AN5051
Application note

Graphic memory optimization with STM32 Chrom-GRC™

Introduction

LCD technology used to be exclusive to rectangular-shape displays. Recent developments enable the creation of non-rectangular LCD displays. These displays are becoming very attractive for a wide variety of applications such as wearable devices.

The STM32 Chrom-GRC™ (GFXMMU) peripheral is a new addition to STM32 microcontrollers (refer to *Table 1: Applicable products*) aiming to efficiently support this emerging trend towards non-rectangular displays.

GFXMMU enables to store only the visible parts of the non-rectangular displays and in the case of round displays, this peripheral reduces by 20% the memory requirements to store the graphic framebuffer. As a consequence, GFXMMU avoids the need to add an external component for SRAM/SDRAM to the microcontroller.

By removing the need for external RAM and taking advantage of the low-power and high-performance features of the internal RAM, the STM32 microcontrollers embedding GFXMMU offer a suitable solution for wearable applications that require both low-power management functions and a high-quality user interface.

Table 1. Applicable products

Type	Part number
Microcontrollers	STM32L4+ Series

Contents

List of tables

List of figures

1 STM32 Chrom-GRC™ (GFXMMU) introduction

The GFXMMU is a graphical oriented memory management unit which aims to optimize the memory usage depending on the display shape.

This peripheral allows the microcontroller to store only the visible parts of non-rectangular displays in a contiguous physical memory area, reducing the framebuffer memory footprint.

By enabling the framebuffer to be stored in the internal RAM and eliminating the need for external RAM, the GFXMMU provides a highly integrated solution for graphic applications. This peripheral leads to better performance, lower power consumption and lower system cost.

1.1 GFXMMU features

The main features of GFXMMU are listed below:

- Lower memory usage according to display shape
- Fully configurable display shape
- Transparent integration
- Works with any memory of the system

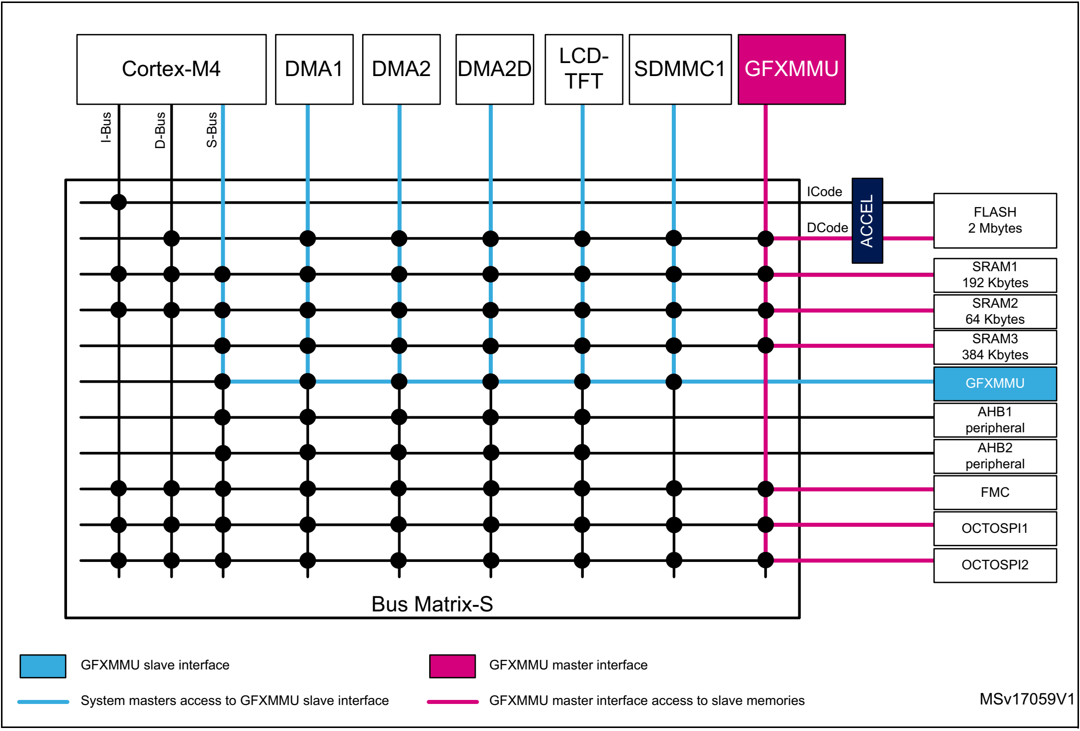
1.2 GFXMMU in a smart architecture

The GFXMMU has both a master and a slave interface. The master interface manages the access to the different slave memories (Flash, SRAM, FMC, OCTOSPI). The slave interface is accessed by different masters (LTDC, DMA2D, Cortex M, DMA, SDMMC).

The system masters access the graphic framebuffer through the GFXMMU. The GFXMMU receives read or write requests on its slave interface and performs an address resolution to determine the physical address targeted. Then it redirects the transfer request through its master interface to the memory which holds the physical address.

The STM32L4+ Series are the first STM32 products to integrate the GFXMMU. *Figure 1* shows the STM32L4+ Series system architecture embedding the GFXMMU.

Figure 1. Example STM32L4+ Series system architecture with GFXMMU



2 GFXMMU virtual buffers

The GFXMMU virtual buffer allows only the visible parts of non rectangular displays to be stored in contiguous physical memory area.

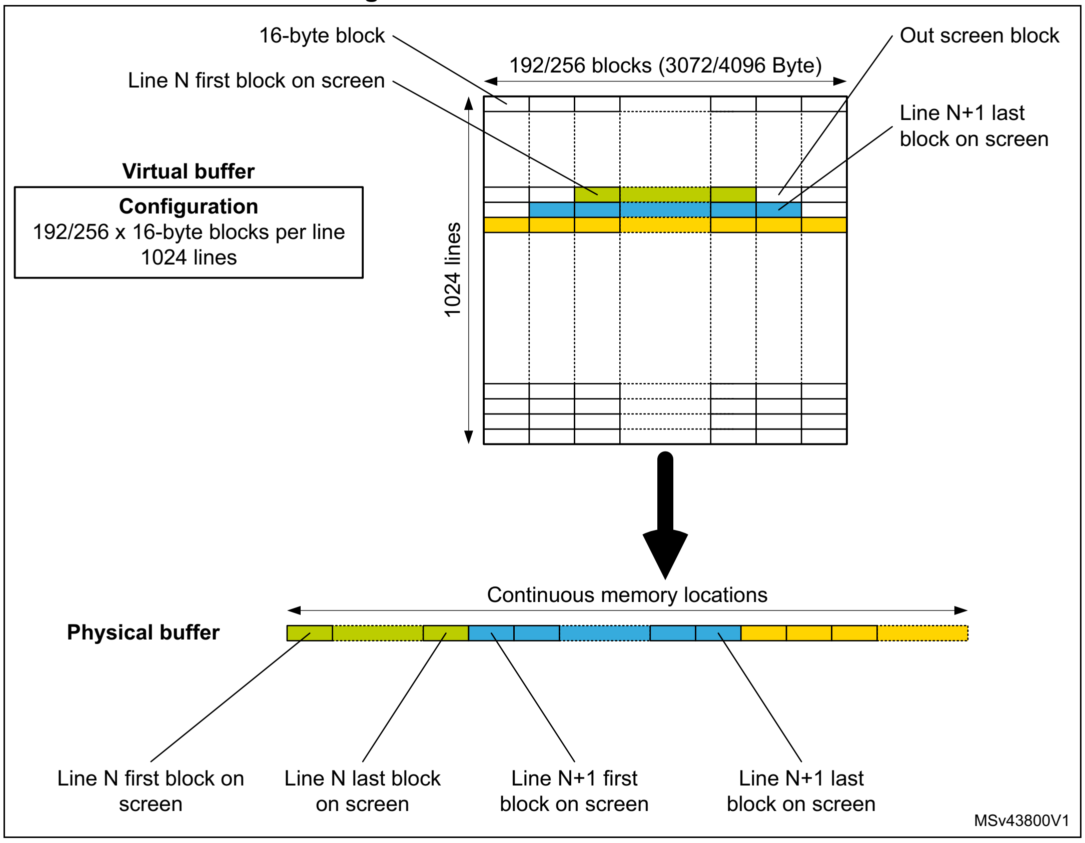
2.1 Virtual buffer overview

The virtual buffer has 3072 or 4096 bytes per line and 1024 lines.

