# AN5051
# 应用笔记

## 使用STM32 Chrom-GRC™进行图形存储器优化

## 前言

LCD技术曾经是矩形显示器的专有技术。最新发展创造出了非矩形LCD显示器。这些显示器对于可穿戴设备等各种应用而言是非常有吸引力的。

STM32 Chrom-GRC™（GFXMMU）外设是STM32微控制器的新成员（参考 *表 1: 适用产品*），可以有效支持这种非矩形显示器发展趋势。

GFXMMU只存储非矩形显示的可见部分，而在圆形显示的情况下，该外设存储图形帧缓冲器的内存需求可减少20%。因此，GFXMMU让微控制器无需额外增加SDRAM/SDRAM存储设备。

由于不需要外部RAM和充分利用内部RAM的低功耗和高性能特性，嵌入GFXMMU的STM32微控制器为需要低功耗管理功能和高品质用户接口的可穿戴应用提供了合适的解决方案。

**表1. 适用产品**

| 类型 | 产品编号 |
|---|---|
| 微控制器 | STM32L4+系列 |

# 目录

# 表格索引

# 图片目录

# 1 STM32 Chrom-GRC™（GFXMMU）说明

GFXMMU是面向图形的内存管理单元，旨在根据显示形状来优化内存使用。

该外设允许微控制器仅将非矩形显示的可见部分存储在连续物理存储区域中，从而可以减少帧缓冲存储器占用空间。

由于可将帧缓冲区存储在内部RAM中，无需使用外部RAM，GFXMMU为图形应用提供了高度集成的解决方案。该外设可提高性能、降低功耗并降低系统成本。

## 1.1 GFXMMU特性

GFXMMU的主要特性如下所列：

- 根据显示形状而降低内存使用率
- 显示形状完全可配置
- 透明集成
- 适用于任何系统的内存

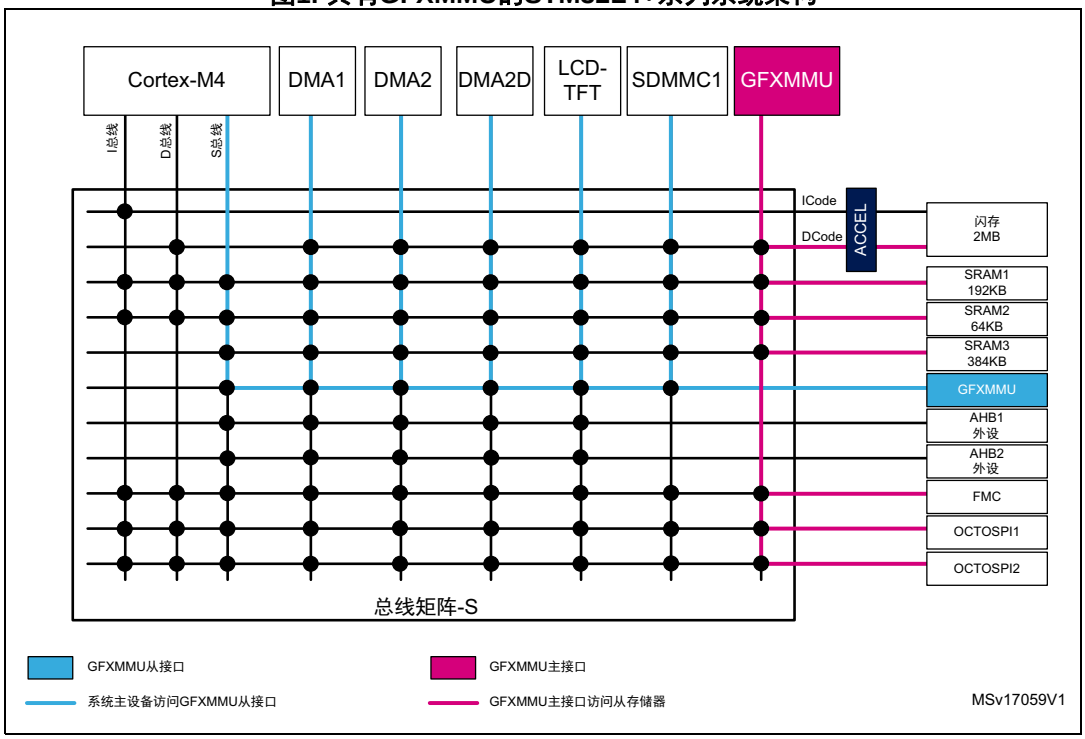
## 1.2 智能架构中的GFXMMU

GFXMMU同时拥有主、从两个接口。主接口管理从不同存储器（闪存，SRAM，FMC，OCTOSPI）的访问。从接口用来被不同的主设备（LTDC，DMA2D，Cortex M，DMA，SDMMC）访问。

系统主机通过GFXMMU访问图形帧缓冲区。GFXMMU在其从接口上接收读/写请求，并执行地址解析以确定目标物理地址。然后它通过主接口将传输请求重定向到实际存储的物理地址内存中。

