7, STM32F7x8 and STM32F7x9 lines.
3. The DTCM RAM size is:
   - 64 Kbytes for STM32F7x5 and STM32F7x6 lines
   - 128 Kbytes for STM32F7x7, STM32F7x8 and STM32F7x9 lines.
4. The ITCM RAM size is 16 Kbytes for STM32F7x5, STM32F7x6, STM32F7x7, STM32F7x8 and STM32F7x9 lines.
5. The SRAM1 size is:
   - 240 Kbytes for STM32F7x5 and STM32F7x6 lines
   - 368 Kbytes for STM32F7x7, STM32F7x8 and STM32F7x9 lines.
6. The SRAM2 size is 16 Kbytes for STM32F7x5, STM32F7x6, STM32F7x7, STM32F7x8 and STM32F7x9 lines.

2.3.4 System architecture of STM32L496xx and STM32L4A6xx devices

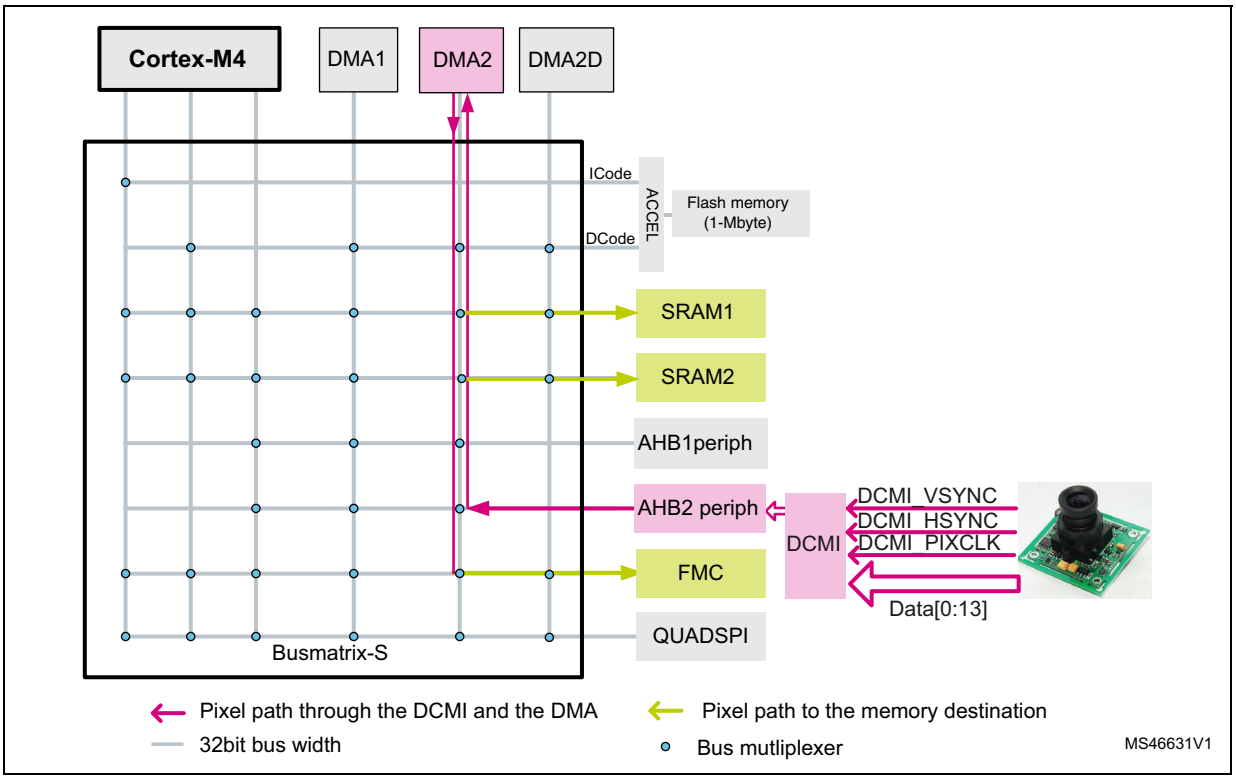
STM32L496xx and STM32L4A6xx devices are based on a 32-bit multi-layer bus matrix, allowing the interconnection between six masters and eight slaves.

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.

In STM32L496xx and STM32L4A6xx MCUs, the DMA has only one port (not like STM32F2, STM32F4, STM32F7 and STM32H7 series 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 9 shows the DCMI interconnection and the data path in STM32L496xx and STM32L4A6xx devices.

Figure 9. DCMI slave AHB2 peripheral in STM32L496xx and STM32L4A6xx devices smart architecture



2.3.5 System architecture of STM32H7x3 line

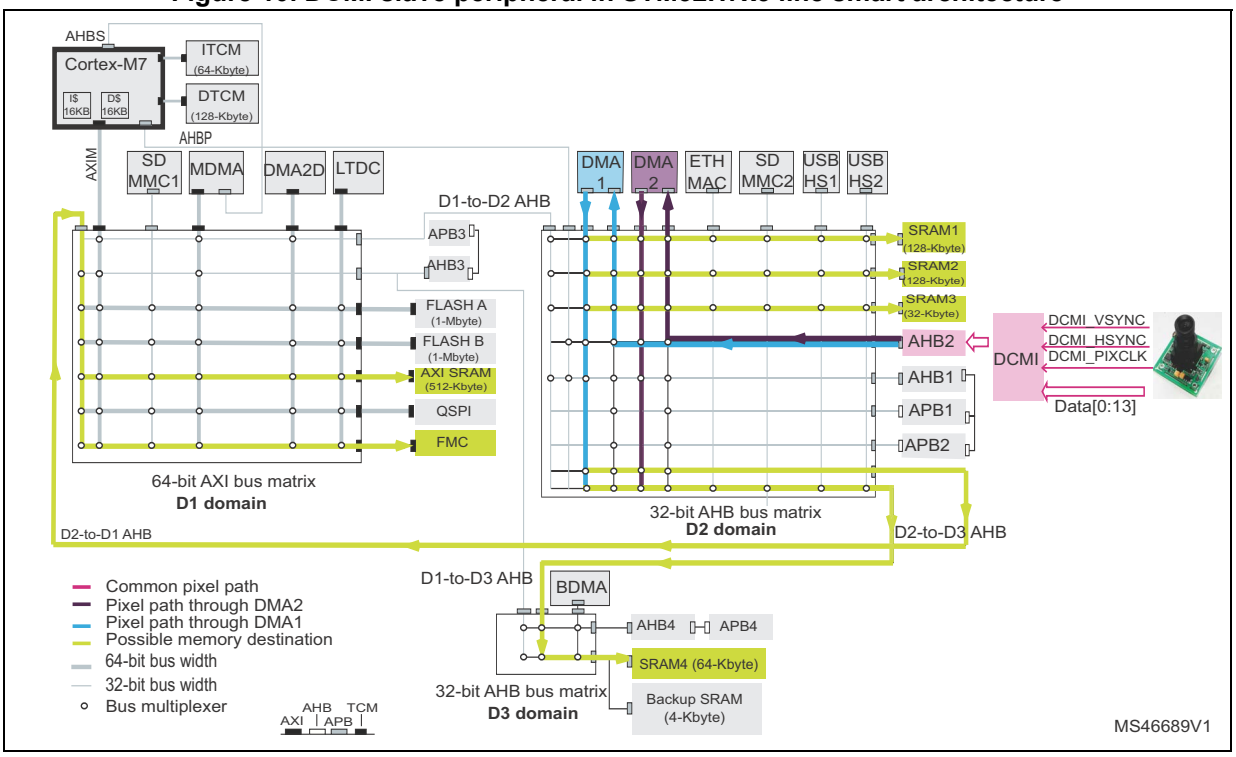
STM32H7x3xx devices are based on an AXI bus matrix, two AHB bus matrices and bus bridges allowing the interconnection between 18 masters and 20 slaves.

The DCMI is a slave AHB2 peripheral. The DMA1 or the DMA2 can perform the data transfer from the DCMI to internal SRAMs or external memories through the FMC.

The DMA1 and DMA2 are located in D2 domain. They are able to access slaves in D1 domain and D3 domain. As a result, the DMA1 or the DMA2 can transfer the data received by the DCMI (located in domain 2) to memories located in domain 1 or domain 3.

Figure 10 shows the DCMI interconnection and the data path in STM32H7x3xx devices.

Figure 10. DCMI slave peripheral in STM32H7x3 line smart architecture



2.4 Reference boards with DCMI and/or camera modules

Many STM32 reference boards are available, such as NUCLEO, Discovery and EVAL boards. Most of them embed the DCMI and some of them have an on-board camera module.

The board selection depends on the application and hardware resources.

Table 4 summarizes the DCMI, the camera modules and the memories availability across various STM32 reference boards.

Table 4. DCMI and camera modules on various STM32 boards(1)

STM32 line	Board	Camera module	CMOS sensor	Internal SRAM (Kbytes)	External SDRAM bus width (bits)	External SRAM bus width (bits)
STM32F2x7	STM3220G-EVAL	Yes(2)	OV2640 or OV9655	132	NA	
	STM3221G-EVAL	Yes(2)				
STM32F407/417	STM32F4DISCOVERY	Yes(3) or (4)	OV9655	196		
	STM3240G-EVAL	Yes(2)				
	STM3241G-EVAL					
STM32F429/439	32F429IDISCOVERY	NA(3)	NA	256	16	NA
	STM32429I-EVAL	Yes(2)	OV2640 or OV9655		32	16
	STM32439I-EVAL					
STM32F446	STM32446E-EVAL	Yes(2)	S5k5CAGA	128	16	NA
STM32F469/479	32F469IDISCOVERY	NA(3)	NA	324	32	NA
	STM32469I-EVAL	Yes(2)	S5k5CAGA			16
	STM32479I-EVAL					
STM32F7x6	32F746GDISCOVERY	Yes(4)	OV9655	320	16	NA
	STM32746G-EVAL	Yes(2)	S5k5CAGA		32	16
	STM32756G-EVAL					
STM32F7x9	32F769IDISCOVERY	NA(3)	NA	512	32	NA
	STM32F769I-EVAL	Yes(2)	S5k5CAGA			16
	STM32F779I-EVAL					
STM32L4x6	32L496GDISCOVERY	Yes(4)	OV9655	320	NA	NA
STM32H7x3	STM32H743I-EVAL STM32H753I-EVAL	NA(3)	NA	864	32	16

1. NA: not available. The user should use the desired camera module compatible with the DCMI interface.
2. For the different EVAL boards, a specific connector allows the connection between the DCMI and the camera module.
   - For STM3220G-EVAL, STM3221G-EVAL, STM32F40G- EVAL and STM32F41G- EVAL, there are two possible cameras to be connected: module CN01302H1045-C (CMOS sensor OV9655, 1.3 Megapixels) and module CN020VAH2554-C (CMOS sensor OV2640, 2 Megapixels).
   - For STM32429I- EVAL and STM32439I- EVAL, the camera module daughterboard MB1066 is connected.
   - For STM32446E-EVAL, STM32469I- EVAL, STM32F479I- EVAL, STM32746G- EVAL, STM32756G-EVAL, STM32F769I-EVAL and STM32F779I-EVAL the camera module daughterboard MB1183 is connected.
3. The camera module can be connected to the DCMI through the GPIO pins.
4. The camera module can be connected to the DCMI through an FFC (flexible flat cable):
   - For the STM32F4DISCOVERY, the STM32F4DIS-EXT expansion board should be used to connect the STM32F4DIS-CAM camera module.
   - For the 32F746IDISCOVERY and 32L496GDISCOVERY, the STM32F4DIS-CAM board can be connected directly.
   For more details on STM32F4DIS-EXT and STM32F4DIS-CAM, please visit STMicroelectronics website

3 DCMI description

This section describes in detail the DCMI and its manner of dealing with the image data and the synchronization signals.

Note: The DCMI supports only the slave input mode.

3.1 Hardware interface

The DCMI consists of:

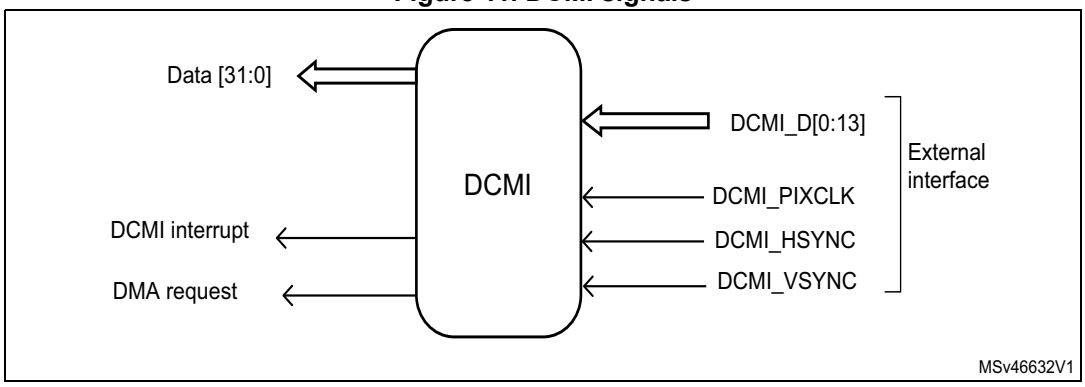
- up to 14 data lines (D13-D0)
- the pixel clock line DCMI_PIXCLK
- the DCMI_HSYNC line (horizontal synchronization)
- the DCMI_VSYNC line (vertical synchronization).

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 function. 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.

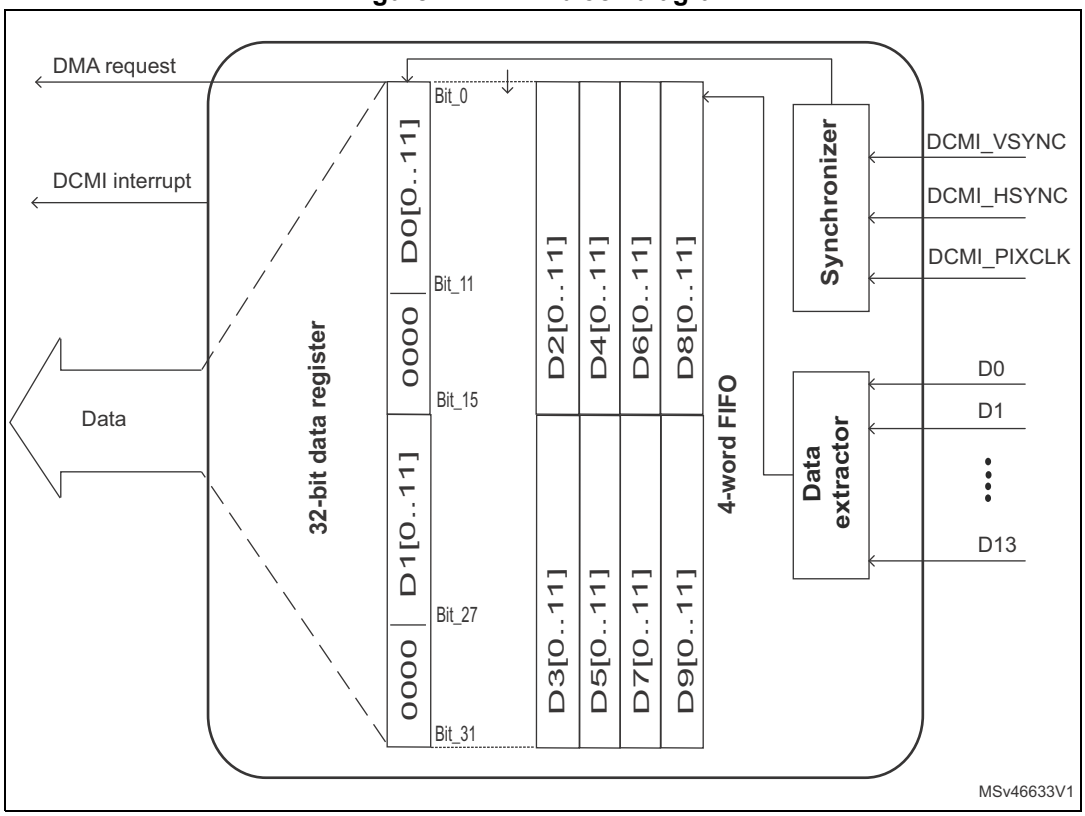
Figure 11. DCMI signals



If x-bit data width is chosen (x data lines are enabled and x is 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.

