life.augmented

AN5632

Application note

Migrating a graphic application from STM32L4+ to STM32U59x/5Ax/5Fx/5Gx MCUs

Introduction

For designers of STM32 microcontroller (MCU) applications, the ability to replace one microcontroller type with another from the same product family easily is an important asset. Migrating an application to a different microcontroller is often needed when product requirements grow, putting extra demands on memory size, or increasing the number of I/Os.

This application note analyzes the steps required to migrate a design based on the STM32L4+ series to STM32U595/5A5, STM32U599/5A9, STM32U5F7/5G7, and STM32U5F9/5G9 MCUs (named STM32U59x/5Ax and STM32U5Fx/5Gx in this document). This document is graphic-oriented, including only major peripherals dealing with graphic applications. For a more complete view on STM32L4+ to STM32U5 series migration, refer to the application note *Migrating from STM32L4 and STM32L4+ to STM32U5 MCUs* (AN5372).

Hardware, peripherals, and graphic software are the main aspects considered in this application note.

This document lists the full set of graphic features available for STM32L4+ and STM32U59x/5Ax/5Fx/5Gx devices.

Note: Only STM32U59x/5Ax/5Fx/5Gx devices embed advanced graphic peripherals in the STM32U5 series; STM32U535/545/575/585 devices do not.

Note: To benefit from this application note, the user can refer to the STM32 microcontroller documentation available on www.st.com, with particular focus on the reference manual and datasheets.

1 STM32U59x/Ax/5Fx/5Gx overview

This document applies to the STM32U59x/5Ax and STM32U5Fx/5Gx Arm®-based microcontrollers.

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

These devices are ultra-low-power and security MCUs, with enhanced efficiency and performance, such as:

- Up to 4 Mbytes of flash memory with ECC accelerated by instruction cache
- Up to six SRAMs with optional ECC split as follows:
  - SRAM1: 768 Kbytes (12 x 64-Kbyte blocks)
  - SRAM2: 64 Kbytes (8-Kbyte + 56-Kbyte blocks)
  - SRAM3: 832 Kbytes (13 x 64-Kbyte blocks)
  - SRAM4: 16 Kbytes
  - SRAM5: 832 Kbytes (13 x 64-Kbyte blocks)
  - SRAM6: 512 Kbytes (8 x 64-Kbyte blocks)
  - BKPSRAM (backup SRAM): 2 Kbytes retaining data in all low-power modes except Shutdown mode. The backup SRAM can be optionally retained in V_{BAT} mode.

The SRAM memory offer fits the graphic applications perfectly, with the fastest embedded memories to manage the double-frame buffer processing.

STM32U59x/5Ax/5Fx/5Gx devices use the embedded Arm® Cortex®-M33 32-bit core running at 160 MHz, versus 120 MHz for the STM32L4+ devices based on the Arm® Cortex®-M4 32-bit core. Cortex®-M33 provides improved security features with the ultra-low-power Arm® TrustZone® for Armv8-M, and the STMicroelectronics instruction/data caches (ICACHE/DCACHE) that support both internal and external memories. The instruction cache is implemented for external and internal memory access, whereas the data cache is implemented only for external memories.

STM32U59x/5Ax/5Fx/5Gx devices include a larger set of peripherals with more advanced features compared to STM32L4+, such as the ones listed below:

- Power consumption
  - Optimized power consumption in dynamic, using DC/DC and LDO in parallel (on-the-fly selection)
  - Optimized power consumption in low-power modes:
    - Low-power background autonomous mode (LPBAM): autonomous peripherals with DMA, functional down to Stop 2 mode
    - Possibility to power on or off some SRAM banks and to keep them in low-power modes
    - Timers running in Stop mode with input capture mode
    - Optimized RTC consumption
    - Advanced 14-bit ADC and ultra-low-power 12-bit ADC
- Security
  - AES and PKA (public key accelerator), side attack resistant (by hardware).
  - HUK (hardware unique key) to get a secure storage resistant to logical, side, and physical attack.
  - Life-cycle/RDP (readout protection): possibility to enable RDP regression with password.
  - TrustZone® and securable peripherals.
  - Up to eight configurable SAU regions.
  - Octo-SPI memory encryption.
  - Active tampering, secure firmware upgrade support, secure hide protection.
  - Temperature, voltage, and frequency protection monitoring for tamper detection.
  - PKA intended for the computation of cryptographic public key primitives, specifically those related to RSA, Diffie-Hellmann, or ECC (elliptic curve cryptography) over GF(p) (Galois fields). To achieve high performance at a reasonable cost, these operations are executed in the Montgomery domain.
  - On-the-fly Octo-SPI memory decryption by OTFDEC module.

- System
  - Performance
    - Cortex®-M33 at 160 MHz
    - 100 k cycles for 256 Kbytes per bank of flash memory (the rest at 10 k cycles)
    - Programmable ECC for the SRAM
  - New coprocessors
    - FMAC and CORDIC (mathematics accelerator coprocessors)
    - Instruction cache for internal and external memories and data cache for external memories only (ART Accelerator)
    - Multifunction digital filters with advanced features
- USB OTG high-speed peripheral with embedded PHY
- Graphic subsystem
  In addition to the peripherals included in both STM32L4+ and STM32U59x/5Ax/5Fx/5Gx devices, the latter offer additional peripherals that increase performance and image processing capabilities, such as:
  - GPU2D for dedicated graphics processing such as graphical user interface (GUI), menu display, or animations (such as rotation, 3D perspective, mirroring, stretching, or texture mapping), as well as hardware support for vector graphics on STM32U5Fx/5Gx.
  - Hexadeca-SPI interface (HSPI) to support most external memories such as PSRAMs, serial NAND and serial NOR flash memories, HyperRAM™ and HyperFlash™ memories. It offers a parallel interface up to 16 bits, supporting SDR or DDR modes for the data transfer rate.

Note: This document describes only the differences between STM32U59x/5Ax/5Fx/5Gx and STM32L4+, based on their system and peripherals targeting graphic applications.

2 Memories

STM32U5 devices offer larger embedded memories than STM32L4+ devices, as shown in the table below.

Table 1. Memories in STM32L4+ and STM32U59x/5Ax/5Fx/5Gx

Product	FLASH(1)	RAM size (Kbytes)							Comment
	Size (Kbytes)	SRAM1	SRAM2	SRAM3	SRAM4	SRAM5	SRAM6	BKPSRAM	
STM32U5F9	2048 to 4096	768	64	832	16	832	512	2	OTG_HS, LTDC, and/or DSI
STM32U5G9	4096								OTG_HS, LTDC, cryptography, and/or DSI
STM32U5F7	2048 to 4096								OTG_HS, LTDC
STM32U5G7	4096								OTG_HS, LTDC, and cryptography
STM32U599	2048 to 4096						N/A		OTG_HS, LTDC, and/or DSI
STM32U5A9	4096								OTG_HS, LTDC, cryptography, and/or DSI
STM32U595	2048 to 4096								OTG_HS
STM32U5A5	4096								OTG_HS and cryptography
STM32L4R9	1024 to 2048	192	64	384	N/A				OTG_FS and DSI
STM32L4S9									OTG_FS, DSI, and cryptography
STM32L4R7									OTG_FS and LTDC
STM32L4S7	2048								OTG_FS, LTDC, and cryptography
STM32L4R5	1024 to 2048								OTG_FS
STM32L4S5	2048								OTG_FS and cryptography
STM32L4P5	512 to 1024	128		128					OTG_FS
STM32L4Q5	1024								OTG_FS and cryptography

1. *Dual bank for all devices.*

STM32U59x/5Ax/5Fx/5Gx devices embed many internal SRAMs to meet the specific requirements for typical graphic applications (for example, smart-watch devices). These can be used, depending on the screen resolution, to handle the double-frame buffers in the internal SRAMs to increase the overall graphic performance and memory bandwidth (as well as latency).

3 Graphic resources

Most of the graphic resources are shared between STM32U59x/5Ax/5Fx/5Gx and STM32L4+ devices.

More powerful peripherals have been introduced from STM32U59x/5Ax onwards to increase the overall graphic performance (very beneficial for animation purposes, for example).

GPU2D is one of these new peripherals contributing to offloading the CPU for image processing operations. The HSPI peripheral improves access to the external PSRAM/HyperRAM™, or NorFlash/HyperFlash™, offering 16-bit high-speed I/Os. This considerably speeds up the data transfer to and from the external memory GPU2D is connected to for image processing, for instance (performance also increases when any controller peripheral uses the H-SPI interface to communicate with external memories).

The table below details the set of peripherals for the various products.

Table 2. Peripherals involved in the graphic system

Peripheral		STM32 L4x7	STM32 L4x9	STM32 L4P5/4Q5	STM32 L4R5/4S5	STM32U 595/5A5	STM32U 599/5A9	STM32U 5F7/5G7	STM32U 5F9/5G9	Comment
Graphic peripherals	DMA2D	X	X	X	X	X	X	X	X	Refer to Section 3.1
	GPU2D	-	-	-	-	-	X	X	X	New peripheral actively participating in the overall graphic performance increase. Refer to Section 3.2
	GFXMMU	X	X	-	-	-	X	X	X	Refer to Section 3.3
	LTDC	X	X(1)	X	-	-	X	X	X	Refer to Section 3.4
	JPEG codec	-	-	-	-	-	-	X	X	Refer to Section 3.5
Memory interfaces	OCTOSPI1	X	X	X	X	X	X	X	X	Refer to Section 3.6
	OCTOSPI2	X(2)	X(1)	X(3)	X	X	X	X(4)	X	
	FSMC	X	X	X(5)	X	X(5)	X(5)	X	X	Refer to Section 3.7
	SDMMC	X	X	X	X	X	X	X(6)	X	Refer to Section 3.10
	HSPI	-	-	-	-	-	X	-	X	New peripheral to interface with high-speed external memories. Refer to Section 3.8
Graphic system interfaces	DCMI	X	X	X	X	X	X	X	X	Refer to Section 3.9
	DSI	-	X	-	-	-	X(7)	-	X	Refer to Section 3.11

1. *Not available on STM32L4x9VI/VG.*
2. *Not available on STM32L4x7VI.*
3. *Not available for packages below 132 pins.*
4. *Not available for packages below 208 pins.*
5. *Not available for packages below 100 pins.*
6. *Not available for LQFP100 DSI SMPS.*
7. *Available on STM32U5x9ZI/JY, STM32U5x9BJY, and STM32U5x9NI/JH.*

The peripheral memory mapping differences are detailed in the table below.

