Developing high-performance graphics with low-power optimization on PSOC™ Edge MCU

About this document

Scope and purpose

This application note guides developers in designing advanced, high-performance GPU graphics applications with a focus on low-power optimization for the PSOC™ Edge MCU. Developers will learn how to configure the graphics subsystem, balance power and performance, and apply proven optimization techniques. The document explains how to measure and reproduce performance metrics, and provides practical API references and code snippets tailored to the target audience. The practical guidance and implementation-focused insights covered in this application note enables developers to create graphics solutions that are both powerful and energy-efficient for modern embedded applications.

Intended audience

This application note is tailored for hardware and software engineers, and system architects who are involved in the design and development of graphical applications using the PSOC™ Edge MCU.

Table of contents

1 PSOC™ Edge graphics subsystem overview

PSOC™ Edge MCU comprises a high-performance and low-power graphics subsystem to enable human machine interface (HMI)-based application development.

The graphics subsystem of the PSOC™ Edge E84 MCU consists of the following three blocks:

1. 2.5D graphics processing unit (GPU)
2. Display controller (DC)
3. Mobile Industry Processor Interface Display Serial Interface (MIPI DSI) host controller

The following Figure 1 illustrates the architecture of the graphics subsystem, showcasing its various blocks and their arrangement.

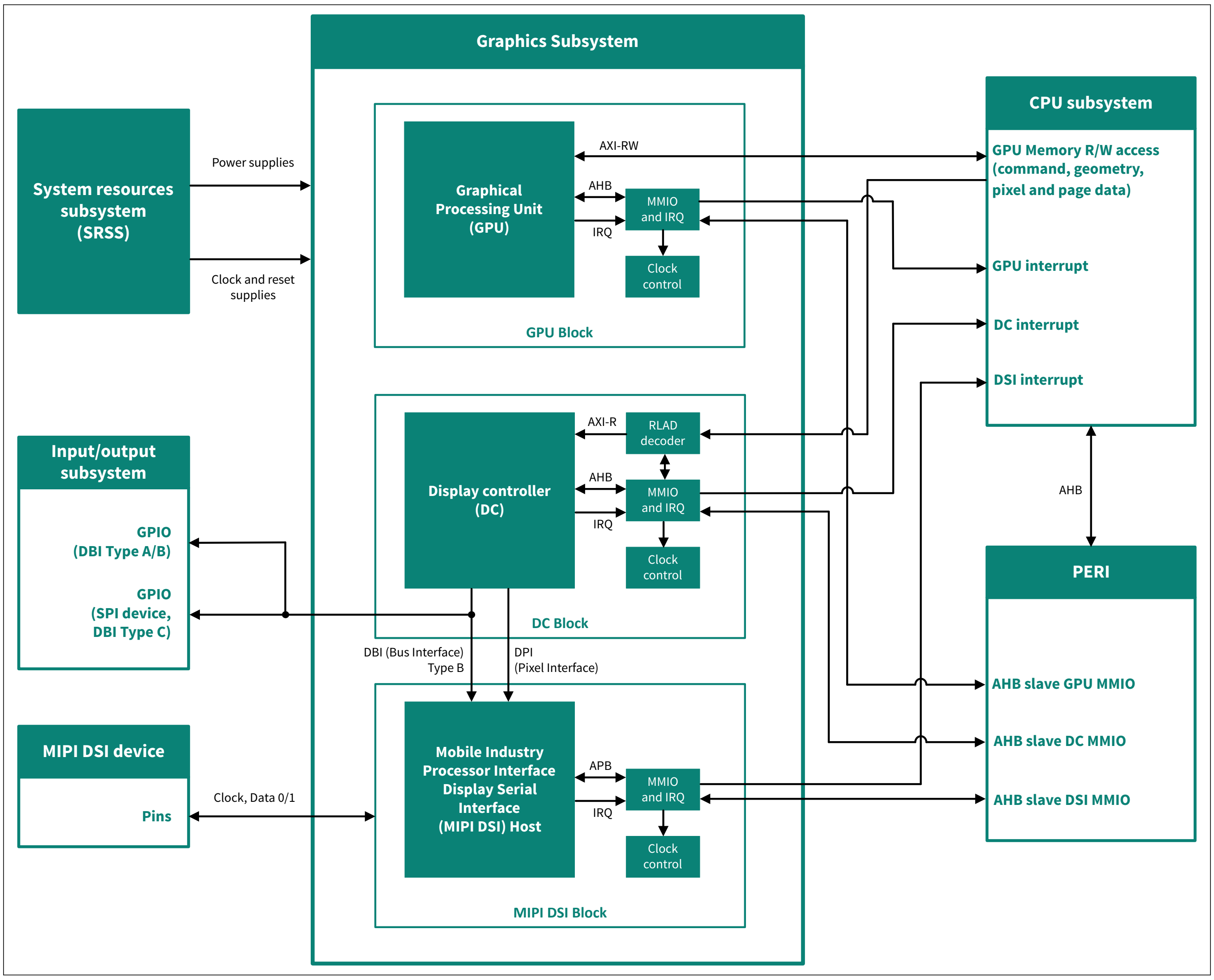


Figure 1 Graphics subsystem architecture

2 System architecture

The PSOC™ Edge MCU is a dual-CPU microcontroller featuring a Cortex®-M33 and a Cortex®-M55 core, with internal volatile memories including 1 MB of internal SRAM, ITCM, and DTCM on the CM55 side, and 5 MB of system SRAM shared between both cores.

This application note uses the smartwatch code example running on the PSOC™ Edge E84 Evaluation Kit (KIT_PSE84_EVAL) to demonstrate end-to-end graphics rendering on MIPI DSI display panels and to evaluate the graphics subsystem's performance and power consumption. The document focuses on two representative targets: a 1.43-inch, 466×466 command-mode wearable panel driven by CO5300 display controller and a Waveshare 4.3-inch, 800×480 video-mode panel. Configuration steps, timing/clocking, and optimization techniques are presented in the context of these displays, providing repeatable measurements and practical guidance for both high-performance and low-power graphics use cases. For more information on these display panels, please refer to the smartwatch code example readme.

The KIT_PSE84_EVAL offers PSRAM, quad and octal NOR flashes connected to PSOC™ Edge MCU via SMIF interface, as shown in block diagram (Figure 2). Additionally, the KIT_PSE84_EVAL board can be connected to different DSI display panels.

The 16 MB S70KS1283 PSRAM connected via SMIF 1 interface features an octal SPI interface that supports a maximum of 200 MHz DDR access. It can be utilized for frame buffers but is primarily used for data storage, particularly for large and infrequently accessed data.

Both 16 MB S25FS128S QSPI and 128 MB S28HS01GT OSPI NOR flashes, connected via SMIF 0 interface, are used for code execution and persistent data storage.

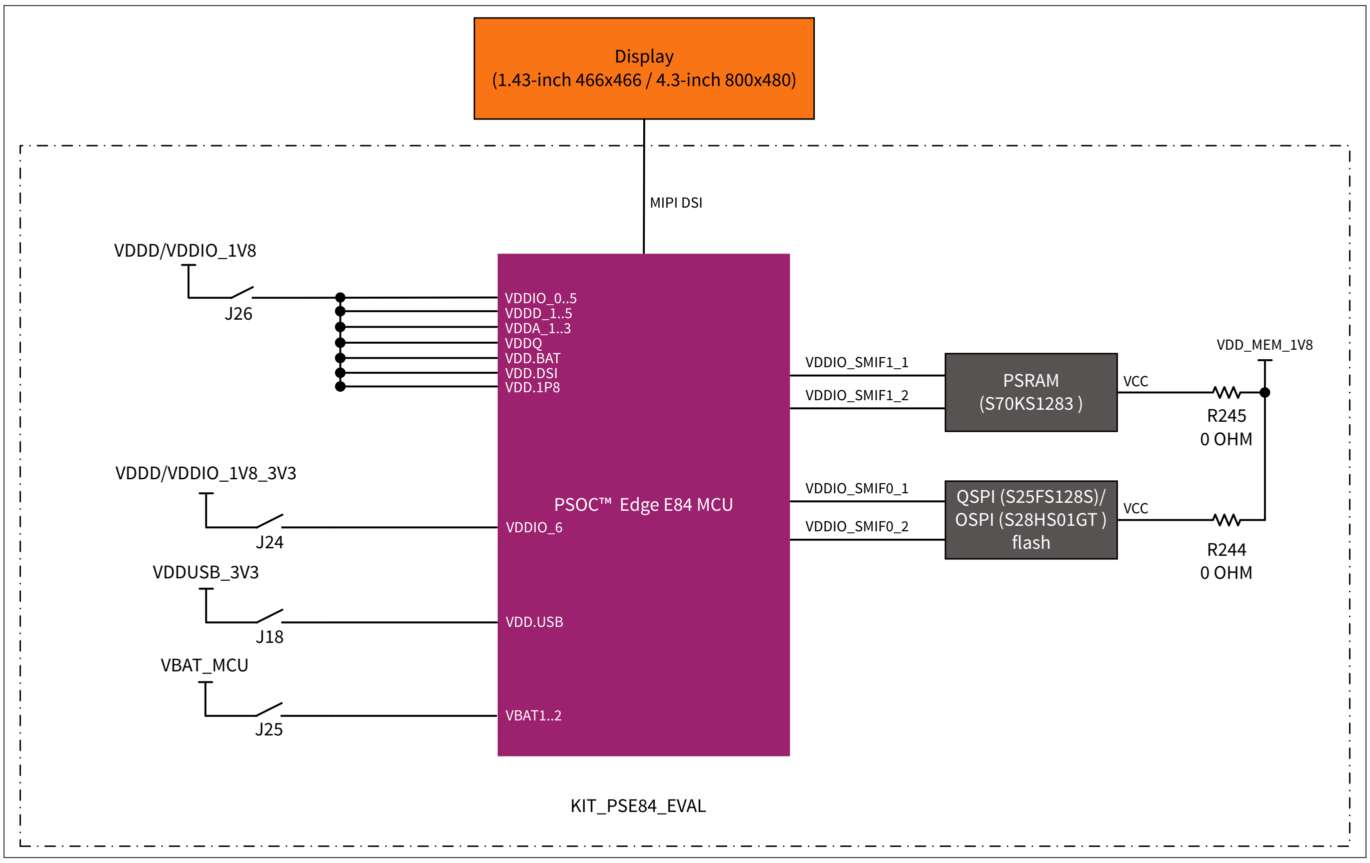


Figure 2 **KIT_PSE84_EVAL block diagram (on-board memory and display)**

***Note**:* *At any given time, either QSPI or OSPI flash can be utilized as they share the same SMIF 0 interface with the PSOC™ Edge MCU.*

3 Creating a graphical application with PSOC™ Edge MCU

This section outlines the steps required before and during the development of a graphical application using the PSOC™ Edge MCU. First, identify the application's requirements, then verify whether the chosen display is compatible with the hardware configuration.

3.1 Identifying graphical application requirements

Identifying the requirements of the graphical application is an essential first step. Before development, establish the key parameters such as display resolution, color depth, and the type of data to be displayed (static images, text, or animation). After defining these fundamental parameters, assess the graphical hardware architecture of the application and the necessary hardware resources by considering the following:

- Is external memory needed for the framebuffer like PSRAM?
- Is the external memory interface bus width sufficient?
- Is the DC interface set to RGB565 or ARGB8888 depending on the target application requirement and display module?
- Is external memory needed to store graphic primitives like QSPI or OSPI flash?
- Is the display type aligned with supported display interface standards such as MIPI DBI, MIPI DPI, and MIPI DSI?
- Is the required FPS within the bandwidth supported by the display panel?

3.2 Matching display resolution and color depth to hardware configuration

When kicking off an PSOC™ Edge based graphics project, the target display resolution and color depth are often predetermined. The critical checkpoint is verifying that these choices are practically feasible on the intended hardware platform. To make that determination:

1. Calculate the framebuffer footprint and decide where it will reside in memory
2. Ensure the display’s refresh rate and resolution match the available framebuffer memory bandwidth
3. Validate the compatibility of the display panel interface with the MIPI DSI host and the display controller (DC)

The following sections break down each points and present recommended actions and examples to support application.

3.2.1 Framebuffer memory size requirements and location

Determining the framebuffer memory size and location is crucial for display compatibility. The memory space needed for the framebuffer must be contiguous and is calculated by multiplying the display's resolution (width x height) by its color depth (bits per pixel), then divided by 8 to convert to bytes. For example:

Resolution: 800 x 480 pixels

Color depth: 16 bpp (which is 16 bits)

Memory space required: (800 x 480 x 16) / 8 = 768,000 bytes

It is not necessary for the framebuffer color depth to match the display color depth; for example, an RGB888 display can use an RGB565 framebuffer.

***Note**:* *The required framebuffer size is doubled for double-buffer configurations, commonly used in graphics applications where one buffer holds the current image while the other prepares the next.*

Depending on the required framebuffer size, the framebuffer can be located either in an internal SRAM (SRAM, System SRAM) or in an external memory such as PSRAM connected to the SMIF interface.

Using internal system SRAM instead of an external memory offers several advantages:

- Using an internal system SRAM as a framebuffer allows the maximum performances and avoids any bandwidth limitation issues for the GPU and DC
- Significantly higher throughput with zero wait-state access
- Reduces pin counts and lowers BOM cost when internal memory is adequate, eliminating the need for external memory

The only limitation of the internal system SRAM is its size, especially when it is shared among other application functions, such as for Machine Learning (ML) use cases. When the framebuffer size exceeds available memory, the external memory (via the SMIF interface) must be utilized. However, users should consider the potential bandwidth limitations with external memories. Refer to AN239774 for more information on the selection and configuration of the various available memories on the PSOC™ Edge MCU.

3.2.2 Verifying display compatibility based on memory bandwidth requirements

This section outlines key bandwidth aspects and explains how to determine the bandwidth requirements based on the pixel clock, GPU, and DC. It also presents a straightforward method to assess whether a desired display size is compatible with a specific hardware configuration.