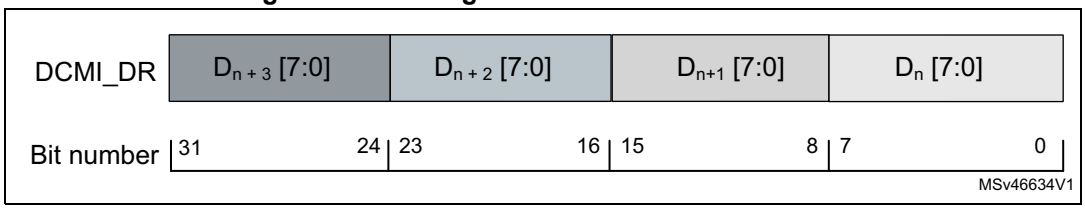
As shown in *Figure 12*, the DCMI is composed of four main components:

Figure 12. DCMI block diagram



- **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.
- **Data extractor**: ensures the extraction of the data received by the DCMI.
- **FIFO**: this 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, FIFO is reset and the DCMI waits for a new start of frame.
- **32-bit register**: 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 **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. So, in this case, a 32-bit data word is made up every four pixel clock cycles.

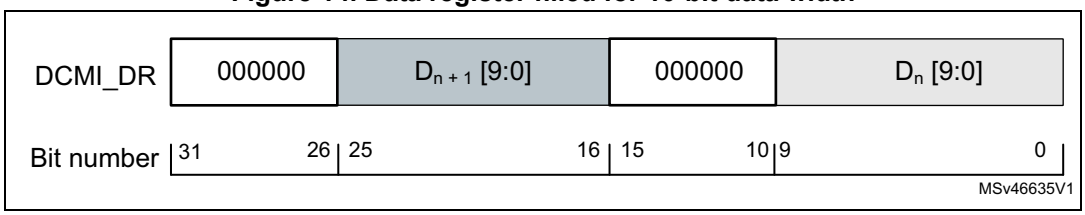
Figure 13. Data register filled for 8-bit data width



for more details, refer to *Section 3.6: Data formats and storage*.

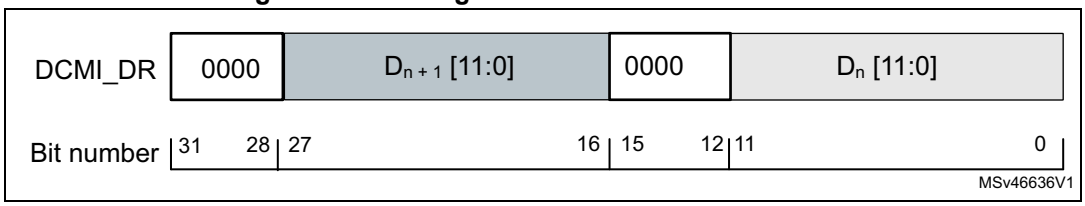
– For **10-bit data width**, the DCMI captures the 10 LSBs (the four other inputs D[13:10] are ignored). The first 10 bits captured are placed as the 10 LSBs of a 16-bit word. The remaining MSBs in the 16-bit word of the DCMI_DR register (bits 10 to 15) are cleared.
So, in this case, a 32-bit data word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

Figure 14. Data register filled for 10-bit data width



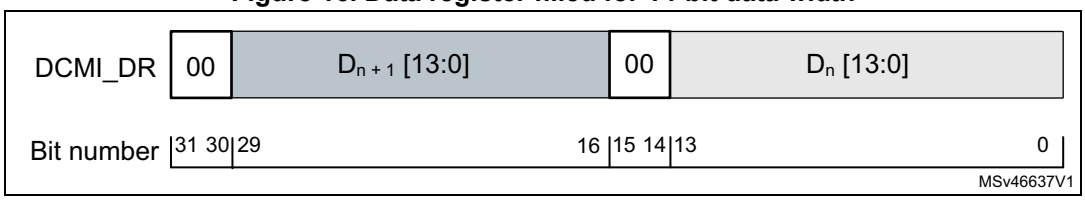
– For **12-bit data width**, the DCMI captures the 12-bit LSBs (the two other inputs D[13:12] are ignored). The first 12 bits captured are placed as the 12 LSBs of a 16-bit word. The remaining MSBs in the 16-bit word of the DCMI_DR register (bits 12 to 15) are cleared.
So, in this case, a 32-bit data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

Figure 15. Data register filled for 12-bit data width



– For **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.
So, in this case, a 32-bit data word is made up every two pixel clock cycles.

Figure 16. Data register filled for 14-bit data width

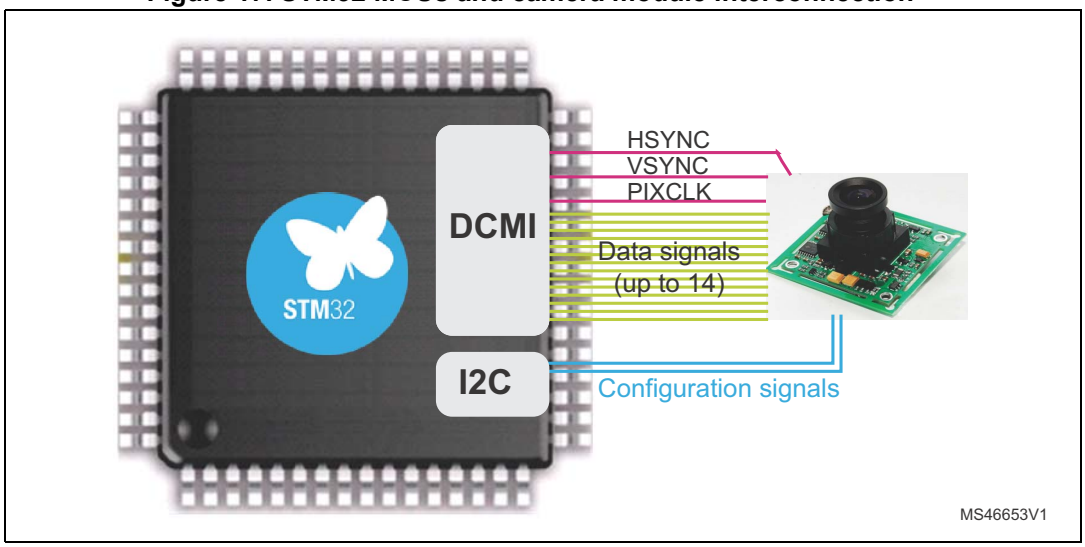


3.2 Camera module and DCMI interconnection

As mentioned in *Section 1.2.2: Camera module interconnect (parallel interface)*, the camera module is connected to the DCMI through three types of signals:

- DCMI clock and data signals
- I2C configuration signals

Figure 17. STM32 MCUs and camera module interconnection(1)



1. For embedded synchronization, the DCMI_HSYNC and DCMI_VSYNC signals are ignored and only 8 data signals are used

3.3 DCMI functional description

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

- After receiving the different signals, the synchronizer controls the data flow through the different components of the DCMI (data extractor, FIFO and 32-bit data register).
- Being extracted by the extractor, the data are packed in the 4-word FIFO then ordered in the 32-bit register.
- Once the 32-bit data block is packed in the register, a DMA request is generated.
- The DMA transfers the data to the corresponding memory destination.
- 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.

收到32bit数据就触发一次DMA中断，
DMA就搬运一次数据到RAM。

3.4 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), horizontal synchronization line, DCMI_HSYNC, and 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.

3.4.1 Hardware (or external) synchronization

In this mode, the two DCMI_VSYNC and DCMI_HSYNC signals are used for 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.

The data is synchronized with DCMI_PIXCLK and changes on the rising or the 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 the VSYNC is programmed active high:

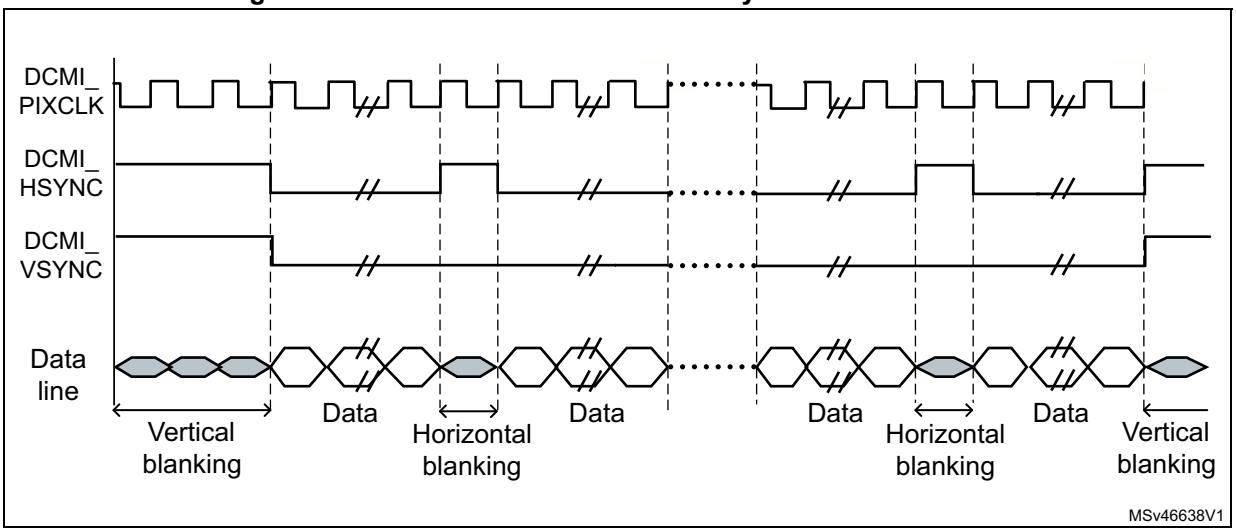
这里要注意，VSYNC设置为Active Low，示波器上就是VSYNC在High时数据有效！！！

- when the VSYNC is low, the data is valid
- when the VSYNC is at the high level, the data is not valid (vertical blanking).

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

Figure 18 shows an example of data transfer when DCMI_VSYNC and DCMI_HSYNC are active high and the capture edge for DCMI_PIXCLK is the rising edge.

Figure 18. Frame structure in hardware synchronization mode



VSYNC配置成Active High，则现实中就是当VSYNC=Low时，数据有效！！！！（正好相反吧）

Compressed data synchronization

For compressed data (JPEG), the DCMI supports only the hardware synchronization. Each JPEG stream is divided into packets. These packets 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.
这句话比较关键！如果VSYNC没有发生，则DCMI补零触发End-of-Frame中断！！！！

3.4.2 Embedded (or internal) synchronization

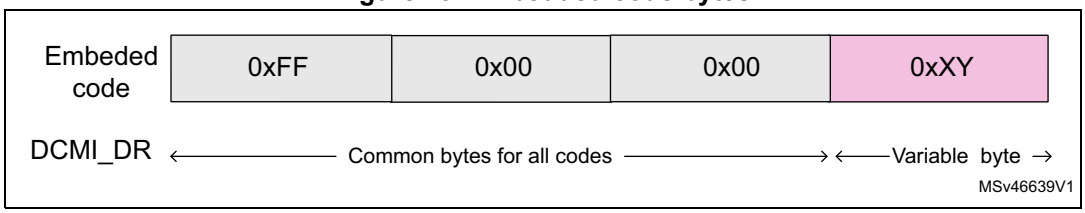
In this case, delimiter codes are used for synchronization. These codes are embedded within the data flow to indicate the start/end of line or the start/end of frame.

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 end/start of line or 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. For this reason, image data can have only 254 possible values (0x00 < image data value < 0xFF).

Each synchronization code consists of 4-byte sequence **0xFF 00 00 XY,** 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 19. Embedded code bytes



Mode 1

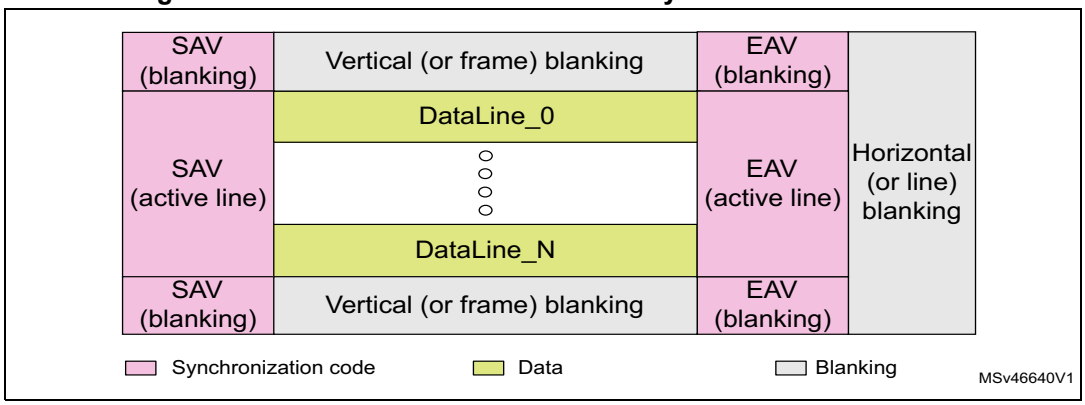
This mode is ITU656 compatible (ITU656 is the digital video protocol ITU-R BT.656).

There are four reference codes indicating 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 20 illustrates the frame structure using this mode.

Figure 20. Frame structure in embedded synchronization mode 1



Mode 2

In this mode, embedded synchronization codes signal another set of events:

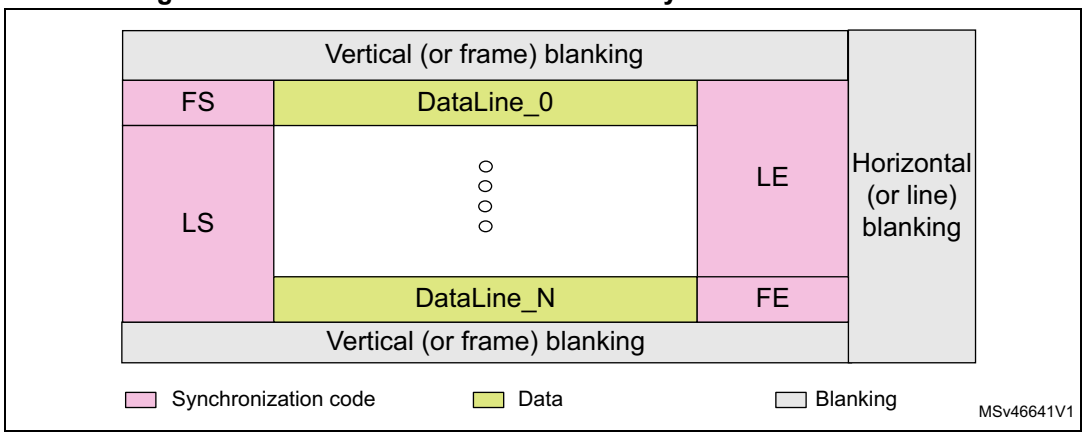
- frame-start (**FS**)
- frame-end (**FE**)
- line-start (**LS**)
- line-end (**LE**)

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 21 illustrates the frame structure when using this mode.

Figure 21. Frame structure in embedded synchronization mode 2



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

Embedded unmask codes

These codes are also used to signal start/end of line or start/end of 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.