Table 3. Peripheral memory mapping in STM32L4+ and STM32U59x/5Ax/5Fx/5Gx

Peripheral		STM32L4+	STM32U59x/5Ax/5Fx/5Gx
OCTOSPI1	Nonsecure	0xA000 1000 - 0xA000 13FF	0x420D 1400 - 0x420D 17FF
	Secure	-	0x520D 1400 - 0x520D 17FF
OCTOSPI2	Nonsecure	0xA000 1400 - 0xA000 17FF	0x420D 2400 - 0x420D 27FF
	Secure	-	0x520D 2400 - 0x520D 27FF
OCTOSPIM	Nonsecure	0x5006 1C00- 0x5006 1FFF	0x420C 4000- 0x420C 43FF
	Secure	-	0x520C 4000- 0x520C 43FF
HSPI(1)	Nonsecure	-	0x420D 3400 - 0x420D 37FF
	Secure		0x520D 3400 - 0x520D 37FF
DMA2D	Nonsecure	0x4002 B000 - 0x4002 BBFF	0x4002 B000 - 0x4002 BBFF
	Secure	-	0x5002 B000 - 0x5002 BBFF
GFXMMU(1)	Nonsecure	0x4002 C000 - 0x4002 EFFF	0x4002 C000 - 0x4002 EFFF
	Secure	-	0x5002 C000 - 0x5002 EFFF
LTDC(1)	Nonsecure	0x4001 6800 - 0x4001 6BFF	0x4001 6800 - 0x4001 6BFF
	Secure	-	0x5001 6800 - 0x5001 6BFF
FSMC	Nonsecure	0xA000 0000 - 0xA000 03FF	0x420D 0400 - 0x420D 07FF
	Secure	-	0x520D 0400 - 0x520D 07FF
SDMMC1	Nonsecure	0x5006 2400 - 0x5006 27FF	0x420C 8000 - 0x420C 83FF
	Secure	-	0x520C 8000 - 0x520C 83FF
SDMMC2(1)	Nonsecure	0x5006 2800 - 0x5006 2BFF	0x420C 8C00 - 0x420C 8FFF
	Secure	-	0x520C 8C00 - 0x520C 8FFF
GPU2D(1)	Nonsecure	-	0x4002 F000 - 0x4002 FFFF
	Secure		0x5002 F000 - 0x5002 FFFF
DCMI	Nonsecure	0x5005 0000 - 0x5005 03FF	0x4202 C000 - 0x4202 C3FF
	Secure	-	0x5202 C000 - 0x5202 C3FF
DSI(1)	Nonsecure	0x4001 6C00 - 0x4001 73FF	0x4001 6C00 - 0x4001 7BFF
	Secure	-	0x5001 6C00 - 0x5001 7BFF
JPEG	Nonsecure	-	0x4002 A000 - 0x4002 AFFF
	Secure	-	0x5002 A000 - 0x5002 AFFF

1. *For devices having this feature.*

3.1 Chrom-ART Accelerator (DMA2D)

The Chrom-ART Accelerator (DMA2D) is a graphic-dedicated peripheral allowing image manipulation without using the CPU. DMA2D is a hardware accelerator for graphical operations (such as plane blending, pixel format conversions, or antialiasing fonts with specific modes). DMA2D is built around a graphic 2D DMA for fast data copy operations.

There is no major difference between the STM32L4+ and the STM32U59x/5Ax/5Fx/5Gx devices that are fully compatible.

STM32U59x/5Ax/5Fx/5Gx devices offer new trigger capabilities from the Chrom-ART Accelerator compared to STM32L4+ devices. These new trigger capabilities can trigger a system GPDMA channel. This brings more flexibility for synchronizing the application software based on specific events, as shown in the table below.

Table 4. Additional trigger connections on STM32U59x/5Ax

DMA2D trigger source	GPDMA trigger connection
Transfer complete	GPDMA_CxTR2.TRIGSEL[5:0] = 50
CLUT transfer complete	GPDMA_CxTR2.TRIGSEL[5:0] = 51
Transfer watermark complete	GPDMA_CxTR2.TRIGSEL[5:0] = 52

3.2 Neo-Chrom graphic processor (GPU2D)

GPU2D is a dedicated graphic processing unit accelerating numerous 2.5D graphic applications, such as graphical user interfaces (GUIs), menu displays, or animations. GPU2D works alongside an optimized software stack designed for state-of-the-art graphic rendering (TouchGFX). For example, the texture mapper is now fully hardware accelerated, with a x10 factor compared to a classical software implementation and no code modification on the user side (additional material and software packages can be provided on demand).

GPU2D is mainly used to transform images (3D perspective correct projections, texture mapping with bilinear filtering, or sampling point). GPU2D supports blit operations like rotation or mirroring, stretching, color keying, and pixel format conversions.

GPU2D can be used for 2D drawing with pixel and line drawing, or filling rectangles, triangles, and quadrilaterals. GPU2D supports text rendering (A1, A2, A4, and A8 antialiasing bitmap) and alpha blending with a hardware blender.

GPU2D uses the embedded graphic peripherals in STM32U599/5A9 devices and internal and external memory resources to improve graphic performances, resulting in a state-of-the-art graphic system. STM32U5Fx/5Gx devices further enhance GPU2D with the hardware support of vector graphic calculation, offering high-end performances.

3.3 Chrom-GRC (GFXMMU)

The Chrom-GRC (GFXMMU) is a graphical memory management unit aiming to optimize memory use according to the display shape. GFXMMU operates an address translation from the virtual buffer space to the physical address memory in a linear way. There are up to four virtual memory spaces (and so four physical memory spaces as well).

GFXMMU acts as the controller on the AHB bus to target the physical memory when performing the address translation to read or write the physical memory.

The table below summarizes the connection of GFXMMU for each product and for each controller/target.

Note: This peripheral is not present on STM32U595/5A5 devices.

Table 5. GFXMMU connection to controller/target ports for STM32L4+ and STM32U59x/5Ax/5Fx/5Gx

Peripheral		STM32L4+(1)	STM32U59x/5Ax/5Fx/5Gx
Controller	CPU	X	X
	LTDC	X	X
	DMA2D	X	X
	DMA	X	-
	SDMMC	X	-
	GPU2D	-	X
Target	FLASH	X	X
	SRAM1	X	X
	SRAM2	X	X
	SRAM3	X	X

Peripheral		STM32L4+[1]	STM32U59x/5Ax/5Fx/5Gx
Target	SRAM4	-	-
	SRAM5	-	X
	BKPSRAM	-	-
	OCTOSPI	X	X
	FSMC	X	X
	HSPI	-	X

1. *No GFXMMU on STM32L4R5x/4S5x/4P5x/4Q5x devices.*

3.4 LCD-TFT display controller (LTDC)

The LCD-TFT display controller (LTDC) provides a parallel digital RGB (Red, Green, Blue), pixel clock, data enable, and synchronization signals, to interface directly with a wide range of LCD or TFT panels. LTDC is the same for STM32U59x/5Ax/5Fx/5Gx and STM32L4+ devices (no functional differences).

Note: This peripheral is not present on STM32U595/5A5 devices.

Table 6. Additional LTDC trigger connection on STM32U59x/5Ax/5Fx/5Gx

DMA2D trigger source	GPDMA trigger connection
LTDC line interrupt (ltdc_li)	GPDMA_CxTR2.TRIGSEL[5:0] = 47

The LTDC pixel clock is connected differently to the PLL depending on the targeted device (see the table below).

Table 7. LTDC pixel clock connection to RCC

STM32L4+[1]	STM32U59x/5Ax/5Fx/5Gx
PLLSAI2 (/R)	PLL2 (/R) or PLL3 (/R)

1. *No LTDC on SMT32L4R5x/4S5x devices.*

LTDC uses several I/Os to connect an external display to the MCU. The alternate function (AF) number used to map the LTDC output to the I/Os is different depending on the device, as shown in the table below.

Table 8. Alternate function to map the LTDC to the external I/Os

STM32L4+[1]	STM32U59x/5Ax/5Fx/5Gx
AF11	AF7, AF8

1. *No LTDC on the STM32L4R5x/4S5x devices.*

The LTDC output signals mapped on the GPIOs are strictly compatible when porting software from STM32L4+ to STM32U59x/5Ax/5Fx/5Gx devices.

The table below shows an additional remapping for STM32U59x/5Ax/5Fx/5Gx devices. Only the alternate function number is different, and the software developer needs to be cautious when porting the part of the code that configures LTDC.