- **Memory bandwidth aspect**

  To determine if the selected graphics memory bandwidth can support the GPU and DC required bandwidth, account for any concurrent memory accesses. Generally, a small framebuffer and GPU heap in internal system SRAM do not demand high bandwidth due to its low pixel clock, resulting in lower GPU and DC required bandwidth. A more complex scenario arises when the framebuffer and even GPU heap is in external memory
- **Memory bus concurrency**
  - DC, GPU, and CPU masters

    In a typical graphic application that uses an external memory for graphics memory, two or three main AHB masters concurrently use the same memory. The GPU or CPU updates the next image to be displayed while the DC fetches and renders the graphics frame. The memory bus load depends mainly on the GPU and DC required bandwidth.
  - Other AXI/AHB masters

    External memory is often shared by multiple masters, not just those for graphics. This concurrency can create a heavy bus load, impacting graphic performance
- **External PSRAM memory bus width**

  When locating the framebuffer and GPU memory in external PSRAM, consider that the external memory's running frequency is approximately half of the system frequency. As a result, the memory bandwidth is often regarded as the bottleneck of the entire graphics system. Concurrent access by multiple masters to the same external memory results in increased latency and affects its throughput

3.2.3 Verifying display interface compatibility

After addressing display resolution, color depth, and memory, select the display panel by aligning PSOC™ Edge's interfaces (MIPI DBI A/B/C, MIPI DPI, MIPI DSI) with UI dynamics, bandwidth, memory, pin budget, and power targets. Generally, choose DBI for low-power partial updates using the panel's GRAM, DPI for continuous pixel streaming with deterministic timing, and DSI for higher bandwidth over fewer pins, using command-mode for sparse updates or video-mode for fluid animation.

Here are the key points to consider when finalizing the display based on the available display interfaces from PSOC™ Edge.

- MIPI DBI A/B (parallel 68xx/8080) and DBI-C (serial SPI/QSPI)
  - Best for small/medium resolutions, static UIs, and partial updates; uses panel GRAM
  - Low pin count (DBI-C) or moderate (A/B); lowest average power (link idles)
  - Bandwidth-limited for full-screen 60 Hz; rely on Tearing Effect (TE) for tear-free partial refresh
  - Verify pixel format, readback, alignment, and command set
- MIPI DPI (parallel RGB)
  - Continuous video stream with strict HSYNC/VSYNC/DE/pixel-clock timing; predictable latency
  - High pin count and EMI; needs full-frame buffers and DMA/display controller
  - Validate porch/sync timing, pixel clock range, color coding, and drive levels
- MIPI DSI
  - High bandwidth with few lanes; supports command-mode and video-mode
  - Command-mode: partial updates, lower average power
  - Video-mode: smooth 60+ Hz motion; requires sustained memory bandwidth and buffering
  - Check lane count/rate headroom, D-PHY routing, DCS/initialization, TE, pixel formats, and power sequencing

4 Graphics subsystem and display configuration

At this stage of the graphical application development, with the application requirements already determined, configure the graphics subsystem of PSOC™ Edge to refine the application. See AN239191 Getting started with graphics on PSOC™ Edge MCU as a quick-start guide to develop graphics application on the PSOC™ Edge MCU.

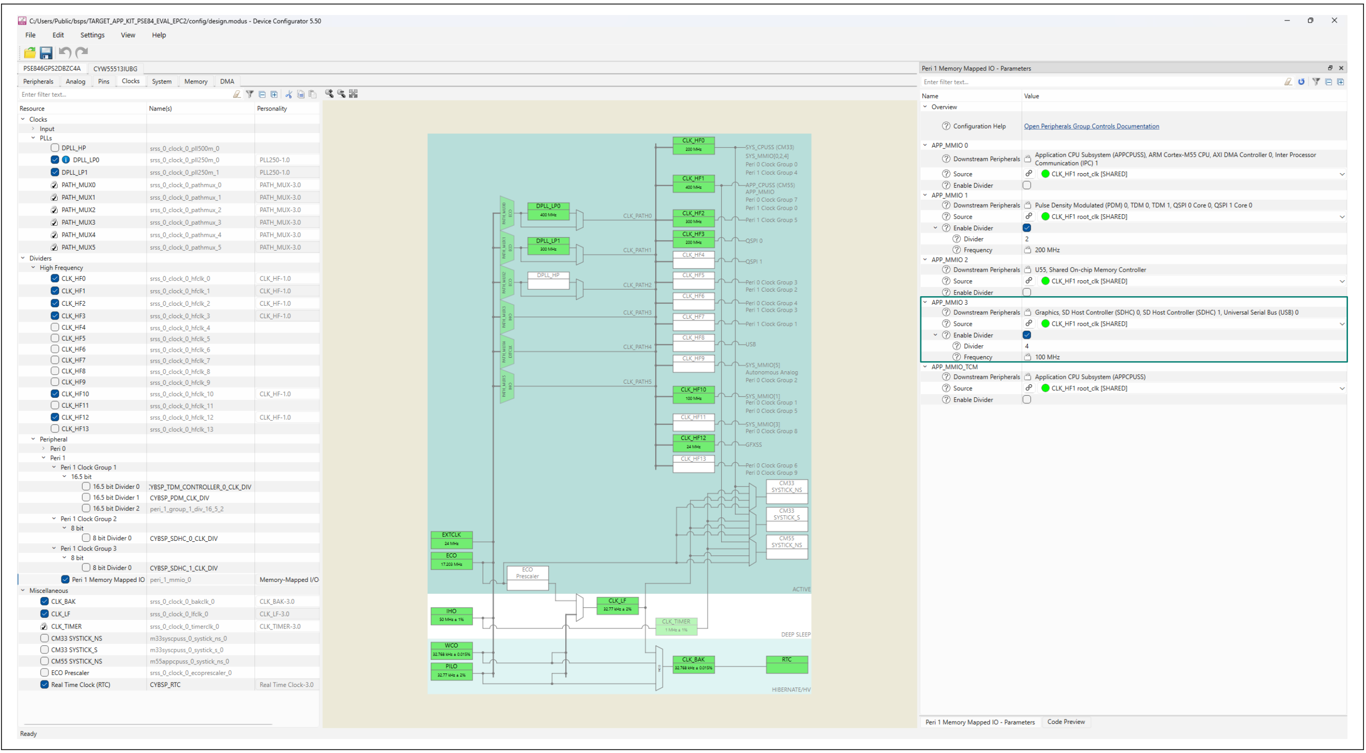
4.1 Clock configuration

The graphics subsystem has the following asynchronous clock domains:

- System clock and high frequency main clock
- MIPI DSI host clock

4.1.1 System and high frequency main clocks

The system clock (clk_sys) supplies inputs for the GPU system clock (clk_sys_gpu), DC (clk_sys_dc), and MIPI DSI (clk_sys_mipidsi), facilitating their configuration through MMIO and interrupt ports for each of the three aforementioned sub components. These clocks are configured with a 1:N ratio to the system's peripheral clock, up to a maximum of 100 MHz, as shown below in the Figure 3.



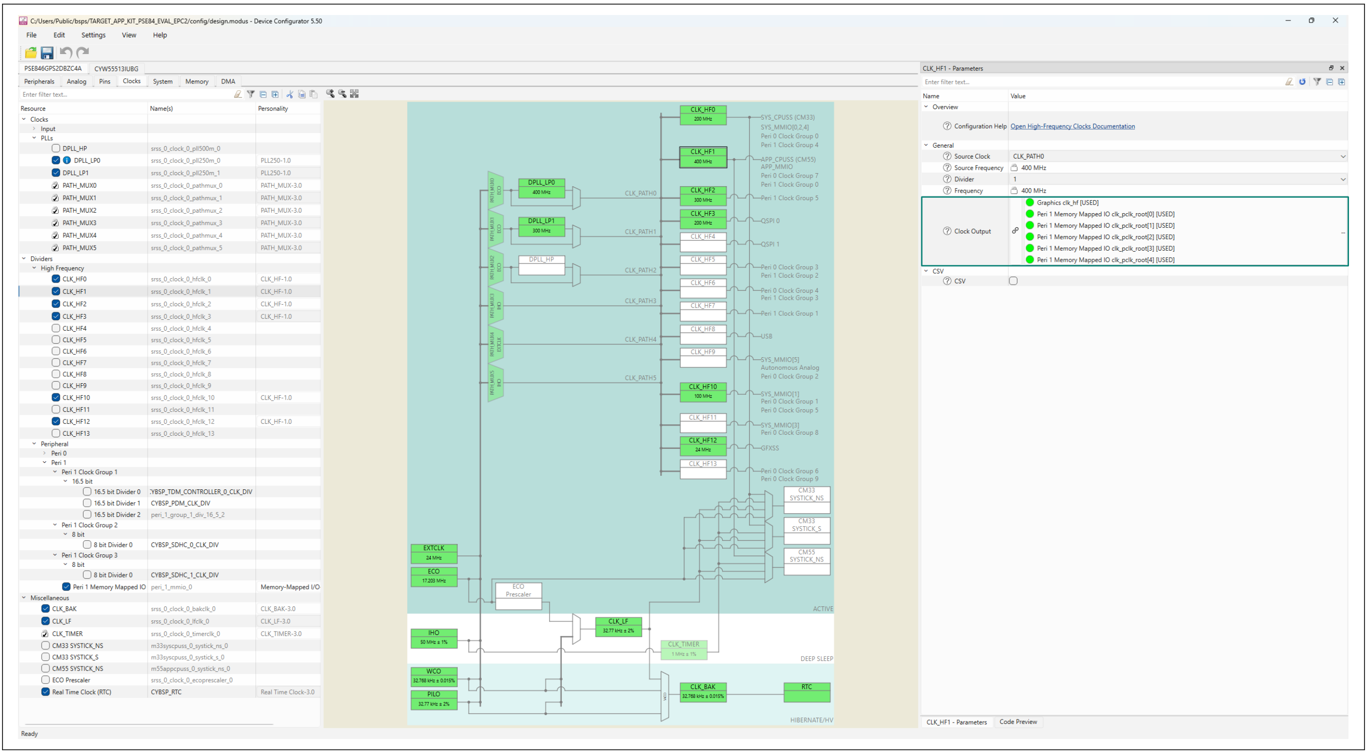
Peri 1 Memory Mapped IO - Parameters

Name	Value
Overview	
Configuration Help	Open Peripherals Group Controls Documentation
APP_MMIO 0	
Downstream Peripherals	Application CPU Subsystem (APPCPUSS), ARM Cortex-M55 CPU, AXI DMA Controller 0, Inter Processor Communication (IPC) 1
Source	CLK_HF1 root_clk [SHARED]
Enable Divider	☐
APP_MMIO 1	
Downstream Peripherals	Pulse Density Modulated (PDM) 0, TDM 0, TDM 1, QSPI 0 Core 0, QSPI 1 Core 0
Source	CLK_HF1 root_clk [SHARED]
Enable Divider	✓
Divider	2
Frequency	200 MHz
APP_MMIO 2	
Downstream Peripherals	U55, Shared On-chip Memory Controller
Source	CLK_HF1 root_clk [SHARED]
Enable Divider	☐
APP_MMIO 3	
Downstream Peripherals	Graphics, SD Host Controller (SDHC) 0, SD Host Controller (SDHC) 1, Universal Serial Bus (USB) 0
Source	CLK_HF1 root_clk [SHARED]
Enable Divider	✓
Divider	4
Frequency	100 MHz
APP_MMIO_TCM	
Downstream Peripherals	Application CPU Subsystem (APPCPUSS)
Source	CLK_HF1 root_clk [SHARED]
Enable Divider	☐

Figure 3 **GFXSS system clock configuration**

High frequency main clock (clk_hf) sourced from CLK_HF1, provides the high-frequency timing reference for the AXI ports used by the GPU and DC. It sets the pace for memory transactions (framebuffer reads/writes, textures, layers) and feeds dividers that generate the functional clocks for GPU core, DC scan-out logic, and standard display interfaces (e.g., DSI/DPI/DBI). In short, CLK_HF determines throughput, latency, and timing determinism across the graphics pipeline, while its divided versions shape logic and I/O rates.

clk_hf can be configured with a minimum frequency of 100 MHz and a maximum frequency of 400 MHz, as illustrated in Figure 4. The minimum is not a strict limitation. Lower frequencies for clk_hf are permitted (e.g., for ULP operation); however, they will affect the maximum frequencies achievable on display interfaces, in addition to having a general impact on rendering performance.



CLK_HF1 - Parameters

Name	Value
Overview	
Configuration Help	Open High-Frequency Clocks Documentation
General	
Source Clock	CLK_PATH0
Source Frequency	400 MHz
Divider	1
Frequency	400 MHz
Clock Output	Graphics clk_hf [USED] Peri 1 Memory Mapped IO clk_pclk_root[0] [USED] Peri 1 Memory Mapped IO clk_pclk_root[1] [USED] Peri 1 Memory Mapped IO clk_pclk_root[2] [USED] Peri 1 Memory Mapped IO clk_pclk_root[3] [USED] Peri 1 Memory Mapped IO clk_pclk_root[4] [USED]
CSV	
CSV	☐

Figure 4 GFXSS high frequency clock configuration

The following list of internal clocks for all three components of the graphics sub-system is auto-generated from the system and the high-frequency main clock.

Table 1 GFXSS internal clocks

Clock	Value	Description
clk_2d	Minimum 50/64 MHz Maximum 200 MHz	GPU core clock. Generated within the graphics sub-system core with a DIV2 from clk_hf. The frequency limits the peak pixel fill rate that the GPU can achieve (= 1 pixel/cycle). The min value ensures a comfortable margin in fill rate for a use case with 640x480 @60 Hz (24 MHz pixel clock). The maximum value is optimal for energy consumption.
clk2x_2d	Minimum 100 MHz Maximum 400 MHz	GPU 2x core clock (by-pass of the internal DIV2). Used to realize dual port access to internal memories by the pixel engine.
clk_hf_core	Minimum 50 MHz Maximum 200 MHz	Core and AXI clock for the DC. Division by 2 from clk_hf.

(table continues...)

Table 1 (continued) GFXSS internal clocks

Clock	Value	Description
clk_hf_div	Minimum 1 MHz Maximum 200 MHz	Fractional division from clk_hf. 50% duty cycle. Systematic jitter on both edges is maximum 1/f_{clk_hf}. DPI Mode via GPIO or MIPI-DSI Same as pixel clock frequency. DBI Mode Type C via GPIO (SPI) Twice the serial clock frequency
clk_sys_cfg	Minimum 17 MHz Maximum 52 MHz	Configuration clock for the D-PHY. Generated by DIV2 from clk_sys_mipidsi.

4.1.2 DSI host clock

A correctly configured MIPI DSI host clock ensures that the display link can carry the required pixel stream (or command bursts) with sufficient margin while maintaining stability across the high speed/low power (HS/LP) transitions. Table 2 presents the necessary clocks for the DSI host controller.