Only the visible parts of the display are mapped into the physical memory space.

Figure 2 presents an overview of the GFXMMU virtual buffer.

Figure 2. Virtual buffer overview


16-byte block
Line N first block on screen
192/256 blocks (3072/4096 Byte)
Out screen block
Line N+1 last block on screen
Virtual buffer
Configuration
192/256 x 16-byte blocks per line
1024 lines
1024 lines
Continuous memory locations
Physical buffer
Line N first block on screen
Line N last block on screen
Line N+1 first block on screen
Line N+1 last block on screen
MSv43800V1

2.2 Virtual buffer usage

When using the GFXMMU, the graphic framebuffer is accessed through the virtual buffer. In this case, the LTDC and the DMA2D must have a specific configuration to take into account the virtual buffer line width.

2.2.1 Virtual buffer usage with LTDC

The LTDC layer pitch is the increment in bytes from the start of one line to the start of the next line. It is configured in the LTDC_LxCFBLR register and it is expressed in bytes.

When the LTDC is using the GFXMMU virtual buffer, the LTDC layer pitch is equal to the virtual buffer line width in bytes which is either 3072 or 4096.

2.2.2 Virtual buffer usage with DMA2D

The DMA2D buffer line offset is added at the end of each line to determine the starting address of the next line. The DMA2D buffer line offset is expressed in pixels.

The DMA2D buffer line offset, when using GFXMMU virtual buffer, is given by following formula:

Line offset = virtual buffer line width in pixels - image width in pixels

When the DMA2D is using the virtual buffer, the line width must have an integer number of pixels.

Note: In case of a 24 bpp framebuffer, having an integer number of pixels is guaranteed only when the virtual buffer has 3072 bytes width which corresponds to 1024 pixels.

2.3 Virtual buffer operating mode

To insure that the virtual buffer has an integer number of pixels per line with different framebuffer color depths, two operating modes can be used:

- 256 block mode
  In this mode the virtual buffer has 256 blocks of 16 bytes per line.
  This mode corresponds to a line width of 256 x 16 = 4096 bytes.
- 192 block mode
  In this mode the virtual buffer has 192 blocks of 16 bytes per line.
  The Line width in this mode is 192 x 16 = 3072 bytes.

The 192 block mode is introduced to have an integer number of pixels per line when using 24 bpp buffers.

Table 2 presents the virtual buffer line width in pixels for different framebuffer color depths.

Table 2. Virtual buffer line width in pixels

-	32 bpp	24 bpp	16 bpp	8 bpp
192 block mode	768	1024	1536	3072
256 block mode	1024	1365.3(1)	2048	4096

1. The 256 block mode shall be avoided in 24 bpp framebuffers in order to have an integer number of pixels per line.

3 Display shape description

The GFXMMU allows the MCU to map only the necessary blocks to a physical memory location depending on the display shape and size. The display shape description is stored in a look up table (LUT).

3.1 LUT configuration

The LUT must be configured by specifying for each line:

- The enable of the line
- The number of the first visible block
- The number of the last visible block
- The address offset of the line within the physical buffer

The visible blocks can be arranged in the physical buffer in a continuous way by programming the address offset of each line.

3.2 LUT calculation example

The GFXMMU LUT entries calculation is explained in this section.

The example is based on the 390 x 390 round display of the 32L4R9IDISCOVERY kit. In this example, the framebuffer has a 16 bpp color format.

The first and last visible pixels of each line are provided by the display manufacturer.

Table 3 describes the first and last visible pixels of the round display. Only the first four lines are presented in this example.

Table 3. Visible pixels description for the round display of the 32L4R9IDISCOVERY kit

Line number	First visible pixel	Last visible pixel
Line 0	181	208
Line 1	172	217
Line 2	164	225
Line 3	158	231

Block number calculation

GFXMMU has 16 bytes granularity, which corresponds to a block.

The block number calculation is based on the pixel number and the framebuffer color depth.

The color depth is expressed in bytes per pixel (Bpp) in the following equations.

The first visible block is the block that holds the first byte of the first pixel.

First visible block = (first visible pixel x Bpp) / block size

The last visible block is the block that holds the last byte of the last pixel:

Last visible block = (last visible pixel x Bpp + Bpp - 1) / block size

Using the *Table 3* as an example, the visible pixels of line 0 are comprised between pixels 181 and 208, so:

First visible block of line 0 = (181 x 2) / 16 = 22

Last visible block of line 0= (208 x 2 + 1) / 16 = 26

Line offset calculation

The line offset defines the offset of the first visible block of a line in the physical buffer. It allows the visible blocks to be arranged in the physical buffer in a continuous way.

The line offset is coded on 22 bits and can have negative values, it is calculated as follows:

Line offset = (number of visible blocks already used - first visible block) x block size

where:

- number of visible blocks already used includes the visible blocks of all the previous lines
- first visible block refers to the current line
- block size = 16 bytes

  Line 0 offset = (0-22) x 16 = -352 = 0x3F:FEA0

  Line 1offset = (5-21) x 16 = -252 = 0x3F:FF00

After calculating the first and last visible blocks and the line offset for each line, the LUT entries are programmed as follows:

- LUT entry x low
  - LUTxL[23:16] is programmed with the last visible block value
  - LUTxL[15:8] is programmed with the first visible block value
  - LUTxL[0] is set when the line is enabled

  Line 0 LUT entry Low: LUT0_L=0x001A1601
- LUT entry x high
  - LUTxH[21:4] is programmed with line offset of the line

  Line 0 LUT entry high: LUT0_H=0x003FFEA0

Table 4 summarizes steps to calculate the LUT entry content for the first four lines of the round display of the 32L4R9IDISCOVERY kit.

Table 4. LUT calculation example

Line number	First pixel	Last pixel	First block	Last block	Visible blocks per line	visible blocks already used	Line offset	LUTxL	LUTxH
Line 0	181	208	22	26	5	0	-352	0x001A1601	0x003FFEA0
Line 1	172	217	21	27	7	5	-256	0x001B1501	0x003FFF00
Line 2	164	225	20	28	9	12	-128	0x001C1401	0x003FFF80
Line 3	158	231	19	28	10	21	32	0x001C1301	0x00000020

4 Memory optimization with GFXMMU

The required framebuffer size with GFXMMU optimization, is calculated as follows:

GFXMMU framebuffer optimized size = number of blocks used x block size

The number of blocks used is the sum of all the blocks used for all the lines. It is known after calculating the LUT.

The framebuffer size without GFXMMU optimization is calculated with the following formula:

Square size = frame width in pixels x Bpp x frame height

Then the gain is calculated with below formula:

Gain in size = (square size - GFXMMU optimized size) / square size

Example of memory gain calculation

For the 390 x 390 round display of the 32L4R9IDISCOVERY kit, the number of used blocks when using a 16 bpp framebuffer is 15248 blocks.

GFXMMU framebuffer optimized size = 15248 x 16 / 1024 = 238.25 Kbytes

Square size = 390 x 2 x 390 / 1024 = 297.07 Kbytes

Gain in size = (297.07 - 238.25) / 297.07 = 0.198

The memory gain is then 19.8%. The gain in size on the graphical framebuffer depends on the number of bits per pixel. For round displays the gain is generally of about 20%.

Table 5 presents the memory gain for the 390 x 390 round display of the 32L4R9IDISCOVERY kit for different color depths.