In other words, 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:

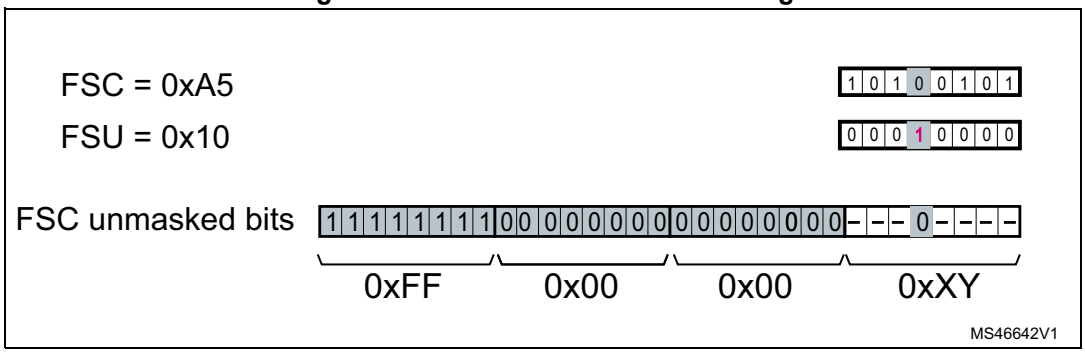
- The most significant byte is the frame end delimiter unmask (FEU): each bit set to 1, implies that this bit, in the frame-end-code, must be compared with the received data to know if it is a frame-end event or not.
- The second byte is the Line end delimiter unmask (LEU): each bit set to 1, implies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

As a result, there can be different codes for each event (line-start or line-end or 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.

In this case the frame-start information is embedded in the bit number 4 of the FS code. As a result, 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 22. Embedded codes unmasking



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

3.5 Capture modes

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

Depending on the DCMI_CR register configuration, the user can control the capture rate by selecting the bytes, the lines and the frames to capture.

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 will be divided by 2).

For more details, refer to *Section 3.7.2: Image resizing (resolution modification)*.

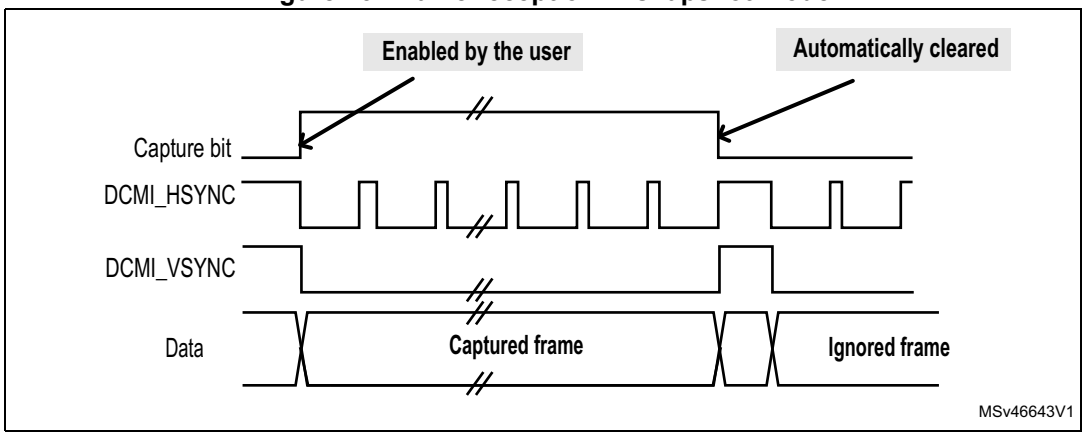
3.5.1 Snapshot mode

In the snapshot mode, a single frame is captured. After the capture is enabled by setting the CAPTURE bit of the DCMI_CR register, 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 (the CAPTURE bit is 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 23. Frame reception in snapshot mode



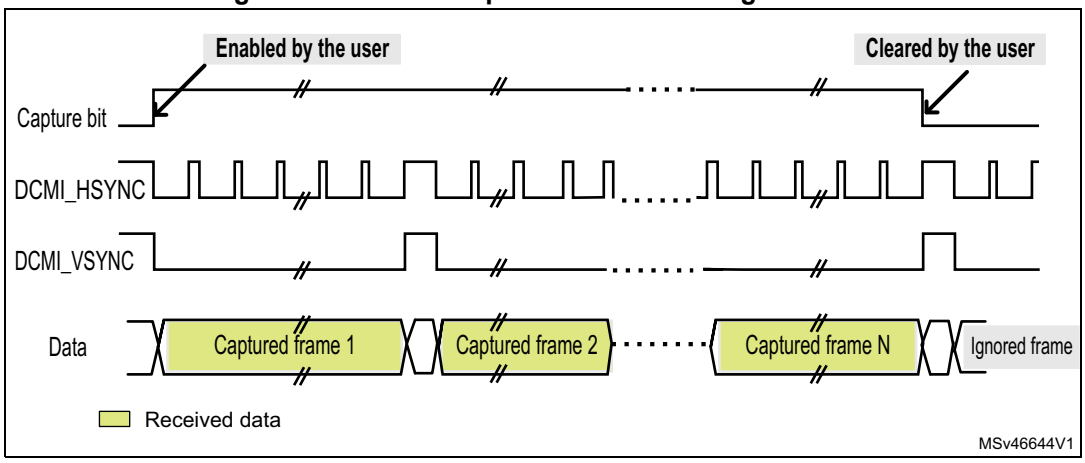
3.5.2 Continuous grab mode

Once this mode is selected and the capture is enabled (CAPTURE bit set), 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).

In this case, the camera interface is not automatically disabled but the user must disable it by setting the CAPTURE bit to zero. After being disabled by the user, the DCMI continues to grab data until the end of the current frame.

Figure 24. Frame reception in continuous grab mode



3.6 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).

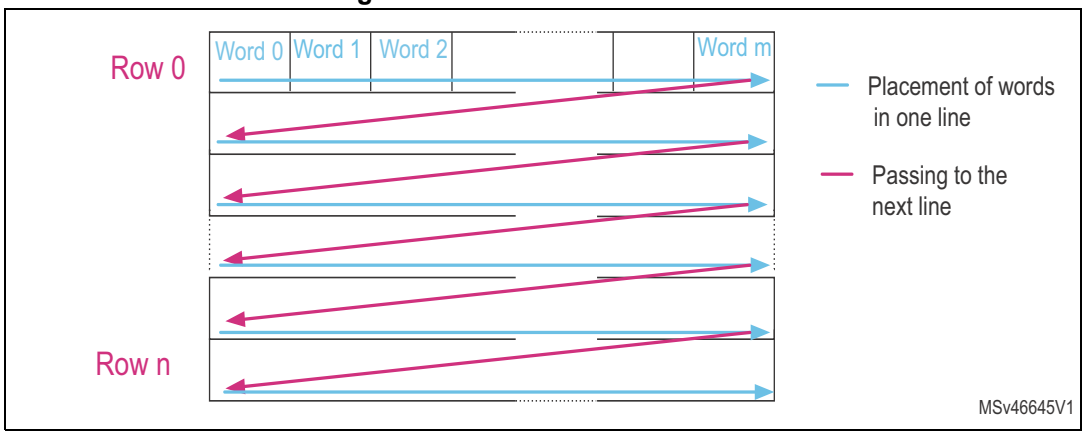
For monochrome, RGB or YCbCr data:

- the maximum input size is 2048 * 2048 pixels
- 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 25*.

Figure 25. 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).

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

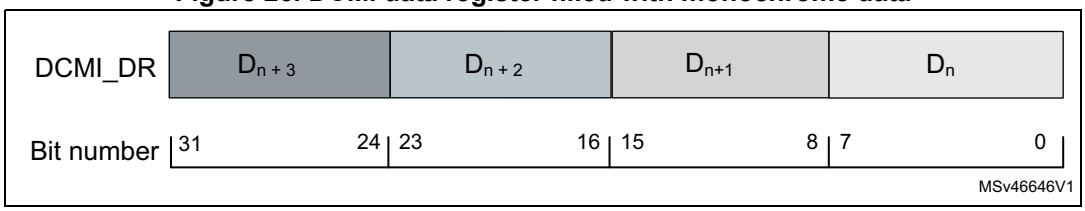
The number of bytes in a line may not be a multiple of four. The user should therefore be careful when handling this case since a DMA request is generated each time a complete 32-bit word has been constructed from the captured data. When an end of frame is detected and the 32-bit word to be transferred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

3.6.1 Monochrome

The DCMI supports the monochrome format 8 bits per pixel.

In the case of 8-bit data width is selected when configuring the DCMI, the data register has the structure shown in *Figure 26*.

Figure 26. DCMI data register filled with monochrome data



3.6.2 RGB565

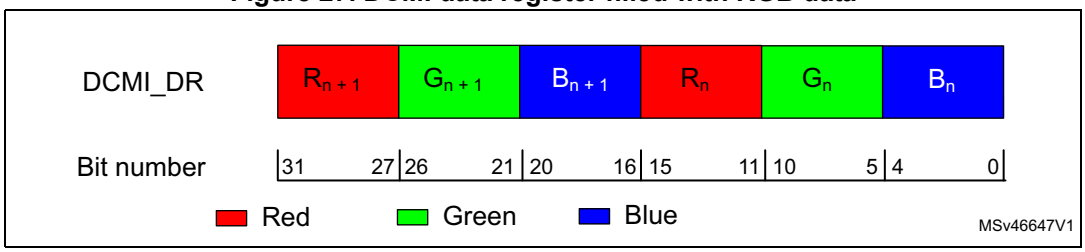
RGB refers to **red**, **green** and **blue**, which represent the three hues of light. Any color is obtained by mixing these three colors.

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

- **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). In other words, each sample has a red (R), a green (G) and a blue (B) component. *Figure 27* shows the DCMI data register containing RGB data, when 8-bit data width is selected.

Figure 27. DCMI data register filled with RGB data



3.6.3 YCbCr

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

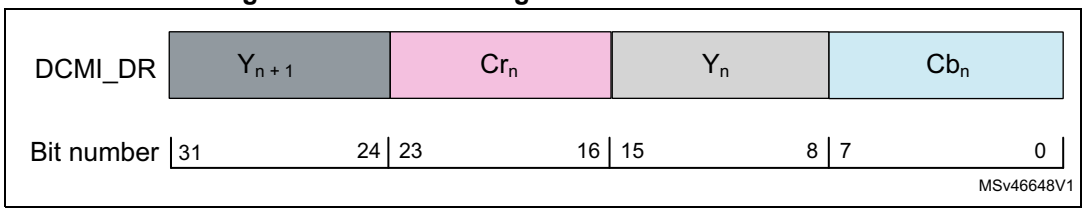
YCbCr consists of three components:

- **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 sub- sampling scheme, requiring 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 28* shows the DCMI data register containing YCbCr data when 8-bit data width is selected.

Figure 28. DCMI data register filled with YCbCr data



3.6.4 YCbCr, Y only

Note: only for STM32F446 line, STM32F469/479 line, STM32L496xx, STM32L4A6xx, STM32F7xxxx devices and STM32H7x3 line.

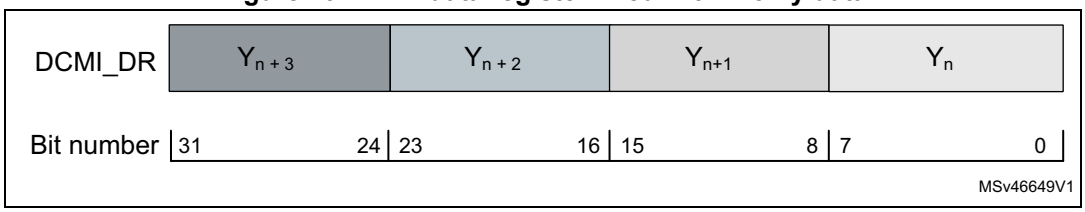
The buffer contains only the Y information - monochrome image.

In this mode, 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 29 shows the DCMI register when 8-bit data width is selected.

Figure 29. DCMI data register filled with Y only data



3.6.5 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, that have programmable size. The packets dispatching depends on the image content and results in a variable blanking duration between two packets.

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

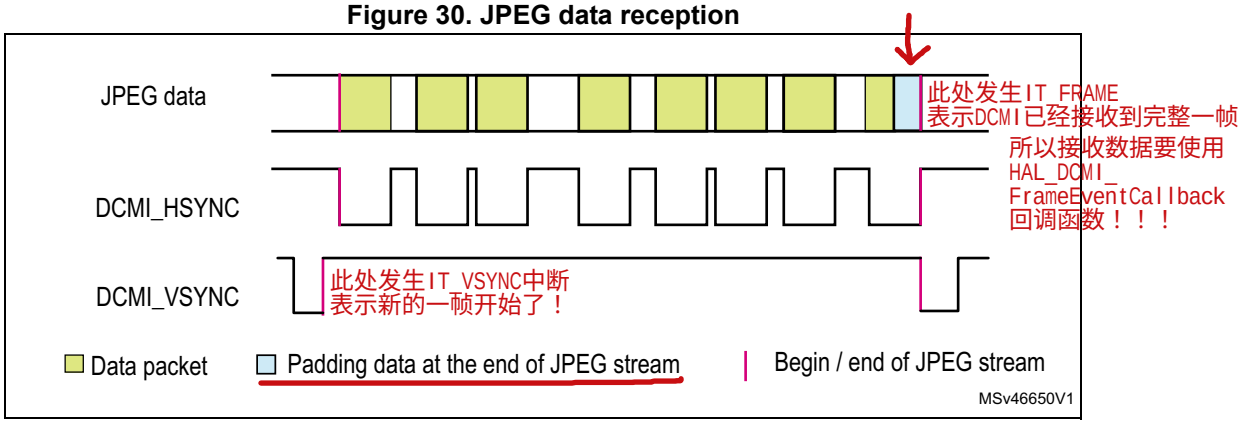
就是使用VSYNC同步的，HSYNC是数据有效信号！

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. In other words, 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.

虽然JPEG数据流结束了，但由于不是4的倍数，所以DCMI检测到0xFF 0xD9后，会在后面补0.

Figure 30. JPEG data reception



3.7 Other features

3.7.1 Crop feature

With the crop feature, the camera interface selects a rectangular window from the received image.

The start coordinates (upper-left corner) is specified in the 32-bit register DCMI_CWSTRT.

The window size is specified in number of pixel clocks (horizontal dimension) and in number of lines (vertical dimension), in the 32-bit register DCMI_CWSIZE.

3.7.2 Image resizing (resolution modification)

Note: Image resizing feature is only available in STM32L496xx, STM32L4A6xx, STM32F446 line, STM32F469/479 line, STM32F7x5 line, STM32F7x6 line, STM32F7x7 line, STM32F7x8 line, STM32F7x9 line and STM32H7x3 line.

As described in *Section 3.5: Capture modes*, the DCMI capture features are set through the DCMI_CR register.

The DCMI can capture:

- all received lines
- one line out of two (in this case, 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. In other words, only the odd or the even bytes are received)
- one byte out of four
- two bytes out of four

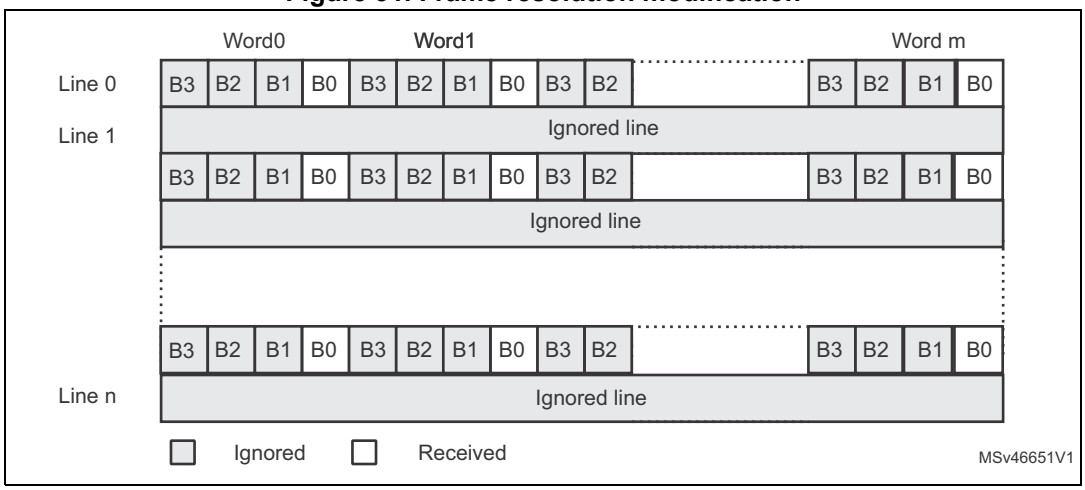
This feature affects the horizontal resolution allowing the user to select one of the following resolutions:

- the full horizontal resolution
- the half of the horizontal resolution
- the quarter of the horizontal resolution (this feature is available only for eight bit per pixel data formats).