STM32L4+系列是首批集成了GFXMMU的STM32产品。*图 1*显示了嵌入GFXMMU的STM32L4+系列系统架构。

**图1. 具有GFXMMU的STM32L4+系列系统架构**



# 2 GFXMMU虚拟缓冲区

GFXMMU虚拟缓冲区只允许将非矩形显示的可见部分存储在连续的物理内存区域中。

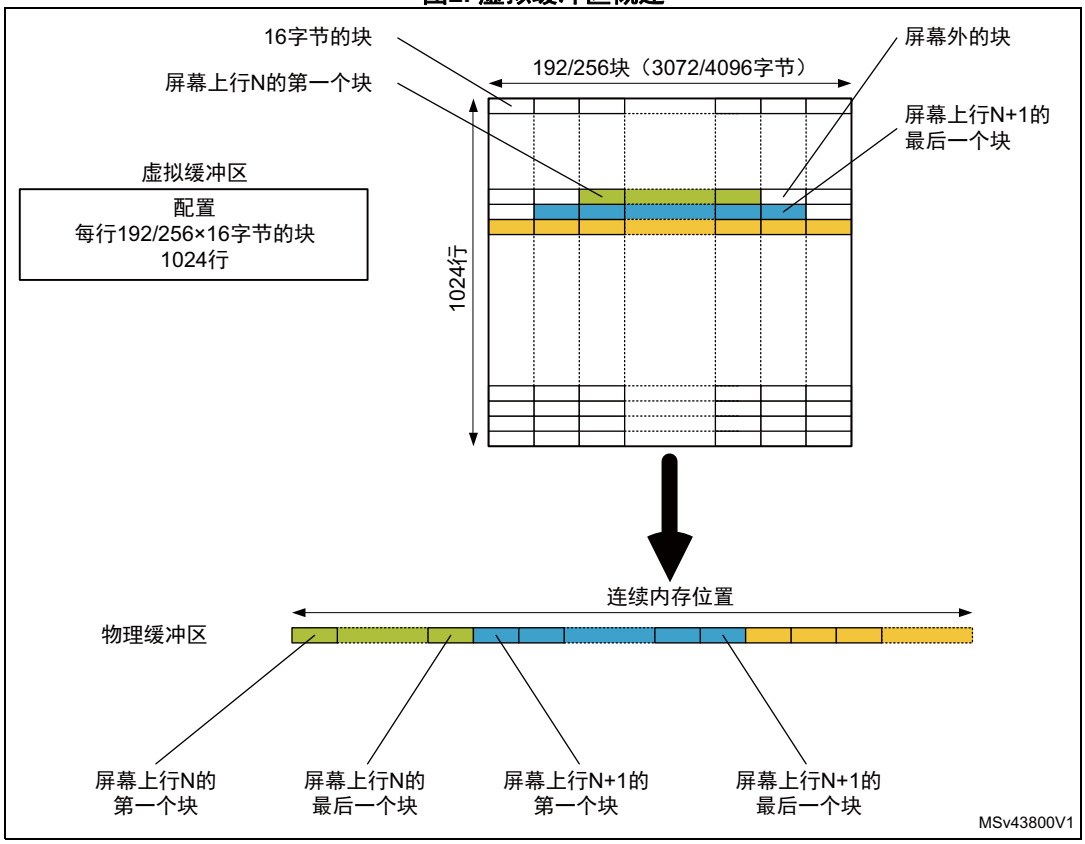
## 2.1 虚拟缓冲区概述

虚拟缓冲区每行有3072或4096个字节，共1024行。

只有显示器的可见部分被映射到物理内存空间。

*图 2*给出了GFXMMU虚拟缓冲区的概述。

**图2. 虚拟缓冲区概述**




## 2.2 虚拟缓冲区使用情况

使用GFXMMU时，通过虚拟缓冲区来访问图形帧缓冲区。这种情况下，考虑到虚拟缓冲区行宽，LTDC和DMA2D必须有特殊配置。

### 2.2.1 使用LTDC时的虚拟缓冲区使用情况

LTDC层间距是从一行开始到下一行开始之间的字节数。它在LTDC_LxCFBLR寄存器中进行配置，并以字节表示。

当LTDC使用GFXMMU虚拟缓冲区时，LTDC层间距等于虚拟缓冲区行宽（以字节为单位，即3072或4096）。

### 2.2.2 使用DMA2D时的虚拟缓冲区使用情况

DMA2D缓冲区行偏移量加到每行末尾，来确定下一行的起始地址。DMA2D缓冲区行偏移量以像素表示。

当使用GFXMMU虚拟缓冲区时，DMA2D缓冲区行偏移由以下公式给出：

*行偏移量 = 以像素计的虚拟缓冲区行宽 - 以像素计的图像宽度*

当DMA2D使用虚拟缓冲区时，行宽必须是整数像素。

*注：* *在24 bpp帧缓冲区的情况下，只有当虚拟缓冲区具有3072个字节宽度（对应于1024个像素）时，才能保证具有整数个像素。*

## 2.3 虚拟缓冲区工作模式

为了确保虚拟缓冲区在不同帧缓冲区色深时每行都具有整数个像素，有两种工作模式可以使用：

- 256块模式
  在这种模式下，虚拟缓冲区每行有256块，每块16字节。
  此模式对应的行宽为256 x 16 = 4096字节。
- 192块模式
  在这种模式下，虚拟缓冲区每行有192块，每块16字节。
  此模式下的行宽为192 x 16 = 3072字节。

使用24 bpp缓冲区时，引入192个块模式，可使得每行具有整数个像素。

*表 2*给出了不同帧缓冲区色深时，以像素计的虚拟缓冲区行宽。

**表2. 虚拟缓冲区行宽（以像素为单位）**

| - | 32 bpp | 24 bpp | 16 bpp | 8 bpp |
|---|---|---|---|---|
| **192块模式** | 768 | 1024 | 1536 | 3072 |
| **256块模式** | 1024 | 1365.3(1) | 2048 | 4096 |

1. 为了使每行具有整数个像素，在24 bpp帧缓冲区中应避免使用256块模式。

# 3 显示形状描述表

GFXMMU允许MCU根据显示形状和大小仅将必要的块映射到物理内存位置。显示形状描述表被存储在查找表（LUT）中。

## 3.1 LUT配置

每行的LUT都必须有特定配置：

- 行使能
- 第一个可见块的编号
- 最后一个可见块的编号
- 物理缓冲区内，行的地址偏移量

通过编程每行的地址偏移量，可见块能够以连续的方式排列在物理缓冲区中。

## 3.2 LUT计算示例

本节介绍GFXMMU LUT条目计算。

该示例基于32L4R9IDISCOVERY套件的390x390圆形显示屏。在这个例子中，帧缓冲区具有16bpp的颜色格式。

每行的第一个和最后一个可见像素由显示器制造商提供。

*表 3*描述了圆形显示器的第一个和最后一个可见像素。本例中只有前四行。

**表3. 32L4R9IDISCOVERY套件的圆形显示器的可见像素说明**

| 行号 | 第一个可见像素 | 最后一个可见像素 |
|---|---|---|
| 行0 | 181 | 208 |
| 行1 | 172 | 217 |
| 行2 | 164 | 225 |
| 行3 | 158 | 231 |