Table 5. Memory optimization for 390 x 390 display

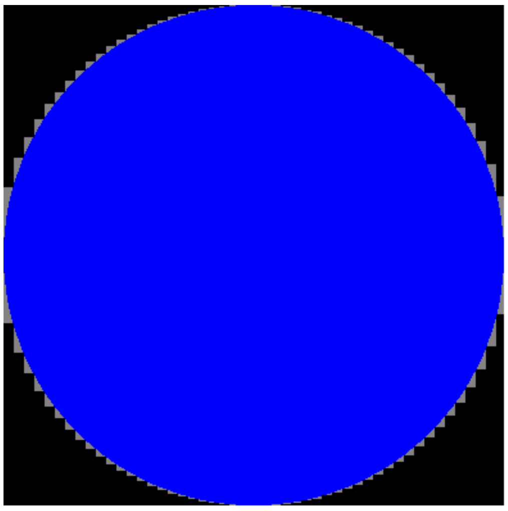
Mode	Square size (Kbytes)	GFXMMU optimized size (Kbytes)	Size reduction (%)
16 bpp	297.1	238.3	19.8
24 bpp	445.6	355.4	20.2
32 bpp	594.1	471.0	20.7

Figure 3 shows the output of the LTDC when used with the round display of the 32L4R9IDISCOVERY kit.

The black area represents the pixels that are not mapped into the physical memory. It is the actual memory gain saved by GFXMMU. When LTDC is fetching these pixels which are not mapped into physical memory, GFXMMU returns a default value programmed in the GFXMMU_DVR register (0x00 in this case).

The blue area represents the pixels that are mapped into the physical memory and visible on the display screen.

Figure 3. Memory optimization example for a round display

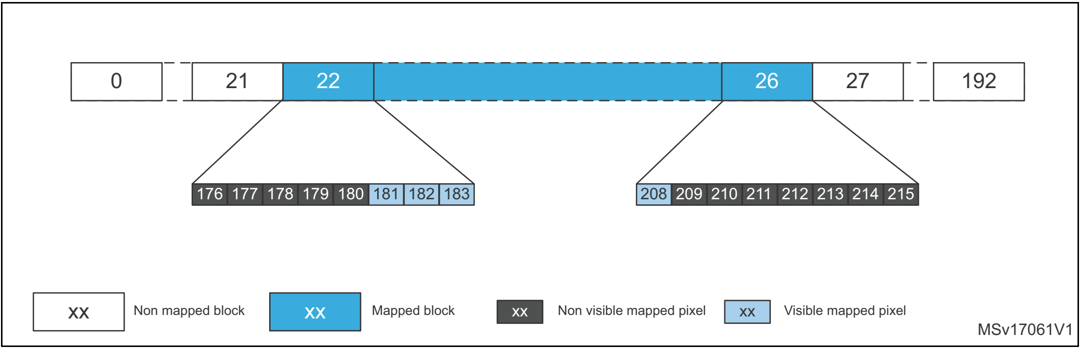


Block granularity overhead

The grey area in *Figure 3* corresponds to the pixels that are mapped into the physical memory but not visible on the display screen. This is what is called the block granularity overhead. The GFXMMU has a 16-byte block granularity, so one block can hold more than one pixel.

For blocks on the edges (first and last blocks), some pixels may not be visible on the screen. For example, if the visible pixels of a line are between pixel 181 and 208, with a 16 bpp framebuffer, the first block that must be enabled is block 22. Block 22 holds pixels 176 to 183. All these pixels are mapped and physically allocated in memory, but only pixels 181 to 183 are visible in the display screen (see *Figure 4*).

Figure 4. Block granularity overhead



Despite the overhead of the pixels that are mapped but not visible on the display, the GFXMMU allows an economy of about 20% of the framebuffer memory footprint.

5 GFXMMU system level operation

When a master tries to access the framebuffer, it uses the GFXMMU virtual framebuffer address. This address maps the transaction to the GFXMMU slave interface.

The GFXMMU resolves the address mapping and redirects the request to the corresponding physical address via its master port.

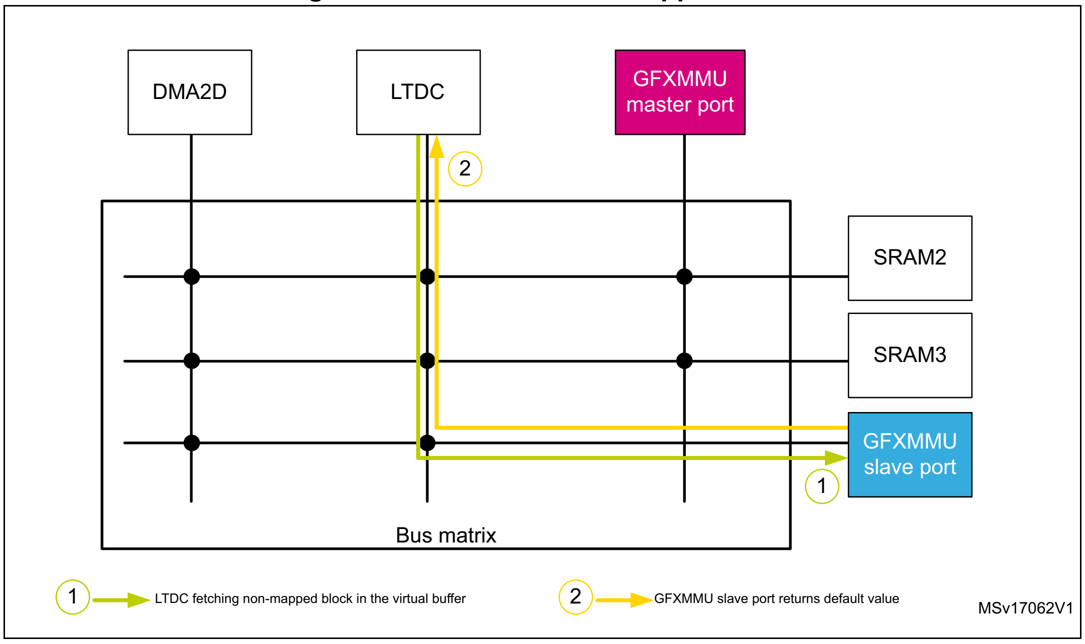
Some typical usage scenarios are described here after.

Read from a non-mapped block

When a master (LTDC for example) tries to read an address from the virtual buffer which is not mapped into a physical address, the GFXMMU responds with a default value to the read request (see *Figure 5*).

The default value is programmed in the graphic MMU default value register (GFXMMU_DVR).

Figure 5. Read from a non-mapped block


DMA2D
LTDC
GFXMMU master port
2
SRAM2
SRAM3
GFXMMU slave port
1
Bus matrix
1 LTDC fetching non-mapped block in the virtual buffer
2 GFXMMU slave port returns default value
MSv17062V1

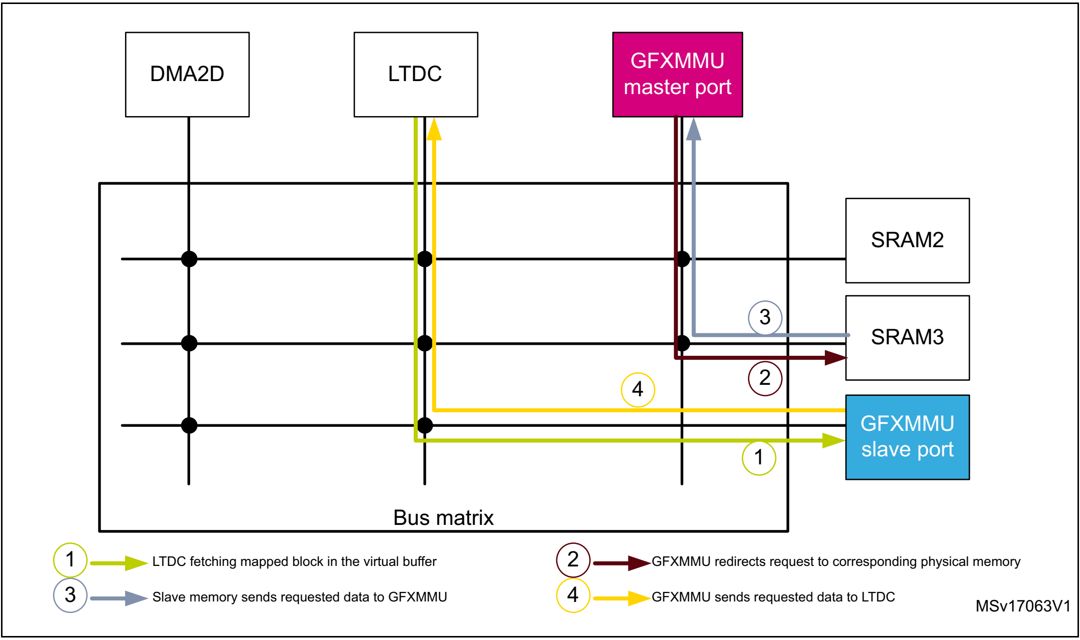
Read from a mapped block

When a master (LTDC for example) tries to read an address from the virtual buffer which is mapped to a physical address, the GFXMMU receives the request on its slave port and determines the corresponding physical address.