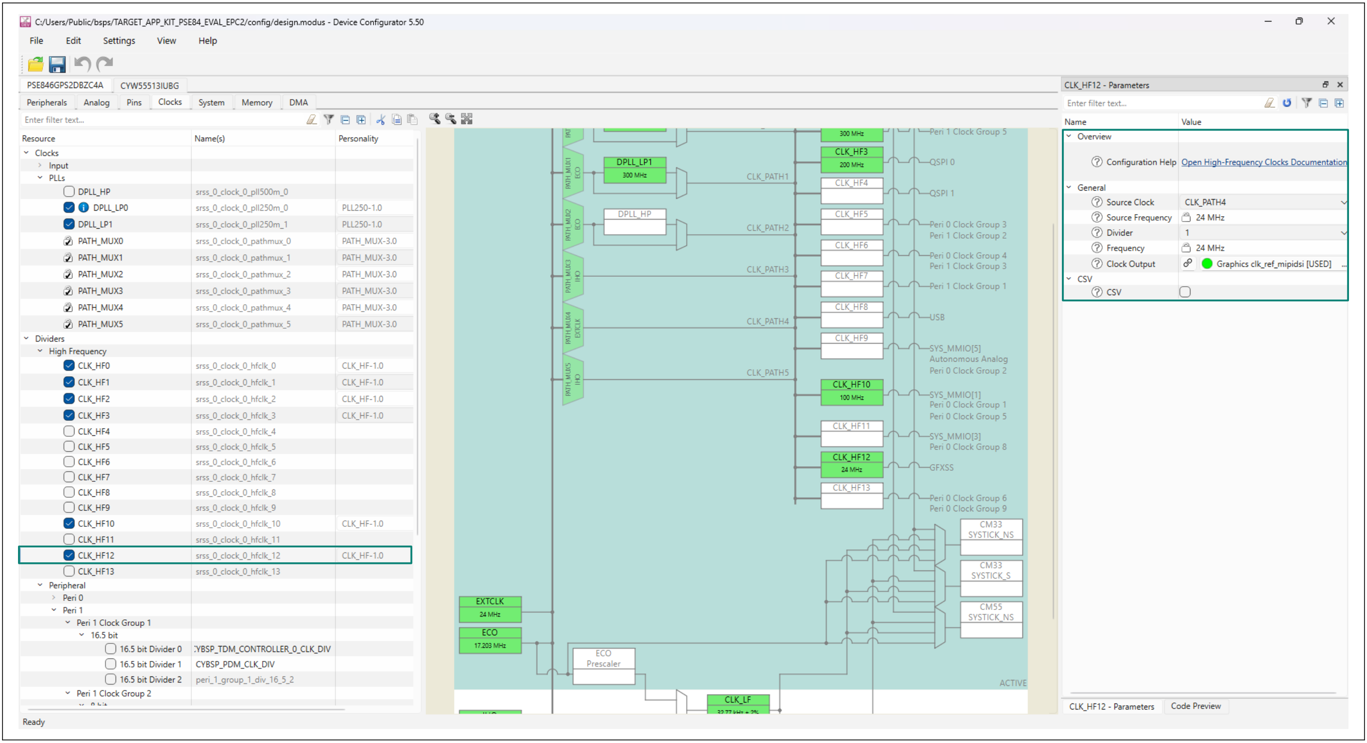
Table 2 DSI host clock

Clock type	Value	Clock alias	Description
Reference clock	Typical 24 MHz	clk_ref_mipidsi	Reference clock for the D-PHY PLL (CLK_HF12).
Lane byte clock	Max. 187.5 MHz	lanebyteclk	Internally generated from clk_ref_mipidsi with the D-PHY PLL, serves as a reference for the DSI host controller's packet interface, synchronizing the transmission of video and command data.
Pixel clock	Min. 1 MHz Max. 64 MHz	dpipclk	Pixel clock for DPI via MIPI-DSI, auto generated by the DC based on video timings and clk_hf_div. Max frequency corresponds to VESA 1024x768 @60 Hz (XGA), min to the slowest VESA mode of 160x120 @50 Hz (QQVGA).
DBI clock	Max. 37.5 MHz	dbiclk	Data clock for DBI Type B via MIPI-DSI. Generated by the DC. Max is 1/4 of max lanebyteclk per MIPI host controller specification.
LP mode clock	Max. 20 MHz	txclkesc rxclkesc	Low-power (escape) clock for command/ULPS operations.

- D-PHY PLL reference clock (configured in device configurator)
  - This needs to be configured as 24 MHz as shown in Figure 5, sourced from either a dedicated external clock (EXTCLK) or clk_hf. This low-jitter reference feeds the D-PHY PLL that synthesizes the high-speed lane bit rate. Its stability directly impacts link lock and signal integrity
- Lane byte clock (auto-derived)
  - This internal clock drives packetization and TX FIFOs; too low and the FIFO may starve, too high and you may hit PHY limits

- The host computes the lane byte clock from the selected lane bit rate: lanebyteclk = lanerate / 8. It drives packetization and TX FIFOs and must sustain your video/command throughput with margin. With the chosen two 1-lane panels capped at 850 Mbps and 700 Mbps respectively, lanebyteclk is:
  - 850 Mbps → 106.25 MHz
  - 700 Mbps → 87.5 MHz
- Never exceed the panel's per-lane limit; if a shared configuration forces a common rate, cap at the lower (700 Mbps) so lanebyteclk = 87.5 MHz

• TX escape (LP) clock (auto-derived, ≤ 20 MHz)
  - Used for low-power commands, ULPS entry/exit, and error recovery. It is derived from lanebyteclk via an integer divider and must not exceed 20 MHz. It affects command/LP latency, not video bandwidth. Typical targets are 10–20 MHz as allowed by lanebyteclk / divider



Resource	Name(s)	Personality
Clocks		
Input		
PLLs		
☐ DPLL_HP	srss_0_clock_0_pll500m_0	
☑ DPLL_LP0	srss_0_clock_0_pll250m_0	PLL250-1.0
☑ DPLL_LP1	srss_0_clock_0_pll250m_1	PLL250-1.0
PATH_MUX0	srss_0_clock_0_pathmux_0	PATH_MUX-3.0
PATH_MUX1	srss_0_clock_0_pathmux_1	PATH_MUX-3.0
PATH_MUX2	srss_0_clock_0_pathmux_2	PATH_MUX-3.0
PATH_MUX3	srss_0_clock_0_pathmux_3	PATH_MUX-3.0
PATH_MUX4	srss_0_clock_0_pathmux_4	PATH_MUX-3.0
PATH_MUX5	srss_0_clock_0_pathmux_5	PATH_MUX-3.0
Dividers		
High Frequency		
☑ CLK_HF0	srss_0_clock_0_hfclk_0	CLK_HF-1.0
☑ CLK_HF1	srss_0_clock_0_hfclk_1	CLK_HF-1.0
☑ CLK_HF2	srss_0_clock_0_hfclk_2	CLK_HF-1.0
☑ CLK_HF3	srss_0_clock_0_hfclk_3	CLK_HF-1.0
☐ CLK_HF4	srss_0_clock_0_hfclk_4	
☐ CLK_HF5	srss_0_clock_0_hfclk_5	
☐ CLK_HF6	srss_0_clock_0_hfclk_6	
☐ CLK_HF7	srss_0_clock_0_hfclk_7	
☐ CLK_HF8	srss_0_clock_0_hfclk_8	
☐ CLK_HF9	srss_0_clock_0_hfclk_9	
☑ CLK_HF10	srss_0_clock_0_hfclk_10	CLK_HF-1.0
☐ CLK_HF11	srss_0_clock_0_hfclk_11	
☑ CLK_HF12	srss_0_clock_0_hfclk_12	CLK_HF-1.0
☐ CLK_HF13	srss_0_clock_0_hfclk_13	
Peripheral		
Peri 0		
Peri 1		
Peri 1 Clock Group 1		
16.5 bit		
☐ 16.5 bit Divider 0	CYBSP_TDM_CONTROLLER_0_CLK_DIV	
☐ 16.5 bit Divider 1	CYBSP_PDM_CLK_DIV	
☐ 16.5 bit Divider 2	peri_1_group_1_div_16_5_2	
Peri 1 Clock Group 2		

CLK_HF12 - Parameters

Name	Value
Overview	
Configuration Help	Open High-Frequency Clocks Documentation
General	
Source Clock	CLK_PATH4
Source Frequency	24 MHz
Divider	1
Frequency	24 MHz
Clock Output	Graphics clk_ref_mipidsi [USED]
CSV	
CSV	☐

Figure 5 **D-PHY PLL reference clock configuration**

4.2 Display type and interface

PSOC™ Edge supports MIPI DBI A/B/C, MIPI DPI, and MIPI DSI (Video and Command) display types; however, this application note and its code example focus exclusively on MIPI DSI single-lane displays in Video mode and Command mode used by the reference application. This section first shows how to configure a 4.3-inch single-lane DSI panel in Video Mode, followed by a 1.43-inch single-lane DSI panel in Command mode. In the Device Configurator, users begin by selecting the display type (Video or Command) and then choose the corresponding interface format and mode-specific parameters. Even when defaults are auto-filled, understanding these options helps making the right trade-offs for image quality, bandwidth, and power. Other interface types, such as DBI/DPI, are supported by the platform but are out of scope for this document.

4.2.1 MIPI DSI video mode

MIPI DSI video mode streams pixels continuously at fixed display timings, making it an ideal choice for smooth animations, video playback, and predictable end-to-end latency. It is used with displays that do not embed graphic RAM capable of holding an entire frame and rely on the MIPI DSI host to provide a continuous pixel stream, as they lack an internal controller.

The MIPI DSI host video mode supports the following three transfer types:

Table 3 Video mode transfer type

Transfer type	Description	When to use
Burst mode	Active pixels are transmitted in brief high-speed bursts, and the remaining line/frame interval is left as blanking or low power idle, thereby reducing the high-speed duty cycle. This transmission mode requires that the DSI payload pixel FIFO has the capacity to store a full line of active pixel data inside of it	• Panel can receive a full pixel line in a single packet burst to prevent overflow in the receiving buffer • Lane rate has ample headroom vs payload (e.g., lower color depth or refresh) • Desire to drop to low-power once per line
Non-burst mode with sync pulses	HSYNC and VSYNC are sent as explicit pulse packets, while blanking and active periods stream continuously. This aligns the DC interface input pixel bandwidth with the DSI link bandwidth. In this mode, the host controller and display do not need to store a full line of pixel data, only the content of one video packet, minimizing memory requirements	• Default choice for broad compatibility and straightforward timing • When lane rate is close to pixel payload
Non-burst mode with sync events	Signals HSYNC/VSYNC with only “start” and “end” events; the blanking interval is the time between these events, not explicit sync-pulse packets. The link streams continuously in high speed during active video, with lower control overhead than non-burst with sync pulses. Clock-lane operation is continuous, making precise porch/sync programming essential, as timing defines blanking	• If the panel datasheet lists non-burst mode with sync events as supported/preferred and you want marginally leaner control traffic than non-burst mode with sync pulses • When your per-lane rate is close to the pixel payload and you want to preserve a bit of headroom by trimming control overhead (without switching to Burst) • In systems where you need deterministic continuous streaming but want to minimize unnecessary packetization during blanking • When power behavior with continuous clock is acceptable and compatibility has been verified

Now in a MIPI DSI Video pipeline, “DPI-x” defines the on-wire color coding and packing used to encapsulate pixels into DSI video packets. This choice directly sets bits per pixel, required lane throughput, visual fidelity, and HS duty cycle. “Configuration 2” denotes packed mappings (no pad bits), which are the bandwidth-efficient variants for 18 bpp and 16 bpp. Always confirm that the target panel supports the selected coding and packing.

Following display interfaces are supported in MPI DSI video mode:

Table 4 **Video mode display interface**

Display interface	Description	When to use
DPI-24 (RGB888, 24 bpp)	Delivers true color at 24 bpp (8 bits per R, G, B) with no precision loss, yielding excellent gradients and minimal banding but it imposes the highest payload and memory/ display bandwidth demand.	• When color quality and gradient smoothness are critical (photos, video, rich UI themes) • When the single-lane link has ample headroom at the target resolution and refresh rate • When the rendering pipeline already produces 24/32-bpp frames, avoiding conversion overhead ***Note**: When using 24-bit color depth, the physical interface remains in the high-speed transmit state for a larger portion of each line or frame, increasing switching activity and potentially raising power consumption compared to lower-bpp modes.*
DPI-18 (configuration 2, RGB666 packed, 18 bpp)	Uses 6 bits per channel in a tightly packed format, preserving good visual quality with modest precision loss, while cutting payload by roughly 25% compared to RGB888 to improve lane-rate margin and reduce high-speed duty cycle.	• When we need extra throughput headroom on a single lane without a large quality compromise • When the panel offers internal dithering or the graphics stack can dither to smooth gradients • When power consumption must be trimmed relative to RGB888 ***Note**: Verify explicit support for "RGB666 packed/ config 2 for the chosen display." Check color order (RGB/BGR) and byte packing.*

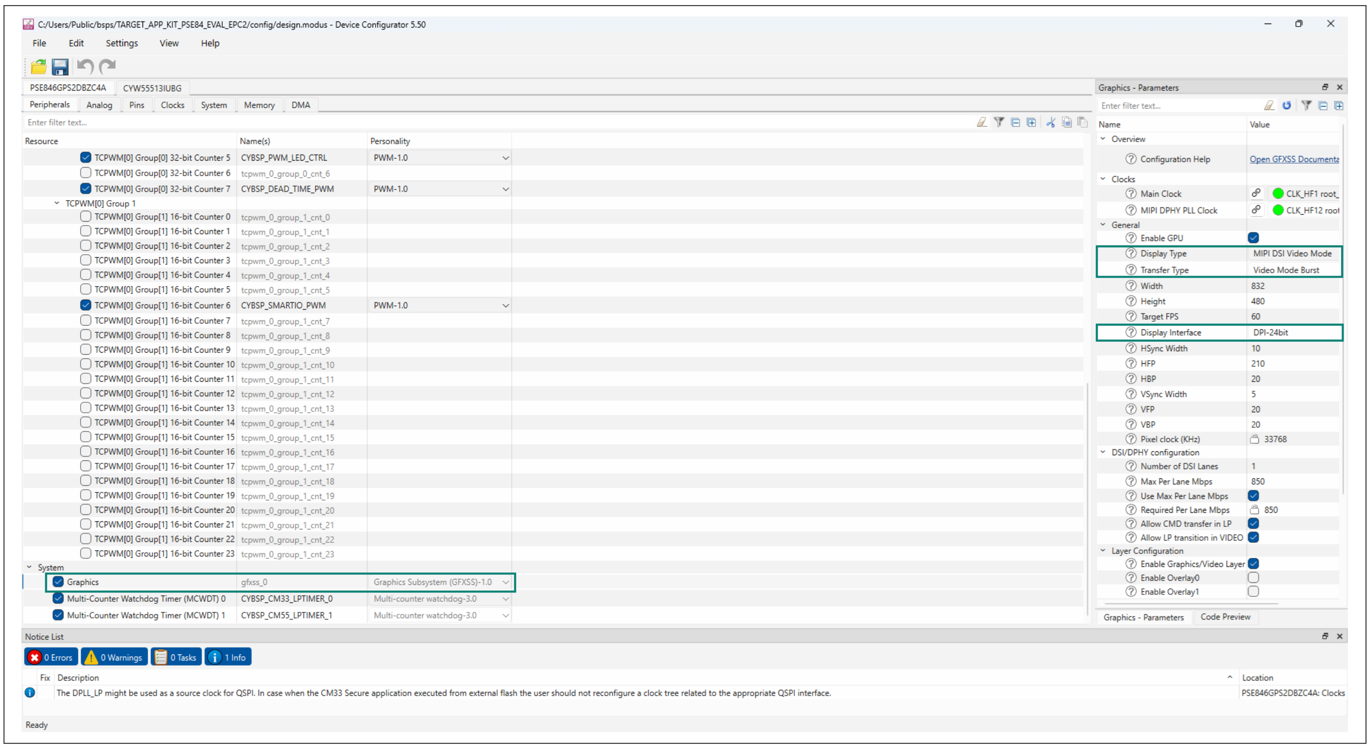
(table continues...)