Table 9. LTDC I/O port mapping in STM32L4+ and STM32U59x/5Ax/5Fx/5Gx

Pin name	STM32L4+	STM32U59x/5Ax/5Fx/5Gx
LCD_VSYNC	-	PD13

3.5 JPEG codec

The hardware 8-bit JPEG codec encodes uncompressed image data streams and decodes JPEG-compressed image data streams. It also fully manages JPEG headers. The main JPEG codec features are:

- Fully synchronous, high-speed operations.
- Configurable as encoder or decoder.
- Single-clock-per-pixel encode/decode.
- RGB, YCbCr, YCMK, and BW (gray scale) image color space support.
- 8-bit depth per image component for encode/decode.
- JPEG header generator/parser with enable/disable.
- Four programmable quantization tables.
- Single-clock Huffman coding and decoding.
- Fully programmable Huffman tables (two AC and two DC).
- Fully programmable minimum coded unit.
- Concurrent input and output data stream interface.

3.6 Octo-SPI interface (OCTOSPI)

Octo-SPI supports most external serial memories, including serial PSRAMs, serial NANDs and serial NORFlash memories, and HyperRAM™ and HyperFlash™ memories, with different modes (indirect, automatic status-polling, or memory-mapped).

The Octo-SPI interface can be used to store graphic primitives, pointed by the graphic application software, for instance. STM32U59x/5Ax/5Fx/5Gx and STM32L4+ devices embed two Octo-SPI instances.

The kernel clock connection for each Octo-SPI instance is slightly different.

Table 10. OCTOSPI kernel clock source connection

STM32L4+	STM32U59x/5Ax/5Fx/5Gx
PLL48M1CLK (PLL/Q)	MSIK, PLL1/Q, PLL2/Q, and SYSCLK

STM32U59x/5Ax/5Fx/5Gx devices support the new features listed below:

- Differential clock for 1.8 V HyperBus™ mode.
- Support of AP memory Quad- and Octal-SPI PSRAMs.
- CS boundary and refresh
- OTFDEC protecting the flash memory code
- TrustZone® security

The I/O port mapping differences on the Octo-SPI I/O manager (OCTOSPIM) are detailed in the table below.

Table 11. OCTOSPIM I/O port mapping on STM32L4+ and STM32U59x/5Ax/5Fx/5Gx

Pin name	STM32L4+	STM32U59x/5Ax/5Fx/5Gx
OCTOSPIM_P1_IO7	-	PC0
OCTOSPIM_P1_NCLK	-	PF11, PE9, PB12, PB5
OCTOSPIM_P2_IO0	PI11	PI3
OCTOSPIM_P2_IO1	PI10	PI2
OCTOSPIM_P2_IO2	PI9	PI1
OCTOSPIM_P2_NCLK	-	PF5, PH7, PI7
OCTOSPIM_P2_NCS	PI8	PA0, PA12, PF6

3.7 Flexible static memory controller (FSMC)

The FSMC includes two memory controllers:

- A NOR/PSRAM memory controller.
- A NAND memory controller.

The FMSC is almost the same for STM32L4+ and STM32U59x/5Ax/5Fx/5Gx, except for a new PSRAM counter timing embedded in STM32U59x/5Ax/5Fx/5Gx devices, which can also be secure using the TrustZone® controller (refer to the reference manual for more details).

The FSMC can be used to interface the LCD_TFT display through an 8- or 16-bit parallel interface (called MCU interface or MIPI DBI).

This solution offers a low pin-count cost to connect to the display and there is no specific memory refresh performed by the MCU to consider. All operations are managed by the external LCD-TFT display controller. The FSMC signals needed to interface the external LCD-TFT display controller are the following:

- FSMC [D0:D15]: FSMC databus: 16-bit width
- FSMC NEx: FSMC chip select
- FSMC NOE: FSMC output enable
- FSMC NWE: FSMC write enable
- FSMC Ax (x = 0 to 25): one address line used to select between command and data

Table 12. Signals correspondence between the FSMC and the external LCD display

FSMC signals	External LCD display signals
FSMC_Ax	RS
FSMC_NEx	CSn
FSMC_NWE	WRn/SCL
FSMC_NOE	RDn
FSMC [D0:D15]	D0-D15

The FSMC can also be used to connect external PSRAMs.

On STM32U599/5A9/5F9/5G9 devices, it is recommended to use HSPI to connect external memories for graphics. This high-speed interface offers outstanding performance to store graphic primitives or, if necessary, to interface external memories to store application frame buffers pointed by GPU2D.

The STM32U59x/5Ax/5Fx/5Gx FSMC is mapped using two alternate function (AF) numbers (a single one for STM32L4+). The table below presents this difference.

Table 13. Alternate function to map the FSMC to the I/O ports

STM32L4	STM32U59x/5Ax
AF11	AF11, AF12

There is one additional I/O port mapping on STM32U59x/5Ax/5Fx/5Gx devices as detailed in the table below.

Table 14. Additional FSMC I/O port mapping on STM32U59x/5Ax/5Fx/5Gx

Pin name	STM32U59x/5Ax/5Fx/5Gx
FMC_NBL1	PB15 (AF11)

3.8 Hexadeca-SPI (HSPI)

HSPI supports most of the external serial memory types (such as serial PSRAMs, serial NAND/NOR flash memories, HyperRAM™, and HyperFlash™ memories), with the following functional modes:

- Indirect mode: all operations are performed using the HSPI registers.
- Automatic status-polling mode: the external memory status register is periodically read and an interrupt can be generated in the case of flag setting.
- Memory-mapped mode: the external memory is memory mapped. The system sees it as if it was an internal memory supporting read and write operations.

Data access can be 8, 16, and 32 bits wide. HSPI supports quad, dual-quad, octal, dual-octal, and 16-bit configurations.

HSPI runs at up to 160 MHz and is new in STM32U599/5A9/5F9/5G9 devices. It offers better performance for data access and reduced latency (higher for applications using Octo-SPI or FSMC to target external memories). It is an added value when migrating a graphic application from STM32L4+ to STM32U599/5A9/5F9/5G9 devices.

Note: This peripheral is not present on STM32U595/5A5/5F7/FG7.

For more details, see the reference manual or the datasheet.

3.9 Digital camera interface (DCMI)

The DCMI is a synchronous parallel interface able to receive a high-speed data flow from an external 8-, 10-, 12-, or 14-bit CMOS camera module. The DCMI supports different data formats: YCbCr4:2:2/RGB565 progressive video and compressed data (JPEG). This interface can be used with black-and-white, X24, and X5 image sensors, provided preprocessing (such as resizing) is performed in the camera module.

The DCMI is the same in STM32L4+ and STM32U59x/5Ax/5Fx/5Gx devices. The only difference is highlighted in the table below.

Table 15. DCMI I/O port mapping difference on STM32L4+ and STM32U59x/5Ax/5Fx/5Gx

Pin name	STM32L4+	STM32U59x/5Ax/5Fx/5Gx
DCMI_D12	PI8(AF10)	PF6 (AF4)

3.10 Secure digital input output multimedia card interface (SDMMC)

The SD/SDIO embedded multimedia card (eMMC) host interface (SDMMC) provides an interface between the AHB bus and SD memory cards, SDIO cards, and eMMC devices. The multimedia card system specifications are available through the multimedia card association website (www.mmca.org), published by the MMCA technical committee. SD memory card and SDIO card system specifications are available through the SD card association website (www.sdcard.org).

STM32L4+ and STM32U59x/5Ax/5Fx/5Gx devices embed two SDMMC instances (except STM32L4Sx, which has only one SDMMC instance). The main feature differences are described in the table below.

Table 16. SDMMC features of STM32L4+ and STM32U59x/5Ax/5Fx/5Gx

Feature	STM32L4+	STM32U59x/5Ax
Full compliance with MultiMediaCard system specification	Version 4.5	Version 5.1
Full compliance with SD memory card specification	Version 4.1	Version 6.0
Data transfer	Up to 104 Mbyte/s for the 8-bit mode	Up to 208 Mbyte/s for the 8-bit mode(1)
IDMA linked list	Not supported	Supported

1. *Depending on GPIO performance. Refer to product datasheet.*

The SDMMC clock connection sources into the RCC (reset and clock control) are described in the table below.

Table 17. SDMMC clock connection to the RCC for STM32L4+ and STM32U59x/5Ax/5Fx/5Gx

STM32L4+	STM32U59x/5Ax/5Fx/5Gx
PLL/P (PLLSAI3CLK)	PLL1/P (pll1_p_ck)
MSI	MSIK
PLL/Q (PLL48M1CLK)	PLL1/Q (pll1_q_ck)
PLLSAI1/Q (PLL48M2CLK)	PLL2/Q (pll2_q_ck)
HSI48	HSI48

The alternate function (AF) numbers used to map the SDMMC signals on the I/O ports are not exactly the same for STM32U59x/5Ax/5Fx/5Gx and STM32L4+ devices (refer to the product datasheet), as shown in the table below.

Table 18. Alternate function to map the SDMMC to the I/O ports

Instance	STM32L4+	STM32U59x/5Ax/5Fx/5Gx
SDMMC1	AF7	AF8
SDMMC2	AF11	AF11
SDMMC1/2	AF8, AF12	AF12

There are some differences in the I/O port mapping between STM32L4+ and STM32U59x/5Ax/5Fx/5Gx devices, as detailed in the table below.