Then GFXMMU sends a read request via its master port to the corresponding memory holding the physical address.

The slave memory responds with the requested data to GFXMMU which redirects it to the LTDC. (See *Figure 6*).

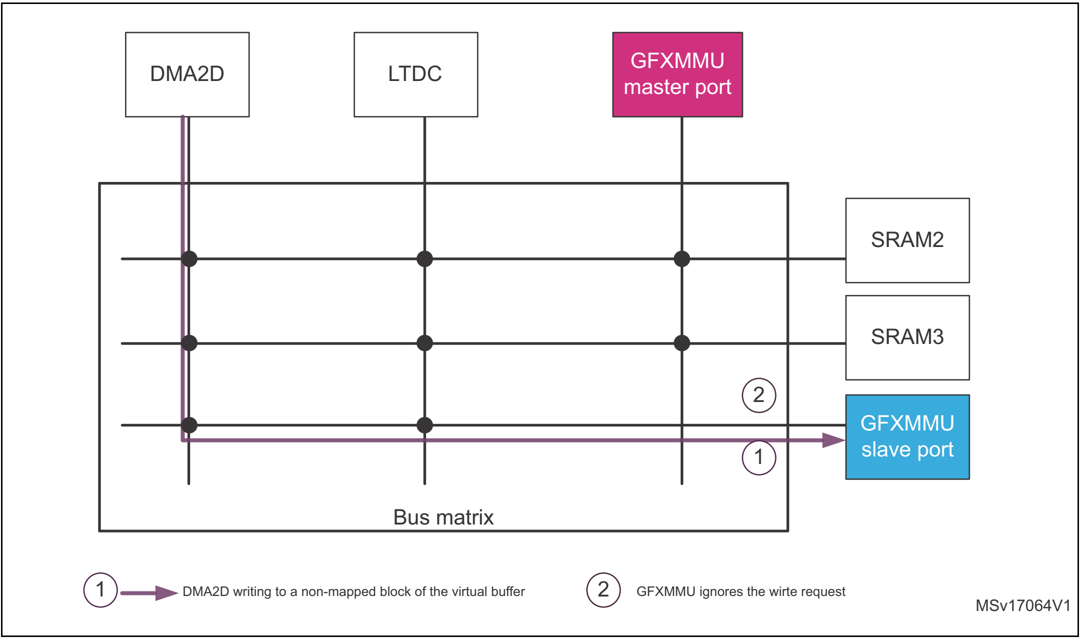
Figure 6. Read from a mapped block


DMA2D
LTDC
GFXMMU master port
SRAM2
SRAM3
GFXMMU slave port
Bus matrix
1 LTDC fetching mapped block in the virtual buffer
2 GFXMMU redirects request to corresponding physical memory
3 Slave memory sends requested data to GFXMMU
4 GFXMMU sends requested data to LTDC
MSv17063V1

Write to a non-mapped block

The GFXMMU receives write requests on its slave port from system masters (DMA2D for example). When the virtual address requested corresponds to a non-mapped block, the write operation is ignored. (See *Figure 7*).

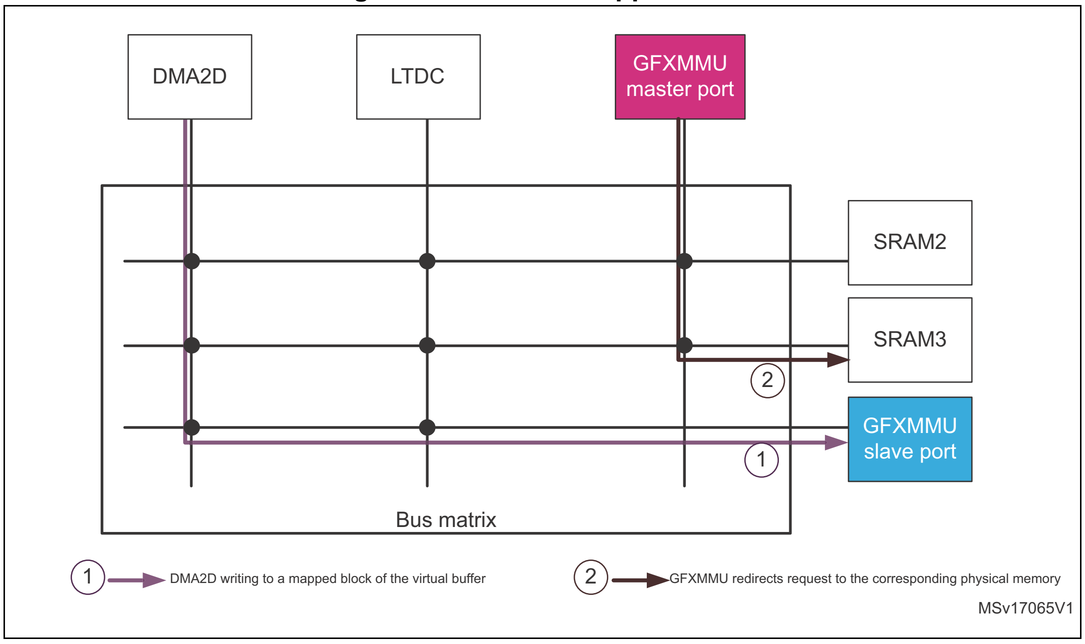
Figure 7. Write to a non-mapped block


DMA2D
LTDC
GFXMMU master port
SRAM2
SRAM3
GFXMMU slave port
Bus matrix
1 DMA2D writing to a non-mapped block of the virtual buffer
2 GFXMMU ignores the wirte request
MSv17064V1

Write to a mapped block

When the write request corresponds to a virtual address that is mapped to a visible block, the GFXMMU translates the virtual address to corresponding physical address. Then it sends the write request via its master port to the memory that holds the targeted physical address (See *Figure 8*).

Figure 8. Write to a mapped block


DMA2D
LTDC
GFXMMU master port
SRAM2
SRAM3
GFXMMU slave port
2
1
Bus matrix
1 DMA2D writing to a mapped block of the virtual buffer
2 GFXMMU redirects request to the corresponding physical memory
MSv17065V1

6 Basic configuration

This section presents the basic GFXMMU configuration together with specific LTDC and DMA2D configurations when used in conjunction with GFXMMU.

6.1 GFXMMU configuration

This section presents the basic parameters to use the GFXMMU.

6.1.1 GFXMMU virtual buffer base address

The GFXMMU allows up to four virtual buffers to be set.

Each virtual buffer has its own base address in the STM32 physical memory map.

The virtual buffer address is used by masters to access the framebuffer.

6.1.2 GFXMMU block mode

The user has to configure the GFXMMU block mode (GFXMMU_CR.192BM) by selecting any of the block modes (except for 24 bpp framebuffers where the 192 block mode must be used to have an integer number of pixels per line).

For other framebuffer color depths, the 256 block mode enables the support of larger display line width.

6.1.3 GFXMMU physical framebuffer

The physical framebuffer memory address can be configured separately for each virtual buffer in the GFXMMU buffer configuration register GFXMMU_BxCR.

For the physical framebuffer address selection, the user must consider its alignment, its size and avoid buffer overflow.

Alignment

The physical framebuffer address must be 16 bytes aligned since the GFXMMU has a 16-bytes block granularity and the four LSB bits of the physical address are considered as 0.

Size

After programming the LUT, the size of the physical framebuffer may be calculated using following formula:

Physical buffer size (Kbytes) = total blocks used x block size / 1024

Buffer overflow

The physical buffer cannot overflow the 8 Mbytes boundary of the zone defined by its base address. So the physical buffer address programmed in GFXMMU_BxCR must guarantee that the first and last mapped blocks of the buffer are in the same 8 Mbytes region of the physical memory to avoid buffer overflow errors.