Table 4 (continued) Video mode display interface

Display interface	Description	When to use
DPI-16 (configuration 2, RGB565 packed, 16 bpp)	Encodes pixels with 5-6-5 bits for R-G-B in a tightly packed format, offering the most bandwidth-efficient standard option by reducing payload about 33% versus RGB888 and yielding the largest headroom for single-lane links.	• When lane-rate budget is tight at chosen fps, or when minimizing HS duty cycle, power, and low power consumption is a priority • For UIs dominated by text, icons, and flat colors where limited precision is acceptable • When memory footprint and bus bandwidth must be reduced (smaller frame buffers, lower read/write traffic) ***Note**: Some banding may appear in smooth gradients; enable dithering if available. Confirm packed mapping and color order with the panel.*

To apply these configurations, the user needs to enable the Graphics subsystem and select the following settings for the 4.3-inch single DSI lane 800x480 video mode display, as shown in the Figure 6.

- Display Type: MIPI DSI Video Mode
- Transfer Type: Video Mode Burst
- Display Interface: DPI-24 bit



C:/Users/Public/bsps/TARGET_APP_KIT_PSE84_EVAL_EPC2/config/design.modus - Device Configurator 5.50

File Edit Settings View Help

PSE846GPS2DBZC4A | CYW55513IUBG

Peripherals | Analog | Pins | Clocks | System | Memory | DMA

Enter filter text...

Resource	Name(s)	Personality
✓ TCPWM[0] Group[0] 32-bit Counter 5	CYBSP_PWM_LED_CTRL	PWM-1.0
☐ TCPWM[0] Group[0] 32-bit Counter 6	tcpwm_0_group_0_cnt_6	
✓ TCPWM[0] Group[0] 32-bit Counter 7	CYBSP_DEAD_TIME_PWM	PWM-1.0
TCPWM[0] Group 1		
☐ TCPWM[0] Group[1] 16-bit Counter 0	tcpwm_0_group_1_cnt_0	
☐ TCPWM[0] Group[1] 16-bit Counter 1	tcpwm_0_group_1_cnt_1	
☐ TCPWM[0] Group[1] 16-bit Counter 2	tcpwm_0_group_1_cnt_2	
☐ TCPWM[0] Group[1] 16-bit Counter 3	tcpwm_0_group_1_cnt_3	
☐ TCPWM[0] Group[1] 16-bit Counter 4	tcpwm_0_group_1_cnt_4	
☐ TCPWM[0] Group[1] 16-bit Counter 5	tcpwm_0_group_1_cnt_5	
✓ TCPWM[0] Group[1] 16-bit Counter 6	CYBSP_SMARTIO_PWM	PWM-1.0
☐ TCPWM[0] Group[1] 16-bit Counter 7	tcpwm_0_group_1_cnt_7	
☐ TCPWM[0] Group[1] 16-bit Counter 8	tcpwm_0_group_1_cnt_8	
☐ TCPWM[0] Group[1] 16-bit Counter 9	tcpwm_0_group_1_cnt_9	
☐ TCPWM[0] Group[1] 16-bit Counter 10	tcpwm_0_group_1_cnt_10	
☐ TCPWM[0] Group[1] 16-bit Counter 11	tcpwm_0_group_1_cnt_11	
☐ TCPWM[0] Group[1] 16-bit Counter 12	tcpwm_0_group_1_cnt_12	
☐ TCPWM[0] Group[1] 16-bit Counter 13	tcpwm_0_group_1_cnt_13	
☐ TCPWM[0] Group[1] 16-bit Counter 14	tcpwm_0_group_1_cnt_14	
☐ TCPWM[0] Group[1] 16-bit Counter 15	tcpwm_0_group_1_cnt_15	
☐ TCPWM[0] Group[1] 16-bit Counter 16	tcpwm_0_group_1_cnt_16	
☐ TCPWM[0] Group[1] 16-bit Counter 17	tcpwm_0_group_1_cnt_17	
☐ TCPWM[0] Group[1] 16-bit Counter 18	tcpwm_0_group_1_cnt_18	
☐ TCPWM[0] Group[1] 16-bit Counter 19	tcpwm_0_group_1_cnt_19	
☐ TCPWM[0] Group[1] 16-bit Counter 20	tcpwm_0_group_1_cnt_20	
☐ TCPWM[0] Group[1] 16-bit Counter 21	tcpwm_0_group_1_cnt_21	
☐ TCPWM[0] Group[1] 16-bit Counter 22	tcpwm_0_group_1_cnt_22	
☐ TCPWM[0] Group[1] 16-bit Counter 23	tcpwm_0_group_1_cnt_23	
System		
✓ Graphics	gfxss_0	Graphics Subsystem (GFXSS)-1.0
✓ Multi-Counter Watchdog Timer (MCWDT) 0	CYBSP_CM33_LPTIMER_0	Multi-counter watchdog-3.0
✓ Multi-Counter Watchdog Timer (MCWDT) 1	CYBSP_CM55_LPTIMER_1	Multi-counter watchdog-3.0

Graphics - Parameters

Enter filter text...

Name	Value
Overview	
Configuration Help	Open GFXSS Documenta
Clocks	
Main Clock	CLK_HF1 root_
MIPI DPHY PLL Clock	CLK_HF12 root
General	
Enable GPU	✓
Display Type	MIPI DSI Video Mode
Transfer Type	Video Mode Burst
Width	832
Height	480
Target FPS	60
Display Interface	DPI-24bit
HSync Width	10
HFP	210
HBP	20
VSync Width	5
VFP	20
VBP	20
Pixel clock (KHz)	33768
DSI/DPHY configuration	
Number of DSI Lanes	1
Max Per Lane Mbps	850
Use Max Per Lane Mbps	✓
Required Per Lane Mbps	850
Allow CMD transfer in LP	✓
Allow LP transition in VIDEO	✓
Layer Configuration	
Enable Graphics/Video Layer	✓
Enable Overlay0	☐
Enable Overlay1	☐

Graphics - Parameters | Code Preview

Notice List

0 Errors | 0 Warnings | 0 Tasks | 1 Info

Fix	Description	Location
	The DPLL_LP might be used as a source clock for QSPI. In case when the CM33 Secure application executed from external flash the user should not reconfigure a clock tree related to the appropriate QSPI interface.	PSE846GPS2DBZC4A: Clocks

Ready

Figure 6 **MIPI DSI video mode display configuration**

In relation to the code example mentioned in this document, the single-lane panel supports up to 850 Mbps per lane and our calculated payload provides substantial margin. MIPI DSI is configured in Video Burst mode to compress active pixels into short high-speed bursts, thereby reducing the high-speed duty cycle. The application renders in RGB565 (16 bpp) for memory/bus efficiency, while the display interface is set to use DPI-24 (RGB888) on-wire color coding to match the panel's preferred input format. The available lane-rate headroom makes this on-the-fly expansion practical without risking under-runs. This combination preserves visual compatibility and quality on the link, benefits from lower average HS activity (potentially improving power), and maintains deterministic, tear-free scan-out.

4.2.1.1 Setting up video timings per display resolution

Once the video mode, transfer type, and display interface have been selected, it is essential to accurately configure the video timing parameters to ensure the display is driven at the exact cadence that the panel expects, delivering a stable image in MIPI DSI video mode. These timings are primarily associated with the display controller, which generates and sends them to the display module to ensure coordinated pixel and frame updates, preventing issues such as screen tearing.

For each resolution, it is necessary to define the active image area and the blanking regions surrounding it. Together with the target refresh rate, these parameters produce the horizontal and vertical totals that set the pixel clock, the rate at which the pixels must be delivered. These timing totals also determine the data payload that the DSI host must packetize and transmit over the lane(s), which in turn establishes the required DSI lane bit rate and the internal lane byte clock.

A display frame is composed of active video pixels (visible area) and several blanking intervals that are used for synchronization and signal processing. The two major timing domains are horizontal timing (per line) and vertical timing (per frame).

Horizontal timing parameters (per line)

- **Horizontal sync width (HSync width)**: Duration of the horizontal sync pulse, signaling the start of a new line

- **Horizontal back porch (HBP)**: Idle period after the horizontal sync pulse before starting to send active data pixels
- **Horizontal active (display active width)**: The number of visible pixels in a single line (equal to horizontal resolution, for example 800 for 800x480 4.3-inch Waveshare display)
- **Horizontal front porch (HFP)**: Idle period after the active pixels before the horizontal sync pulse. Used by the display to prepare for the next line

Formula for total horizontal period:

```
H_TOTAL [in pixels] = HSync width + HBP + Display active width + HFP
```

Vertical timing parameters (per frame)

- **Vertical sync width (VSync width)**: Number of lines during the active vertical sync pulse
- **Vertical back porch (VBP)**: Number of idle lines after the VSync pulse before starting the next frame
- **Vertical active (display active height)**: Number of lines of visible pixels per frame (equal to vertical resolution, for example 480 for 800x480 4.3-inch Waveshare display)
- **Vertical front porch (VFP)**: Number of idle lines after completing active frame lines before the vertical sync pulse

Formula for total vertical period:

```
V_TOTAL [in lines] = VSync width + VBP + Display active height + VFP
```

Following figure illustrates fully programmable timings and resolutions.

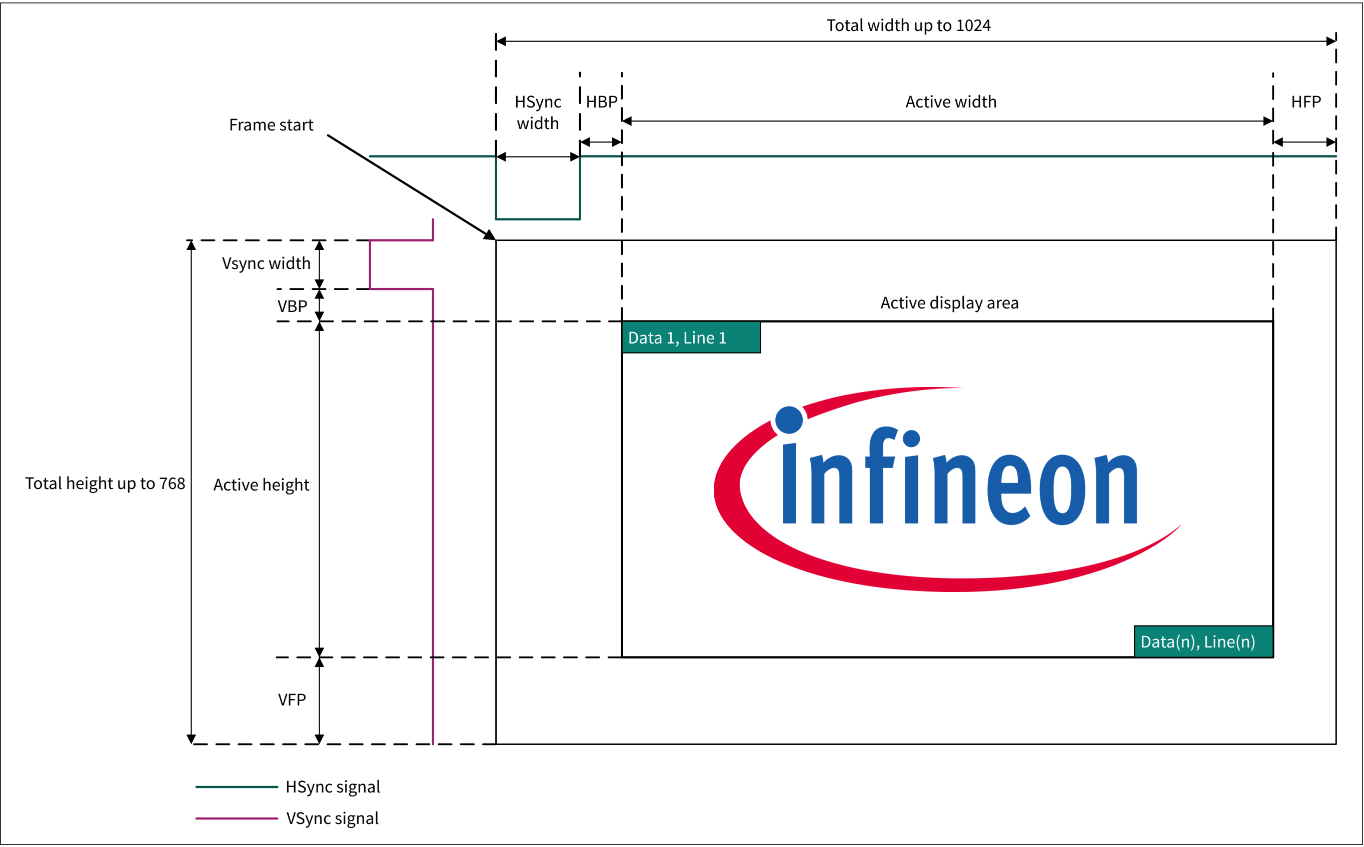


Figure 7 **Fully programmable timings and resolutions**

Calculating pixel clock, DSI clock, and bit clock

The pixel clock (PCLK) defines the rate at which pixels are transmitted to the panel.

```
PCLK [in Hz] = H_TOTAL x V_TOTAL x Frame rate
```

Where:

-
  ```
  Frame rate = Refresh rate or Target FPS (for example 60 Hz)
  ```

-
  ```
  H_TOTAL and V_TOTAL are total horizontal and vertical periods respectively
  ```

For a DSI-based interface:

- **DSI clock**: Determined by the pixel clock, the bits per pixel (BPP), and the number of data lanes in use
- **Bit clock formula**:

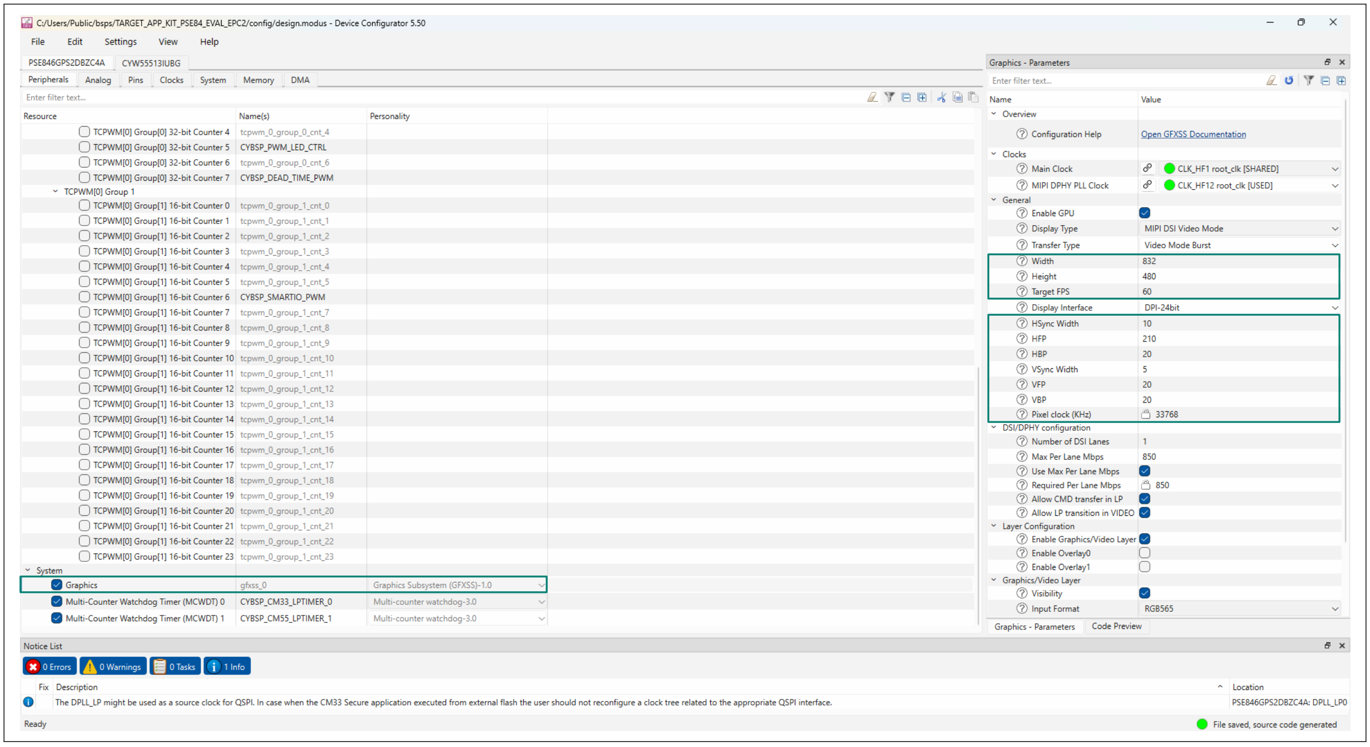
  ```
  Bit clock = (PCLK x BPP) / Number of lanes
  ```

  Where:

  - BPP depends on the color format (for example DPI-24 = 24 bits per pixel)
  - Number of lanes depends on the DSI lanes configured for the display (for example 1 lane for 4.3-inch 800x480 Waveshare display)
  - Final DSI clock may be set slightly higher to account for protocol overhead

In the Device Configurator, set the parameters described above in the "General" section for the Waveshare 4.3-inch 800x480 MIPI DSI video mode display as follows, as depicted in Figure 8.

- Width: 832 (if using RGB565 color format to achieve 128 byte stride alignment per DC requirement) else 800 matching the horizontal resolution
- Height: 480 matching the vertical resolution
- Target FPS: 60 as per the panel spec
- All HSync and VSync timings as per the panel datasheet which in turn auto calculate and set the pixel clock



Resource	Name(s)	Personality
TCPWM[0] Group[0] 32-bit Counter 4	tcpwm_0_group_0_cnt_4	
TCPWM[0] Group[0] 32-bit Counter 5	CYBSP_PWM_LED_CTRL	
TCPWM[0] Group[0] 32-bit Counter 6	tcpwm_0_group_0_cnt_6	
TCPWM[0] Group[0] 32-bit Counter 7	CYBSP_DEAD_TIME_PWM	
TCPWM[0] Group 1		
TCPWM[0] Group[1] 16-bit Counter 0	tcpwm_0_group_1_cnt_0	
TCPWM[0] Group[1] 16-bit Counter 1	tcpwm_0_group_1_cnt_1	
TCPWM[0] Group[1] 16-bit Counter 2	tcpwm_0_group_1_cnt_2	
TCPWM[0] Group[1] 16-bit Counter 3	tcpwm_0_group_1_cnt_3	
TCPWM[0] Group[1] 16-bit Counter 4	tcpwm_0_group_1_cnt_4	
TCPWM[0] Group[1] 16-bit Counter 5	tcpwm_0_group_1_cnt_5	
TCPWM[0] Group[1] 16-bit Counter 6	CYBSP_SMARTIO_PWM	
TCPWM[0] Group[1] 16-bit Counter 7	tcpwm_0_group_1_cnt_7	
TCPWM[0] Group[1] 16-bit Counter 8	tcpwm_0_group_1_cnt_8	
TCPWM[0] Group[1] 16-bit Counter 9	tcpwm_0_group_1_cnt_9	
TCPWM[0] Group[1] 16-bit Counter 10	tcpwm_0_group_1_cnt_10	
TCPWM[0] Group[1] 16-bit Counter 11	tcpwm_0_group_1_cnt_11	
TCPWM[0] Group[1] 16-bit Counter 12	tcpwm_0_group_1_cnt_12	
TCPWM[0] Group[1] 16-bit Counter 13	tcpwm_0_group_1_cnt_13	
TCPWM[0] Group[1] 16-bit Counter 14	tcpwm_0_group_1_cnt_14	
TCPWM[0] Group[1] 16-bit Counter 15	tcpwm_0_group_1_cnt_15	
TCPWM[0] Group[1] 16-bit Counter 16	tcpwm_0_group_1_cnt_16	
TCPWM[0] Group[1] 16-bit Counter 17	tcpwm_0_group_1_cnt_17	
TCPWM[0] Group[1] 16-bit Counter 18	tcpwm_0_group_1_cnt_18	
TCPWM[0] Group[1] 16-bit Counter 19	tcpwm_0_group_1_cnt_19	
TCPWM[0] Group[1] 16-bit Counter 20	tcpwm_0_group_1_cnt_20	
TCPWM[0] Group[1] 16-bit Counter 21	tcpwm_0_group_1_cnt_21	
TCPWM[0] Group[1] 16-bit Counter 22	tcpwm_0_group_1_cnt_22	
TCPWM[0] Group[1] 16-bit Counter 23	tcpwm_0_group_1_cnt_23	
System		
✓ Graphics	gfxss_0	Graphics Subsystem (GFXSS)-1.0
✓ Multi-Counter Watchdog Timer (MCWDT) 0	CYBSP_CM33_LPTIMER_0	Multi-counter watchdog-3.0
✓ Multi-Counter Watchdog Timer (MCWDT) 1	CYBSP_CM55_LPTIMER_1	Multi-counter watchdog-3.0

Name	Value
Overview	
Configuration Help	Open GFXSS Documentation
Clocks	
Main Clock	CLK_HF1 root_clk [SHARED]
MIPI DPHY PLL Clock	CLK_HF12 root_clk [USED]
General	
Enable GPU	✓
Display Type	MIPI DSI Video Mode
Transfer Type	Video Mode Burst
Width	832
Height	480
Target FPS	60
Display Interface	DPI-24bit
HSync Width	10
HFP	210
HBP	20
VSync Width	5
VFP	20
VBP	20
Pixel clock (KHz)	33768
DSI/DPHY configuration	
Number of DSI Lanes	1
Max Per Lane Mbps	850
Use Max Per Lane Mbps	✓
Required Per Lane Mbps	850
Allow CMD transfer in LP	✓
Allow LP transition in VIDEO	✓
Layer Configuration	
Enable Graphics/Video Layer	✓
Enable Overlay0	
Enable Overlay1	
Graphics/Video Layer	
Visibility	✓
Input Format	RGB565

Fix	Description	Location
	The DPLL_LP might be used as a source clock for QSPI. In case when the CM33 Secure application executed from external flash the user should not reconfigure a clock tree related to the appropriate QSPI interface.	PSE846GPS2DBZC4A: DPLL_LP0

Figure 8 **Video timings and display resolution configuration**

The video timings will reflect in the `cy_stc_mipidsi_display_params_t` configuration structure as shown in the snippet below after the configurations are saved in the Device Configurator.

MIPI DSI display parameters

```
cy_stc_mipidsi_display_params_t GFXSS_mipidsi_display_params =
{
    .pixel_clock    = 33768,
    .hdisplay       = 832,
    .hsync_width    = 10,
    .hfp            = 210,
    .hbp            = 20,
    .vdisplay       = 480,
    .vsync_width    = 5,
    .vfp            = 20,
    .vbp            = 20,
    .polarity_flags = 0,
};
```

Polarity flags

Polarity flags define which logic level (high or low) is “active” for HSYNC, VSYNC, and DATAEN (DE) in a DPI-fed MIPI DSI Video mode path. Correct settings are essential for proper image alignment and stable scan-out. “Shutdown” and “Color mode” are auxiliary control signals; depending on the panel ecosystem, they may be implemented as GPIOs or not used at all (DSI panels often rely on DCS commands instead).

It is applicable only in Video mode (DPI input to the DSI Host). Not applicable in Command Mode (DBI input). For the 4.3-inch display, the polarity flags field is 0, indicating active high for HSYNC, VSYNC, and DE.

In case of interfacing a different display panel, manually set the polarity_flags parameter of the `cy_stc_mipidsi_display_params_t` structure before initializing the graphics subsystem, according to the panel datasheet. To quickly validate the configuration, generate test patterns (color bars/checkerboard) for the display. If the image shifts or rolls, swap the HSYNC/VSYNC polarity; if the active area is inverted, adjust the DE polarity.

4.2.2 MIPI DSI command mode

In MIPI DSI command mode, the display panel refresh is driven primarily by commands and data transferred via the DSI link, rather than a continuous pixel stream as in video mode. Pixel data is written into the panel's frame memory (GRAM) through short and long packets. The panel itself is responsible for scanning out the stored image to the display glass at the configured refresh rate. This allows asynchronous updates – only the regions that change need to be updated, reducing bus traffic and power consumption. Clock-lane operation is usually non-continuous to save power, and ULPM can be used during long idle periods.

The MIPI DSI host command mode supports following display interfaces:

8-bit interface modes

1. **RGB332 (8 bpp)**
   - 3 bits Red, 3 bits Green, 2 bits Blue
   - Pros: Very low data rate, extremely low memory footprint; ideal for icon/monochrome-dominant GUIs
   - Cons: Very limited color depth, visible banding for gradients and photos
   - Typical Use: Low-cost/simple GUIs, industrial HMI status displays
2. **RGB444 (8 bpp)** *(packed format)*
   - 4 bits per component, with specific packing rules for DBI
   - Pros: Slightly better fidelity than RGB332 for gradients; still low bandwidth usage
   - Cons: Not full 8 or 6 bits per component, still prone to visible artifacts in rich color content
   - Typical Use: Low-power systems with moderate image quality needs
3. **RGB565 (8 bpp mode)**
   - Host transmits reduced pixel data with conversion handled internally by panel
   - Pros: Good compromise on quality vs bandwidth even in constrained mode
   - Cons: Requires panel-side mapping; limited flexibility in some controllers
4. **RGB666 and RGB888 (8 bpp mode)**
   - Encoded in compressed or reduced form for transmission; reconstructed by panel
   - Pros: Achieves richer color depth with minimal transfer size
   - Cons: Panel must internally expand and map correctly

9-bit interface modes

1. **RGB666 (9 bpp)**
   - Allocates 6 bits per channel, sent with a 9-bit parallel structure in command packets
   - Pros: Offers true 18-bit color resolution; good balance of fidelity and bandwidth
   - Cons: Not as widely supported as 8 or 16-bit wide bus modes; marginally more complex hardware/firmware handling
   - Typical Use: Medium-end GUIs where better gradients are needed without 24-bit transfers

16-bit interface modes

1. **RGB332 (16-bit)**
   - Padding to meet 16-bit word alignment
   - Pros: Very low bandwidth and processing cost; alignment can simplify DMA transfers
   - Cons: Same fidelity limitations as 8-bit RGB332

2. **RGB444 (16-bit)**
   - Padding/bus alignment improves per-pixel access overhead
   - Pros: Slightly richer color representation than RGB332; efficient transfer in 16-bit systems
   - Cons: Not photo-quality; visible quantization
3. **RGB565 (16-bit)**
   - 5 bits Red, 6 bits Green, 5 bits Blue (packed into 2 bytes)
   - Pros: Most common 16-bit mode; good compromise between fidelity and data rate; available in Option 1 and Option 2 packing
   - Cons: Not true 24-bit color; minor gradients loss in certain images
   - Typical Use: Embedded GUIs, wearable devices, control systems
4. **RGB666 (16-bit)** *(Option 1 and Option 2)*
   - Encoded with specific padding/packing rules to match DCS protocol requirements
   - Pros: Delivers near 18-bit fidelity with reasonable transfer width
   - Cons: Requires careful configuration; not all panels support both packing options
5. **RGB888 (16-bit)** *(Option 1 and Option 2)*
   - Two packing schemes used to fit 24-bit pixels into 16-bit aligned transfers
   - Pros: Achieves full 24-bit color representation over a 16-bit interface; essential for photo-grade content
   - Cons: Significantly higher data rate; bandwidth and memory impact; careful configuration required
   - Typical Use: High-end GUIs, photo viewers, video rendering in command mode

When transmitting pixel formats such as RGB666 (18 bpp) or RGB888 (24 bpp) over a 16-bit wide DSI command mode interface (DBI-B), the pixel data does not align perfectly with the 16-bit word size. As a result, the MIPI DCS specification defines two common packing schemes for these formats: Option 1 and Option 2. The difference lies in whether pixels are word-aligned (loose packing) or packet-optimized (tight packing).