Note: Caution is required when using this feature. 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 31 shows one frame when receiving only one byte out of four and one line out of two.

Figure 31. Frame resolution modification



3.8 DCMI interrupts

疑问：IT_VSYNC与IT_FRAME是不是同一个？
为什么要分开两个中断？？？
难道他们不一样？？？

Five 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 codes order (only in embedded synchronization mode).

IT_VSYNC表示检测到一个新帧开始！！！
由内部逻辑检测到外部信号，开始接收新的一帧数据了！

而IT_FRAME表示DCMI模块接收到完整的一帧数据了！
报告给用户！！！

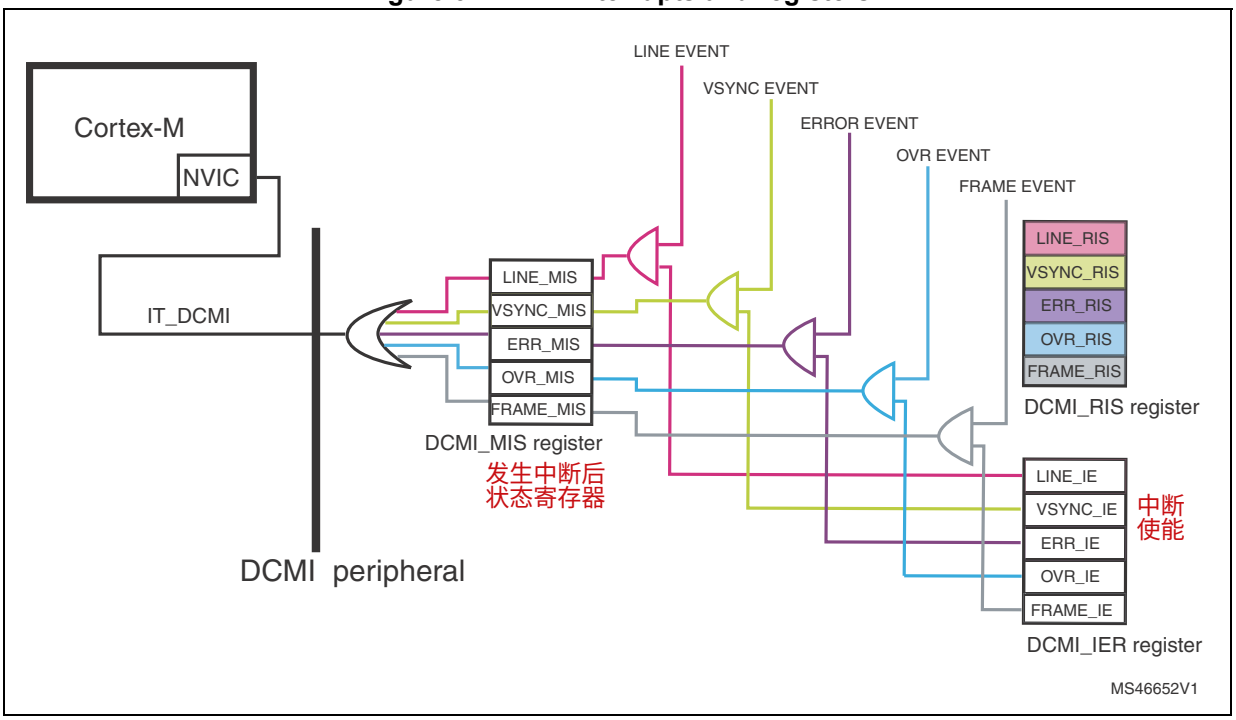
All interrupts can be masked by software. The global interrupt **dcmi_it** is the logic OR of all the individual interrupts.

As shown in *Figure 32*, the DCMI interrupts are handled through three 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 the DCMI_IER register (each bit gives the status of the interrupt that can be enabled or disabled in the DCMI_IER register).
- **DCMI_MIS**: read-only register providing the current masked status of the corresponding interrupt, depending on the DCMI_IER and the DCMI_RIS registers.

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

Figure 32. DCMI interrupts and registers



3.9 Low-power modes

The STM32 power mode has a direct effect on the DCMI peripheral. For this reason, it is essential to know the DCMI peripheral operation over the different power modes.

In **Run** mode, the DCMI and all peripherals operate normally.

In **Sleep** mode, the DCMI and all the peripherals work normally and generate interrupts to wake up the CPU.

In **Stop** mode and **Standby** mode, the DCMI does not work.

For STM32L496xx and STM32L4A6xx 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, Stop1, Stop 2** mode: the peripheral registers content is kept.
- **Shutdown** mode: the peripheral must be reinitialized when exiting Shutdown mode.

Table 5 summarizes the DCMI operation in the different modes.

Table 5. DCMI operation in low-power modes

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

1. Only for STM32L496xx and STM32L4A6xx devices.

4 DCMI configuration

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

To correctly implement his 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, data format.

Note: It is recommended to reset the DCMI peripheral 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.

4.1 GPIO configuration

To easily configure the DCMI GPIOs (such as data pins, control signals pins, camera configuration pins) and to avoid any pins 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 the EDM bits in the DCMI_CR register, the DCMI receives 8, 10, 12 or 14 bits per pixel clock (DCMI_PIXCLK). The user needs to configure 11, 13, 15 or 17 GPIOs for the DCMI in case of hardware synchronization. 17=14bits(D0~D13)+VSYNC+HSYNC+PCLK

In case of embedded synchronization, only nine GPIOs must be configured (eight pins for data and one pin for DCMI_PIXCLK)

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)

Interrupts enabling

To be able to use the DCMI interrupts, the user should 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.

In hardware synchronization mode, only four interrupts can be used (IT_LINE, IT_FRAME, IT_OVR and IT_DCMI_VSYNC) but in embedded synchronization mode all the five interrupts can be used.

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

4.2 Clocks and timings configuration

This section describes the timings and clocks configurations steps.

4.2.1 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 should 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 FMC, the maximum SDRAM clock is 90 MHz (HCLK/2).
- STM32F7 Series: the maximum system speed is 216 MHz. With this speed and HCLK/2 prescaler, the SDRAM speed exceeds the maximum allowed speed (see products datasheet 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 line, HCLK @ 120 MHz and SRAM @ 60 MHz
- for STM32F407/417 line, HCLK @ 168 MHz and SRAM @ 60 MHz
- for STM32L4x6 line, HCLK @ 80 MHz and SRAM @ 40 MHz

4.2.2 DCMI clocks and timings 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 3.4: Data synchronization*, there are two types of synchronization: embedded and hardware. To select the desired synchronization mode for his application, the user needs to configure the ESS bit in the DCMI_CR register.

Hardware (external) synchronization

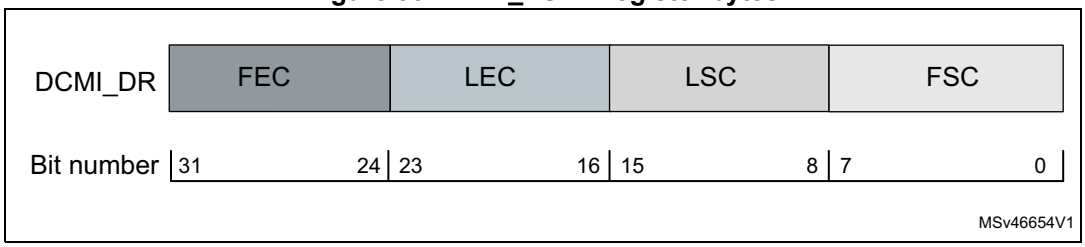
The DCMI_HSYNC and DCMI_VSYNC signals are used. The configuration of these two signals is defined by selecting each signal active level (high or low) in the VSPOL and HSPOL bits in DCMI_CR register.

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 bit of the DCMI_CR register cleared to zero), the IT_VSYNC interrupt is generated (if enabled), even when the CAPTURE bit of the DCMI_CR register is cleared to zero. 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.

Embedded (internal) synchronization

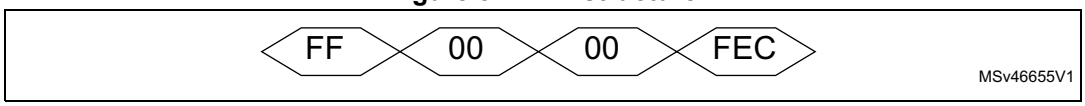
The line-start or line-end and frame-start or frame-end are determined by codes or markers embedded within the data flow. The embedded synchronization codes are supported only for 8-bit parallel data interface width. The synchronization codes must be programmed in the DCMI_ESCR register, as defined in *Figure 33*.

Figure 33. 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 34*.

Figure 34. FEC structure

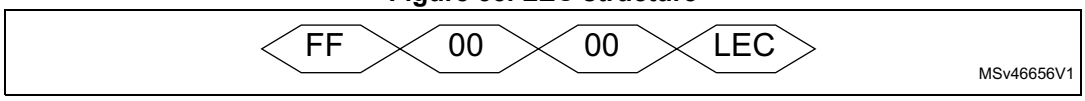


Before the reception of this FEC code, the value of VSYNC bit in the DCMI_SR register must be set to 1 to indicate a valid frame. After the reception of the FEC, the value of VSYNC bit must be 0 to indicate that it is synchronization between frames. This VSYNC bit value must remain 0 until the reception of the next frame-start code.

If FEC value is equal to 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).

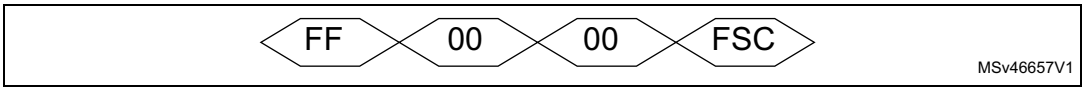
- **LEC (line-end code)**: this byte specifies the line-end marker. The code received from the camera to indicate the end of line is 0xFF 00 00 XY with XY = LEC code.

Figure 35. 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.

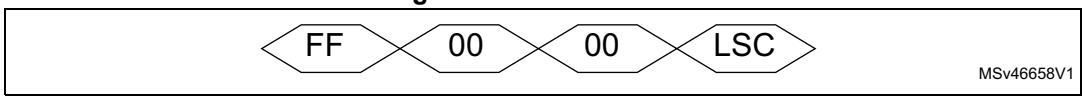
Figure 36. FSC structure



- **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 is programmed to 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 37. LSC structure

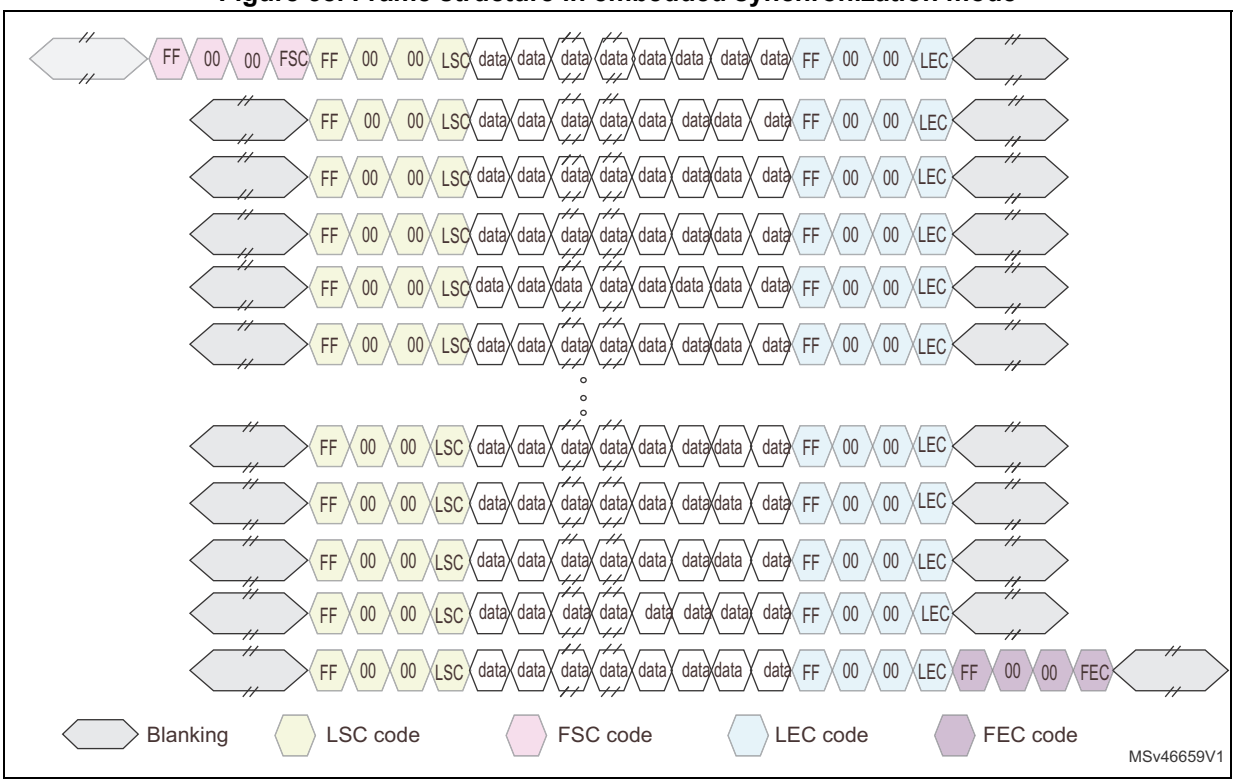


In this embedded synchronization mode, the HSPOL and VSPOL bits are ignored. While the DCMI is receiving data (CAPTURE bit set in the DCMI_CR register), the user can monitor the data flow, to know if it is an active line / frame or synchronization between lines / frames, by reading the VSYNC and HSYNC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

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

Figure 38 shows a frame received in embedded synchronization mode.

Figure 38. Frame structure in embedded synchronization mode



4.3 DCMI configuration

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

4.3.1 Capture mode

The user can capture an image or a video by selecting:

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

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

4.3.2 Data format

As mentioned previously, the DCMI allows the reception of the compressed data (JPEG) or many uncompressed data formats (such as monochrome, RGB, YCbCr).

For more details, refer to *Section 3.6: Data formats and storage*.

4.3.3 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. It is up to the DMA to ensure 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. The user can also configure and enable the crop feature to select a rectangular window from the received image.

For more details on these two features, please refer to *Section 3.7: Other features*.

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

4.4 DMA configuration

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

As mentioned in *Section 2.3: DCMI in a smart architecture*, the DMA2 ensures the transfer from the DCMI to the memory (internal SRAM or external SRAM/SDRAM) for all STM32 devices embedding the DCMI, except for STM32H7x3xx devices where the DMA1 can also access the AHB2 peripherals and ensure the transfer of the received data from the DCMI to the memory frame buffer.