6.1.4 GFXMMU default value

The GFXMMU default value (GFXMMU_DVR) is returned by GFXMMU when the virtual address read does not belong to a physically mapped block.

6.1.5 GFXMMU LUT

The GFXMMU LUT must be programmed depending on the display shape. Refer to *Section 3.2* for an example on the calculation of LUT entries.

Note: The display shape description must be stored in a non volatile memory (internal Flash for example), then it is used to program the GFXMMU LUT entries.

6.2 LTDC configuration

This section describes the LTDC configuration specific to the use of LTDC in conjunction with GFXMMU.

6.2.1 LTDC framebuffer

When accessing the framebuffer, the LTDC must use one of the four GFXMMU virtual framebuffers.

The LTDC layerx color framebuffer address register (LTDC_LxCFBAR) must be programmed with the address of the GFXMMU virtual buffer.

6.2.2 LTDC layer pitch

The LTDC layer color framebuffer pitch (LTDC_LxCFBLR.CFBP) must be carefully set when the LTDC is used in conjunction with GFXMMU.

The LTDC layer pitch is expressed in bytes. It depends on the GFXMMU block mode (GFXMMU_CR.192BM):

- GFXMMU_CR.192BM = 1 --> LTDC_LxCFBLR.CFBP = 3072 bytes
- GFXMMU_CR.192BM = 0 --> LTDC_LxCFBLR.CFBP = 4096 bytes

6.3 DMA2D configuration

Specific DMA2D configuration must be followed when either DMA2D source or destination is using the virtual buffer.

6.3.1 DMA2D framebuffer

The DMA2D framebuffer must be programmed to one of the four GFXMMU virtual buffers.

The DMA2D framebuffer registers to be programmed are:

- DMA2D output memory address (DMA2D_OMAR) if the destination is a virtual buffer.
- DMA2D foreground or background memory address register (DMA2D_FGMAR or DMA2D_BGMAR) if the source is a virtual buffer.

6.3.2 DMA2D line offset

The DMA2D line offset must be calculated based on the virtual buffer width in pixels (see *Table 2: Virtual buffer line width in pixels*).

The DMA2D line offset registers to be programmed are:

- DMA2D output line offset (DMA2D_OOR.LO) if the destination address is a virtual framebuffer.
- DMA2D layer line offset (DMA2D_FGOR.LO and DMA2D_BGOR.LO) if the source address is a virtual framebuffer.

7 Software example

This section presents software example to configure the GFXMMU.

Examples for LTDC and DMA2D are also presented in this section.

7.1 GFXMMU configuration example

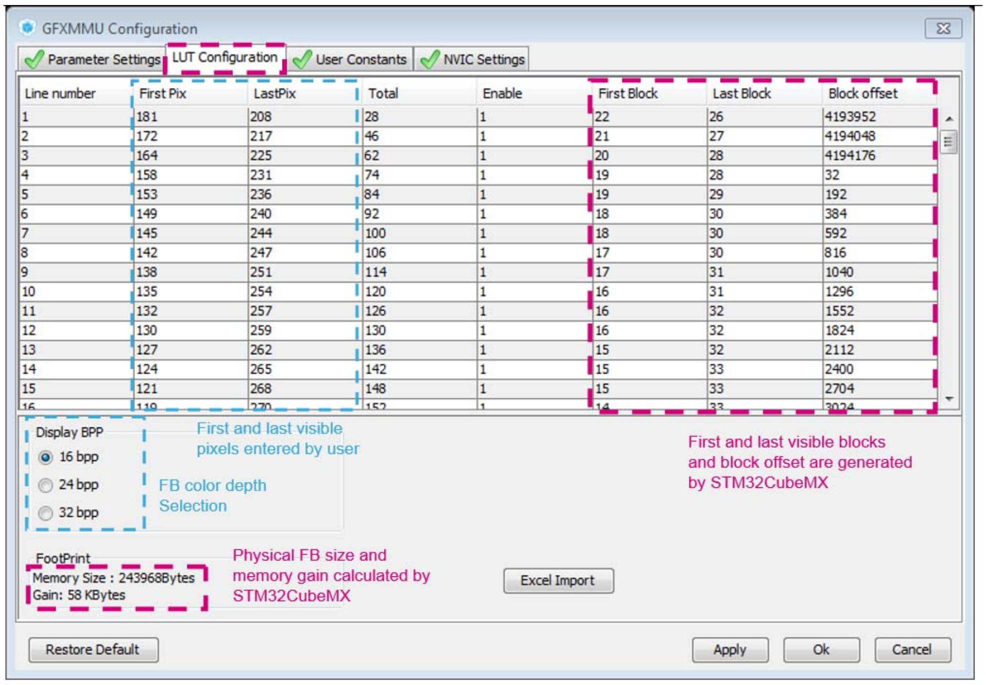
This sections presents the GFXMMU configuration with STM32CubeMX and the corresponding initialization code.

7.1.1 GFXMMU configuration with STM32CubeMX

In the GFXMMU parameter settings, the user selects the block mode and the virtual buffer to be used. The user may also change the default value returned by GFXMMU when a master tries to access an unmapped block.

In the LUT configuration interface (see *Figure 9*), the user has to enter the first and last visible pixels for each line and must select the framebuffer color depth. STM32CubeMX automatically generates the first and last block and the block offset. The memory footprint required for the physical framebuffer is also calculated.

Figure 9. GFXMMU LUT configuration in STM32CubeMX


GFXMMU Configuration
Parameter Settings
LUT Configuration
User Constants
NVIC Settings
First and last visible pixels entered by user
FB color depth Selection
First and last visible blocks and block offset are generated by STM32CubeMX
Physical FB size and memory gain calculated by STM32CubeMX
Excel Import
Restore Default
Apply
Ok
Cancel

Line number	First Pix	LastPix	Total	Enable	First Block	Last Block	Block offset
1	181	208	28	1	22	26	4193952
2	172	217	46	1	21	27	4194048
3	164	225	62	1	20	28	4194176
4	158	231	74	1	19	28	32
5	153	236	84	1	19	29	192
6	149	240	92	1	18	30	384
7	145	244	100	1	18	30	592
8	142	247	106	1	17	30	816
9	138	251	114	1	17	31	1040
10	135	254	120	1	16	31	1296
11	132	257	126	1	16	32	1552
12	130	259	130	1	16	32	1824
13	127	262	136	1	15	32	2112
14	124	265	142	1	15	33	2400
15	121	268	148	1	15	33	2704
16	119	270	152	1	14	33	3024

Display BPP: [✓ 16 bpp] 24 bpp 32 bpp

FootPrint
Memory Size : 243968Bytes
Gain: 58 KBytes

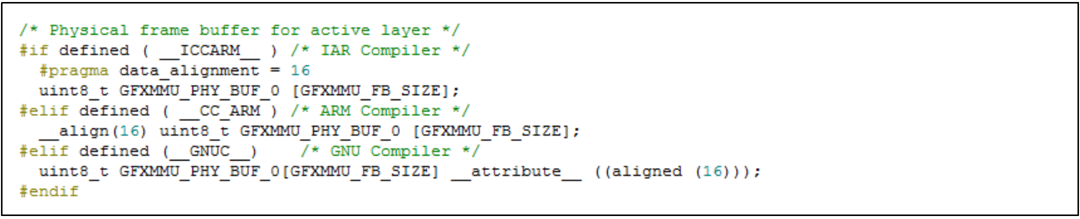
STM32CubeMX automatically generates the LUT configuration in the “gfxmmu_lut.h” header file.

7.1.2 GFXMMU initialization code

- Physical buffer: the physical framebuffer must be 16 bytes aligned.

The example on *Figure 10* shows how to align the physical framebuffer to 16 bytes with three different compilers (IAR, GNU and Arm® compilers).