1. **Option 1 - loose packing (word-aligned)**
   - **Description:**

     Each pixel's data is sent starting at a new 16-bit word boundary. Any unused bits in the word are filled with padding (don't-care values)
   - **RGB888:**

     A 24-bit pixel is expanded to two 16-bit words. This introduces unused bits in the second word to keep pixel boundaries aligned
   - **RGB666:**

     An 18-bit pixel is padded to 2 bytes (16 bits), again introducing unused bits
   - **Outcome:**

     Simplifies both host and panel logic because each pixel is cleanly separated, but data rate is slightly higher due to padding
2. **Option 2 - tight packing (bit-optimized)**
   - **Description:**

     Pixels are packed back-to-back in memory without padding, and their bits can cross 16-bit word boundaries. Three consecutive bytes (RGB for one pixel in RGB888 or 18 bits for RGB666) are split between words as needed
   - **RGB888:**

     For example, the Blue component of one pixel and the Red component of the next may share the same 16-bit word
   - **RGB666:**

Multiple pixels' bits are interleaved so that no padding is required, each word is fully utilized

- **Outcome:**

  Delivers maximum bandwidth efficiency, but increases packing/unpacking complexity in both the host and the panel

The MIPI DCS standard provides both options so that system designers can trade off between simplicity (Option 1) and link efficiency (Option 2). The correct choice depends on system bandwidth, panel support, and implementation complexity tolerance.

In our case, we utilize a 1.43-inch single lane 466x466 MIPI DSI command mode round display, where we configure the display's resolution and display interface to RGB888 16-bit Option 1, as shown in the Figure 9 below. This choice is made on the following reasons:

1. **Simplicity of integration**:

   Option 1 keeps each pixel aligned to a 16-bit word boundary, greatly simplifying the host controller's memory-to-DSI data mapping, and minimizing firmware complexity

2. **Panel compatibility**:

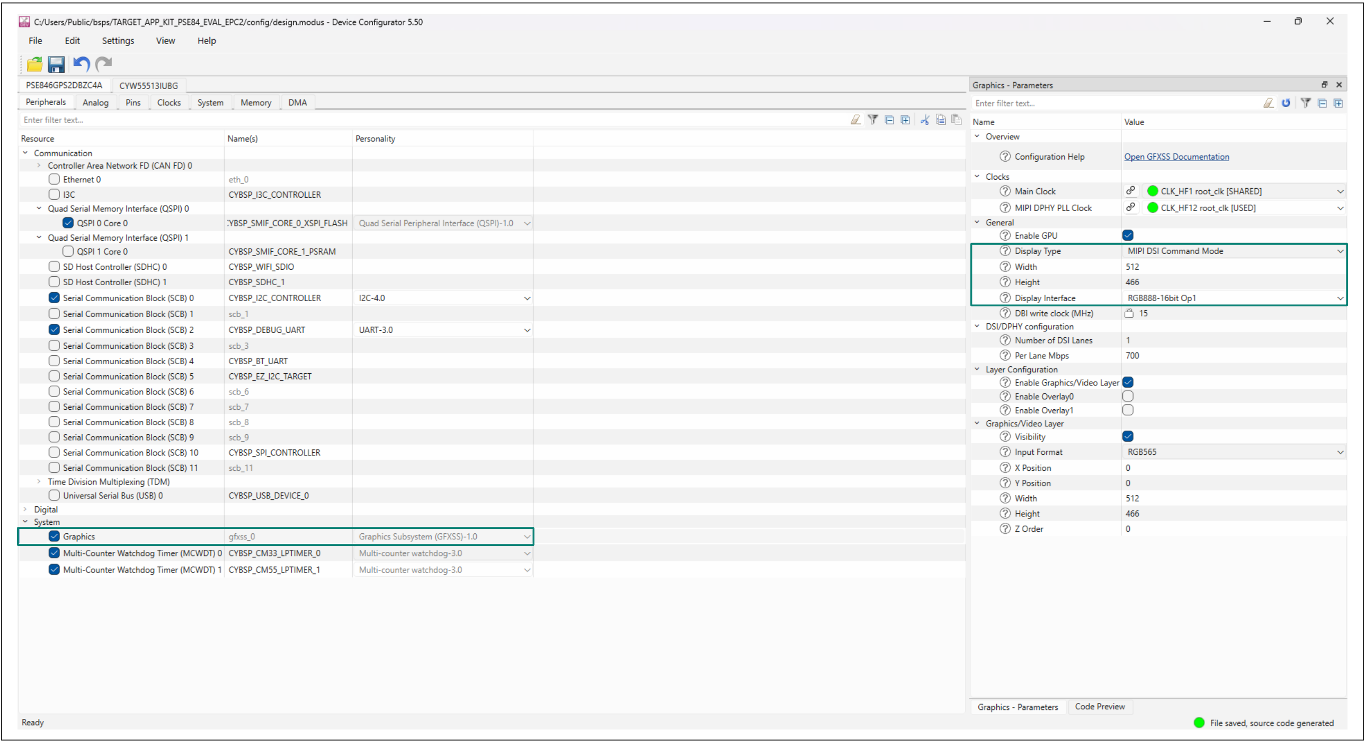
   The selected 1.43" panel fully supports RGB888 Option 1 in command mode, ensuring straightforward compliance with the panel's datasheet without requiring custom tight-packing logic

3. **Adequate bandwidth headroom**:

   Even though Option 1 is less efficient than Option 2, the actual pixel clock and DSI lane data rate for a small-format single-lane panel are well within the lane budget. Maintaining 20-30% margin avoids under-runs without the need for tight packing

4. **Reliability priority**:

   For this application, robustness of the link and ease of debugging outweigh the marginal bandwidth gain from Option 2. With Option 1, data alignment issues are less likely to cause visible artifacts


C:/Users/Public/bsps/TARGET_APP_KIT_PSE84_EVAL_EPC2/config/design.modus - Device Configurator 5.50

Resource	Name(s)	Personality
Communication		
Controller Area Network FD (CAN FD) 0		
☐ Ethernet 0	eth_0	
☐ I3C	CYBSP_I3C_CONTROLLER	
Quad Serial Memory Interface (QSPI) 0		
✓ QSPI 0 Core 0	:YBSP_SMIF_CORE_0_XSPI_FLASH	Quad Serial Peripheral Interface (QSPI)-1.0
Quad Serial Memory Interface (QSPI) 1		
☐ QSPI 1 Core 0	CYBSP_SMIF_CORE_1_PSRAM	
☐ SD Host Controller (SDHC) 0	CYBSP_WIFI_SDIO	
☐ SD Host Controller (SDHC) 1	CYBSP_SDHC_1	
✓ Serial Communication Block (SCB) 0	CYBSP_I2C_CONTROLLER	I2C-4.0
☐ Serial Communication Block (SCB) 1	scb_1	
✓ Serial Communication Block (SCB) 2	CYBSP_DEBUG_UART	UART-3.0
☐ Serial Communication Block (SCB) 3	scb_3	
☐ Serial Communication Block (SCB) 4	CYBSP_BT_UART	
☐ Serial Communication Block (SCB) 5	CYBSP_EZ_I2C_TARGET	
☐ Serial Communication Block (SCB) 6	scb_6	
☐ Serial Communication Block (SCB) 7	scb_7	
☐ Serial Communication Block (SCB) 8	scb_8	
☐ Serial Communication Block (SCB) 9	scb_9	
☐ Serial Communication Block (SCB) 10	CYBSP_SPI_CONTROLLER	
☐ Serial Communication Block (SCB) 11	scb_11	
Time Division Multiplexing (TDM)		
☐ Universal Serial Bus (USB) 0	CYBSP_USB_DEVICE_0	
Digital		
System		
✓ Graphics	gfxss_0	Graphics Subsystem (GFXSS)-1.0
✓ Multi-Counter Watchdog Timer (MCWDT) 0	CYBSP_CM33_LPTIMER_0	Multi-counter watchdog-3.0
✓ Multi-Counter Watchdog Timer (MCWDT) 1	CYBSP_CM55_LPTIMER_1	Multi-counter watchdog-3.0

Graphics - Parameters

Name	Value
Overview	
Configuration Help	Open GFXSS Documentation
Clocks	
Main Clock	CLK_HF1 root_clk [SHARED]
MIPI DPHY PLL Clock	CLK_HF12 root_clk [USED]
General	
Enable GPU	✓
Display Type	MIPI DSI Command Mode
Width	512
Height	466
Display Interface	RGB888-16bit Op1
DBI write clock (MHz)	15
DSI/DPHY configuration	
Number of DSI Lanes	1
Per Lane Mbps	700
Layer Configuration	
Enable Graphics/Video Layer	✓
Enable Overlay0	☐
Enable Overlay1	☐
Graphics/Video Layer	
Visibility	✓
Input Format	RGB565
X Position	0
Y Position	0
Width	512
Height	466
Z Order	0

File saved, source code generated

Figure 9 **Command mode display configuration**

***Note**:* *When using the RGB565 color format in Command Mode, set the horizontal width parameter to 512 instead of the actual active resolution of 466 pixels. This ensures a 128-byte stride alignment, which will be explained in detail in the Display controller configuration section.*

5 GPU configuration

GPU is a standalone functional block which does not have any direct interfacing to other modules (DC and MIPI DSI host) within graphics subsystem. Data exchange is software controlled via frame buffers in the system's video RAM which can be System SRAM or external PSRAM based on the application requirement. From a configuration standpoint, we simply enable the GPU in the device configurator for our application. It can also be disabled via the device configurator before the application build or dynamically enabled/disabled using following software APIs.

Enable GPU

```
cy_en_gfx_status_t Cy_GFXSS_Enable_GPU(GFXSS_Type *base, cy_stc_gfx_context_t *context)

Enable GPU for composition

Parameters
    base    Pointer to the graphics sub system base address
    context    context information used by the driver
Returns
    SUCESS/TIMEOUT status
```

Disable GPU

```
cy_en_gfx_status_t Cy_GFXSS_Disable_GPU(GFXSS_Type *base, cy_stc_gfx_context_t *context)

Disable GPU and let display controller refresh display based on already composed buffer stored
in frame buffer address

Parameters
    base    Pointer to the graphics sub system base address
    context    context information used by the driver
Returns
    SUCESS/TIMEOUT status
```

6 Display controller configuration

The Display Controller (DC) is the hardware block responsible for fetching image data from system memory, processing it according to display pipeline requirements, and outputting it to the target panel interface. Alternatively, it can drive standard I/O pins directly. It supports interfacing with panels having a total horizontal width of up to 1024 pixels and a total vertical height of up to 768 lines.

It supports operation in two distinct output modes:

- Video mode through the DPI interface to continuously drive HSync, VSync, and pixel clock signals for the video mode display panel, which is a 4.3-inch 800x480 display in our case
- Command mode via the DBI interface (including MIPI DSI in Command Mode) using DCS write transactions to update the panel's onboard frame buffer (GRAM), which is a 1.43-inch 466x466 display in our case

This unified architecture allows the same controller to be configured for either real-time continuous streaming (video mode) or asynchronous partial/full updates (command mode), depending on panel type and application needs.

The DC supports three independent display layers, composited into a final frame throughper-pixel alpha blending.

6.1 Three programmable DC layers

This section outlines the necessary steps to configure the DC layers, taking into account the display size and color depth. As previously mentioned, the DC features three independently configurable layers: one video/graphic layer and two overlay layers. The users can enable one, two, or all of the layers. By default, the graphics layer is enabled.

User must configure the following DC layer parameters separately for each enabled layer:

- Layer visibility
- Input format (RGB or YUV)
- Tiling type (tiled or linear)
- Window size and position
- Z-order of the layer
- Framebuffer start address (by default set to system SRAM (also known as SoCMEM) base address)
- YUV buffer start address (by default set to system SRAM base address)

Based on the Input format selection, the tiling type gets configured.

If the Input Format is RGB565/ARGB8888/ARGB1555/ARGB4444, then the Tiling type will be Linear. However, if the Input Format is Packed YUV422/YUV 420 semi-planar, then the Tiling type will be Tiled.

The user needs to configure the Windows X and Y positions according to the application requirements, ensuring that the Windows Width and Height are less than or equal to the resolution of the interfaced display to meet the application specifications.

By default, the framebuffer start address, or YUV buffer start address, will be configured to the system SRAM, also known as SoCMEM (used interchangeably throughout the document), base address. These parameters can be set manually by overriding the corresponding enabled graphic and overlay layer structures prior initializing the graphics subsystem as shown in the snippet below.