Table 19. SDMMC I/O port mapping on STM32L4+ and STM32U59x/5Ax/5Fx/5Gx

Pin name	STM32L4+	STM32U59x/5Ax/5Fx/5Gx
PA0	-	SDMMC2_CMD
PA1	SDMMC2_CMD	-
PB12	SDMMC2_CK	
PC0	SDMMC2_CKIN/SDMMC1_CMD	SDMMC1_D5
PC1	-	SDMMC2_CK
PD4	SDMMC2_CKIN	-
PG2	SDMMC2_D4	
PG3	SDMMC2_D5	
PG4	SDMMC2_D6	
PG5	SDMMC2_D7	
PG9	SDMMC2_D0	
PG10	SDMMC2_D1	
PG11	SDMMC2_D2	
PG12	SDMMC2_D3	

3.11 DSI host (DSI)

The DSI is part of a group of communication protocols defined by the MIPI Alliance. The MIPI DSI® host is a digital core that implements all protocol functions defined in the MIPI DSI® specification. The DSI host provides an interface between the system (LTDC and APB interfaces) and the MIPI D-PHY, allowing the user to communicate with a DSI-compliant display.

The kernel of the DSI host is compatible with both STM32U599/5A9/5F9/5G9 and STM32L4+ devices. The D-PHY is different, with the following updates:

- Wrapper to control the D-PHY
- Power supply
- PLL source

The D-PHY physical layer configuration phase needs to be adapted to the STM32U599/5A9/5F9/5G9 devices when porting a graphic application code from a STM32L4+ device.

Note: This peripheral is not present on STM32U595/5A5/5F7/5G7.

The differences in power supply are highlighted in the table below.

Table 20. DSI power supply for STM32L4+ and STM32U599/5A9/5F9/5G9

Feature	STL32L4R9/S9	STM32U599/5A9/5F9/5G9
Internal voltage regulator	Available	N/A
DSI host power supply	V_{DDDSI} (connected to the internal voltage regulator)	V_{DDDSI}
DSI DPHY transceiver power supply	$V_{DD12DSI}$ (an external capacitor of 2.2µF must be connected to the VDD12DSI pin)	$V_{DD11DSI}$ (must be connected to V_{DD11})
Output DSI regulator	V_{CAPDSI} (to be connected externally to VDD12DSI pin)	N/A

The source clock connections from the RCC to the DSI are detailed in the table below.

Table 21. DSI clock source connections of STM32L4+ and STM32U599/5A9/5F9/5G9

STL32L4R9/S9	STM32U599/5A9/5F9/5G9
DSI_PHY PLL clock	DSI_PHY PLL clock
PLLSAI2/Q	PLL3/P

The mapping of the tearing effect input pin is only software compatible if the PF11 pin is used when porting the graphic application code from STM32L4+ to STM32U599/5A9/5F9/5G9 devices.

Table 22. DSI I/O port mapping on STM32L4+ and STM32U599/5A9/5F9/5G9

Pin name	STL32L4R9/S9	STM32U599/5A9/5F9/5G9
DSI_TE	PB7, PB11, PF11, PG6	PF10, PF11, PG5

D-PHY configuration parameters

D-PHY transceivers are intrinsically linked to the targeted technology that is different between STM32U599/5A9/5F9/5G9 and STM32L4+ devices. The transceiver configuration is also different and must be adjusted to match the device specificities. The major ones are described below to help the user port the graphic application from STM32L4+ to STM32U599/5A9/5F9/5G9:

- The UIX4[4:0] bitfield defining the bit period in high-speed mode (in units of 0.25 ns) is in DSI_WPRCR0 for STM32L4+, but does not exist for STM32U599/5A9/5F9/5G9. For the latter, the software must configure the frequency band of:
  - The clock line in BC[4:0] of DSI_DPCBCR.
  - The data lanes in BC[4:0] of DSI_DPDL0BCR and DSI_DPDL1BCR.
- In STM32U599/5A9/5F9/5G9, the slew rate of the clock and the data lines must be set to 0x0E (not the reset value) in SRC[7:0] of DSI_DPCSRCR, DSI_DPDL0SRCR, and DSI_DPDL1SRCR, respectively.
- In STM32U599/5A9/5F9/5G9, the reference bias must be powered up by setting PWRUP in DSI_BCFGR (not available on STM32L4+).
- In STM32U599/5A9/5F9/5G9, the PLL has to be configured according to DSI_WPTR (PLL loop filter control, as well as charge pump) and DSI_WPRPCR, knowing that STM32U599/5A9/5F9/5G9 devices no longer have a regulator (the REGEN bit on STM32L4+ is not present on STM32U599/5A9/5F9/5G9).

After these mandatory configurations above, the D-PHY PLL can be enabled by setting PLLEN in DSI_WRPCR.

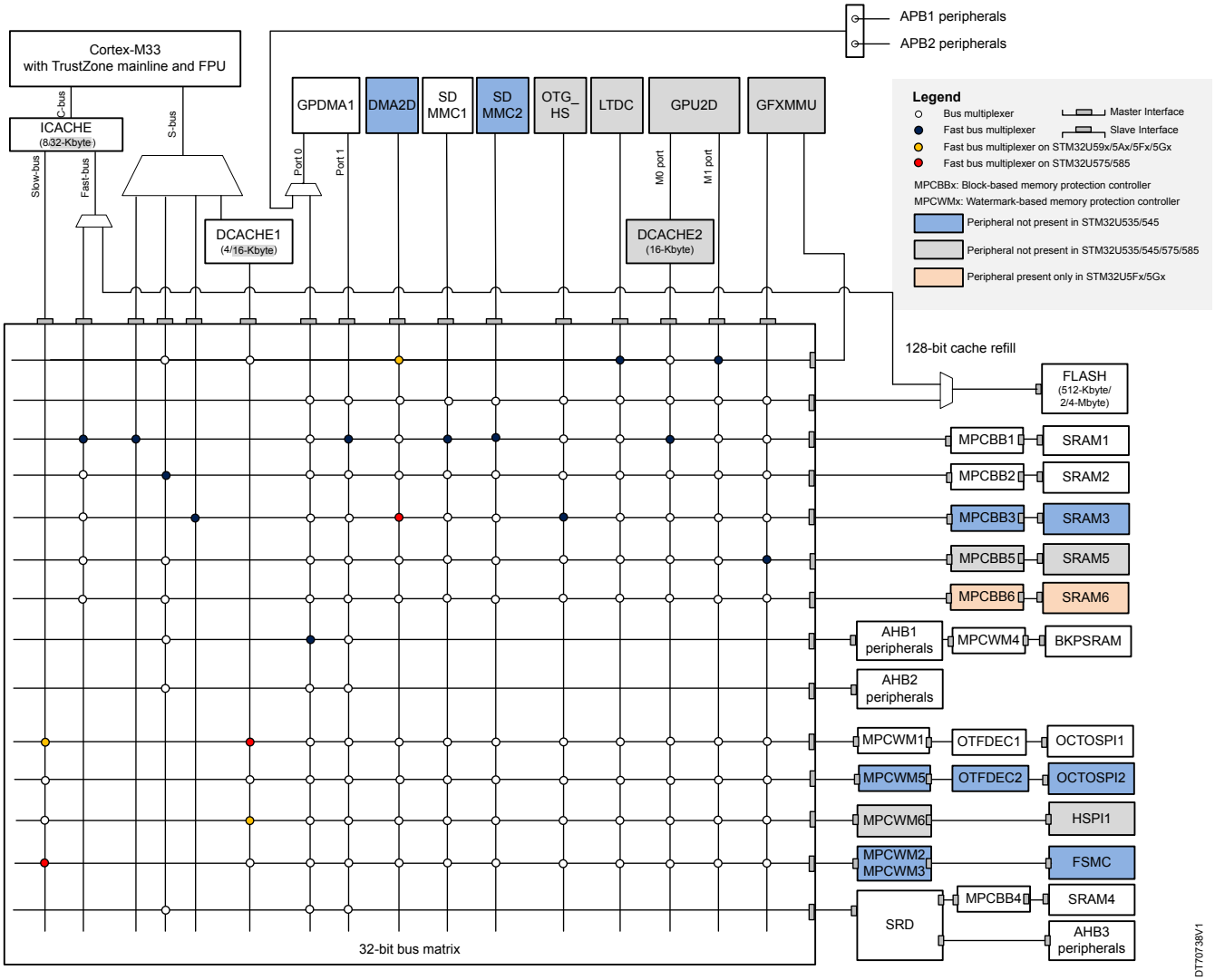
4 Neo-Chrom software integration

The GPU2D is able to accelerate most graphic operations required by modern applications: for example, classic 2D blitting operations with rotations and alpha-blending, Porter/Duff compositing, perspective-correct texture mapping, point-sampling and bilinear filtering, 8x MSAA antialiasing when rendering triangles and quadrilaterals. All these operations are available for a wide range of supported pixel formats.

4.1 GPU2D and DCACHE2

The figure below describes the interconnections between the GPU2D and the rest of the system.

Figure 1. STM32U5 system architecture



The GPU2D has access to both internal SRAM and external memories through Octo-SPI, HSPI, and the FMC. A dedicated 16-Kbyte data cache (DCACHE2) is placed in front of the GPU2D (on the M0 port) in order to cache data fetched from external memories with high-access latencies. The DCACHE2 is used exclusively by the GPU2D, and caches read transactions only. The DCACHE2 is similar to the DCACHE1 that is attached to the Cortex®-M33 CPU. The same software driver (stm32u5xx_hal_dcache) can operate the DCACHE1 and DCACHE2.

The HSPI controller operates an external 16-bit PSRAM memory, whereas external octal flash memory modules can be attached to the OCTOSPI1/2 controllers (HyperFlash™ and HyperRAM™ memories are also supported). The unified HAL XSPI driver, which is part of the STM32Cube MCU Package for the STM32U5 series (STM32CubeU5), drives the HSPI and Octo-SPI memory controllers. These memories are then memory mapped into the system and made accessible to the software application and other peripherals on the platform.