4.4.1 DMA common configuration for DCMI-to-memory transfers

In the case of DCMI-to-memory transfer:

- The transfer direction must be peripheral-to-memory by configuring the DIR[1:0] bits in the DMA_SxCR register. In this case:
  - The source address (DCMI data register address) must be written in the DMA_SxPAR register.
  - The destination address (frame buffer address in internal SRAM or external SRAM/SDRAM) must be written in DMA_SxMAR register.
- To ensure the data transfer from the DCMI data register, the DMA waits for the request to be generated from the DCMI. So the relevant stream and channel must be configured. For more details refer to *Section 4.4.3: DCMI channels and streams configuration*.
- Since a DMA request is generated each time the DCMI data register is filled, the data transferred from the DCMI to the DMA2 (or the DMA1 for STM32H7x3xx devices) must have 32-bit width.So, The peripheral data width programmed in the PSIZE bits in the DMA_SxCR register must be 32-bit words.
- The DMA is the flow controller: the number of 32-bit data words to be transferred is software programmable from 1 to 65535 in the DMA_SxNDTR register (called DMA_CNDTRx in STM32L4x6 lines). For more details on this register, refer to *Section 4.4.4: DMA_SxNDTR register*.

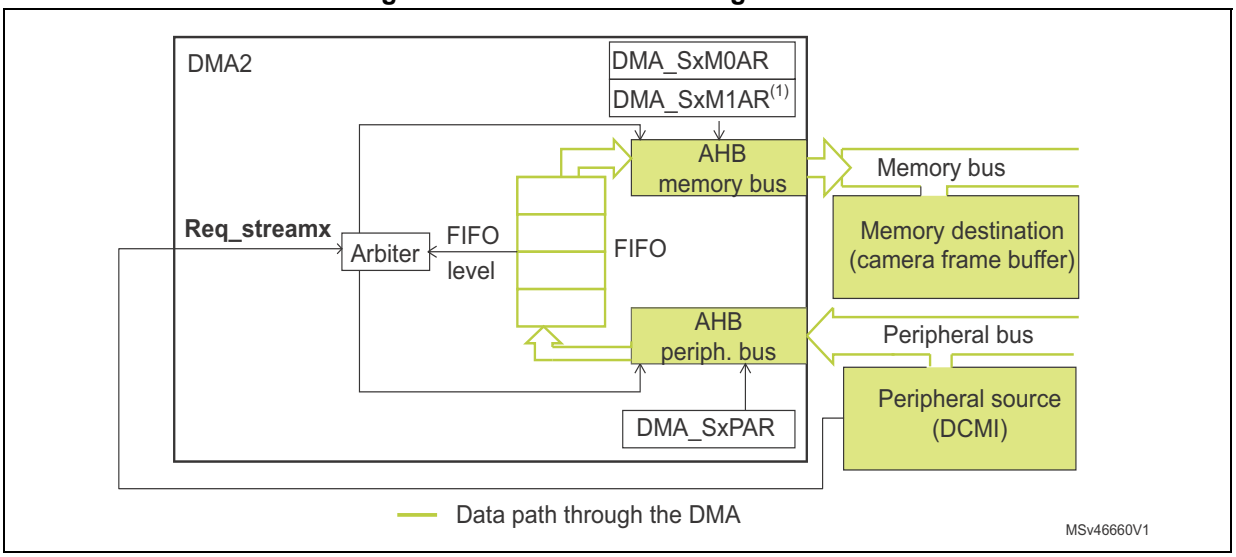
The DMA can operate in two modes:

- 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 4.4.5: FIFO and burst transfer configuration*.

Figure 39 shows the DMA2 (or the DMA1 for STM32H7x3xx devices) operation in peripheral-to-memory mode (except for STM32L496xx and STM32L4A6xx devices because the DMA2 in these devices has only one port).

Figure 39. Data transfer through the DMA



1. DMA_SxM1AR register is configured in case of double-buffer mode.

4.4.2 Setting DMA depending on the image size and capture mode

The DMA must be configured according to the image size (color depth and resolution) and 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. For more detailed description of this mode, refer to *Section 4.4.6: Normal mode for low resolution in snapshot capture*.
  - If the image size in words is between 65535 and 131070, the stream can be configured in double buffer mode. For more detailed description of this mode, refer to *Section 4.4.8: Double-buffer mode for medium resolutions (snapshot or continuous capture)*.
  - If the image size in words exceeds 131070, the stream can not be configured in double-buffer mode. For more detailed description of the mode that must be used, refer to *Section 4.4.9: DMA configuration for higher resolutions*.
- 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. For more detailed description of this mode, refer to *Section 4.4.7: Circular mode for low resolution in continuous capture*.
  - If one image size in words is between 65535 and 131070, the stream can be configured in double buffer mode. For more detailed description of this mode, refer to *Section 4.4.8: Double-buffer mode for medium resolutions (snapshot or continuous capture)*.
  - If one image size in words exceeds 131070, the stream can not be configured in double-buffer mode. For more detailed description of the mode that must be used, refer to *Section 4.4.9: DMA configuration for higher resolutions*.

65535*sizeof(int)
=65535*4=262140(byte)
=262.14KB

163840

Yantai iRay Infrared Camera
640*512*16bits
=655360(bytes)
=163840(int)

4.4.3 DCMI channels and streams configuration

The user must also configure the corresponding DMA2 (or the DMA1 for STM32H7x3xx devices) stream and channel to ensure the DMA acknowledgment each time the DCMI data register is fulfilled.

Table 6 summarizes the DMA channels enabling DMA request from the DCMI.

Table 6. DMA stream selection across STM32 devices

STM32 Series	DMA stream	Channel
STM32F2	Stream 1	Channel 1 or channel 7
STM32F4		
STM32F7		
STM32L4	Stream 0	Channel 6
	Stream 4	Channel 5
STM32H7	Stream 0 Stream 1 Stream 2 Stream 3 Stream 4 Stream 5 Stream 6 Stream 7	Multiplexer1 request 74

Note: For a step by step description of the stream configuration procedure, refer to the relevant STM32 reference manual.

4.4.4 DMA_SxNDTR register

```
#define __HAL_DMA_GET_COUNTER(__HANDLE__) ((__HANDLE__)->Instance->CNDTR)
```

*Note: This register is called **DMA_CNDTRx** in STM32L496xx and STM32L4A6xx devices.*

The total number of words to transfer from the peripheral source (DCMI) to the memory destination is programmed in this register by the user.

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).

Table 7 resumes the number of bytes corresponding to the programmed value and the peripheral data width (PSIZE bits):

Table 7. Maximum number of bytes transferred during one DMA transfer

DMA_SxNDTR programmed value	Peripheral size	Number of bytes
65535	Words	262140
N(1)	Words	4 * N

1. 0 < N < 65535.

Note: To avoid data corruption, the value programmed in the DMA_SxNDTR must be a multiple of MSIZE value / PSIZE value.

4.4.5 FIFO and burst transfer configuration

The DMA performs the transfer with or without enabling the 4-word FIFO. As mentioned previously, when the FIFO is enabled the source data width (programmed in PSIZE bits) can differ from the destination data width (programmed in MSIZE bits). 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 the PSIZE and MSIZE bits in the DMA_SxCR register. 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 to guarantee better performance.

4.4.6 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 frame having low resolution (see *Table 7*).

Table 8 summarizes the maximum image sizes that can be transferred using the normal mode.

Table 8. Maximum image resolution in normal mode

Item	Maximum number of bytes	Bit depth (bytes per pixel)(1)	Maximum number of pixels	Maximum resolution
Word	262140 262140/4=65535	1	262140	720x364
		2	131070	480x272

1. 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.

4.4.7 Circular mode for low resolution in continuous capture

Low resolution images are the ones having size (in 32-bit word) less than 65536.

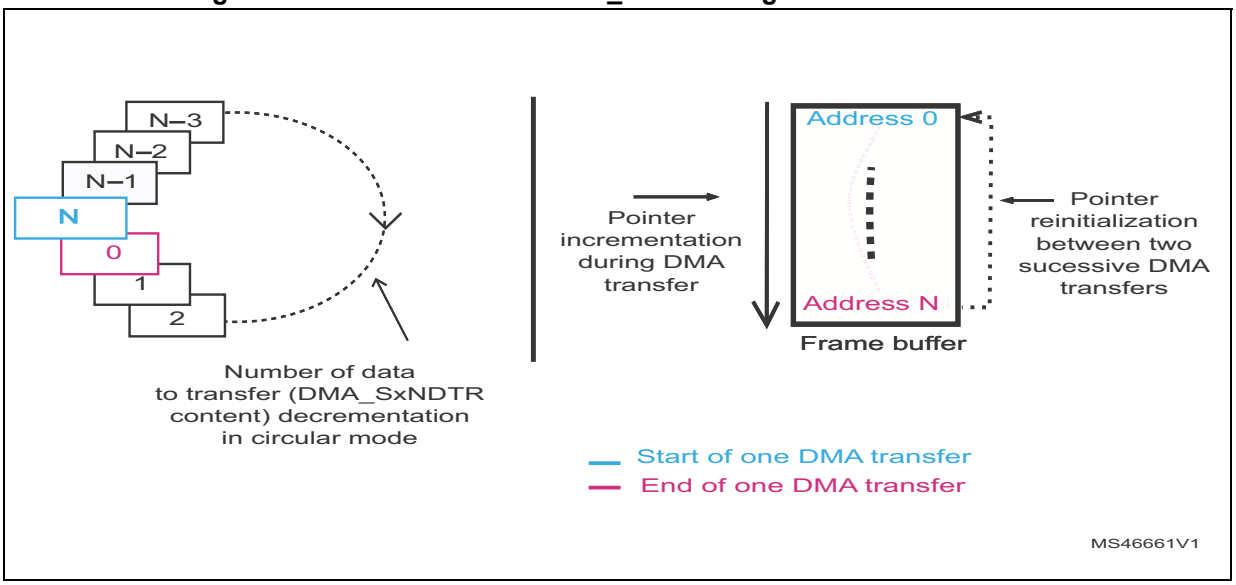
This circular mode allows the process of successive frames (continuous data flows), providing that one frame size (the initial programmed value in the DMA_SxNDTR register (DMA_CNDTR for STM32L4 Series)) is less than 65535.

Each time the number of data decrementing reaches the zero, the number of data words is automatically reloaded to the initial value. And each time the DMA pointer reaches the end of the frame buffer, it is reinitialized (returns to the programmed address in DMA_SxM0AR) and the DMA ensures the transfer of the next frame.

Resolutions listed in *Table 8* are also valid for the low resolution in continuous mode.

Figure 40 shows the DMA_SxNDTR value and the frame buffer pointer modifications during a DMA transfer and between two successive DMA transfers.

Figure 40. Frame buffer and DMA_SxNDTR register in circular mode



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

Note: This mode is not available in STM32L4A6xx and STM32L496xx 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 8* 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 the received data is stored in two buffers, each one maximum size (in 32-bit words) is 65535 (the maximum frame size is 131070 words or 524280 bytes). As a result the images sizes and resolutions allowed to be received by the DCMI and transferred by the DMA are doubled, as shown in *Table 9*.

Table 9. Maximum image resolution in double-buffer mode

Item	Maximum number of bytes	Bit depth (bytes per pixel)	Programmed value in SxNDTR register	Number of pixels	Maximum resolution
Word	524280	1	65535	524280	960x544
		1	N(1)	8 * N	960x544
		2	65535	262140	720x364
		2	N(1)	4 * N	720x364

1. 0 < N < 65536.

640*512=327680

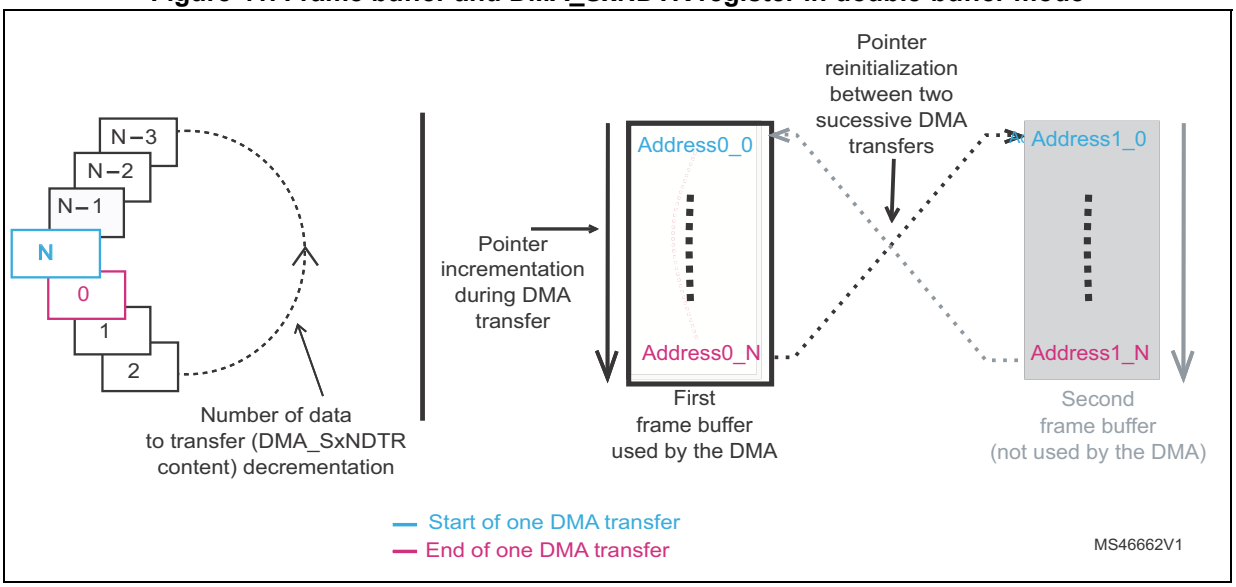
In this mode, the double-buffer stream has two pointers (two buffers for storing data), switched each end of transaction:

- 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 SxNDTR register is reinitialized to the programmed value and the DMA pointer switches to the second frame buffer), the data is transferred to the second buffer. In fact, the frame total size (in words) is divided by two and programmed in the SxNDTR register and 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 SxNDTR 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 the DBM bit in the DMA_SxCR register.

Figure 41 shows the two pointers and the DMA_SxNDTR value modifications during the DMA transfers.

Figure 41. Frame buffer and DMA_SxNDTR register in double-buffer mode



4.4.9 DMA configuration for higher resolutions

高分辨率，只能使用这种模式了！

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 9*, 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 6.3.6: SxGA resolution capture (YCbCr data format).

是不是只有这几个型号支持呀？？？怎么L496没有这2个寄存器呢？？？

STM32F2, STM32F4, STM32F7 and STM32H7 Series 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 the **CT bit** is set to **zero** in the DMA_SxCR register (current target memory is memory 0), the DMA_Sx**M1**AR register can be written.

Attempting to write to this register while CT is set to one, generates an error flag (TEIF) and the stream is automatically disabled.

- When the **CT bit** is set to **one** in the DMA_SxCR register (current target memory is memory 1), the DMA_Sx**M0**AR register can be written.
  Attempting to write to this register while CT is set to zero, 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 the DMA_SxCR register.

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 more than two buffers management:

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

The DMA allows then to use its two registers DMA_SxM0AR and DMA_SxM1AR, 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.