### 块号计算

GFXMMU的每一个块对应为16字节的最小单元。

块号计算基于像素数量和帧缓冲区色深。

在以下等式中，色深以每像素的字节数（Bpp）表示。

第一个可见块是保存第一个像素的第一个字节的块。

*第一个可见块号 = (第一个可见像素 x Bpp) / 块大小*

最后一个可见块是保存最后一个像素的最后一个字节的块：

*最后一个可见块号 = (最后一个可见像素 x Bpp + Bpp - 1) / 块大小*

以*表 3*为例，行0的可见像素包含在像素181至208之间，因此：

*行0的第一个可见块号 = (181 x 2) / 16 = 22*

*行0的最后一个可见块号 = (208 x 2 + 1) / 16 = 26*

### 行偏移量计算

行偏移量的定义为物理缓冲区中行的第一个可见块的偏移量。它允许可见块以连续方式排列在物理缓冲区中。

行偏移量编码为22位，可以有负值，计算方法如下：

*行偏移量=（已使用的可见块数 - 第一个可见块号）x 块大小*

其中：

- 已经使用的可见块数包括所有先前行的可见块
- 第一个可见块号，指的是当前行的第一个可见块号
- 块大小 = 16 字节

*行0偏移量 = (0-22) x 16 = -352 = 0x3F:FEA0*

*行1偏移量 = (5-21) x 16 = -252 = 0x3F:FF00*

在计算每行的第一个和最后一个可见块以及行偏移之后，LUT条目按如下方式编程：

- LUT条目x的低位寄存器
  - LUTxL[23:16]使用最后一个可见块的值来编程
  - LUTxL[15:8]使用第一个可见块的值来编程
  - 当该行使能时，LUTxL[0]置位

  行0 LUT条目的低位寄存器：LUT0_L=0x001A1601
- LUT条目x的高位寄存器
  - LUTxH[21:4]使用该行的行偏移量来编程

  Line 0 LUT条目的高位寄存器：LUT0_H=0x003FFEA0

*表 4*总结了计算32L4R9IDISCOVERY套件圆形显示屏前四行的LUT条目内容的步骤。

**表4. LUT计算示例**

| 行号 | 第一个像素 | 最后一个像素 | 第一个块 | 最后一个块 | 每行的可见块 | 已经使用的可见块 | 行偏移量 | LUTxL | LUTxH |
|---|---|---|---|---|---|---|---|---|---|
| 行0 | 181 | 208 | 22 | 26 | 5 | 0 | -352 | 0x001A1601 | 0x003FFEA0 |
| 行1 | 172 | 217 | 21 | 27 | 7 | 5 | -256 | 0x001B1501 | 0x003FFF00 |
| 行2 | 164 | 225 | 20 | 28 | 9 | 12 | -128 | 0x001C1401 | 0x003FFF80 |
| 行3 | 158 | 231 | 19 | 28 | 10 | 21 | 32 | 0x001C1301 | 0x00000020 |

# 4 利用GFXMMU进行存储器优化

GFXMMU优化所需的帧缓冲区大小计算如下：

*GFXMMU帧缓冲区优化大小 = 所用块数 x 块大小*

所用块数是所有行所用的所有块总和。计算LUT后才能知道它的值。

不进行GFXMMU优化的帧缓冲区大小用以下公式计算：

*方形大小 = 帧宽度（以像素计） x Bpp x 帧高度*

因此增益可按以下公式计算：

*增益大小 = (方形大小 - GFXMMU优化后的大小) / 方形大小*

### 内存增益计算示例

对于32L4R9IDISCOVERY套件的390x390圆形显示屏，使用16bpp帧缓冲区时所用的块数为15248块。

*GFXMMU帧缓冲区优化后的大小 = 15248 x 16 / 1024 = 238.25 KB*

*方形大小 = 390 x 2 x 390 / 1024 = 297.07 KB*

*增益大小 = (297.07 - 238.25) / 297.07 = 0.198*

因此内存增益为19.8%。图形帧缓冲区上的增益大小取决于每像素位数。对于圆形显示屏通常增益约为20%。

*表 5*显示了不同色深下，32L4R9IDISCOVERY套件的390 x 390圆形显示屏的内存增益。

**表5. 针对390 x 390显示器的内存优化**

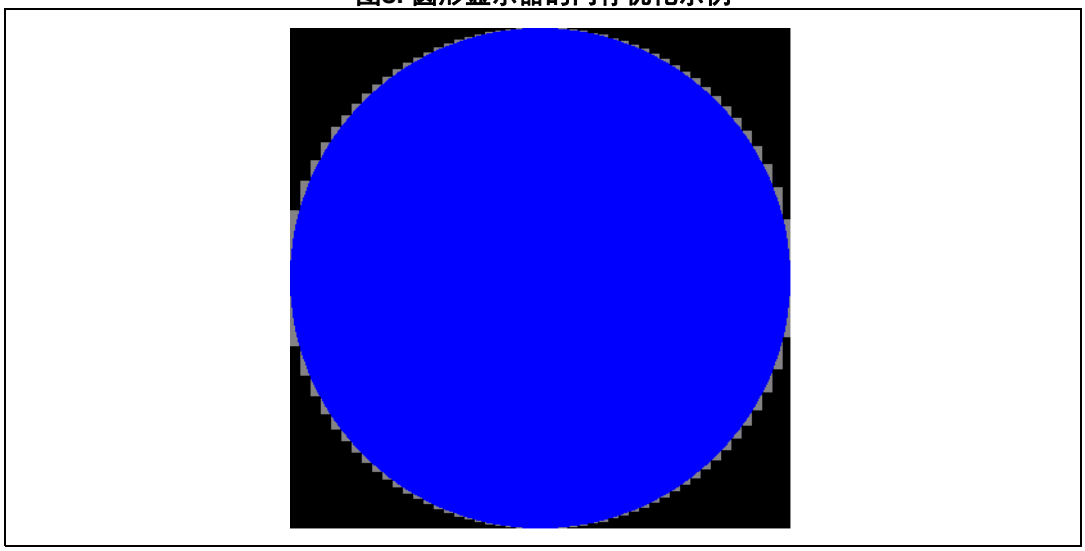
| 模式 | 方形尺寸 (KB) | GFXMMU 优化后尺寸 (KB) | 尺寸减少量（%） |
|---|---|---|---|
| 16 bpp | 297.1 | 238.3 | 19.8 |
| 24 bpp | 445.6 | 355.4 | 20.2 |
| 32 bpp | 594.1 | 471.0 | 20.7 |

*图 3*显示了与32L4R9IDISCOVERY套件的圆形显示器一起使用时的LTDC输出。

黑色区域表示未映射到物理内存的像素。这是GFXMMU带来的实际内存增益。当LTDC读取这些未映射到物理内存的像素时，GFXMMU返回一个默认值，该值在GFXMMU_DVR寄存器中编程（这种情况下为0x00）。