Both the GPU2D and the DCACHE2 are clocked at the same clock rate: hclk system clock.

4.2 NemaGFX/NemaVG API

At software level, the GPU2D is exclusively operated using the NemaGFX/NemaVG library. NemaGFX/NemaVG acts as a device driver, and as the API interface towards middleware and applications that want to leverage the graphic hardware acceleration.

The NemaGFX/NemaVG library is provided as a precompiled library to customers through the NeoChromSDK and X-CUBE-TOUCHGFX packages.

4.3 GPU2D initialization

The following code snippet initializes the GPU2D:

```
/* Enable GPU2D */
__HAL_RCC_GPU2D_CLK_ENABLE();

NVIC_SetPriority(GPU2D_IRQn, 5);
NVIC_EnableIRQ(GPU2D_IRQn);
```

Once the DCACHE2 is enabled, it must be invalidated before actual use, as shown in Section 4.10.

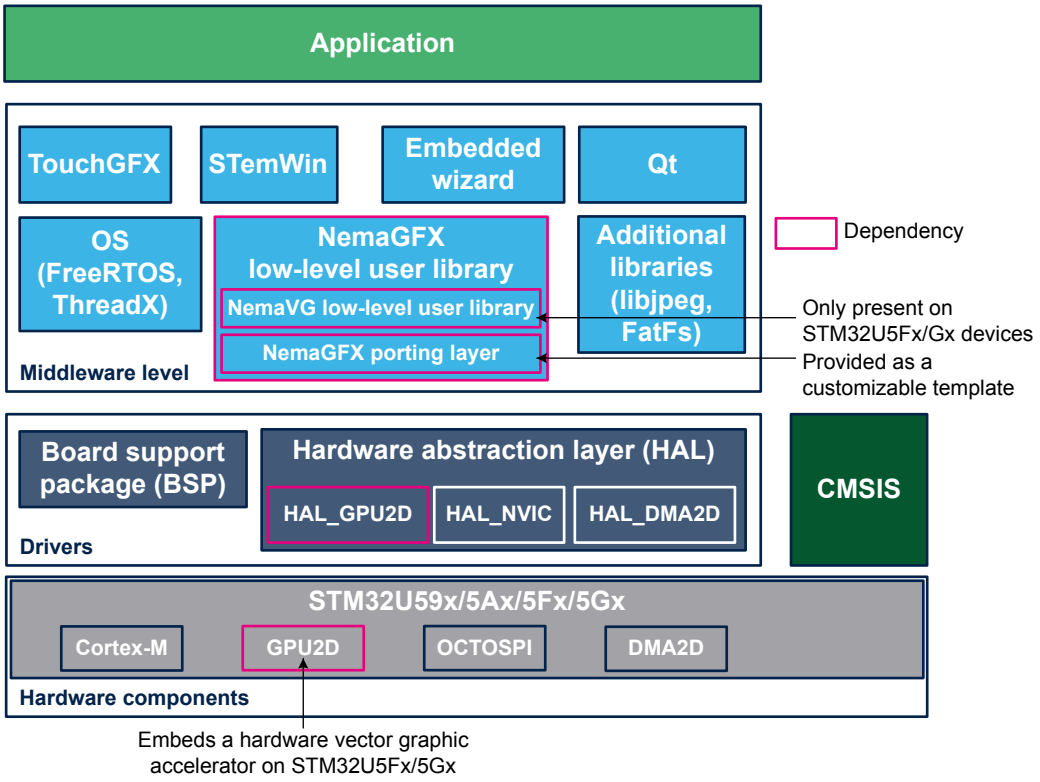
The application prepares command lists and source textures, and submits these to the GPU2D for execution. The command lists are attached to a master ring buffer (a singleton circular DMA buffer), which is also allocated by the application and initialized in the `nema_sys_init()` API.

The underlying buffers (nema_buffer_t objects) of the ring buffer and commands lists, when allocated (or explicitly placed using the linker script) must be 64-bit aligned in the system memory. The allocation of such buffers is described in the next section.

4.4 GPU2D platform integration

The figure below shows the software architecture that pertains to graphics applications.

Figure 2. GPU2D graphic software architecture



In terms of platform integration, the GPU2D is managed by two software components:

- the HAL_GPU2D module that handles the device initialization and interrupts servicing
- the NemaGFX porting layer that handles command lists, texture buffer allocations, and CPU / GPU2D synchronization
  The NemaGFX library expects the NemaGFX porting layer to provide the adequate implementation of the functions listed in the table below, for the correct operations on the STM32 target.

Table 23. NemaGFX porting layer functions

Function	Description

`int32_t nema_sys_init(void);`
`int nema_wait_irq(void);`
`int nema_wait_irq_cl(int cl_id);`
`int nema_wait_irq_brk(int brk_id);`
`uint32_t nema_reg_read(uint32_t reg);`
`void nema_reg_write(uint32_t reg, uint32_t value);`
	Device and interrupt management

`nema_buffer_t nema_buffer_create(int size);`
`nema_buffer_t nema_buffer_create_pool(int pool, int size);`
`void *nema_buffer_map(nema_buffer_t *bo);`
`void nema_buffer_unmap(nema_buffer_t *bo);`
`void nema_buffer_destroy(nema_buffer_t *bo);`
`uintptr_t nema_buffer_phys(nema_buffer_t *bo);`
`void nema_buffer_flush(nema_buffer_t *bo);`
`void nema_host_free(void *ptr);`
`void *nema_host_malloc(unsigned size);`
	GPU2D buffer management

`int nema_mutex_lock(int mutex_id);` `int nema_mutex_unlock(int mutex_id);`	Multi-threading synchronization

The NemaGFX porting layer implementation depends on the operating system (OS) used by the application, because the CPU/GPU2D synchronization uses the task synchronization primitives offered by this OS. The porting layer calls into the HAL_GPU2D module for operations such as register access and interrupt management. To ease the integration in STM32 platforms, the NeoChromSDK package provides templates of the NemaGFX porting layer (for example, for FreeRTOS™ or baremetal).

The code below is an implementation example of the NemaGFX porting layer, for a baremetal application configuration. It relies on the C heap for the buffer allocations (key exported functions highlighted in bold).

```
#include <nema_core.h>
#include <nema_sys_defs.h>

#include <stdlib.h>
#include <assert.h>

#define RING_SIZE 1024          /* Ring buffer size in byte */

static nema_ringbuffer_t ring_buffer_str = {{0}};

volatile static int last_cl_id = -1;
GPU2D_HandleTypeDef hgpu2d = { 0 };

#if (USE_HAL_GPU2D_REGISTER_CALLBACKS == 1)
static void GPU2D_CommandListCpltCallback(GPU2D_HandleTypeDef *hgpu2d, uint32_t CmdListID)
#else
void HAL_GPU2D_CommandListCpltCallback(GPU2D_HandleTypeDef *hgpu2d, uint32_t CmdListID)
#endif
{
    UNUSED(hgpu2d);
    last_cl_id = CmdListID;
}

int32_t nema_sys_init(void)
{
    /* Initialize the GPU2D device */
```

```
    hgpu2d.Instance = GPU2D;
    HAL_GPU2D_Init(&hgpu2d);

#if (USE_HAL_GPU2D_REGISTER_CALLBACKS == 1)
    HAL_GPU2D_RegisterCommandListCpltCallback(&hgpu2d, GPU2D_CommandListCpltCallback);
#endif

    /* Allocate ring buffer memory */
    ring_buffer_str.bo = nema_buffer_create(RING_SIZE);
    (void)nema_buffer_map(&ring_buffer_str.bo);

    /* Initialize the ring buffer */
    int ret = nema_rb_init(&ring_buffer_str, 1);
    if (ret < 0)
    {
        return ret;
    }

    /* Reset last_cl_id counter */
    last_cl_id = 0;

    return 0;
}
int nema_wait_irq(void)
{
    return 0;
}

int nema_wait_irq_cl(int cl_id)
{
    while (last_cl_id < cl_id)
    {
        (void)nema_wait_irq();
    }

    return 0;
}
int nema_wait_irq_brk(int brk_id)
{
    while (nema_reg_read(GPU2D_BREAKPOINT) == 0U) {
        (void)nema_wait_irq();
    }

    return 0;
}

uint32_t nema_reg_read(uint32_t reg)
{
    return HAL_GPU2D_ReadRegister(&hgpu2d, reg);
}

void nema_reg_write(uint32_t reg, uint32_t value)
{
    HAL_GPU2D_WriteRegister(&hgpu2d, reg, value);
}

nema_buffer_t nema_buffer_create(int size)
{
    nema_buffer_t bo;
    bo.base_virt = malloc(size);
    assert(bo.base_virt);
    bo.base_phys = (uint32_t)bo.base_virt;
    bo.size = size;
    bo.fd = 0;
    return bo;
}

nema_buffer_t nema_buffer_create_pool(int pool, int size)
{
    UNUSED(pool);
```

```
    return nema_buffer_create(size);
}

void *nema_buffer_map(nema_buffer_t *bo)
{
    return bo->base_virt;
}

void nema_buffer_unmap(nema_buffer_t* bo)
{
}

void nema_buffer_destroy(nema_buffer_t* bo)
{
    assert(bo->base_virt);
    free(bo->base_virt);

    bo->base_virt = (void*)0;
    bo->base_phys = 0;
    bo->size = 0;
    bo->fd = -1;
}

uintptr_t nema_buffer_phys(nema_buffer_t* bo)
{
    return bo->base_phys;
}
void nema_buffer_flush(nema_buffer_t* bo)
{
}

void nema_host_free(void* ptr)
{
    if (ptr)
    {
        free(ptr);
    }
}

void* nema_host_malloc(unsigned size)
{
    return malloc(size);
}

int nema_mutex_lock(int mutex_id)
{
    return 0;
}

int nema_mutex_unlock(int mutex_id)
{
    return 0;
}
```

The code below is the typical GPU2D_IRQHandler implementation for servicing GPU2D interrupts.

```
extern GPU2D_HandleTypeDef hgpu2d;

void GPU2D_IRQHandler(void)
{
    HAL_GPU2D_IRQHandler(&hgpu2d);
}
```

To ease the integration in STM32 platforms, the NeoChromSDK package provides some template implementations of the NemaGFX porting layer (such as for FreeRTOS™ and CMSIS).