Figure 10. GFXMMU physical framebuffer declaration

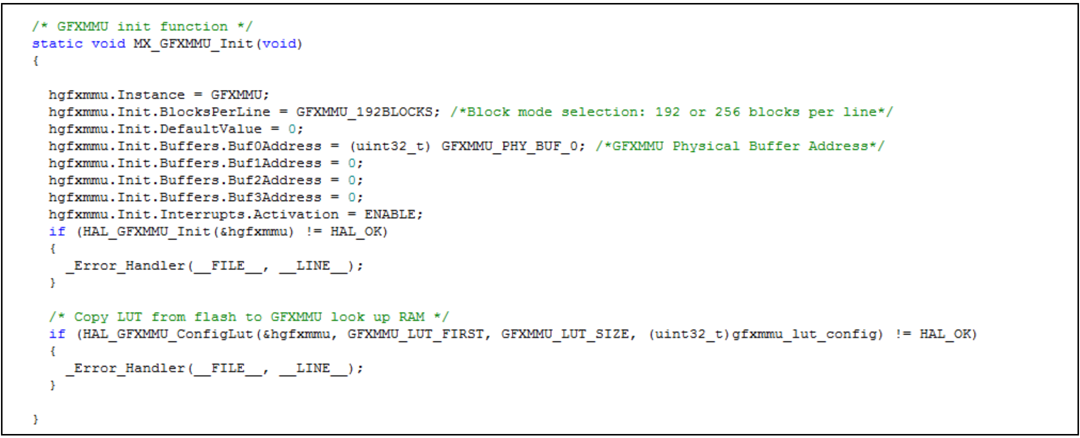


```
/* Physical frame buffer for active layer */
#if defined ( __ICCARM__ ) /* IAR Compiler */
  #pragma data_alignment = 16
  uint8_t GFXMMU_PHY_BUF_0 [GFXMMU_FB_SIZE];
#elif defined ( __CC_ARM ) /* ARM Compiler */
  __align(16) uint8_t GFXMMU_PHY_BUF_0 [GFXMMU_FB_SIZE];
#elif defined (__GNUC__)     /* GNU Compiler */
  uint8_t GFXMMU_PHY_BUF_0[GFXMMU_FB_SIZE] __attribute__ ((aligned (16)));
#endif
```

The physical framebuffer size is calculated by STM32CubeMX based on the LUT configuration.

- GFXMMU initialization: see *Figure 11* for an example.

Figure 11. GFXMMU initialization



```
/* GFXMMU init function */
static void MX_GFXMMU_Init(void)
{

  hgfxmmu.Instance = GFXMMU;
  hgfxmmu.Init.BlocksPerLine = GFXMMU_192BLOCKS; /*Block mode selection: 192 or 256 blocks per line*/
  hgfxmmu.Init.DefaultValue = 0;
  hgfxmmu.Init.Buffers.Buf0Address = (uint32_t) GFXMMU_PHY_BUF_0; /*GFXMMU Physical Buffer Address*/
  hgfxmmu.Init.Buffers.Buf1Address = 0;
  hgfxmmu.Init.Buffers.Buf2Address = 0;
  hgfxmmu.Init.Buffers.Buf3Address = 0;
  hgfxmmu.Init.Interrupts.Activation = ENABLE;
  if (HAL_GFXMMU_Init(&hgfxmmu) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  /* Copy LUT from flash to GFXMMU look up RAM */
  if (HAL_GFXMMU_ConfigLut(&hgfxmmu, GFXMMU_LUT_FIRST, GFXMMU_LUT_SIZE, (uint32_t)gfxmmu_lut_config) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}
```

The gfxmmu_lut_config is automatically generated by STM32CubeMX in the “gfxmmu_lut.h” header file based on the display shape’s description entered by the user in the “LUT Configuration” window. It is used to initialize the LUT.

7.2 LTDC configuration example

When using GFXMMU, the LTDC fetches data from the GFXMMU virtual buffer, hence a specific LTDC configuration must be set.

7.2.1 LTDC configuration with STM32CubeMX

The LTDC layer framebuffer address is programmed with the GFXMMU virtual buffer.

STM32CubeMX automatically calculates the LTDC layer pitch in pixels based on the virtual buffer line width.

See *Figure 12* for an example of LTDC layer settings in STM32CubeMX.

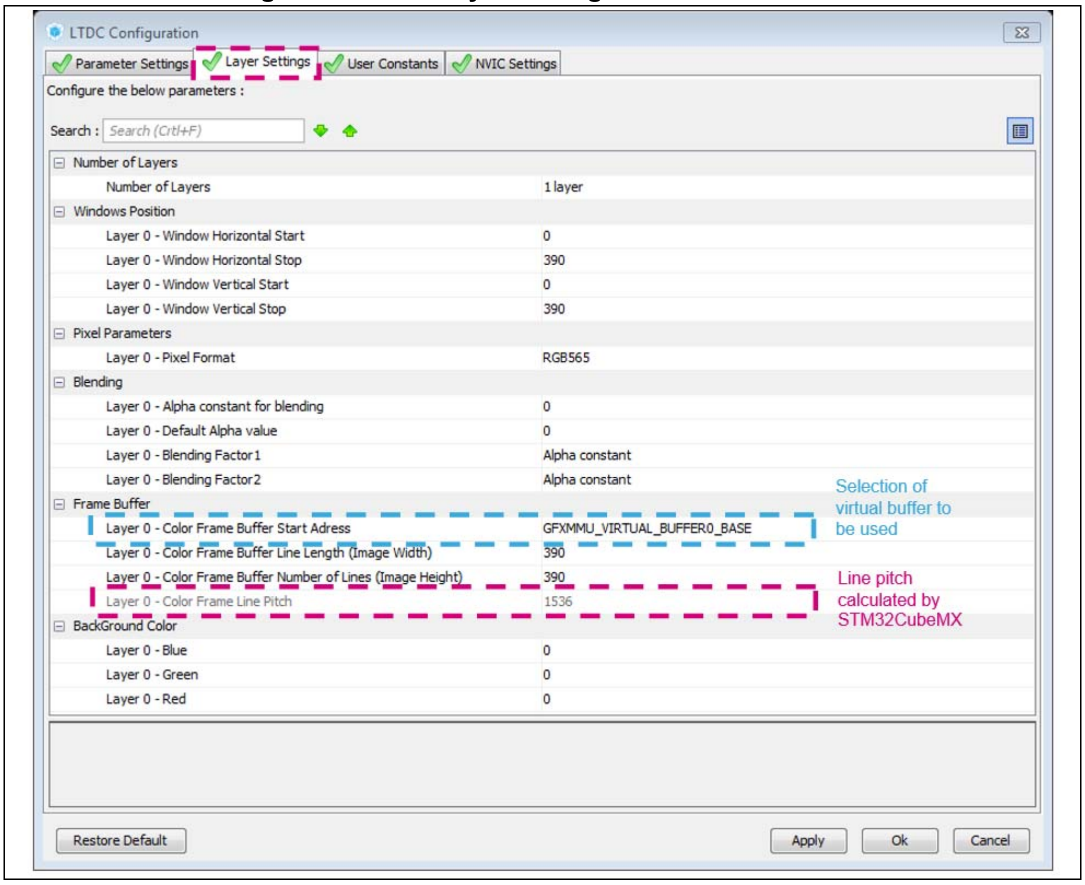


Figure 12. LTDC layer settings in STM32CubeMX

Number of Layers	
Number of Layers	1 layer
Windows Position	
Layer 0 - Window Horizontal Start	0
Layer 0 - Window Horizontal Stop	390
Layer 0 - Window Vertical Start	0
Layer 0 - Window Vertical Stop	390
Pixel Parameters	
Layer 0 - Pixel Format	RGB565
Blending	
Layer 0 - Alpha constant for blending	0
Layer 0 - Default Alpha value	0
Layer 0 - Blending Factor1	Alpha constant
Layer 0 - Blending Factor2	Alpha constant
Frame Buffer	
Layer 0 - Color Frame Buffer Start Adress	GFXMMU_VIRTUAL_BUFFER0_BASE
Layer 0 - Color Frame Buffer Line Length (Image Width)	390
Layer 0 - Color Frame Buffer Number of Lines (Image Height)	390
Layer 0 - Color Frame Line Pitch	1536
BackGround Color	
Layer 0 - Blue	0
Layer 0 - Green	0
Layer 0 - Red	0