蓝色区域表示映射到物理内存并在显示屏幕上可见的像素。

图3. 圆形显示器的内存优化示例

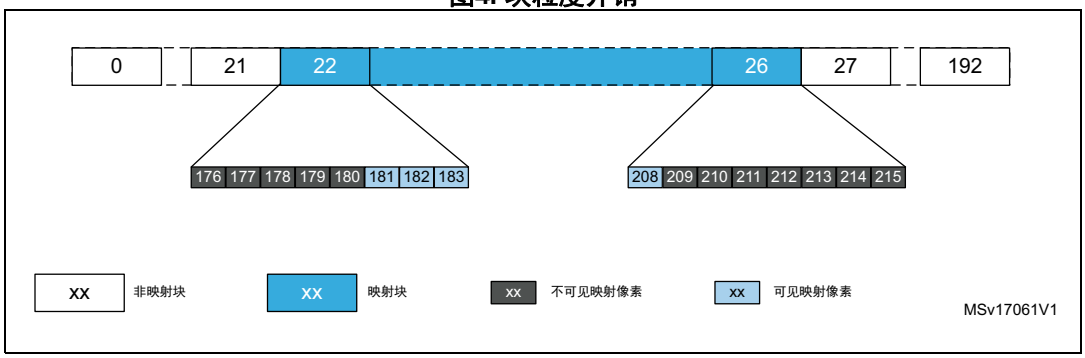


## 块粒度开销

*图 3*中的灰色区域对应着映射到物理内存但在显示屏幕上不可见的像素。这就是所谓的块粒度开销。GFXMMU具有16字节的块粒度，所以一个块可以容纳多个像素。

对于边缘上的块（第一个和最后一个块），一些像素可能在屏幕上不可见。例如，如果一行中的可见像素位于像素181和208之间，那么当使用16 bpp帧缓冲区时，必须使能的第一个块是块22。块22容纳了像素176至183。所有这些像素都被映射并物理分配到内存中，但只有像素181到183在显示屏幕上可见（见*图 4*）。