1280*1024*16bits/32bits=655360

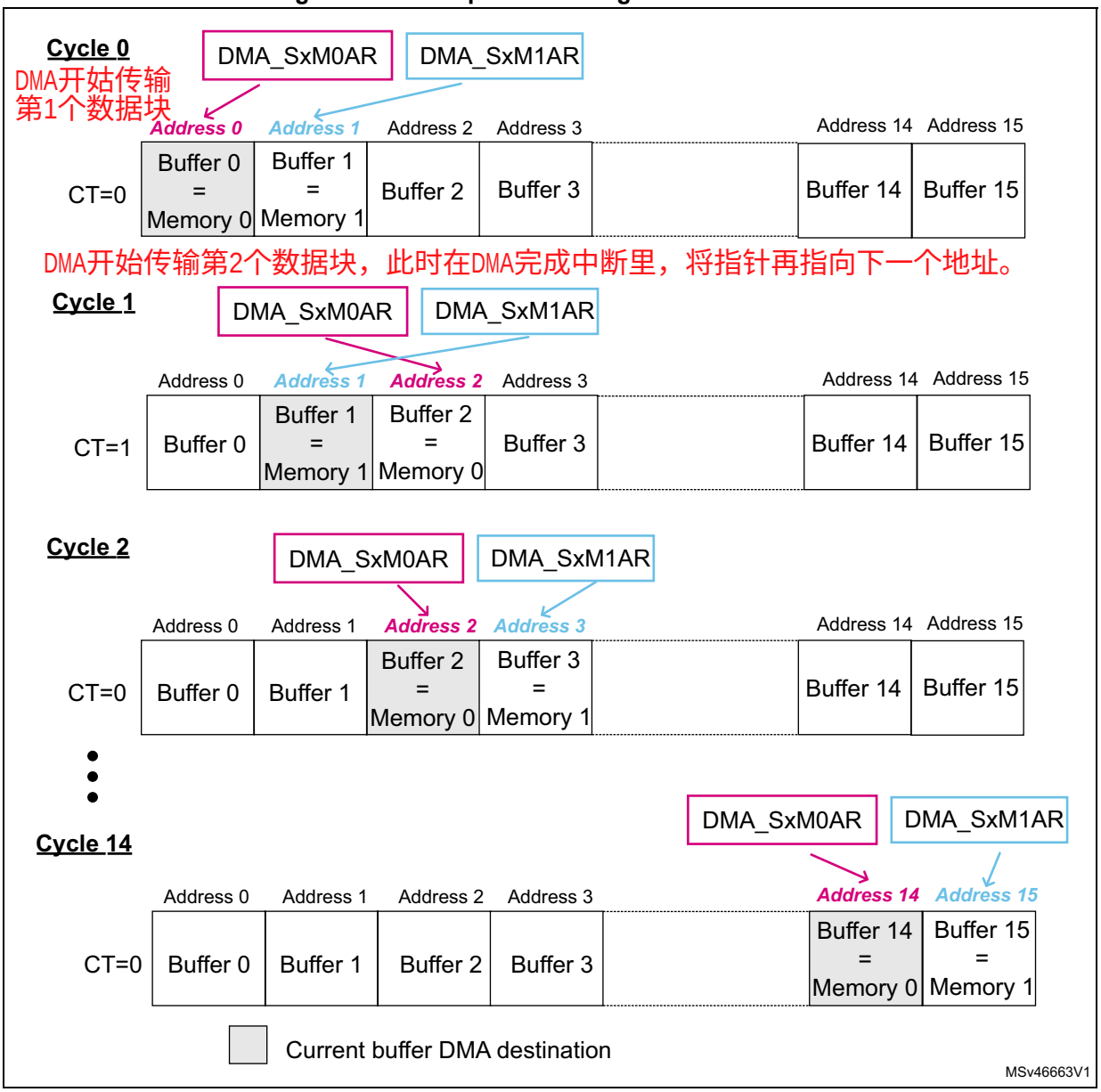
Example: In case of SxGA resolution (1280x1024), the image size is 655360 words (32 bits). This size must be divided into equal buffers, with a maximum size of 65535 for each of them. To be correctly received, the image must then be divided into 16 frame buffers, each frame buffer size equal to 40960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

655360/16=40960<<65535

Figure 42 illustrates the update of the DMA_SxM0AR and DMA_SxM1AR registers during the DMA transfer:

为什么没有这2个寄存器呢？难道改名了？？？
DMA_CMAR0 ???

Figure 42. DMA operation in high resolution case



4.5 Camera module configuration

To properly configure the camera module, the user needs to refer to its datasheet.

The following steps allow a correct configuration of the camera module:

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

5 Power and performance considerations

5.1 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 entering the STM32 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 drops to **1 mA** in case of I2C-initiated Standby (the internal circuit activity is suspended but the clock is not halted) and to **10 µA** in case of pin-initiated Standby (the internal device clock is halted and all internal counters are reset). For more details refer to relevant camera datasheet.

5.2 Performance considerations

For all STM32 MCUs, the number of bytes to be transferred 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.

Table 10 summarizes the maximum data flow depending on the data width configuration.

Table 10. Maximum data flow at maximum DCMI_PIXCLK(1)

STM32 Series		Extended data mode			
		8-bit	10-bit	12-bit	14-bit
Bytes per PICXCLK		1	1.25	1.5	1.75
Data flow (max Mbyte/s)	STM32F2	46.875	58.594	70.312	82.031
	STM32F4	52.734	65.918	79.101	92.285
	STM32F7	52.734	65.918	79.101	92.285
	STM32H7	78.125	97.656	117.187	136.718
	STM32L4	31.25	39.062	46.875	54.687

1. These values are calculated for the maximum DCMI_PIXCLK described in *Section Table 2.: DCMI and related resources availability*.

- In some applications, the DMA2 (or the DMA1 for STM32H7x3 devices) is configured to serve in parallel other requests together with the DCMI request. In this case, the user

must pay attention to the streams priorities configurations and consider the performance impact when the DMA is serving 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 STM32 DCMI to avoid the overrun.
- It is recommended to use the highest system speed HCLK for better performance, but the user must consider all the used peripherals speed (for example external memories speed) to avoid the overrun and to guarantee the success of his application.
- The DCMI is not the only AHB2 peripheral but there are many other peripherals and 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 and the user must consider its impact on the performance.

6 DCMI application examples

This section depicts a bunch of information connected to using the DCMI and provides step-by-step examples implementation.

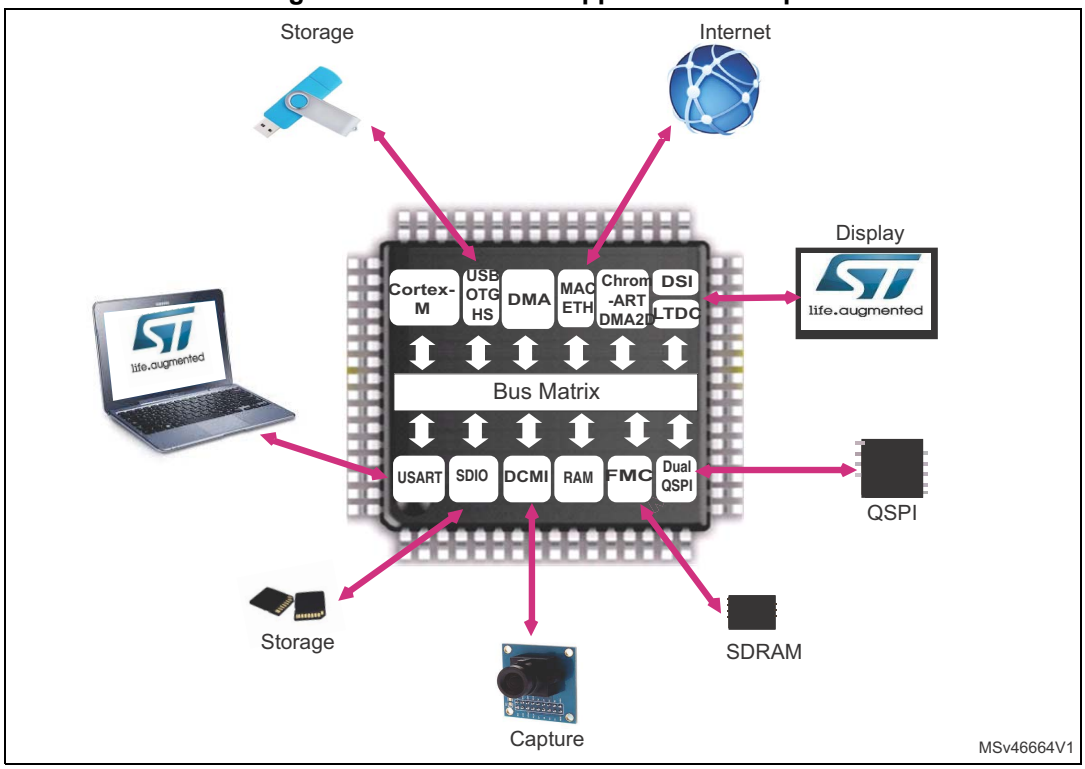
6.1 DCMI use case examples

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

Figure 43. STM32 DCMI application example



6.2 STM32Cube firmware examples

The STM32CubeF2, STM32CubeF4, STM32CubeF7 and STM32CubeL4 firmware packages offer a large set of examples implemented and tested on the corresponding boards. *Table 11* offers an overview of the DCMI examples and applications across the different STM32Cube firmware.

Table 11. STM32Cube DCMI examples

Firmware package	Project name(1)	Board
STM32CubeF2	DCMI_CaptureMode	STM3220G-EVAL STM3221G-EVAL
	SnapshotMode	
	Camera_To_USBDisk	
STM32CubeF4	DCMI_CaptureMode	STM32446E-EVAL STM324x9I-EVAL STM324xG-EVAL STM32446E-EVAL
	SnapshotMode	
	Camera_To_USBDisk	
STM32CubeF7	DCMI_CaptureMode	STM32756G-EVAL STM32F769I-EVAL
	SnapshotMode	
	Camera_To_USBDisk	
STM32CubeL4	DCMI_CaptureMode	32L496GDISCOVERY
	SnapshotMode	

1. All the 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.

6.3 DCMI examples based on STM32CubeMX

This section provides the description of five typical examples of using the DCMI:

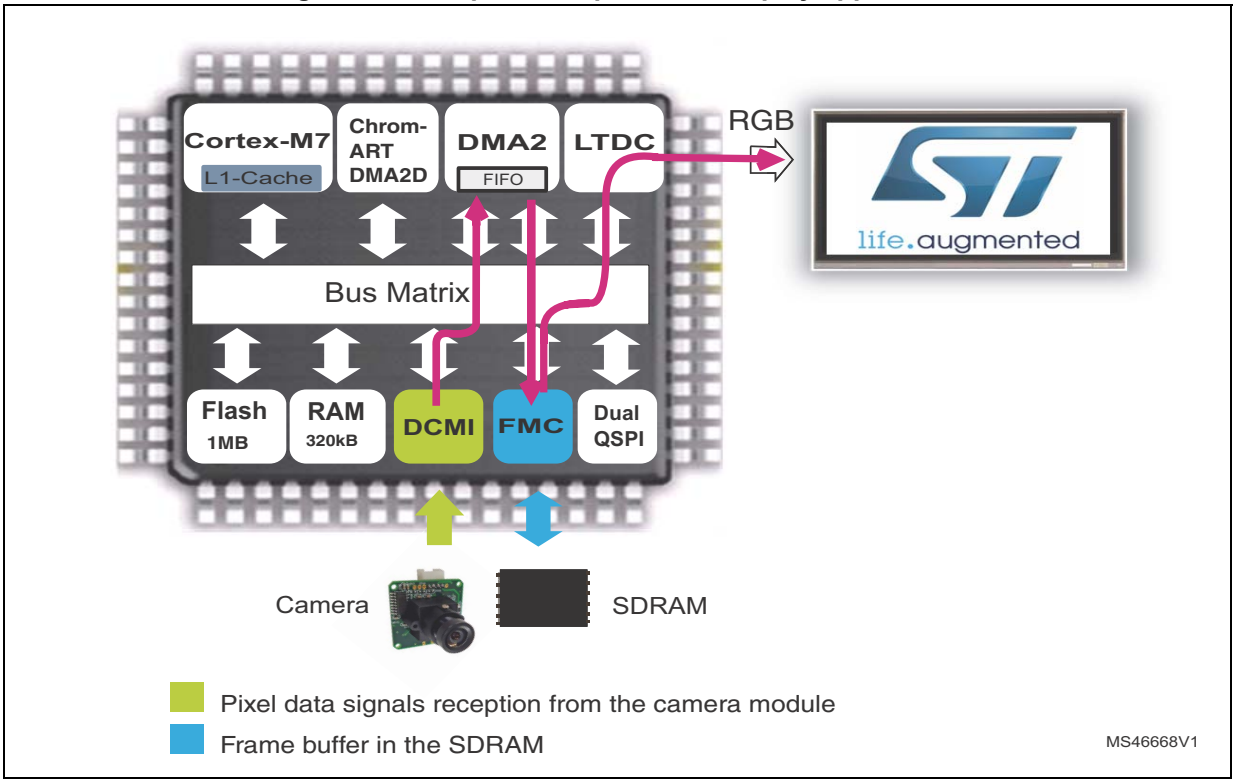
- Capture and display of RGB data: the data is captured in RGB565 format with QVGA (320x240) resolution, stored in the SDRAM and displayed on the LCD-TFT.
- Capture of YCbCr data: the data is captured in YCbCr format with QVGA (320x240) resolution and stored in the SDRAM.
- Capture of Y-only data: the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

All these examples were 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 44*, the application consists of three main steps:

- importing the received data from the DCMI to the DMA (to be stored in FIFO temporarily) through its peripheral port.
- transferring the data from the FIFO to the SDRAM
- importing 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 44. 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 following 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:

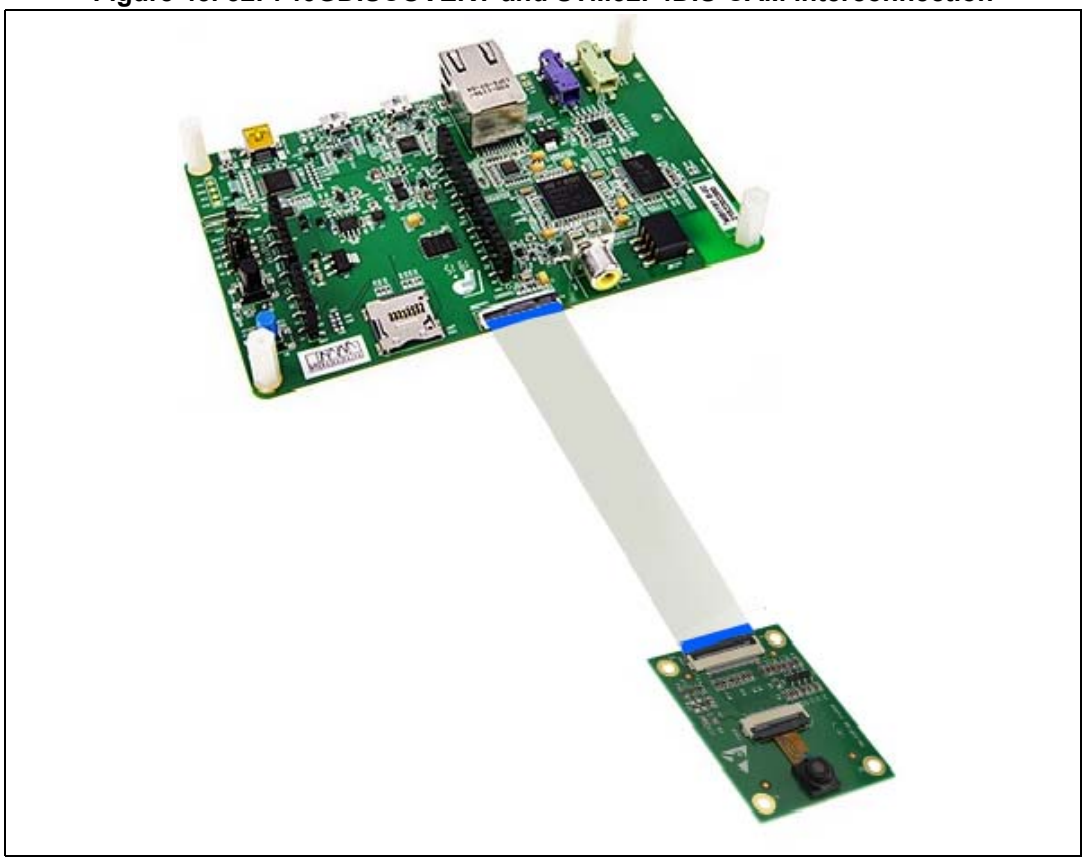
- DCMI peripheral configuration
- Camera module configuration

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

6.3.1 Hardware description

The following examples (except the JPEG capture example) were implemented on 32F746GDISCOVERY using the camera board STM32F4DIS-CAM.