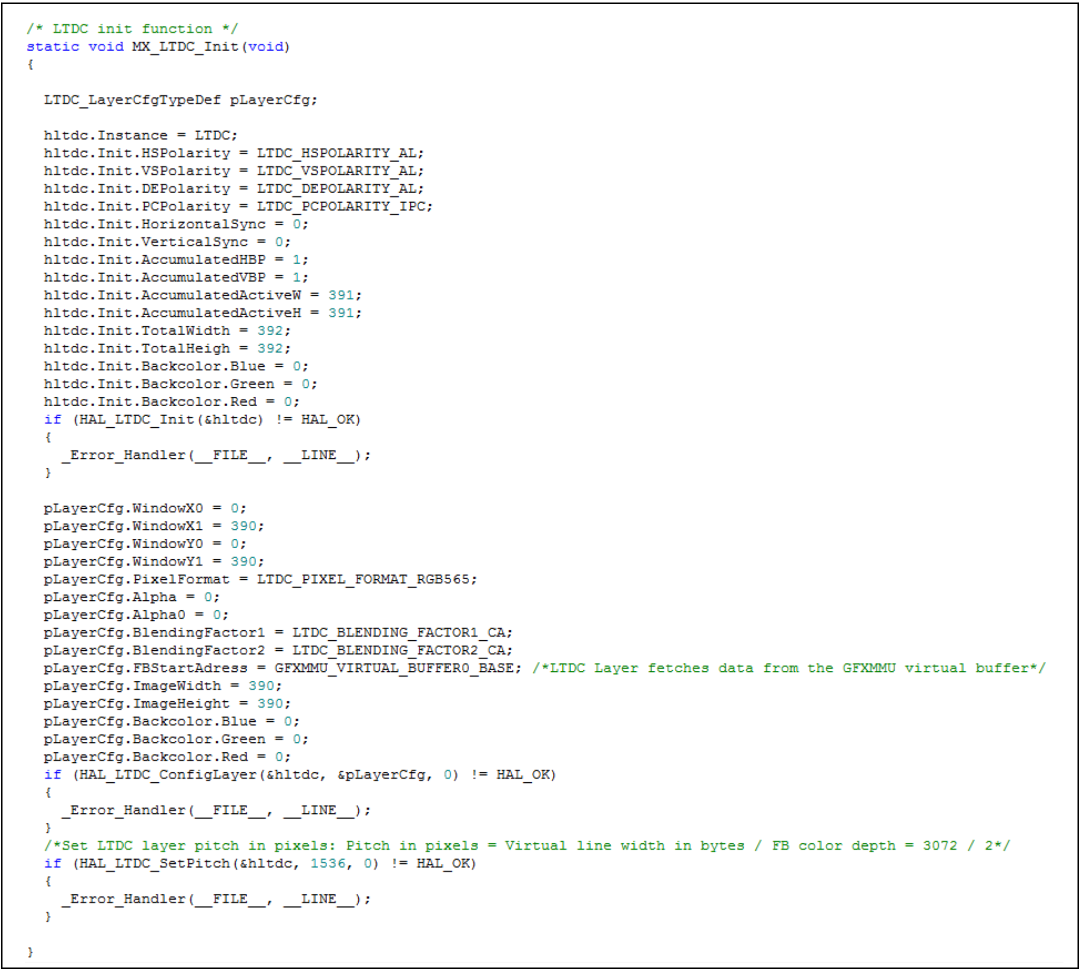
Selection of virtual buffer to be used

Line pitch calculated by STM32CubeMX

7.2.2 LTDC initialization code

See an example of LTDC initialization code in *Figure 13*.

Figure 13. LTDC initialization code



```
/* LTDC init function */
static void MX_LTDC_Init(void)
{

  LTDC_LayerCfgTypeDef pLayerCfg;

  hltdc.Instance = LTDC;
  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.Init.HorizontalSync = 0;
  hltdc.Init.VerticalSync = 0;
  hltdc.Init.AccumulatedHBP = 1;
  hltdc.Init.AccumulatedVBP = 1;
  hltdc.Init.AccumulatedActiveW = 391;
  hltdc.Init.AccumulatedActiveH = 391;
  hltdc.Init.TotalWidth = 392;
  hltdc.Init.TotalHeigh = 392;
  hltdc.Init.Backcolor.Blue = 0;
  hltdc.Init.Backcolor.Green = 0;
  hltdc.Init.Backcolor.Red = 0;
  if (HAL_LTDC_Init(&hltdc) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

  pLayerCfg.WindowX0 = 0;
  pLayerCfg.WindowX1 = 390;
  pLayerCfg.WindowY0 = 0;
  pLayerCfg.WindowY1 = 390;
  pLayerCfg.PixelFormat = LTDC_PIXEL_FORMAT_RGB565;
  pLayerCfg.Alpha = 0;
  pLayerCfg.Alpha0 = 0;
  pLayerCfg.BlendingFactor1 = LTDC_BLENDING_FACTOR1_CA;
  pLayerCfg.BlendingFactor2 = LTDC_BLENDING_FACTOR2_CA;
  pLayerCfg.FBStartAdress = GFXMMU_VIRTUAL_BUFFER0_BASE; /*LTDC Layer fetches data from the GFXMMU virtual buffer*/
  pLayerCfg.ImageWidth = 390;
  pLayerCfg.ImageHeight = 390;
  pLayerCfg.Backcolor.Blue = 0;
  pLayerCfg.Backcolor.Green = 0;
  pLayerCfg.Backcolor.Red = 0;
  if (HAL_LTDC_ConfigLayer(&hltdc, &pLayerCfg, 0) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }
  /*Set LTDC layer pitch in pixels: Pitch in pixels = Virtual line width in bytes / FB color depth = 3072 / 2*/
  if (HAL_LTDC_SetPitch(&hltdc, 1536, 0) != HAL_OK)
  {
    _Error_Handler(__FILE__, __LINE__);
  }

}
```

7.3 DMA2D configuration example

In this example the DMA2D is used to copy an image from Flash memory to framebuffer.

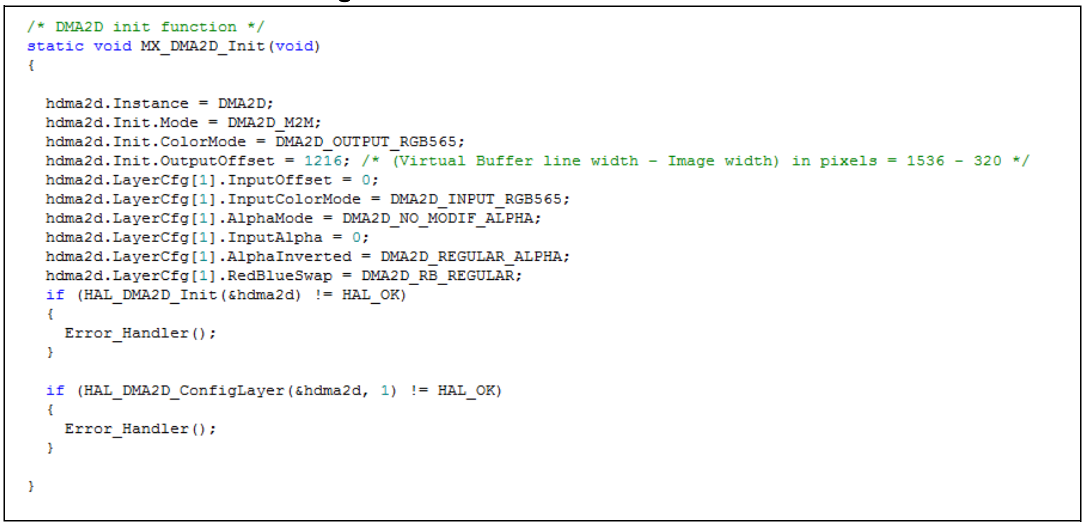
The access to the framebuffer is done through the GFXMMU, so the destination address of the DMA2D is the GFXMMU virtual buffer.

The output offset must take into account the virtual buffer line width in pixels.

7.3.1 DMA2D initialization

An example of DMA2D initialization code is presented in *Figure 14*.