图4. 块粒度开销




尽管在内存中映射了一些不可见的像素，GFXMMU还是节省了大约20%的帧缓冲存储器空间。

# 5 GFXMMU系统级操作

当主设备尝试访问帧缓冲区时，它使用GFXMMU虚拟帧缓冲区地址。该地址将事务映射到GFXMMU从接口。

通过其主端口，GFXMMU解析地址映射并将请求重定向到相应的物理地址。

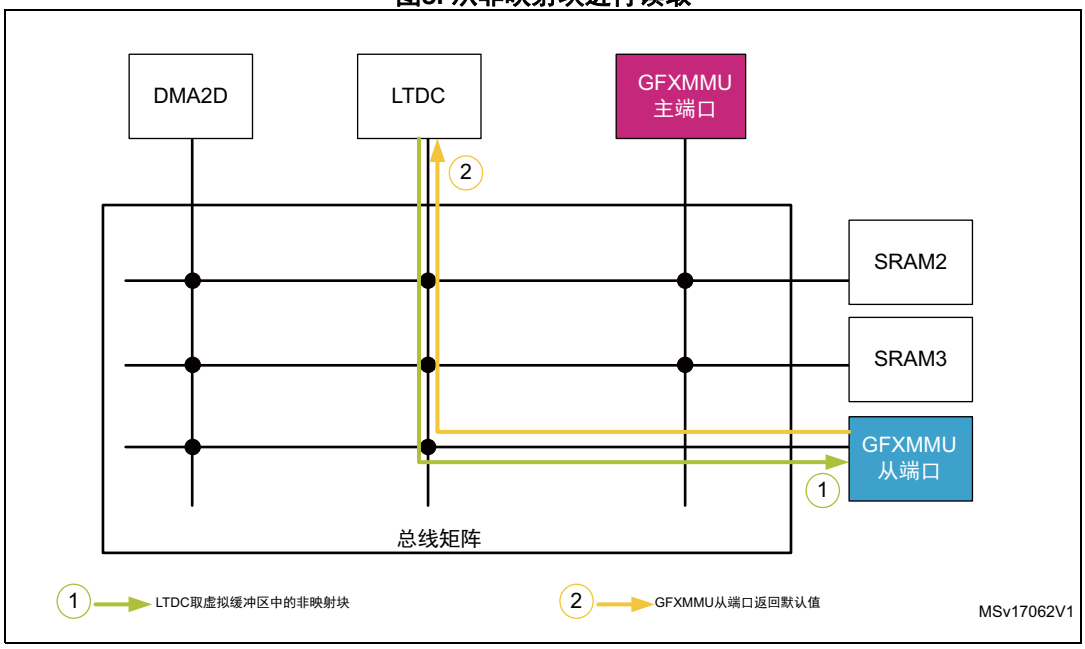
下面介绍一些典型的使用场景。

### 从非映射块进行读取

当主设备（例如LTDC）尝试从未映射到物理地址的虚拟缓冲区中读取地址时，GFXMMU响应为读取请求的默认值（参见*图 5*）。

该默认值在图形MMU默认值寄存器（GFXMMU_DVR）中编程。

**图5. 从非映射块进行读取**



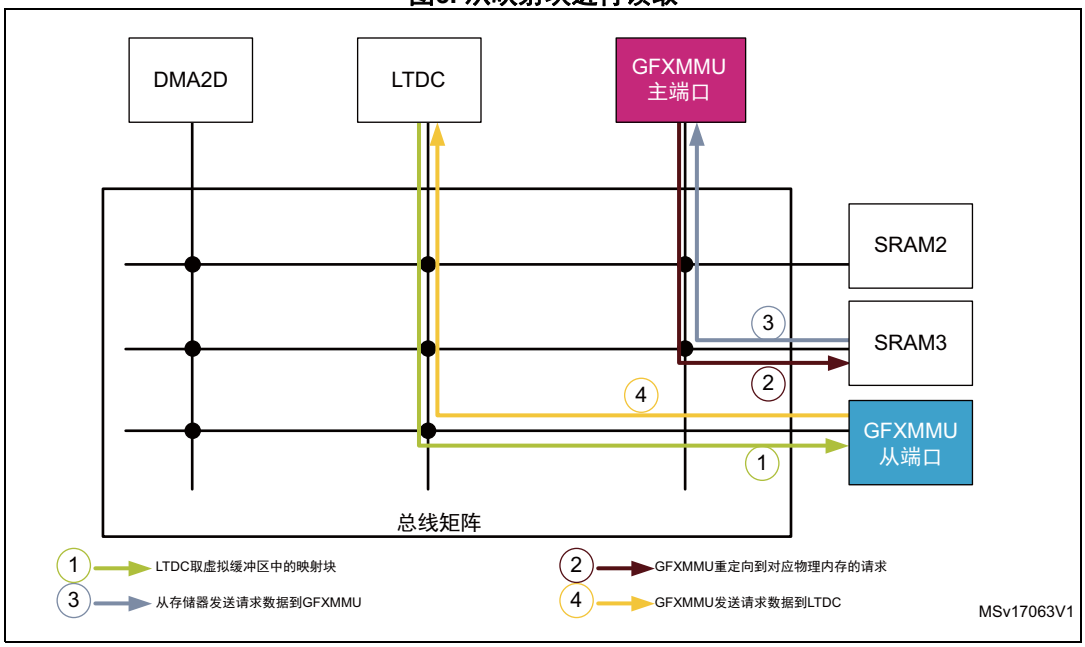

### 从映射块进行读取

当主设备（例如LTDC）尝试从映射到物理地址的虚拟缓冲区中读取地址时，GFXMMU在其从端口上接收请求并确定相应物理地址。

然后GFXMMU通过其主端口发送读取请求到保存物理地址的相应存储器。

从属存储器将所请求的数据响应给GFXMMU，后者将其重定向到LTDC。（参见*图 6*）。

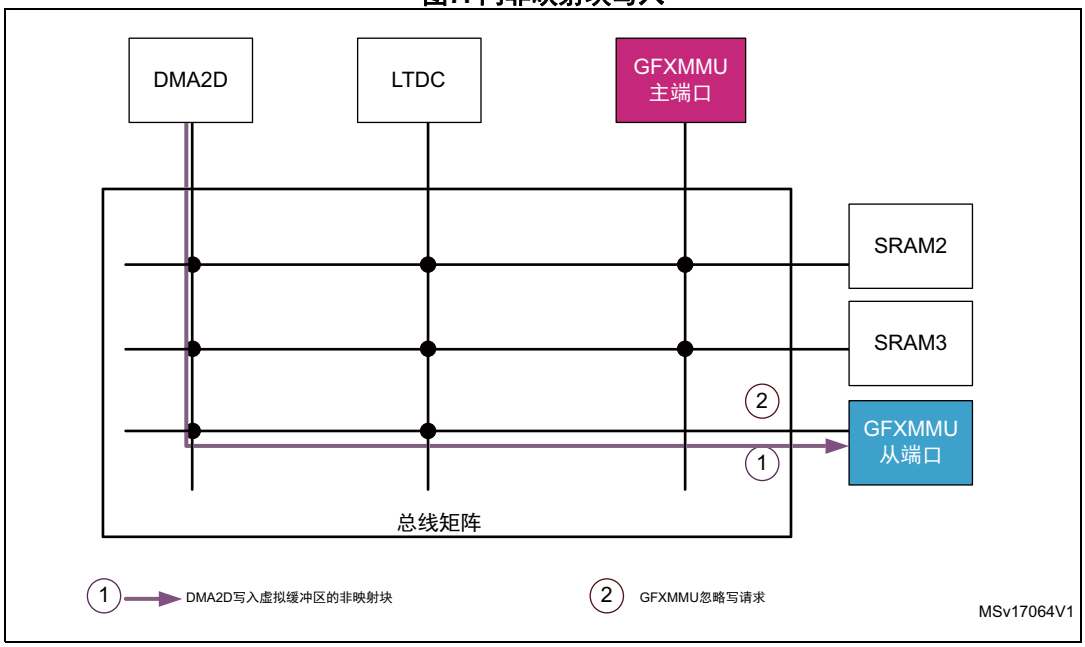
图6. 从映射块进行读取




## 向非映射块写入

GFXMMU在其从端口上从系统主设备（例如DMA2D）接收写入请求。当请求的虚拟地址对应于非映射块时，忽略该写入操作。（请参见*图 7*）。

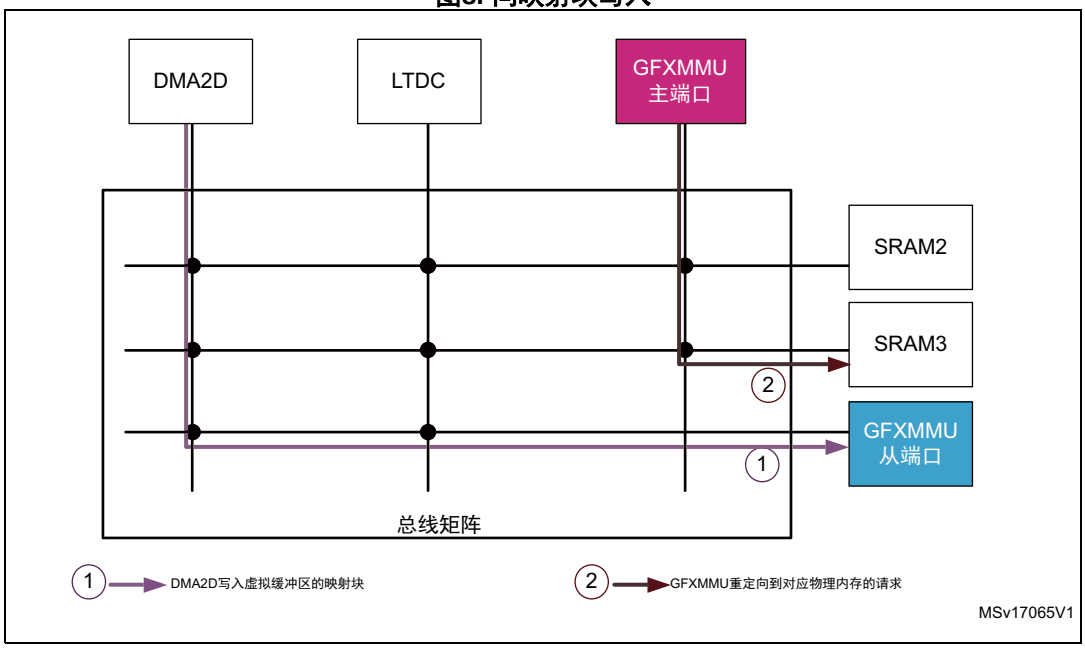
图7. 向非映射块写入



### 向映射块写入

当写入请求对应于映射到可见块的虚拟地址时，GFXMMU将虚拟地址转换为相应物理地址。然后它通过其主端口将写入请求发送到保存目标物理地址的存储器（参见*图 8*）。