4.5 TSi memory allocation (tsi_malloc)

The NemaGFX library comes with a custom memory allocator, called tsi_malloc. It is used with the GPU2D and its associated NemaGFX API to allocate buffers in RAM. This includes buffers for storing commands to be executed by the GPU2D. It also includes source textures and destination framebuffers or render targets that the GPU2D reads and writes into respectively. Thanks to the tsi_malloc, a portion of the system RAM can be dedicated to the GPU2D and application needs (at runtime). The tsi_malloc provides also a clear RAM partitioning versus the application requirements, thus improving the software design and the application maintainability.

The tsi_malloc provides a memory region in system RAM at initialization time: this constitutes a pool of memory from which buffers are allocated. The tsi_malloc allows the application to register up to eight memory pools: each pool resides in a particular memory (for example, one pool in the internal SRAM, a second pool in an external PSRAM, a third pool in an external SDRAM). The pools are represented by their integer IDs, starting with zero.

The NemaGFX library expects to have at least one pool declared, with pool ID = 0. This pool is used to allocate command buffers to store instructions for the GPU2D.

```
/* tsi_malloc_init_pool() for initializing a new memory pool with ID pool, physical
   address base_pyhs and size in bytes. On MCU systems, base_virt equals base_phys. */
int tsi_malloc_init_pool(int pool, void *base_virt, uintptr_t base_phys, int size, int reset)
;

/* tsi_malloc_pool() to allocate a buffer of size bytes from memory pool ID pool. */
void *tsi_malloc_pool(int pool, int size);

/* tsi_free() to free a buffer previously allocated by tsi_malloc_pool(). */
void tsi_free(void *ptr);
```

The NemaGFX porting layer can use the tsi_malloc for the graphic buffer allocation, and to free functions as shown in the code below (differences highlighted in bold).

```
#define POOL_ADDR 0x200D0000U
#define POOL_SIZE (2 * 832 * 1024) /* SRAM3 + SRAM5 slices */

int32_t nema_sys_init(void)
{
    /* Initialize the GPU2D device */
    hgpu2d.Instance = GPU2D;
    HAL_GPU2D_Init(&hgpu2d);

#if (USE_HAL_GPU2D_REGISTER_CALLBACKS == 1)
    HAL_GPU2D_RegisterCommandListCpltCallback(&hgpu2d, GPU2D_CommandListCpltCallback);
#endif

    /* register pool 0, located at address 0x200D0000 and of size 2 * 832 * 1024 bytes */
    tsi_malloc_init_pool(0, (void*)POOL_ADDR, POOL_ADDR, POOL_SIZE, 1);

    /* Allocate ring buffer memory */
    ring_buffer_str.bo = nema_buffer_create(RING_SIZE);
    (void)nema_buffer_map(&ring_buffer_str.bo);

    /* Initialize the ring buffer */
    int ret = nema_rb_init(&ring_buffer_str, 1);
    if (ret < 0)
    {
        return ret;
    }

    /* Reset last_cl_id counter */
    last_cl_id = 0;

    return 0;
}

nema_buffer_t nema_buffer_create(int size)
{
    nema_buffer_t bo = { 0 };
    bo.base_virt = tsi_malloc(size);
    assert(bo.base_virt);
    bo.base_phys = (uint32_t)bo.base_virt;
    bo.size = size;
```

```
    return bo;
}

nema_buffer_t nema_buffer_create_pool(int pool, int size)
{
    UNUSED(pool);
    return nema_buffer_create(size);
}

void nema_buffer_destroy(nema_buffer_t* bo)
{
    assert(bo->base_virt);
    tsi_free(bo->base_virt);

    bo->base_virt = (void*)0;
    bo->base_phys = 0;
    bo->size = 0;
}

void nema_host_free(void* ptr)
{
    if (ptr)
    {
        tsi_free(ptr);
    }
}

void* nema_host_malloc(unsigned size)
{
    return tsi_malloc(size);
}
```

4.6 Framebuffer memory allocation

The STM32U5x9 and STM32U5F7/5G7 microcontrollers operate with 16-, 24-, and 32-bit framebuffers, in RGB and ARGB pixel formats respectively. The application can allocate one or two framebuffers, depending on whether it wants to drive a single-buffered or double-buffered display. Such framebuffers must be allocated from the internal SRAM memory on the MCU. The application must reserve a dedicated part of the internal SRAM for the command lists and framebuffers.

The TSi memory allocator manages this memory.

```
#include <nema_core.h>

nema_buffer_t framebuffer_bo[2] = { { 0 } };

uint32_t stride = nema_stride_size(NEMA_BGR24, 0, 480);

/* allocate two 480x480 RGB24 Framebuffers from NEMA_MEM_POOL_FB pool */
framebuffer_bo[0] = nema_buffer_create_pool(NEMA_MEM_POOL_FB, stride * 480);
framebuffer_bo[1] = nema_buffer_create_pool(NEMA_MEM_POOL_FB, stride * 480);

nema_cmdlist_t cl = nema_cl_create();

/* make cl current */
nema_cl_bind(&cl);
nema_cl_rewind(&cl);

/* bind framebuffer_bo[0] as the destination buffer (GPU2D writing into it) */
nema_bind_dst_tex((uint32_t)framebuffer_bo[0].base_phys, 480, 480, NEMA_BGR24, stride);
/* set scissor to the entire buffer */
nema_set_clip(0, 0, 480, 480);

/* fill the entire buffer with red color */
nema_fill_rect(0, 0, 480, 480, nema_rgba(255, 0, 0, 255));

/* submit commands to GPU2D and wait for their completion */
nema_cl_submit(&cl);
nema_cl_wait(&cl);

/* put framebuffer_bo[0] onscreen */
swap_framebuffer();
```

4.7 Framebuffer configuration across GPU2D, LTDC, and DSI

Once allocated by the tsi_malloc, the framebuffer physical address can be accessed via the `nema_buffer_t::base_phys` field. This address is passed to the LTDC and the DSI HAL drivers to access the same framebuffer (for example, to scan it out or to transmit it over the DSI bus to the display module).

The following code snippets show how it is done.

```
#include <stm32u5xx_hal.h>

LTDC_HandleTypeDef LtdcHandle;
LTDC_LayerCfgTypeDef LayerCfg;

HAL_LTDC_Init(&LtdcHandle);

LayerCfg.WindowX0        = 0;
LayerCfg.WindowX1        = 480
LayerCfg.WindowY0        = 0;
LayerCfg.WindowY1        = 480;
LayerCfg.PixelFormat     = LTDC_PIXEL_FORMAT_RGB888;
LayerCfg.Alpha           = 0xFF;
LayerCfg.Alpha0          = 0;
LayerCfg.BlendingFactor1 = LTDC_BLENDING_FACTOR1_PAxCA;
LayerCfg.BlendingFactor2 = LTDC_BLENDING_FACTOR2_PAxCA;
LayerCfg.FBStartAdress   = framebuffer_bo[0].base_phys;
LayerCfg.ImageWidth      = 480;
LayerCfg.ImageHeight     = 480;
LayerCfg.Backcolor.Red   = 0;
LayerCfg.Backcolor.Green = 0;
LayerCfg.Backcolor.Blue  = 0;
LayerCfg.Backcolor.Reserved = 0xFF;

HAL_LTDC_ConfigLayer(&LtdcHandle, &LayerCfg, LTDC_LAYER_1);
```

4.8 Double-buffered display synchronization

In some applications, a double-buffered display allows the user to use a parallel display interface (DPI), or to achieve a higher and consistent frame-rate onscreen for example. In this case, the application allocates two framebuffers:

- a front-buffer sent to display (scanned out by LTDC), what is made visible onscreen
- a back-buffer composed by the GPU2D (and the DMA2D or the CPU), the next frame to be presented to the user

When the application finishes preparing the next frame, it swaps the front-buffer and back-buffer in a synchronized manner, usually on the “enter active area” event, as received from the display controller.

The code below shows how to swap the front- and back-buffers, using the LTDC line interrupt events and FreeRTOS™, in a double-buffered display configuration:

1. Call `init_tft_display()` to allocate the two framebuffers, the synchronization object, and to configure the LTDC.
2. Get into `app_main_loop()`, the main rendering loop whereby it renders a frame into the current back-buffer.
3. Call `swap_buffers()` to post it to display. This function blocks and only returns after the LTDC line event interrupt corresponding to the “enter active area” is fired.
4. The corresponding interrupt callback `HAL_LTDC_LineEventCallback` swaps the back- and front-buffer indices, so that the now-read back-buffer becomes the current front-buffer.
5. The LTDC scans out the current front-buffer, and becomes the new back-buffer, where the application renders to.