Figure 14. DMA2D initialization code

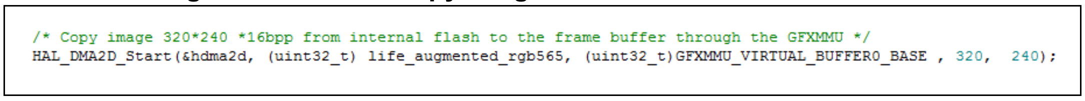


```c
/* DMA2D init function */
static void MX_DMA2D_Init(void)
{

  hdma2d.Instance = DMA2D;
  hdma2d.Init.Mode = DMA2D_M2M;
  hdma2d.Init.ColorMode = DMA2D_OUTPUT_RGB565;
  hdma2d.Init.OutputOffset = 1216; /* (Virtual Buffer line width - Image width) in pixels = 1536 - 320 */
  hdma2d.LayerCfg[1].InputOffset = 0;
  hdma2d.LayerCfg[1].InputColorMode = DMA2D_INPUT_RGB565;
  hdma2d.LayerCfg[1].AlphaMode = DMA2D_NO_MODIF_ALPHA;
  hdma2d.LayerCfg[1].InputAlpha = 0;
  hdma2d.LayerCfg[1].AlphaInverted = DMA2D_REGULAR_ALPHA;
  hdma2d.LayerCfg[1].RedBlueSwap = DMA2D_RB_REGULAR;
  if (HAL_DMA2D_Init(&hdma2d) != HAL_OK)
  {
    Error_Handler();
  }

  if (HAL_DMA2D_ConfigLayer(&hdma2d, 1) != HAL_OK)
  {
    Error_Handler();
  }

}
```

7.3.2 DMA2D copy image from Flash to the framebuffer

Refer to *Figure 15* for the code for DMA2D to copy an image from Flash to framebuffer.

Figure 15. DMA2D copy image from Flash to the framebuffer



```c
/* Copy image 320*240 *16bpp from internal flash to the frame buffer through the GFXMMU */
HAL_DMA2D_Start(&hdma2d, (uint32_t) life_augmented_rgb565, (uint32_t)GFXMMU_VIRTUAL_BUFFER0_BASE , 320,  240);
```

8 Application example

A typical application example of GFXMMU is illustrated in *Figure 16*.

In this example the graphic framebuffer is located into the internal SRAM and the graphic primitives are stored in the Octal-SPI NOR Flash memory.

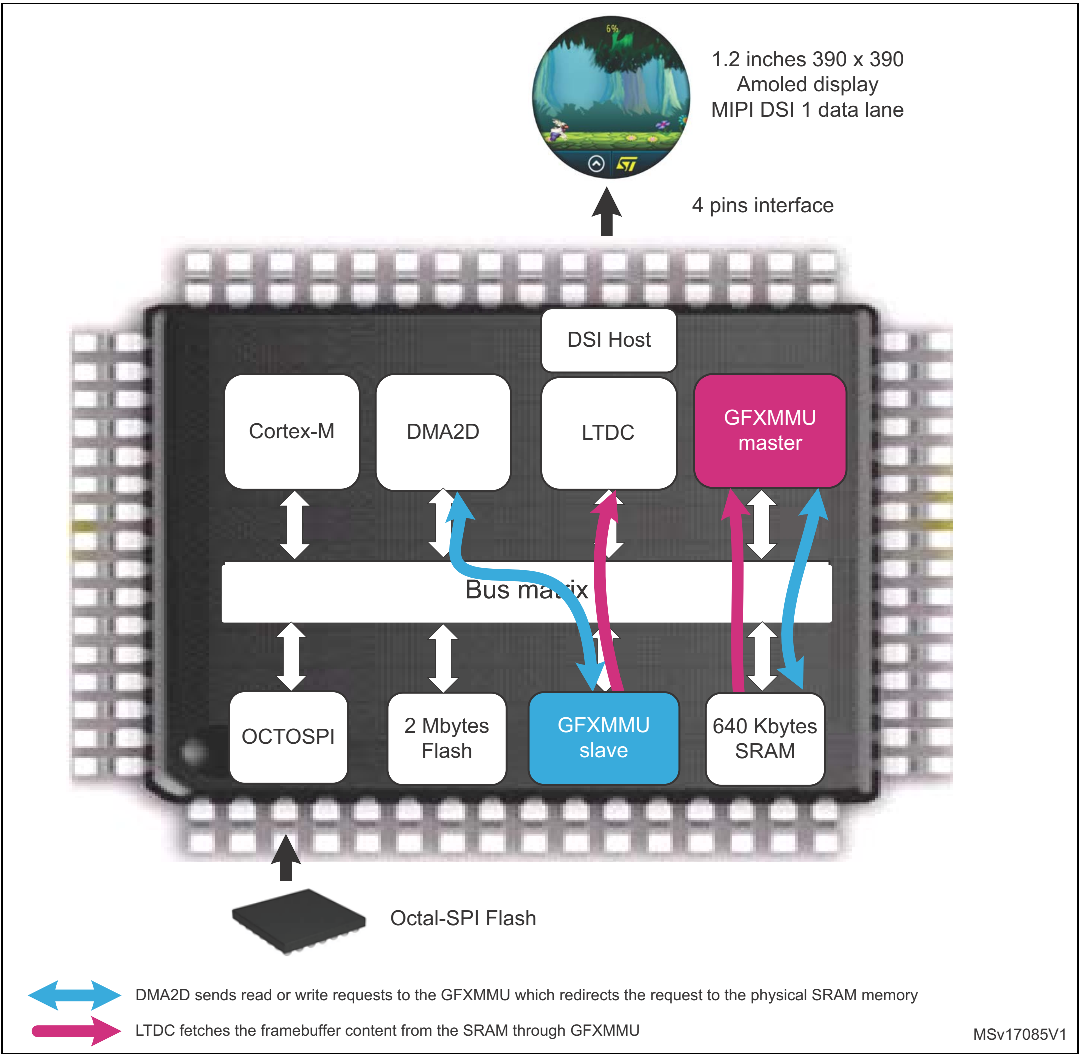
DMA2D fetches graphic primitives from an external Octal-SPI NOR Flash through the OCTOSPI interface.

When creating the graphic framebuffer content, DMA2D sends read or write requests to the GFXMMU which redirects the request to the physical SRAM memory (blue path in *Figure 16*).

During the graphic framebuffer display, LTDC fetches the framebuffer content from the SRAM through the GFXMMU (pink path in *Figure 16*).

The DSI Host serializes the LTDC output allowing the STM32 to be interfaced with a MIPI® DSI display with only four pins.

Figure 16. GFXMMU application example



9 Special recommendations

GFXMMU adds extra wait state (1 WS) when accessing the framebuffer for address resolution. Even with this extra latency, using GFXMMU with internal framebuffer provides much better performance than using a framebuffer located in external memory (which would require tens of cycles). So, the use of GFXMMU is very useful when it allows the device to reduce the graphic framebuffer size so it can fit into internal RAM.

There are cases where the GFXMMU should not be used to avoid extra latency when accessing the framebuffer:

- If the internal RAM is already enough to store the graphic framebuffer.
  In this case it is better to use the internal RAM directly to benefit from the 0 WS execution.

Note: User can still use GFXMMU if further memory optimization is preferred over extra latency.

- If the framebuffer is located in an external memory.
  When the graphic framebuffer does not fit into the internal RAM even with GFXMMU optimization, it is then placed in the external memory. The external memory should be big enough to hold the graphic framebuffers without the need for optimization. So the GFXMMU must not be used to avoid adding extra latency when accessing the external memory.

10 Conclusion

The GFXMMU offers an optimized solution to drive non-rectangular displays by reducing the framebuffer size; providing a fully integrated solution without the need for external RAM.

This application note presented the GFXMMU features and functional behavior and described the system level operation in conjunction with other graphic peripherals.

The GFXMMU LUT programming has been presented based on the round display of the 32L4R9IDISCOVERY kit.

Finally this document provided GFXMMU basic configuration and code examples to ease the applications development.

11 Revision history

Table 6. Document revision history

Date	Revision	Changes
10-Oct-2017	1	Initial release.