Layer configuration during graphics subsystem initialization

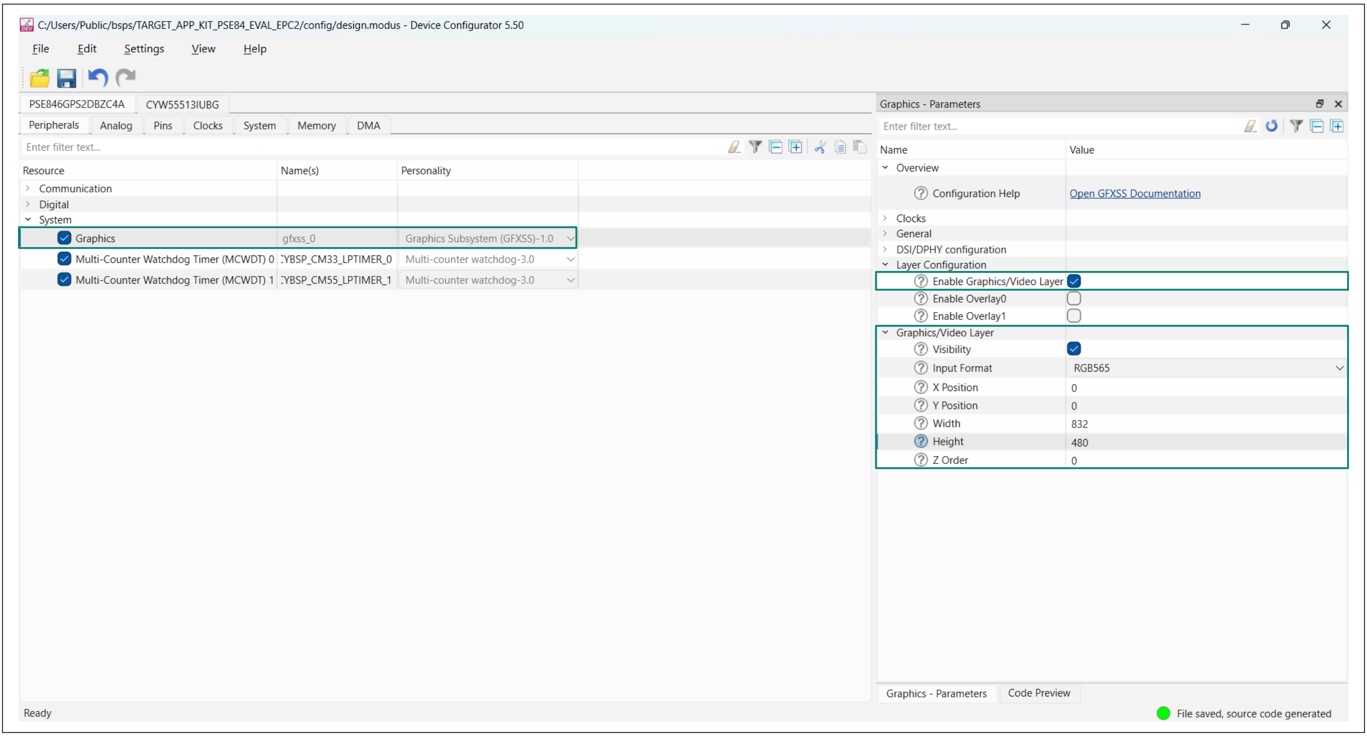
```
/* Graphic layer configuration structure */
cy_stc_gfx_layer_config_t GFXSS_graphics_layer =
{
    .layer_type = GFX_LAYER_GRAPHICS,
    .buffer_address = (gctADDRESS *)CY_SOCMEM_RAM_BASE,
    .uv_buffer_address = (gctADDRESS *)CY_SOCMEM_RAM_BASE,
    .input_format_type = vivRGB565,
    .tiling_type = vivLINEAR,
    .pos_x = 0,
    .pos_y = 0,
    .width = 832,
    .height = 480,
    .zorder = 0,
    .layer_enable = true,
    .visibility = true,
};

/* Display controller configuration structure */
cy_stc_gfx_dc_config_t GFXSS_dc_config =
{
    .gfx_layer_config = &GFXSS_graphics_layer,
    .ovl0_layer_config = &GFXSS_overlay0_layer,
    .ovl1_layer_config = &GFXSS_overlay1_layer,
    .display_type = GFX_DISP_TYPE_DSI_DPI,
    .display_format = vivD24,
    .display_size = vivDISPLAY_CUSTOMIZED,
    .display_width = 832,
    .display_height = 480,
};

/* Frame buffers, where MY_DISP_HOR_RES = 832, MY_DISP_VER_RES = 480 and color format = RGB565
(16bpp ~ 2 bytes) for 4.3-inch display */
CY_SECTION(".cy_gpu_buf") LV_ATTRIBUTE_MEM_ALIGN uint8_t disp_buf1[MY_DISP_HOR_RES *
                                               MY_DISP_VER_RES * 2];
CY_SECTION(".cy_gpu_buf") LV_ATTRIBUTE_MEM_ALIGN uint8_t disp_buf2[MY_DISP_HOR_RES *
                                               MY_DISP_VER_RES * 2];

/* Frame buffers used by GFXSS to render UI */
void *frame_buffer1 = &disp_buf1;
void *frame_buffer2 = &disp_buf2;

/* Set frame buffer address to the GFXSS configuration structure */
GFXSS_config.dc_cfg->gfx_layer_config->buffer_address = frame_buffer1;
GFXSS_config.dc_cfg->gfx_layer_config->uv_buffer_address = frame_buffer1;

/* Initialize Graphics subsystem as per the configuration */
(void)Cy_GFXSS_Init(GFXSS, &GFXSS_config, &gfx_context);
```

Additionally, calling the following listed `Set_FrameBuffer_YUV` or `Set_FrameBuffer` APIs for the respective graphic or overlay layer will set the framebuffer or YUV buffer start address parameters and render the frame to the display.

APIs to set frame and YUV buffers

```
cy_en_gfx_status_t Cy_GFXSS_Set_FrameBuffer_YUV(GFXSS_Type *base, uint32_t* y_buffer, uint32_t*
uv_buffer, cy_stc_gfx_context_t *context)

Sets Video/Graphics layer YUV buffer addresses

Parameters
    base         Holds the base address of the Graphics block registers
    y_buffer     Pointer to the Y buffer address to be used by Display Controller for
transferring frame
    uv_buffer    Pointer to the UV buffer address to be used by Display Controller for
transferring frame
    context      Pointer to the graphics config structure base address
Returns
    CY_GFX_SUCCESS/CY_GFX_BAD_PARAM

Note: YUV Framebuffer base address and stride for linear data should be 64-byte aligned.

cy_en_gfx_status_t Cy_GFXSS_Set_FrameBuffer(GFXSS_Type *base, uint32_t* gfx_layer_buffer,
cy_stc_gfx_context_t *context)

Sets Video/Graphics layer buffer address

Parameters
    base              Holds the base address of the Graphics block registers
    gfx_layer_buffer  Pointer to the frame buffer address to be used by Display Controller for
transferring frame
    context           Pointer to the graphics config structure base address
Returns
    CY_GFX_SUCCESS/CY_GFX_BAD_PARAM

Note: Framebuffer base address and stride for linear data should be 128-byte aligned.

cy_en_gfx_status_t Cy_GFXSS_Set_Overlay0_YUV(GFXSS_Type *base, uint32_t* y_buffer, uint32_t*
uv_buffer, cy_stc_gfx_context_t *context)
cy_en_gfx_status_t Cy_GFXSS_Set_Overlay0(GFXSS_Type *base, uint32_t* overlay0_buffer,
cy_stc_gfx_context_t *context)
cy_en_gfx_status_t Cy_GFXSS_Set_Overlay1(GFXSS_Type *base, uint32_t* overlay1_buffer,
cy_stc_gfx_context_t *context)
```

Following snapshot shows DC layer parameters configuration using Device Configurator for 4.3-inch 800x480 display.


C:/Users/Public/bsps/TARGET_APP_KIT_PSE84_EVAL_EPC2/config/design.modus - Device Configurator 5.50

Resource	Name(s)	Personality
Communication		
Digital		
System		
✓ Graphics	gfxss_0	Graphics Subsystem (GFXSS)-1.0
✓ Multi-Counter Watchdog Timer (MCWDT) 0	:YBSP_CM33_LPTIMER_0	Multi-counter watchdog-3.0
✓ Multi-Counter Watchdog Timer (MCWDT) 1	:YBSP_CM55_LPTIMER_1	Multi-counter watchdog-3.0

Graphics - Parameters

Name	Value
Overview	
Configuration Help	Open GFXSS Documentation
Clocks	
General	
DSI/DPHY configuration	
Layer Configuration	
Enable Graphics/Video Layer	✓
Enable Overlay0	☐
Enable Overlay1	☐
Graphics/Video Layer	
Visibility	✓
Input Format	RGB565
X Position	0
Y Position	0
Width	832
Height	480
Z Order	0

Figure 10 **DC layer configuration**

Similarly, for 1.43-inch display, Width is set to 512 and Height as 466 while keeping Input Format as RGB565.

***Note**:* *Overlay1 layer does not support the tiled mode.*

6.2 RLAD decoder

Run-Length Adaptive Dithering (RLAD) is a proprietary image compression format optimized for decoding hardware with low complexity and small design size. It supports both lossless and lossy compression and allows for the decompression of standard Run-Length encoded images according to TGA specifications.

The encoding is designed to be carried out by the RLAD encoder offline, while the RLAD decoder block is integrated into the display controller, which can be enabled for only one display layer to perform on-the-fly decoding.

The RLAD use case is demonstrated in the PSOC™ Edge Graphics using RLAD code example. See the README for more information about this code example.

6.2.1 Use case and application benefits

- Its main purpose is to store pre-rendered image assets such as static backgrounds, splash screens, and UI skins in compressed form, then decode and merge them with dynamic content on the fly
- In addition to image compression, RLAD can be used for RGB packing. For example, RGB888 or RGB666 images require DC to configure a 32 bpp frame buffer layout, despite the actual pixel data being only 24 and 18 bpp, respectively
- Example: Smartwatch face with moving hands, date, and other information being dynamic foreground
  - Smartwatch background stored in compressed form and decoded on the fly
  - For a 400x400 16 bpp pixel background image, compression will reduce image by 33%; reduces memory requirement by over 200 KB (400 KB for 32 bpp)

6.2.2 RLAD encoder tool

This tool allows to encode any PNG image asset into RL/RLA/RLAD/RLAD_UNIFORM format offline. It generates a header file (.h) corresponding to the compressed image.

The tool takes three inputs:

1. Input file name (png only):file to be compressed. e.g., duck.png, selfie.png
2. Output file name(.h): name of a header file to be created. e.g., duck.h, selfie.h
3. Mode of compression: RL/RLA/RLAD/RLAD_UNIFORM

The tool is distributed as part of the mtb-example-psoc-edge-gfx-rlad code example and can be used for any graphics application to leverage RLAD. Refer RLAD encoder readme for more information on its usage.

6.2.3 RLAD decoder configuration

By default, the RLAD decoder is disabled: the memory-to-display data stream is passed through unchanged (bypass mode) and all the RLAD registers remain in their reset state.

To use RLAD, the user must explicitly configure and enable the decoder in the application by following these steps.

1. Set up the RLAD decoder configuration structure (`cy_stc_gfx_rlad_cfg_t`) shown below for one of the available DC layers

   RLAD configuration structure

```
typedef struct
{
  cy_en_gfx_layer_type_t      layer_id;             /**<  Layer ID */
  uint32_t                    image_width;          /**<  Width of the image in number
of pixels minus one */
  uint32_t                    image_height;         /**<  Height of the image in number
of pixels minus one */
  uint32_t                    compressed_image_size; /**<  Size of the the encoded image
in number of 32-bit words minus one. Defined in image header file */
  uint32_t                    *image_address;       /**<  Image address */
  cy_en_gfx_rlad_comp_mode_t  compression_mode;     /**<  RLAD compression mode */
  cy_en_gfx_rlad_fmt_t        rlad_format;          /**<  RLAD display format */
  bool                        enable;               /**<  RLAD state */
} cy_stc_gfx_rlad_cfg_t;
```

- layer_id: Select the display layer (`cy_en_gfx_layer_type_t`) that will receive decoded pixels. Only one layer should have RLAD decoding enabled at a time

  Layer type

```
/* Layer type */
typedef enum {
    GFX_LAYER_GRAPHICS,   /**< Graphics layer */
    GFX_LAYER_OVERLAY0,   /**< Overlay 0 layer */
    GFX_LAYER_OVERLAY1,   /**< Overlay 1 layer */
} cy_en_gfx_layer_type_t;
```

- image_width: Width of the resultant uncompressed image in number of pixels minus one. The DC layer must be setup for the same width

- image_height: Height of the resultant uncompressed image in number of lines minus one. The DC layer must be setup for the same height
- compressed_image_size: Size of the encoded image in number of 32-bit words minus one. The buffer must be readable to the subsequent 128 bytes boundary
- image_address: Base address of the encoded image buffer in multiple of 128 bytes. The DC layer must be configured to the same base address and with the smallest possible stride value (aligned to 128 bytes)
- compression_mode: Encoding/compression mode (`cy_en_gfx_rlad_comp_mode_t`) of the source image

  RLAD compression mode

```
/* RLAD compression mode */
typedef enum
{
     CY_GFX_RLAD_MODE_RLAD,           /**< 'Run Length Adaptive Dithering'
compression. Lossy image compression type. */
     CY_GFX_RLAD_MODE_RLAD_UNIFORM,   /**< 'Run Length Adaptive Dithering' compression
with uniform bits per pixel. Lossy image compression type. */
     CY_GFX_RLAD_MODE_RLA,            /**< 'Run Length Adaptive' compression. Lossless
image compression type. */
     CY_GFX_RLAD_MODE_RL              /**< 'Run Length Encoded' compression. Lossless
image compression type. */
} cy_en_gfx_rlad_comp_mode_t;

Note: The encoded image data is stored as a stream of 32-bit words in memory. RLA
and RLAD data fields are in little endian order within these words (the first field
starts in the MSBits of a word). Standard RL encoding uses big endian. This is
distinct from the byte endianness of the 32-bit words in memory, which is a system
property and does not affect the encoding/decoding process
```

- rlad_format: Color format of the uncompressed image. The RLAD block converts the configured format after decompression to ARGB8888, which is the format the DC layer has to be configured for (in linear order and with component swizzle ARGB). Alpha value is set to 255 for RGB and GRAY formats

RLAD image format

```
/* RLAD image format */
typedef enum
{
     CY_GFX_RLAD_FMT_ARGB4444,
     CY_GFX_RLAD_FMT_ARGB1555,
     CY_GFX_RLAD_FMT_RGB565,
     CY_GFX_RLAD_FMT_ARGB8888,
     CY_GFX_RLAD_FMT_RGB888,
     CY_GFX_RLAD_FMT_RGB666,
     CY_GFX_RLAD_FMT_RGB444,
     CY_GFX_RLAD_FMT_GRAY8,
     CY_GFX_RLAD_FMT_GRAY6,
     CY_GFX_RLAD_FMT_GRAY4
} cy_en_gfx_rlad_fmt_t;
```

- enable: Enable/disable the RLAD decoder

2. Set the image configuration for RLAD during graphics subsystem initialization by calling the `Cy_GFXSS_Init` API. Alternatively, explicitly use the `Cy_GFXSS_RLAD_SetImage` API, followed by `Cy_GFXSS_RLAD_Enable`

3. After RLAD decoder is configured, render the RLAD encoded image frame to the target layer (as per layer_id) using `Cy_GFXSS_Set_FrameBuffer` or `Cy_GFXSS_Set_Overlay0/1` APIs; the decoder streams decoded pixels on-the-fly into the configured layer without additional copying

4. Once the frame rendering is complete, RLAD decoder can be disabled using `Cy_GFXSS_RLAD_Disable` API

The following snippet shows rendering a 800x480 32-bit image encoded in RLAD format using PSOC™ Edge MCU. For more information, see mtb-example-psoc-edge-gfx-rlad code example.

RLAD

```
GFXSS_Type* base = (GFXSS_Type*) GFXSS;
cy_stc_gfx_context_t gfx_context;

/* RLAD configuration */
cy_stc_gfx_rlad_cfg_t rlad_cfg =
{
    .layer_id              = GFX_LAYER_GRAPHICS,           /* Graphics Layer ID */
    .image_width           = (IMG_PTR_WIDTH - 1U),         /* Image width */
    .image_height          = (IMG_PTR_HEIGHT - 1U),        /* Image height */
    .compressed_image_size = (SIZE_IN_WORDS - 1UL),        /* compressed image size */
    .image_address         = (uint32_t*)gfx_buff,          /* Image address */
    .compression_mode      = CY_GFX_RLAD_MODE_RLAD,        /* RLAD compression mode */
    .rlad_format           = CY_GFX_RLAD_FMT_RGB888,       /* RLAD display format */
    .enable                = 1,                            /* Enabled */
};

/* Prior rendering graphics subsystem and display are configured */

/* Copy the stored image asset to ram */
memcpy(gfx_buff, img_ptr, sizeof(img_ptr));

/* Set RLAD config and enable RLAD */
Cy_GFXSS_RLAD_SetImage(base, &rlad_cfg, &gfx_context);
Cy_GFXSS_RLAD_Enable(base, &gfx_context);

/* Set frame address as base address of compressed image */
Cy_GFXSS_Set_FrameBuffer(base, rlad_cfg.image_address, &gfx_context);
```

***Note**: The RLAD configuration must not be changed again before the frame-complete interrupt. Besides causing an inconsistent setup for a display frame, this can lead to data corruption in the active configuration.*

7 MIPI DSI host and D-PHY configuration

The MIPI DSI Host sits between the MCU's display controller and the panel, converting rendered pixels and control commands into DSI packets that the D-PHY transmits over one or more serial lanes. The DSI host transmits the DSI packets in the form of parallel data to the D-PHY through the PHY protocol interface (PPI). The D-PHY serializes the packets and sends them across the serial link.