The cycle continues like that and within the `app_main_loop()` function.

```
// Display mode: 480x480p @ 60Hz
struct DisplayTimingsTypeDef timing = {
    .HSYNC = 2;
    .HBP   = 1;
    .HFP   = 1;
    .VSYNC = 1;
    .VBP   = 12;
    .VFP   = 50;
    .HACT  = 480;
    .VACT  = 481;
};

static nema_buffer_t framebuffer_bo[2] = { { 0 } };

/* FreeRTOS synchronization object */
static SemaphoreHandle_t vsync_sem = NULL;

static volatile uint32_t request_refresh;
static volatile uint32_t refreshing;

static volatile uint8_t cur_fb;     /* current back-buffer index */
static volatile uint8_t present_fb; /* current front-buffer index */

static void LTDC_Init(void)
{
    LTDC_LayerCfgTypeDef pLayerCfg;

    /* Configure and enable the LTDC */
    __HAL_LTDC_RESET_HANDLE_STATE(&hltdc);

    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     = timing.HSYNC - 1;
    hltdc.Init.AccumulatedHBP     = timing.HSYNC + timing.HBP - 1;
    hltdc.Init.AccumulatedActiveW = timing.HACT + timing.HBP + timing.HSYNC - 1;
    hltdc.Init.TotalWidth         = timing.HACT + timing.HBP + timing.HFP + timing.HSYNC - 1;
    hltdc.Init.VerticalSync       = timing.VSYNC - 1;
    hltdc.Init.AccumulatedVBP     = timing.VSYNC + timing.VBP - 1;
```

```
    hltdc.Init.AccumulatedActiveH = timing.VSYNC + timing.VACT + timing.VBP - 1;
    hltdc.Init.TotalHeigh         = timing.VSYNC + timing.VACT + timing.VBP + timing.VFP - 1;

    hltdc.Init.Backcolor.Red      = 0;
    hltdc.Init.Backcolor.Green    = 0;
    hltdc.Init.Backcolor.Blue     = 0;
    hltdc.Init.Backcolor.Reserved = 0xFF;

    HAL_LTDC_Init(&hltdc);

    /* LTDC layer configuration */
    pLayerCfg.WindowX0            = 0;
    pLayerCfg.WindowX1            = timing.HACT;
    pLayerCfg.WindowY0            = 0;
    pLayerCfg.WindowY1            = timing.VACT;
    pLayerCfg.PixelFormat         = LTDC_PIXEL_FORMAT_RGB565;
    pLayerCfg.Alpha               = 0xFF;
    pLayerCfg.Alpha0              = 0;
    pLayerCfg.BlendingFactor1     = LTDC_BLENDING_FACTOR1_PAxCA;
    pLayerCfg.BlendingFactor2     = LTDC_BLENDING_FACTOR2_PAxCA;
    pLayerCfg.FBStartAdress       = framebuffer_bo[0].base_phys;
    pLayerCfg.ImageWidth          = timing.HACT;
    pLayerCfg.ImageHeight         = timing.VACT;
    pLayerCfg.Backcolor.Red       = 0;
    pLayerCfg.Backcolor.Green     = 0;
    pLayerCfg.Backcolor.Blue      = 0;
    pLayerCfg.Backcolor.Reserved = 0xFF;

    HAL_LTDC_ConfigLayer(&hltdc, &pLayerCfg, 0);
}

static int lcd_int_active_line;
static int lcd_int_porch_line;

int init_tft_display(void)
{
    uint32_t stride = nema_stride_size(NEMA_BGR24, 0, 480);

    /* allocate two 480x480 RGB24 Framebuffers from NEMA_MEM_POOL_FB pool */
    framebuffer_bo[0] = nema_buffer_create_pool(NEMA_MEM_POOL_FB, stride * 480);
    framebuffer_bo[1] = nema_buffer_create_pool(NEMA_MEM_POOL_FB, stride * 480);

    LTDC_Init();

    vsync_sem = xSemaphoreCreateBinary();

    lcd_int_active_line = (LTDC->BPCR & 0x7FF) - 1;
    lcd_int_porch_line = (LTDC->AWCR & 0x7FF) - 1;

    /* set the line event position, enable line interrupts */
    LTDC->LIPCR = lcd_int_active_line;
    LTDC->IER |= LTDC_IER_LIE;
}

void HAL_LTDC_LineEventCallback(LTDC_HandleTypeDef* hltdc)
{
    if (LTDC->LIPCR == lcd_int_active_line)
    {
        /* configure line interrupt for next back porch */
        HAL_LTDC_ProgramLineEvent(hltdc, lcd_int_porch_line);

        if (request_refresh && !refreshing)
        {
            if (framebuffers_count == 2) /* when using a double-buffered display */
            {
                /* swap front and back buffers */
                present_fb = cur_fb;
                cur_fb = (cur_fb + 1) % 2;

                /* present the new front-buffer */
```

```
                LTDC_LAYER(hltdc, 0)->CFBAR = framebuffer_bo[present_fb].base_phys;
                __HAL_LTDC_RELOAD_IMMEDIATE_CONFIG(hltdc);

                /* signal the new back-buffer is now available */
                {
                    portBASE_TYPE px = pdFALSE;
                    xSemaphoreGiveFromISR(vsync_sem, &px);
                    portEND_SWITCHING_ISR(px);
                }
            }

            request_refresh = 0;
            refreshing = 1;
        }
    }
    else
    {
        /* configure line interrupt for next active area */
        HAL_LTDC_ProgramLineEvent(hltdc, lcd_int_active_line);

        if (refreshing)
        {
            refreshing = 0;

            if (framebuffers_count == 1) { /* when using a single-buffered display */
                portBASE_TYPE px = pdFALSE;
                xSemaphoreGiveFromISR(vsync_sem, &px);
                portEND_SWITCHING_ISR(px);
            }
        }
    }
}

void swap_buffers(void)
{
    /* request a refresh */
    request_refresh = 1;

    /* wait for vsync before returning */
    xSemaphoreTake(vsync_sem, portMAX_DELAY);
}

nema_buffer_t *get_current_framebuffer(void)
{
    return &framebuffer_bo[cur_fb];
}

void app_main_loop(void)
{
    while (app_running)
    {
        nema_buffer_t *bo = get_current_framebuffer();

        render_frame(bo); /* render a frame into the current back-buffer */

        swap_buffers(); /* request a display refresh + swap front and back buffers */
    }
}
```

4.9 GPU2D external cache

The GPU2D external cache is connected on the GPU2D M0 port, which serves for reading texture data. These textures are usually fetched from external memories. Having the cache in place reduces then the pressure on the external memories, and improves the graphic performance when used adequately. The GPU2D external cache size is 16 Kbytes. It is disabled by default (after reset/boot), and must be explicitly enabled by the application to benefit from it.

4.9.1 GPU2D external cache initialization

The code snippet below enables the external cache.

```
/* Enable GPU2D DCACHE */
__HAL_RCC_DCACHE2_CLK_ENABLE();

hgcache.Instance = DCACHE2;
hgcache.Init.ReadBurstType = DCACHE_READ_BURST_INCR;

HAL_DCACHE_Init(&hgcache);
HAL_DCACHE_Enable(&hgcache);
HAL_DCACHE_Invalidate(&hgcache);

SYSCFG->CFGR1 &= ~(1L << 28);
```

The first time the external cache is enabled, it is recommended to also invalidate it before use.

4.9.2 GPU2D external cache invalidation

With the cache enabled, when the application updates a graphic buffer with a new content (buffer, which has been previously accessed by the GPU2D, so potentially cached), the application needs to invalidate the external cache: this allows the GPU2D to pick up the recent data from this buffer. The application must follow this process as the GPU2D does not know the data state in the texture buffers.

The `HAL_DCACHE_Invalidate()` API from the DCACHE HAL driver invalidates the external cache.

```
HAL_DCACHE_Invalidate(&hgcache);
```

4.9.3 GPU2D external cache and internal SRAM access

The external cache is mainly designed to optimize the access time for textures located in external memories. On the other hand, accessing graphic buffers located in an internal SRAM is very fast, and caching these types of access does not bring any further value.

STM32U5x9 and STM32U5F7/5G7 devices propose the following option in SYSCFG registers to disable caching access to buffers located in an internal SRAM.

```
SYSCFG->CFGR1 &= ~(1L << 28);
```

The cache must be disabled for the internal SRAM buffers (for example, vector graphic applications that require an intermediary stencil buffer stored in an internal SRAM).

The cache monitors help the software developer to direct the efforts to improve the application graphic rendering performance. These counters are exposed through the HAL DCACHE driver, via the APIs listed below.

```
HAL_DCACHE_Monitor_Start(&hgcache, DCACHE_MONITOR_READ_HIT| DCACHE_MONITOR_READ_MISS);
HAL_DCACHE_Monitor_Reset(&hgcache, DCACHE_MONITOR_READ_HIT| DCACHE_MONITOR_READ_MISS);

uint32_t hit = HAL_DCACHE_Monitor_GetReadHitValue(&hgcache);
uint32_t miss = HAL_DCACHE_Monitor_GetReadMissValue(&hgcache);
```

The application developer optimizes then the texture use, and maximizes the cache hit ratio throughout the rendering routine.