**图8. 向映射块写入**



# 6 基本配置

本节介绍GFXMMU的基本配置以及与GFXMMU一起使用时的特定LTDC和DMA2D配置。

## 6.1 GFXMMU配置

本章给出了使用GFXMMU的基本参数。

### 6.1.1 GFXMMU虚拟缓冲区基址

GFXMMU可允许设置最多四个虚拟缓冲区。

在STM32物理内存映射下，每个虚拟缓冲区都有自己的基址。

主设备使用虚拟缓冲区地址来访问帧缓冲区。

### 6.1.2 GFXMMU块模式

用户必须选择某一种块模式来配置GFXMMU（GFXMMU_CR.192BM）（24 bpp帧缓冲区除外，这种情况下必须使用192块模式来使每行具有整数个像素）。

对于其他帧缓冲区色深，256块模式可支持更大的显示行宽。

### 6.1.3 GFXMMU物理帧缓冲区

对于每个虚拟缓冲区，可以在GFXMMU缓冲区配置寄存器GFXMMU_BxCR中分别配置物理帧缓冲存储器地址。

对于物理帧缓冲区地址选择，用户必须考虑其对齐、大小并避免缓冲区溢出。

#### 对齐

由于GFXMMU具有16字节的块粒度并且物理地址的四个LSB位被视为0，因此物理帧缓冲区地址必须是16字节对齐的。

#### 大小

在对LUT进行编程之后，可以使用以下公式来计算物理帧缓冲区的大小：

*物理帧缓冲区大小（KB） = 所用总块数 x 块大小 / 1024*

#### 缓冲区溢出

物理缓冲区不能溢出由其基址定义的区域的8 MB边界。因此，GFXMMU_BxCR中编程的物理缓冲区地址必须保证缓冲区的第一个和最后一个映射块位于物理内存的同一个8 MB区域中，以避免缓冲区溢出错误。

### 6.1.4 GFXMMU默认值

当虚拟地址读取不属于物理映射的块时，GFXMMU默认值（GFXMMU_DVR）由GFXMMU返回。

### 6.1.5 GFXMMU LUT

GFXMMU LUT必须根据显示形状进行编程。关于计算LUT条目的示例请参考*第 3.2 节*。

*注：* *显示形状描述表必须存储在非易失性存储器（例如内部闪存）中，然后用来编程GFXMMU LUT条目。*

## 6.2 LTDC配置

本节介绍LTDC与GFXMMU一起使用的情况下LTDC的配置。

### 6.2.1 LTDC帧缓冲区

访问帧缓冲区时，LTDC必须使用四个GFXMMU虚拟帧缓冲区的其中一个。

LTDC层x颜色帧缓冲区地址寄存器（LTDC_LxCFBAR）必须使用GFXMMU虚拟缓冲区的地址进行编程。

### 6.2.2 LTDC层间距

当LTDC与GFXMMU一起使用时，必须仔细设置LTDC层颜色帧缓冲区间距（LTDC_LxCFBLR.CFBP）。

LTDC层间距以字节表示。它取决于GFXMMU模块模式（GFXMMU_CR.192BM）：

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

## 6.3 DMA2D 配置

当DMA2D源或目标使用虚拟缓冲区时，必须遵循特定的DMA2D配置。

### 6.3.1 DMA2D帧缓冲区

DMA2D帧缓冲区必须编程为四个GFXMMU虚拟缓冲区之一。

要编程的DMA2D帧缓冲寄存器是：

- 如果目标是虚拟缓冲区，则DMA2D输出内存地址（DMA2D_OMAR）。
- 如果源是虚拟缓冲区，则DMA2D输出前景或背景内存地址寄存器（DMA2D_FGMAR或DMA2D_BGMAR）。

### 6.3.2 DMA2D行偏移

必须根据虚拟缓冲区宽度（以像素为单位）来计算DMA2D行偏移量（参见*表 2：虚拟缓冲区行宽（以像素为单位）*）。

要编程的DMA2D行偏移寄存器是：

- 如果目标地址是虚拟帧缓冲区，则DMA2D输出行偏移（DMA2D_OOR.LO）。
- 如果源地址是虚拟帧缓冲区，则DMA2D输出层的行偏移（DMA2D_FGOR.LO和DMA2D_BGOR.LO）。

# 7 软件示例

本节介绍配置GFXMMU的软件示例。

本节还介绍了LTDC和DMA2D的示例。

## 7.1 GFXMMU配置示例

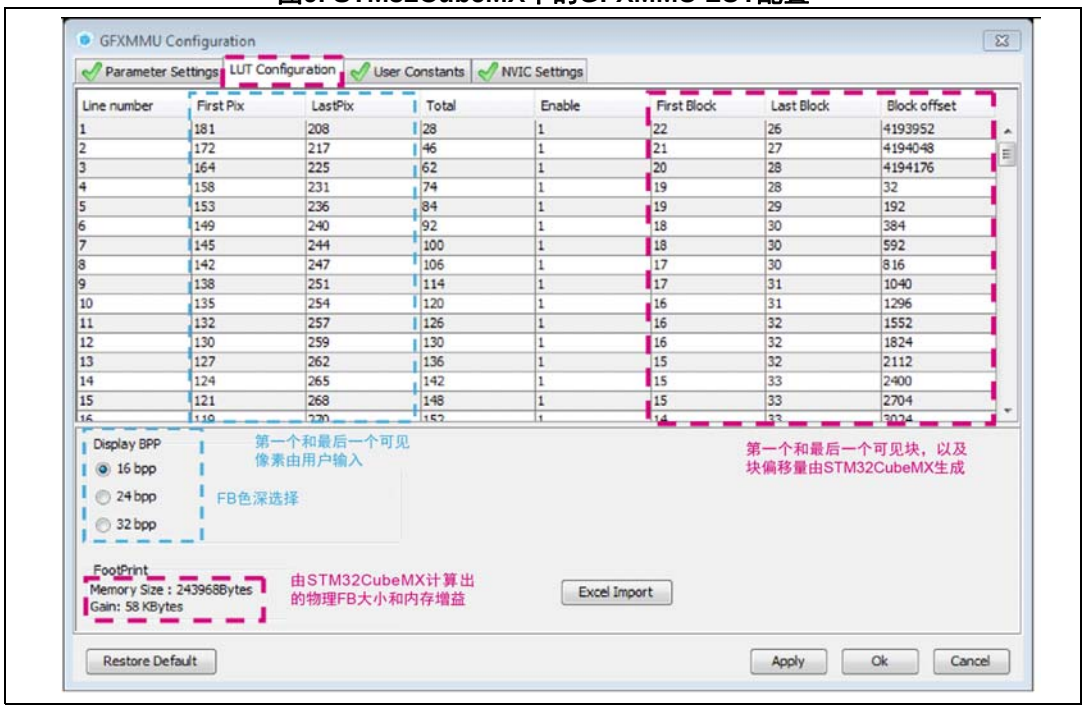
本节介绍使用STM32CubeMX的GFXMMU配置和相应的初始化代码。

### 7.1.1 使用STM32CubeMX的GFXMM配置

在GFXMMU参数设置中，用户选择要使用的块模式和虚拟缓冲区。当主设备尝试访问未映射块时，用户还可以更改GFXMMU返回的默认值。

在LUT配置界面（见*图 9*）中，用户必须输入每行的第一个和最后一个可见像素，并且必须选择帧缓冲区色深。STM32CubeMX自动生成第一个和最后一个块以及块偏移量。还可计算物理帧缓冲区所需的内存占用量。