Figure 45. 32F746GDISCOVERY and STM32F4DIS-CAM interconnection



1. Picture is not contractual.

The STM32F4DIS-CAM board includes an Omnivision CMOS sensor (ov9655), 1.3 mega-pixels. 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 46*, 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:

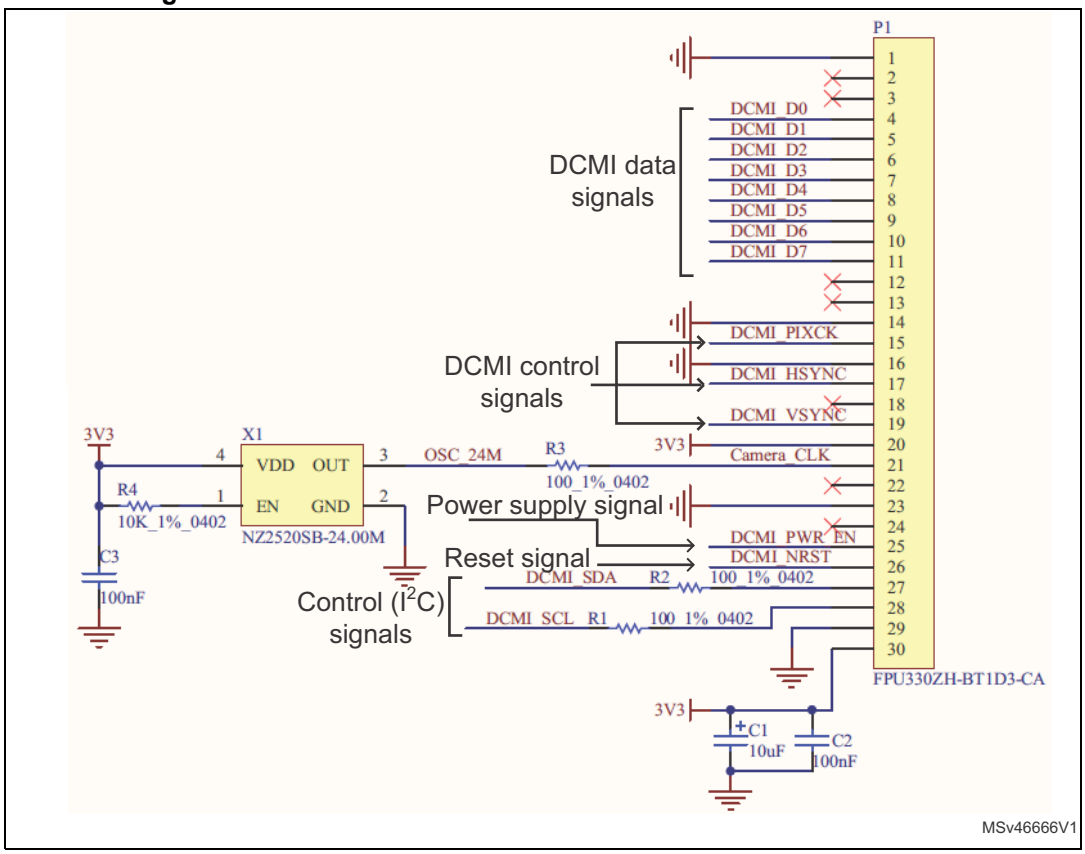
- power supply signals (DCMI_PWR_EN)
- clock for the camera module (Camera_CLK)
- configuration signals (I2C)
- reset signal (DCMI_NRST)

For more details on these signals, please refer to *Section 1.2.2: 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) allowing to reset the camera module is connected to the global MCU reset pin (NRST).

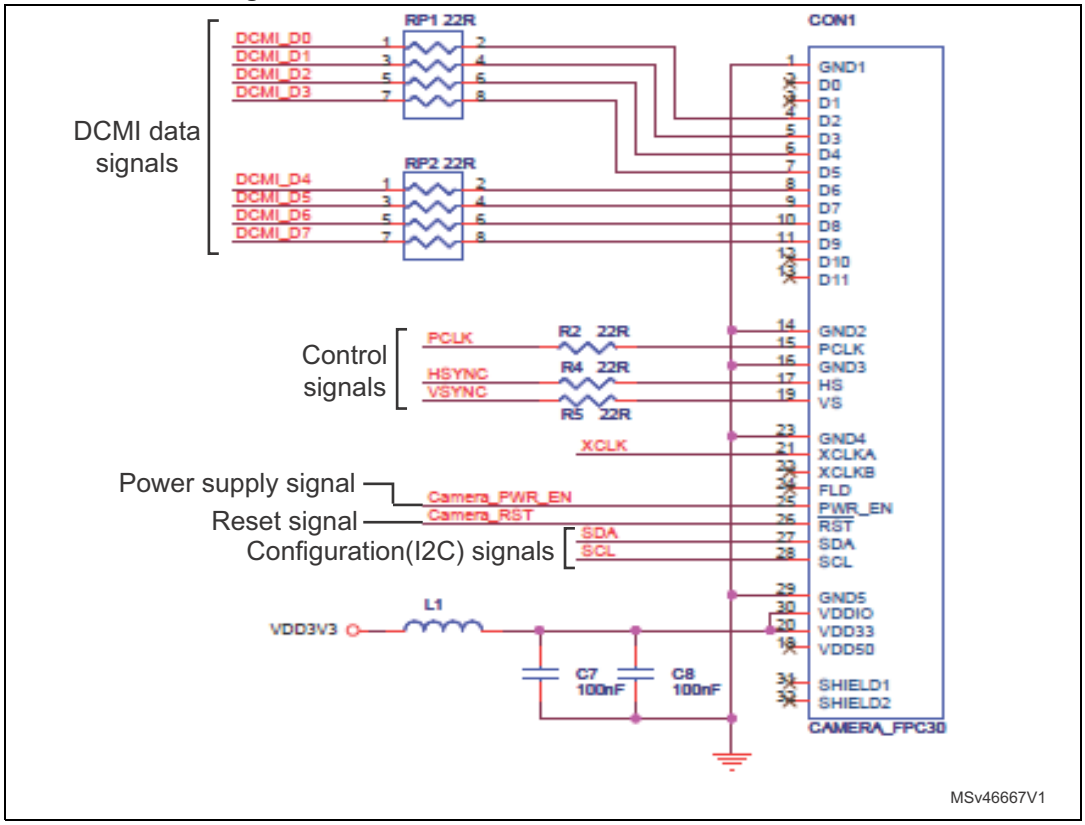
Figure 46. Camera connector on the 32F746GDISCOVERY board



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

The camera module connector implemented on STM32F4DIS-CAM is illustrated in the *Figure 47*.

Figure 47. Camera connector on STM32F4DIS-CAM



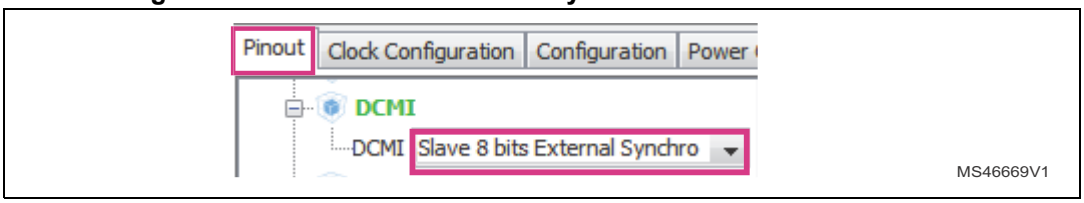
6.3.2 Common examples configuration

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

STM32CubeMX - DCMI GPIOs configuration

1. Select the DCMI and choose “Slave 8 bits External Synchro” in the Pinout tab to configure the DCMI in slave 8-bit external (hardware) synchronization (*Figure 48*).

Figure 48. STM32CubeMX - DCMI synchronization mode selection



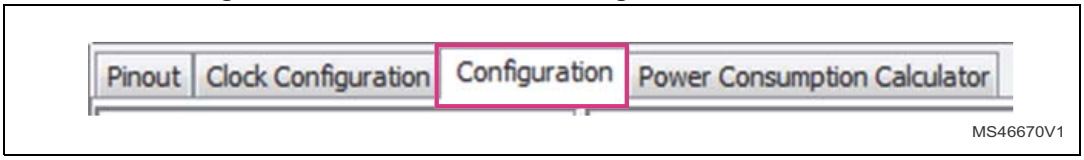
If after selecting one hardware configuration (Slave 8 bits External Synchro), the used GPIOs does not match with the hardware, the user can change the desired GPIO and configure the alternate function directly on the pin.

Another method consists of configuring the GPIO pins manually by selecting the right alternate function for each of them. For more details on the GPIOs that must be configured, refer to *Figure 52: STM32CubeMX - DCMI pins selection*.

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 and speed, as shown in *Figure 51*.

Figure 49. STM32CubeMX - Configuration tab selection

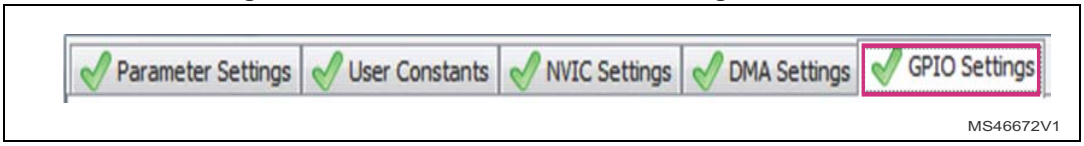


3. Click on the DCMI button in the configuration tab as shown in *Figure 50*.

Figure 50. STM32CubeMX - DCMI button in the Configuration tab

4. When the DCMI configuration window appears, select the GPIO settings tab as shown in *Figure 51*.

Figure 51. STM32CubeMX - GPIO settings selection

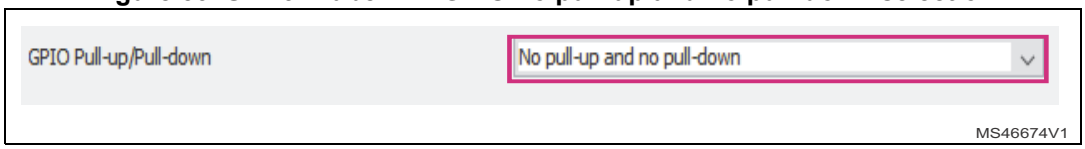


5. Select all the DCMI pins as shown in *Figure 52*.

Figure 52. STM32CubeMX - DCMI pins selection

6. Set the GPIO pull-up / pull-down as shown in *Figure 53*.

Figure 53. STM32CubeMX - GPIO no pull-up and no pull-down selection

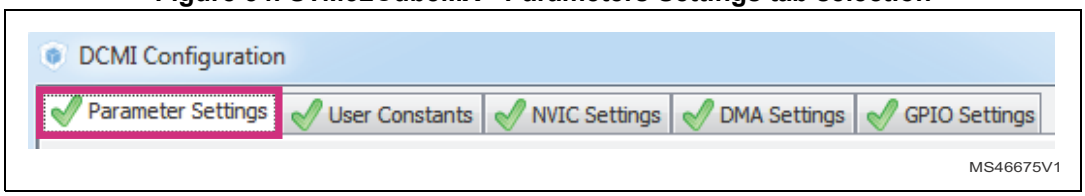


7. Click on Apply and OK.

STMCubeMX - DCMI control signals and capture mode configuration

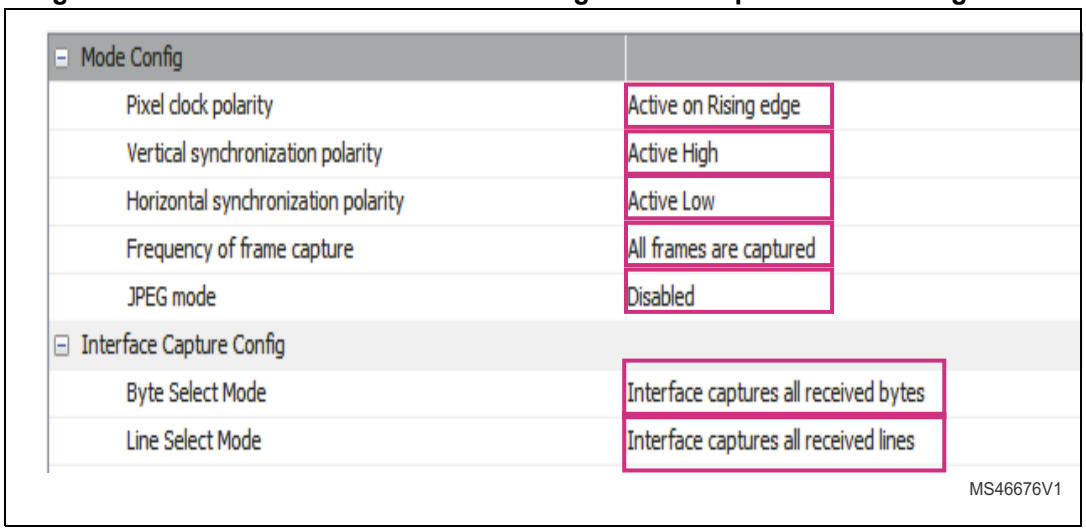
1. Click on the Parameter Settings tab in DCMI Configuration window, then select Parameter Settings tab, as shown in *Figure 54*.

Figure 54. STM32CubeMX - Parameters Settings tab selection



2. Set the different parameters as illustrated in *Figure 55*. The vertical synchronization, horizontal synchronization and pixel clock polarities must be programmed according to the camera module configuration.

Figure 55. STM32CubeMX - DCMI control signals and capture mode configuration



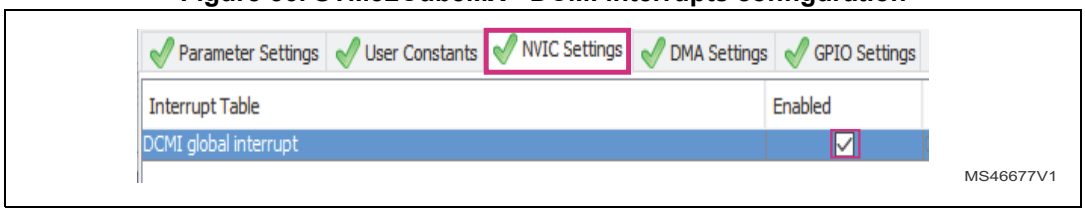
3. Click on Apply and OK.

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.

STM32CubeMX - Enabling DCMI interrupts

1. Select the NVIC Settings tab in the DCMI Configuration window and check the DCMI global interrupt as shown in *Figure 56*.

Figure 56. STM32CubeMX - DCMI interrupts configuration



2. Click on Apply and OK.

STM32CubeMX - DMA configuration

This configuration aims to receive RGB565 data (2 bytes per pixel) and the image resolution is QVGA (320x240). The image size is then 320 * 240 * 2 = 153600 bytes.

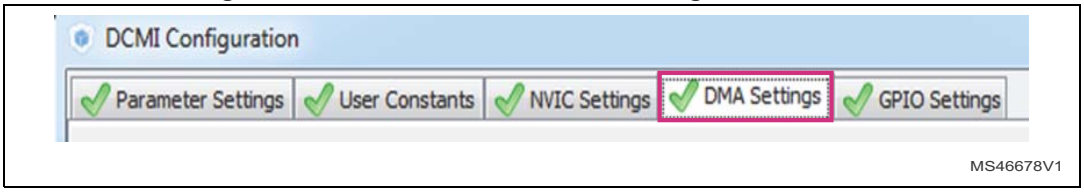
Since the data width sent from the DCMI is 4 bytes (32-bit words sent from the data register in the DCMI), the number of data items in the DMA_SxNDTR register is the number of words to transfer. The number of words is then 38400 (153600 / 4) which is less than 65535.