4.10 GPU2D tiled access to textures

The GPU2D is able to access source textures in a tiled fashion (versus linear access). This mode offers an opportunity for the GPU2D to cache neighboring texels internally. These texels can then be reused (since improving locality) within a rendering operation involving a transformation (such as rotation and perspective projection). The software user code has to call the `nema_enable_tiling` API to enable tiled access.

```
nema_cl_bind(&cl);
nema_cl_rewind(&cl);

nema_bind_dst_tex((uintptr_t)fbo->bo.base_phys, 454, 454, NEMA_BGR24, 3 * 454);
nema_set_blend_blit(NEMA_BL_SRC_OVER);
nema_set_clip(0, 0, 454, 454);

nema_bind_src_tex((uintptr_t)Compass_454x454, 454, 454, NEMA_BGRA8888, 454 * 4, NEMA_FILTER_B
L);

nema_enable_tiling(1);

nema_blit_quad_fit(x1, y1, x2, y2, x3, y3, x4, y4);

nema_cl_submit(&comp_cl);
nema_cl_wait(&comp_cl);
```

4.11 GPU2D interrupts

The GPU2D has two interrupt lines connected to the NVIC:

- gpu2d_irq, used to inform the host CPU about command-list completion events
  When the bit [1] is set to one in the NEMA_INTERRUPT register (at offset 0x00F8), the interrupt signals that a command list has been entirely executed. The NEMA_CLID register contains the 32-bit identifier of this command list. The software application has to clear the bit [1] in NEMA_INTERRUPT and in the respective GPU2D_IRQ interrupt handler, before continuing.
- gpu2d_er_irq, used to raise errors observed at GPU level or at interconnect level (such as relayed by the memory controllers)
  This interrupt is issued to signal system (bus) errors. For example, when the GPU2D tries to access an external memory through the FMC or the OCTOSPI. The bit [0] in the NEMA_SYS_INTERRUPT register (at offset 0x0FF8) indicates whether a bus error has been observed.
  The bits [10:7] in NEMA_SYS_INTERRUPT indicate the source of the error as follows.

  ```
  1000: AHB Slave Port
  0100: AHB M0 Master Port
  0010: AHB M1 Master Port
  ```

  Bus errors are usually considered fatal and non-recoverable. A gpu2d_er_irq interrupt informs the application that such a condition has been observed. The default IRQ handler for gpu2d_er_irq is an infinite loop, which halts the execution.
  The gpu2d_er_irq interrupt line also signals events from the general-purpose lines, detailed in the next section.

4.12 GPU2D general-purpose flags

The GPU2D exposes four general purpose flags that are connected (depending on the device architecture) to the CPU and to other peripherals present on the STM32 microcontroller. These flags are used to synchronize operations between the GPU2D and the peripherals, without intervention of the CPU and software (without incurring overhead).

On the STM32U5x9 devices, the general-purpose flags are connected to the GPDMA and the CPU.

The flags can be individually asserted or deasserted, dynamically using instructions emitted within a command list. This is done via the dedicated `nema_ext_hold` API of the NemaGFX library.

A gpu2d_er_irq interrupt can be associated to a given general-purpose line, and triggered by the GPU2D when this line is set high. The individual bits[3:0] in the NEMA_SYS_INTERRUPT register indicate which general-purpose flag was set, and act accordingly.

```
Bit 0: Indicates that IRQ_SYSERROR due to GP_FLAG line 0.
Bit 1: Indicates that IRQ_SYSERROR due to GP_FLAG line 1.
Bit 2: Indicates that IRQ_SYSERROR due to GP_FLAG line 2.
Bit 3: Indicates that IRQ_SYSERROR due to GP_FLAG line 3.
```

4.12.1 Using the general-purpose flags for CPU and GPU2D synchronization

This section details how to use the general-purpose flags to trigger a processing on the CPU when the GPU2D comes across at a specific point while executing instructions from the command list. The processing to trigger, in this example, is invalidating the external GPU cache.

In the example below, the trigger processing invalidates the external GPU cache. The application renders a texture-mapped rectangle, updates the texture with new content, then redraws the rectangle again. Because the texture content changes, the application needs to invalidate the GPU2D cache before drawing the rectangle the second time.

The general-purpose line events are signaled through the gpu2d_er_irq to the host CPU. The IRQ has to be enabled at application startup with the following code.

```
/* Enable GPU2D IRQs */
    NVIC_SetPriority(GPU2D_IRQn, 5);
    NVIC_EnableIRQ(GPU2D_IRQn);

    NVIC_SetPriority(GPU2D_ER_IRQn, 5);
    NVIC_EnableIRQ(GPU2D_ER_IRQn);

    /* Enable GPU2D DCACHE */
    __HAL_RCC_DCACHE2_CLK_ENABLE();

    hgcache.Instance = DCACHE2;
    hgcache.Init.ReadBurstType = DCACHE_READ_BURST_INCR;

    HAL_DCACHE_Init(&hgcache);
    HAL_DCACHE_Enable(&hgcache);
    HAL_DCACHE_Invalidate(&hgcache);
```

The gpu2d_er_irq interrupt handler must similarly be implemented, forwarding the IRQ to the HAL GPU2D driver to be handled, as shown below.

```
void GPU2D_ER_IRQHandler(void)
{
    HAL_GPU2D_ER_IRQHandler(&hgpu2d);
}
```

`HAL_GPU2D_ER_IRQHandler` calls the `HAL_GPU2D_ErrorCallback` function, which is a weak symbol that can be overridden and implemented alternatively by the application. The specialized processing that is triggered by the GPU2D, at the synchronization points within the command list, has to be implemented in a custom `HAL_GPU2D_ErrorCallback` handler as listed below.

```
void HAL_GPU2D_ErrorCallback(GPU2D_HandleTypeDef *hgpu2d)
{
    uint32_t val = nema_reg_read(GPU2D_SYS_INTERRUPT);

    HAL_DCACHE_Invalidate(&hgcache); /* action to perform on sync points */
    nema_ext_hold_deassert_imm(0); /* immediately deassert gp line 0 */

    nema_reg_write(GPU2D_SYS_INTERRUPT, val); /* clear the ER interrupt */
}
```

`HAL_GPU2D_ErrorCallback` starts by reading the content of the GPU2D_SYS_INTERRUPT register, holding status bits about the origin of the system interrupt (a general-purpose line or a bus error). It then calls to `HAL_DCACHE_Invalidate` (action defined to perform on a sync point), followed by a call to `nema_ext_hold_deassert_imm(0)` to immediately reset the general-purpose line #0 to low. `nema_reg_write(GPU2D_SYS_INTERRUPT, val)` is finally called to clear the gpu2d_er_irq interrupt (otherwise it stays high).

On the synchronization points, these are special instructions emitted within the command list, alongside the rendering instructions. When these special instructions are executed by the GPU2D, the execution is suspended (hold), and the gpu2d_er_irq interrupt is triggered (and thus the `HAL_GPU2D_ErrorCallback` execution).

These special instructions are listed in the code below.

```
nema_ext_hold_enable(0); /* enable gp line 0 */
nema_ext_hold_irq_enable(0); /* enable SYS IRQ generation associated with gp line 0 */

nema_cmdlist_t cl = nema_cl_create();

nema_cl_bind(&cl);
nema_cl_rewind(&cl);

nema_bind_dst_tex((uint32_t)fb.bo.base_phys, 320, 240, NEMA_RGB565, 320 * 2);
nema_set_clip(0, 0, 320, 240);
nema_clear(0xff000000);

nema_bind_src_tex((uint32_t)texture.bo.base_phys, 32, 32, NEMA_BGR24, 32 * 3, NEMA_FILTER_PS)
;

nema_blit_rect_fit(0, 0, 320, 240); /* first draw */

nema_ext_hold_assert(0, 1); /* use general-purpose line 0, GPU2D stops execution once hit */

nema_blit_rect_fit(0, 0, 320, 240); /* second draw */

nema_cl_submit(&cl); /* actual GPU2D execution starts here */
nema_cl_wait(&cl); /* wait for all instructions to complete */
```

The `nema_ext_hold_assert(0, 1)` statement causes a hold instruction emission in the currently bound command list:

- The first argument specifies that line 0 is used to signal the hold condition.
- The second argument indicates that the GPU2D execution is suspended once the hold instruction is encountered.

Important: All NemaGFX API calls in the sequence construct an instruction buffer. These instructions are executed only when the software submits this buffer for execution (via the `nema_cl_submit` API).

4.13 TouchGFX and STM32CubeMX support

The STM32CubeMX version 6.5.0 and X-CUBE-TOUCHGFX version 4.19.0 introduce support for STM32U5x9 and STM32U5F7/5G7 devices, including configuration and graphic hardware acceleration using the GPU2D. Refer to STM32 Graphical User Interface.

Revision history

Table 24. **Document revision history**

Date	Version	Changes
20-Apr-2021	0.1	Initial draft release.
23- Nov-2022	0.2	Updated: • Introduction • Section 2 Memories • Section 3 Graphic resources • New Section 4 Neo-Chrom software integration • Section 4.6 Framebuffer memory allocation • Section 4.8 Double-buffered display synchronization
16-Dec-2022	1	Updated Figure 2. GPU2D graphic software architecture Generated a public version of the document
18-Sep-2023	2	Updated: • Title • Section Introduction • Section 1 STM32U59x/Ax/5Fx/5Gx overview • Section 2 Memories • Section 3 Graphic resources and all subsections • Section 4.1 GPU2D and DCACHE2 • Section 4.2 NemaGFX/NemaVG API • Figure 2. GPU2D graphic software architecture • Section 4.9.3 GPU2D external cache and internal SRAM access • Section 4.13 TouchGFX and STM32CubeMX support Added Section 3.5 JPEG codec

Contents

List of tables

List of figures

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice. Purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgment.

Purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2023 STMicroelectronics – All rights reserved