**图9. STM32CubeMX中的GFXMMU LUT配置**




STM32CubeMX自动在“gfxmmu_lut.h”头文件中生成LUT配置。

### 7.1.2 GFXMMU初始化代码

- 物理缓冲区：物理帧缓冲区必须对齐16个字节。

*图 10*上的示例演示了如何使用三种不同的编译器（IAR，GNU和Arm®编译器）将物理帧缓冲区与16字节对齐。

**图10. GFXMMU物理帧缓冲区声明**

```
/* 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
```

STM32CubeMX根据LUT配置计算物理帧缓冲区大小。

- GFXMMU初始化：示例见*图 11*。

**图11. GFXMMU初始化**

```
/* 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__);
  }

}
```

根据用户在“LUT配置”窗口中输入的显示形状描述，gfxmmu_lut_config可由STM32CubeMX在“gfxmmu_lut.h”头文件中自动生成。它用来初始化LUT。

## 7.2 LTDC 配置示例

当使用GFXMMU时，LTDC从GFXMMU虚拟缓冲区读取数据，因此必须设置特定的LTDC配置。

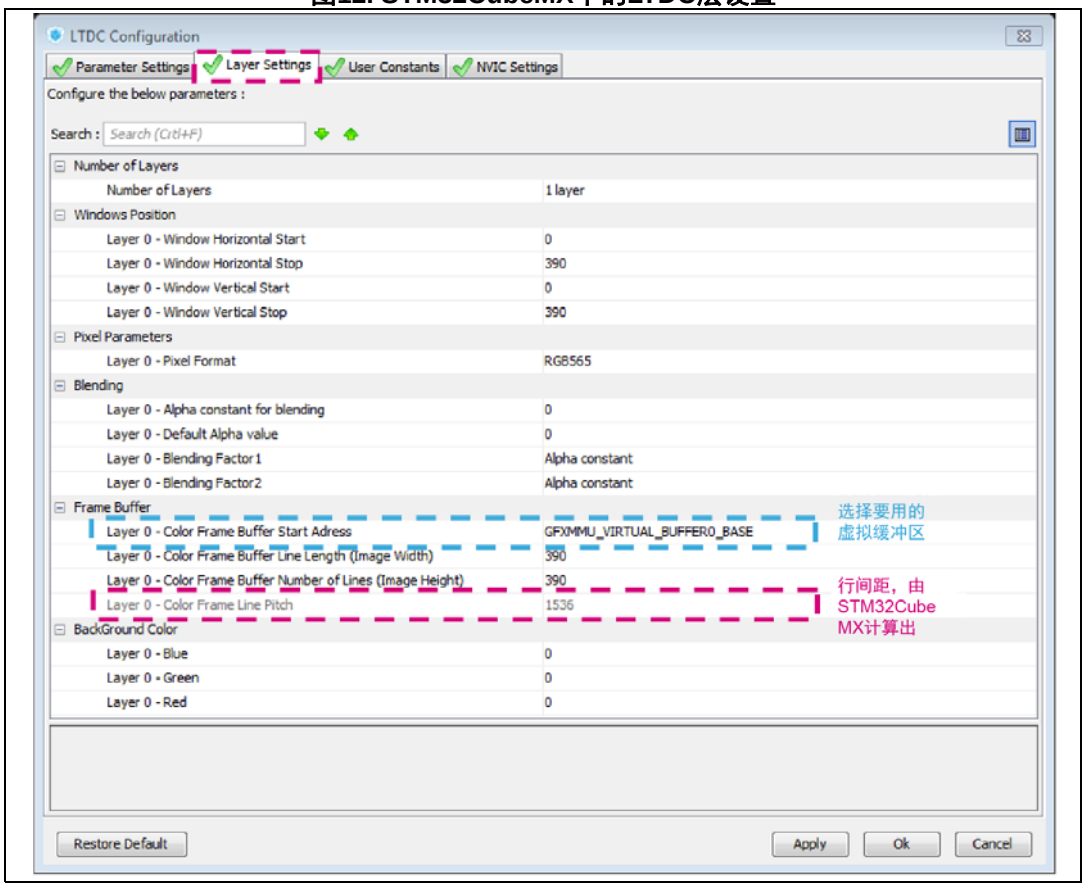
### 7.2.1 使用STM32CubeMX的LTDC配置

LTDC层帧缓冲区地址利用GFXMMU虚拟缓冲区进行编程。

STM32CubeMX根据虚拟缓冲区行宽自动计算LTDC层间距（以像素计）。

STM32CubeMX中的LTDC层设置示例见*图 12*。

图12. STM32CubeMX中的LTDC层设置



### 7.2.2 LTDC初始化代码

LTDC初始化代码示例见*图 13*。

**图13. LTDC初始化代码**

```
/* 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 配置示例

在这个例子中，DMA2D用来将图像从闪存复制到帧缓冲区。

对帧缓冲区的访问是通过GFXMMU完成的，因此DMA2D的目标地址是GFXMMU虚拟缓冲区。

输出偏移量必须考虑虚拟缓冲区行宽（以像素为单位）。

### 7.3.1 DMA2D初始化

*图 14*中是一个DMA2D初始化代码示例。

**图14. DMA2D初始化代码**