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

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

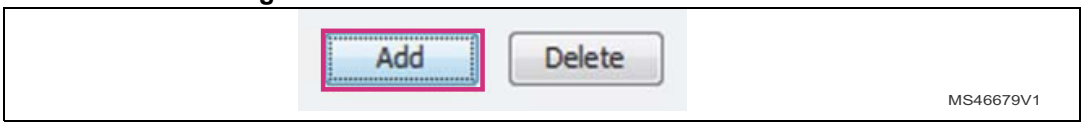
1. Select the DMA Setting tab in the DCMI Configuration window as shown in *Figure 57*.

Figure 57. STM32CubeMX - DMA Settings tab selection



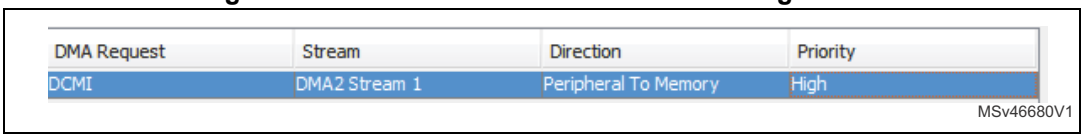
2. Click on the Add button illustrated in *Figure 58*.

Figure 58. STM32CubeMX - Add button selection



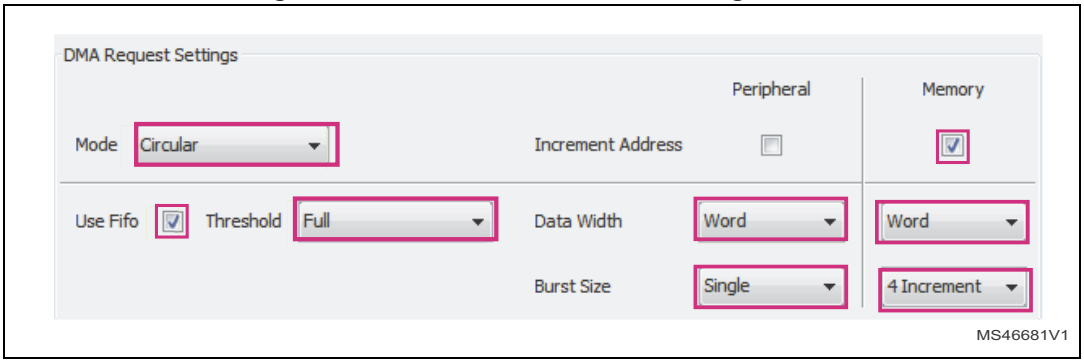
3. Click on Select under DMA Request and choose DCMI. The DMA request is configured as shown in *Figure 59*. The DMA2 Stream 1 channel 1 is configured to transfer the DCMI request each time its time register is fulfilled.

Figure 59. STM32CubeMX - DMA stream configuration



4. Modify the DMA Request Settings as shown in *Figure 60*.

Figure 60. STM32CubeMX - DMA configuration



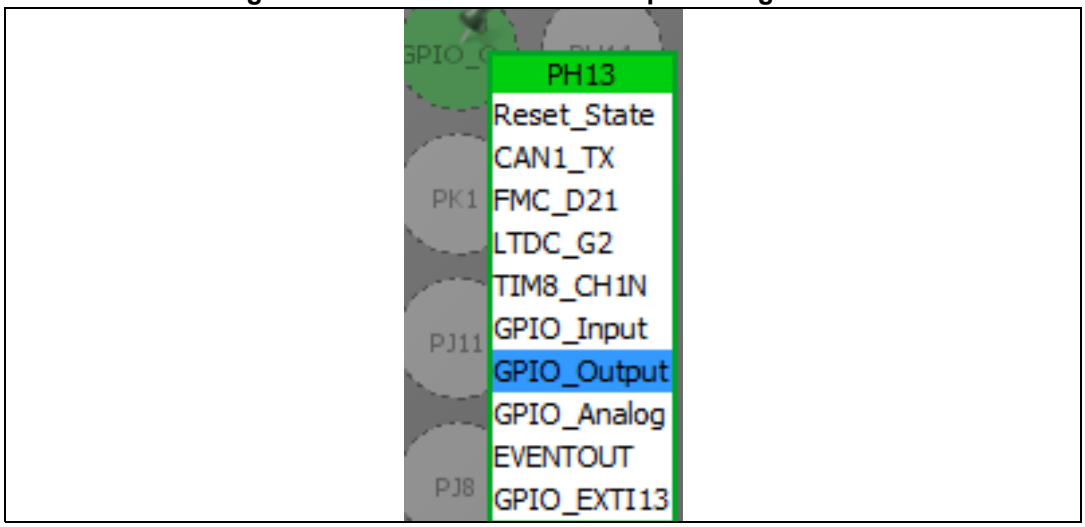
5. Click on Apply and OK.

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, as shown in *Figure 61*.

Figure 61. STM32CubeMX - PH13 pin configuration

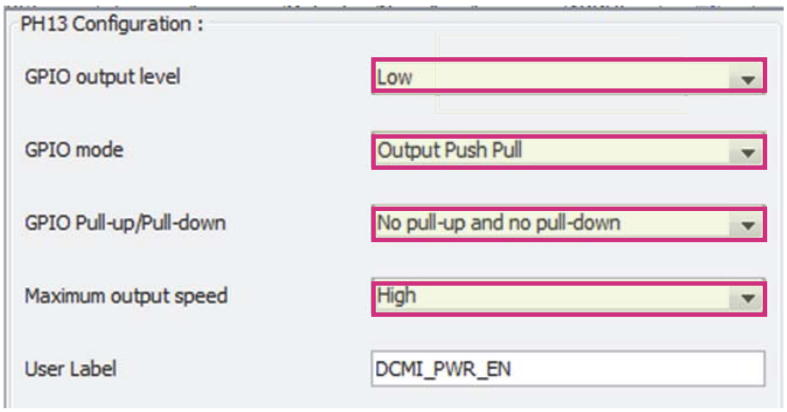


2. In the Configuration tab, click on the GPIO button illustrated in *Figure 62*.

Figure 62. STM32CubeMX - GPIO button in the configuration tab

3. Set the parameters as shown in *Figure 63*.

Figure 63. STM32CubeMX - DCMI power pin configuration



STM32CubeMX - System clock configuration

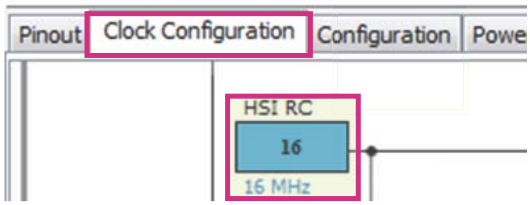
In this example the system clock is configured as follow:

- use of internal HSI RC, where main PLL is used as system source clock.
- HCLK @ 200 MHz, so 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.

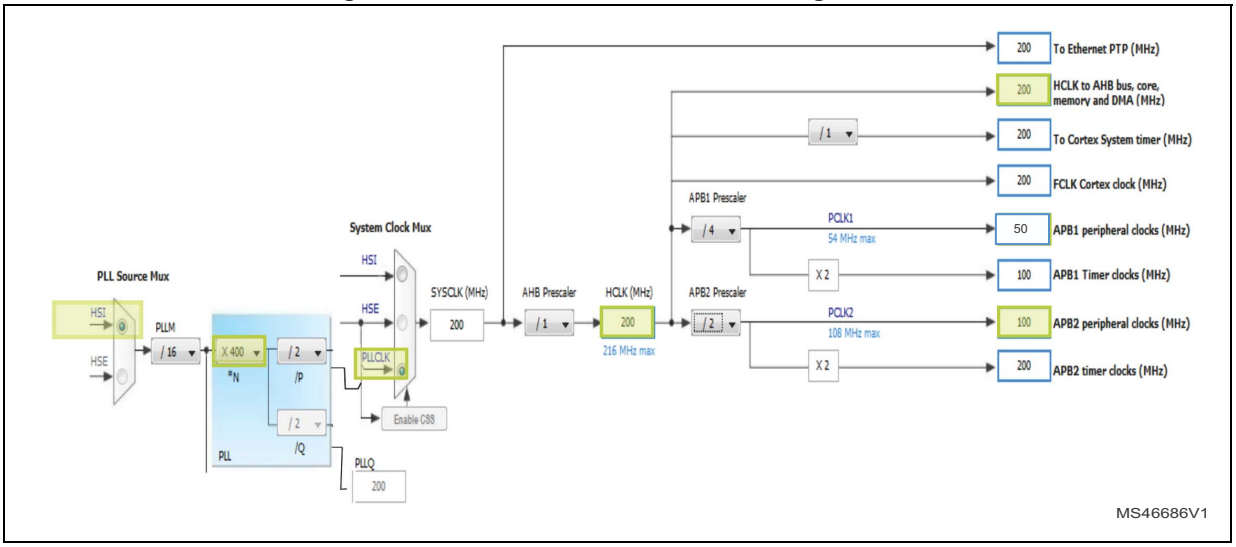
1. Select the Clock Configuration tab as shown in *Figure 64*.

Figure 64. STM32CubeMX - HSI configuration



2. Set the PLLs and the prescalers in the Clock Configuration tab, to get the system clock HCLK @ 200 MHz, as shown in *Figure 65*.

Figure 65. STM32CubeMX - Clock configuration



At this level, the user can generate the project.

Adding files to the project

Generate the code and open the generated project using the preferred toolchain and 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. Uncomment the modules DMA2D, LTDC, SDRAM, UART in **stm32f7xx_hal_conf.h.**
5. Create a new group called, for example, Imported_Drivers.
6. Copy the following files from the STM32746G_Discovery folder in the C: directory to the **Src** folder of the project:
   - **stm32746g_discovery.c**
   - **stm32746g_discovery_sdram.c**
7. Copy the following files from the STM32746G_Discovery folder in the C: directory to the **Src** folder of the project:
   - **stm32746g_discovery.h**
   - **stm32746g_discovery_sdram.h**
8. Copy **ov9655.c** from the Components folder to the Src folder.
9. Copy **ov9655.h** from the Components folder to the Inc folder.
10. Copy **camera.h** from the Component/Common folder to the Inc folder.
11. Add the following files in the new group (called Imported_Drivers in this example):
    - **stm32746g_discovery.h**
    - **stm32746g_discovery_sdram.h**
    - **ov9655.c.**
12. Allow modifications on **ov9655.h** and **camera.h** (read-only by default), by:
    - clicking right on the file
    - unchecking read-only
    - clicking on apply and OK.
13. Modify the **ov9655.h** file by replacing `#include "../Common/camera.h` by `#include "camera.h`.”
14. Copy the following files to the Inc folder:
    - **rk043fn48h.h** from Components folder
    - **fonts.h** and **fonts24.h** from Utilities/Fonts folder.
15. Check that no problem happened by rebuilding all files. There must be no error and no warning.

Modifications in main.c file

1. Update **main.c** by inserting some instructions to include the needed files in the adequate space, indicated in **green 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"
#include "font24.c"
/* USER CODE END Includes */
```

Then, it is necessary to insert some variables declarations in the adequate space indicated in **green 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 */
```

After that, it is necessary to insert the functions prototypes in the adequate space indicated in **green 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 **green bold** below. LTDC_Init function allows the configuration and initialization of the LCD. BSP_SDRAM_Init function allows the configuration and initialization of the SDRAM. CAMERA_Init function allows the configuration of the camera module and the DCMI registers and 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 **green 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 */
```

Modifications in main.h file

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

6.3.3 RGB data capture and display

To simplify this example, the data is captured and displayed in RGB565 format (2 bpp). The image resolution is 320x240 (QVGA). The frame buffer is placed in the SDRAM. The camera and the 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 then to output RGB565 data, QVGA (320x240).

The configuration of this example can be done by following the steps described in *Section 6.3.2: Common examples configuration*.

6.3.4 YCbCr data capture

Description

This example implementation aims to receive the YCbCr data from the camera module and to transfer it into 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, the YCbCr data must be converted into RGB565 data (or RGB888 or ARGB8888, depending on the application needs).

All the configuration steps signaled in *Section 6.3.2: Common examples configuration* must be followed and here are some instructions to be added to obtain the YCbCr data. Only the camera configuration has to be updated.

Camera module configuration:

The new camera module configuration is done by adding:

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

1. The declaration of the table containing the camera module registers configurations has to be added in main.c file 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 file** described in *Section 6.3.2: Common examples configuration* has to be updated. Modify the **main()** function by inserting the following functions in the adequate space, indicated in **green 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 6.3.2: Common examples configuration* 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);
      } }
```

6.3.5 Capture Y only data format

Description

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 and only the Y component is transferred to the frame buffer in the SDRAM.

All the configuration steps signaled in *Section 6.3.2: Common examples configuration* must be followed and here are some instructions to be added to obtain the Y only data. Only the camera and the DCMI configuration must be updated.

To simplify this task, the **main.c file** must be modified as described in *Section 6.3.4: YCbCr data capture* 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 the main.c file) must be modified to have the following configuration:

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;

6.3.6 SxGA resolution capture (YCbCr data format)

Description

This example implementation aims to receive the YCbCr data from the camera module and to transfer it into the SDRAM. The captured image(s) resolution is SxGA (1280x1024).

Displaying the YCbCr received data on the LCD (configured to display RGB565 data) is not correct.

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

All the configuration steps signaled in *Section 6.3.2: Common examples configuration* must be followed and here are some instructions to be added to obtain the YCbCr data. Only the camera and the DMA configuration have to be updated.

DMA configuration

The DMA is configured as described in *Section 4.4.9: DMA configuration for higher resolutions* and the HAL_DMA_START function when called ensures this configuration because the image size exceeds the maximum allowed size for double-buffer mode.

In fact, when calling HAL_DMA_START function, it 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 into 16 frame buffers. Each buffer size is equal to 40960 words.

For the buffers addresses, the HAL_DMA_START function 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)) and the 16^{th} 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 the *Figure 42: DMA operation in high resolution case*.

Camera module configuration:

The new camera module configuration is done by adding:

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

In order to ensure that the camera module is sending image having SxGA resolution and YCbCr format, the CMOS sensor registers must be configured as below:

1. The declaration of the table containing the camera module registers configurations has to be added in main.c file 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 folder** in this example is to update the **main()** function by inserting the following functions in the adequate space, indicated in **green 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 6.3.2: Common examples configuration* 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 TFT-LCD by using the resizing feature of the DCMI.

6.3.7 Capture of JPEG format

Description

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.

So, 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 for example, but this implementation aims only to receive the JPEG data through the DCMI and to store it in the SDRAM.

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

- The DCMI and I2C GPIOs are configured as described in *Section 6.3.2: Common examples configuration*.
- 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 register. To set this bit, the user must simply, in the stm324x9i_eval_camera.c file in "uint8_t BSP_CAMERA_Init(uint32_t Resolution)" function where the DCMI is configured (this function is called in the main() function to configure the DCMI and the camera module), add the instruction written in **bold** below and keep the DCMI previous configuration as shown below:

```
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 the CMOS sensor (ov2640) registers must be inserted in the ov2640.c file as given below:

```
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 I2C; In the same file (ov2640.c), in the function "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;
    }
```

7 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. In the *Table 12*, some camera modules are mentioned.

Table 12. Examples of support camera modules

CMOS sensor	Camera module	Formats	Parallel interface
OV9655	STM32F4DIS-CAM	– RGB – YCbCr	– 8-bit – 10-bit
OV7740	TD7740-FBAC	– RGB – YCbCr	– 8-bit – 10-bit
MT9M001	ArduCAM	– RGB	– 8-bit – 10-bit
OV5642	ArduCAM 5 Megapixels	– RGB – YCbCr	– 8-bit – 10-bit
MT9M111	CMOS camera	– RGB – YCbCr	– 8-bit
MT9P031	HDCAM	– RGB	– 8-bit – 10-bit – 12-bit
OV3640	3 Megapixels	– RGB – YCbCr – JPEG	– 8-bit – 10-bit

8 Conclusion

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

9 Revision history

Table 13. Document revision history

Date	Revision	Changes
3-Aug-2017	1	Initial release.