```
/* 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将图像从闪存复制到帧缓冲区

DMA2D将图像从闪存复制到帧缓冲区的代码，请参考*图 15*。

**图15. DMA2D将图像从闪存复制到帧缓冲区**

```
/* 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 应用程序示例

一个典型的GFXMMU的应用实例如*图 16*中所示。

在这个例子中，图形帧缓冲区位于内部SRAM中，图形基元存储在Octal-SPI NOR闪存中。

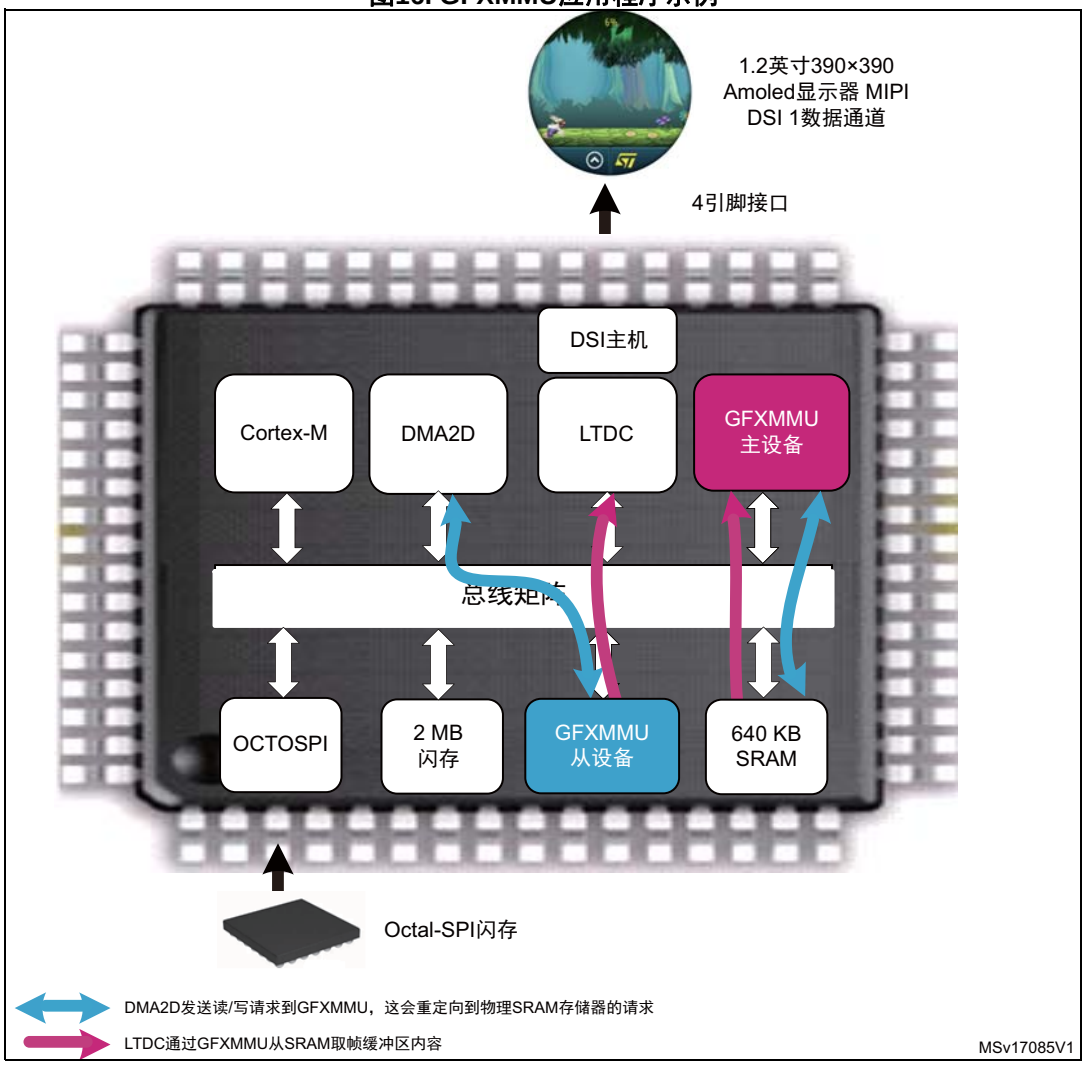
DMA2D通过OCTOSPI接口从外部Octal-SPI NOR闪存获取图形基元。

在创建图形帧缓冲区内容时，DMA2D向GFXMMU发送读/写请求，GFXMMU将请求重定向到物理SRAM内存（*图 16*中蓝色路径）。

在图形帧缓冲区显示期，LTDC通过GFXMMU（*图 16*中粉色路径）从SRAM中获取帧缓冲区内容。

DSI主机串行化LTDC输出，使得STM32能够仅利用4个引脚与MIPI®DSI显示器连接。

**图16. GFXMMU应用程序示例**



# 9 特别建议

当访问用于地址解析的帧缓冲区时，GFXMMU会增加额外的等待状态（1WS）。即使存在这种额外的延迟，相比于使用位于外部存储器中的帧缓冲区（需要数十个周期），使用内部帧缓冲区的GFXMMU性能要好得多。所以，当允许设备减小图形帧缓冲区大小以适应内部RAM时，使用GFXMMU非常有用。

有些情况下，不应使用GFXMMU，以避免访问帧缓冲区时的额外延迟：

- 如果内部RAM已经足够存储图形帧缓冲区。
  这种情况下，最好直接使用内部RAM，以便能够从0 WS执行中获益。

*注：* *如果认为进一步的内存优化要优先于额外的延迟，那么用户仍然可以使用GFXMMU。*

- 如果帧缓冲区位于外部存储器中。
  当图形帧缓冲区即使进行了GFXMMU优化也不适合内部RAM时，则将其放入外部存储器中。外部存储器应该足够大，以容纳图形帧缓冲区，而不需要对其进行优化。因此，在访问外部存储器时，GFXMMU不能用来避免增加额外的延迟。

# 10 结论

GFXMMU提供了一个优化的解决方案，通过减少帧缓冲区大小来驱动非矩形显示器；提供了一种完全集成的解决方案，无需外部RAM。

本应用笔记介绍了GFXMMU的特性和功能行为，并描述了与其他图形外设一起使用时的系统级操作。

还对基于32L4R9IDISCOVERY套件的圆形显示器的GFXMMU LUT编程进行了介绍。

最后，本文档提供了用来简化应用开发的GFXMMU的基本配置和代码示例。

# 11 版本历史

表6. 文档版本历史

| 日期 | 版本 | 变更 |
|---|---|---|
| 2017年10月10日 | 1 | 初始版本。 |

表7. 中文文档版本历史

| 日期 | 版本 | 变更 |
|---|---|---|
| 2018年8月10日 | 1 | 中文初始版本。 |

**重要通知 - 请仔细阅读**

意法半导体公司及其子公司 （“ST”）保留随时对 ST 产品和 / 或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。买方在订货之前应获取关于 ST 产品的最新信息。 ST 产品的销售依照订单确认时的相关 ST 销售条款。

买方自行负责对 ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的 ST 产品如有不同于此处提供的信息的规定，将导致 ST 针对该产品授予的任何保证失效。

ST 和 ST 徽标是 ST 的商标。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。本文档的中文版本为英文版本的翻译件，仅供参考之用；若中文版本与英文版本有任何冲突或不一致，则以英文版本为准。