The DSI host has two types of system interfaces:

1. DC interface
   - In Video mode, DSI host receives a continuous pixel stream via DPI (DPI-24/18/16), packetizes per the selected transfer type (non-burst pulses/events or burst), and transmits in High-Speed with the required clock-lane behavior
   - In Command mode, it ingests panel commands and windowed pixel bursts via DBI, using Low-Power packets for control and High-Speed long writes for image updates
2. APB interface
   - The DSI host also supports an APB Slave Generic Interface to send DCS or manufacturer-specific commands to a display in DBI and DPI mode. In particular, this includes get commands with read cycles, which are not supported by the DC in graphics subsystem

Clocking is provided by a D-PHY PLL locked to the 24 MHz reference, generating the HS lane bit rate and the lane byte clock that drive the Host's packetizer and FIFOs.

Both Video and Command mode are already described above. Next, the following will focus on configurations pertaining to MIPI DSI and D-PHY.

7.1 DSI host PHY parameters

This section presents the D-PHY parameters controlled from the DSI host.

1. **Number of lanes**

   The DSI host provides a scalable architecture using one or two data lanes. Each DSI lane has a maximum of 1.5 Gbps data rate, for a total 3 Gbps rate in dual data lane mode. Even though the D-PHY supports 2 lanes but the panel dictates what we should actually use. Some panels require a single lane, others support two lanes with their own per-lane maximum bit rates. User must match the panel's lane capability and never exceed its per-lane limit, even if the host can drive faster.

   Both the 800x480 4.3-inch video mode and the 466x466 1.43-inch command mode display support only a single lane. Accordingly, set the `Number of DSI Lanes` to 1 under the graphics tab in the device configurator.
2. **Per lane rate**

   The per-lane high-speed bit rate defines how fast each D-PHY data lane transmits payload to the panel. It must always be within the panel's per-lane limit and the DSI host's capability. In this setup, the host supports up to 1.5 Gbps per lane, but the panels constrain the configuration: the 4.3-inch display is single-lane with a maximum of 850 Mbps per lane, and the 1.43-inch display is single-lane with a maximum of 700 Mbps per lane. Users should never exceed these panel caps, even if the host can drive faster.

   - **Per lane rate configuration for the video mode panel**
     - When `Use Max Per Lane Mbps` option is checked

       The DSI host sets the per lane rate (`per_lane_mbps`) to the maximum (e.g., 850 Mbps) specified in the `Max Per Lane Mbps` field in the Device Configurator, regardless of the actual payload determined by video timings, target refresh, DPI format, and lane count. This ensures a deterministic HS duty cycle and provides extra margin, but it may consume more power than necessary

- When `Use Max Per Lane Mbps` option is unchecked

  The DSI host calculates the lane rate based on the configured video timings and format, disregarding the configured maximum lane rate, and then operates only as fast as necessary to transmit the payload. This approach is power-efficient and minimizes the HS duty cycle, but it is important to maintain headroom and stay within the panel's per-lane limit

```
For 4.3-inch display in 16-bit (RGB565) color mode with DPI-24 interface:

per lane rate (per_lane_mbps) = H_TOTAL × V_TOTAL × Total BPP x Frame Rate /
(Number of Lanes)

Total Horizontal Period (H_TOTAL) = HSync width + HBP + Display active width +
HFP [in pixels] = 10 + 20 + 832 + 210 = 1072
Total Vertical Period (V_TOTAL) = VSync width + VBP + Display active height + VFP
[in lines] = 5 + 20 + 480 + 20 = 525
Total BPP = 3 bytes/pixel = 24 (DPI-24)
Frame Rate = 60 Hz
Number of Lanes = 1

per lane rate (per_lane_mbps) = 810 Mbps
```

- **Per lane rate configuration for the command mode panel**
  - The DSI host considers the entered value of `Per Lane Mbps` field and drivers the display accordingly. In case of 1.43-inch command mode display, as per panel specification maximum of 700 Mbps per lane is configured
  - The resulting per-lane high-speed bit rate determines our lane byte clock and, by extension, constrains the maximum DBI ingress clock we can use to feed the DSI TX FIFO. DBI clock determines how fast the display controller can feed commands and pixel data into the DSI host's transmit FIFO in command mode. If the DBI clock is too low, the FIFO starves and bursts take too long, risking missed update deadlines and visible stutter or tearing.

    For example, in case of 1.43-inch command mode the DBI clock gets auto calculated as follows:

```
Lane rate = 700 Mbps (single lane)
Lane byte clock = Lane rate/8 = 87.5 MHz (Lane byte clock >= 3 x Rx escape clock)
DBI clock (max 37.5 MHz) <= Lane byte clock/4 = 21.875 MHz within the limit
```

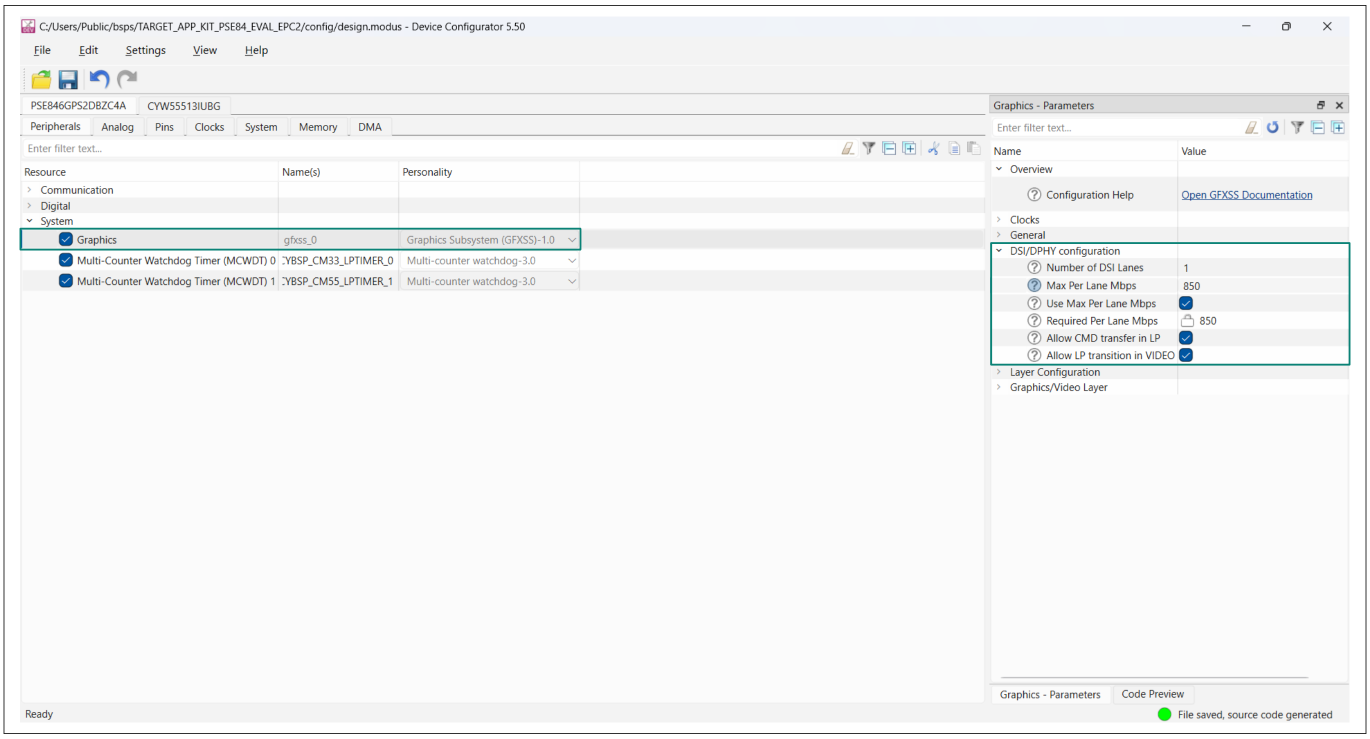
3. **Video mode configuration flags**

   These flags control how the DSI host manages power and link behavior while streaming the video. They let the host send control commands in low-power (LP) mode and drop the link back to LP during vertical and horizontal blanking intervals, reducing the high-speed (HS) duty cycle without disturbing active video. Configuring them correctly can lower power, provided the panel supports these behaviors.

   - **Enable command transmission in low-power mode**
     - Purpose: Send short DCS commands (Display On/Off, Sleep In/Out, Set Pixel Format, TE enable, brightness updates) using LP packets instead of HS, so the clock lane and data lanes need not remain in HS for control traffic

  - Significance: Reduces dynamic power and avoids unnecessary HS toggling; commands are more tolerant of timing and typically specified for LP
  - Why configure: Ensures control operations are reliable and energy-efficient while video streaming continues in HS for active lines
- **Return to low-power inside vertical/horizontal timings (non-continuous clock/LP video)**
  - Purpose: Transition the link from HS to LP during blanking (HFP/HSYNC/HBP and VFP/VSYNC/VBP) and resume HS only for active video. This shrinks HS duty cycle
  - Significance: Cuts average link power by limiting HS activity to the periods that actually carry pixels
  - Why configure: Beneficial for static or moderately animated UIs where blanking intervals are substantial; must be supported by the panel. Many video-mode panels require a continuous clock if so, do not enable LP video

In this use case, the 4.3-inch display panel supports non-continuous clock/LP video; that is why we enable the `Allow CMD transfer in LP` and `Allow LP transition in VIDEO` options along with `Number of DSI Lanes` set to 1, `Max Per Lane Mbps` as 850 and `Use Max Per Lane Mbps` option checked shown in Figure 11.


C:/Users/Public/bsps/TARGET_APP_KIT_PSE84_EVAL_EPC2/config/design.modus - Device Configurator 5.50

Resource	Name(s)	Personality
Communication		
Digital		
System		
✓ Graphics	gfxss_0	Graphics Subsystem (GFXSS)-1.0
✓ Multi-Counter Watchdog Timer (MCWDT) 0	:YBSP_CM33_LPTIMER_0	Multi-counter watchdog-3.0
✓ Multi-Counter Watchdog Timer (MCWDT) 1	:YBSP_CM55_LPTIMER_1	Multi-counter watchdog-3.0

Graphics - Parameters

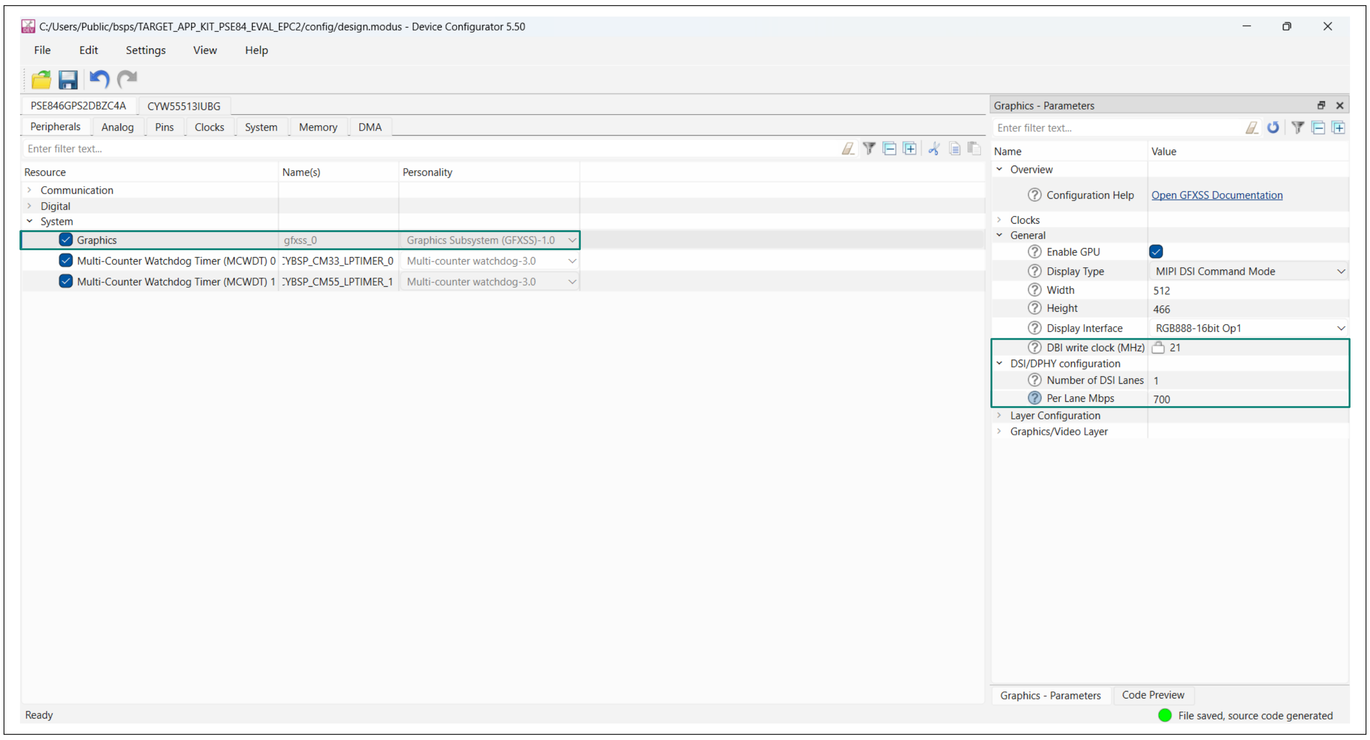
Name	Value
Overview	
Configuration Help	Open GFXSS Documentation
Clocks	
General	
DSI/DPHY configuration	
Number of DSI Lanes	1
Max Per Lane Mbps	850
Use Max Per Lane Mbps	✓
Required Per Lane Mbps	850
Allow CMD transfer in LP	✓
Allow LP transition in VIDEO	✓
Layer Configuration	
Graphics/Video Layer	

Ready

File saved, source code generated

Figure 11 **Video mode DSI/DPHY configuration**

Figure 12 shows DSI/DPHY configuration for 1.43-inch display.


C:/Users/Public/bsps/TARGET_APP_KIT_PSE84_EVAL_EPC2/config/design.modus - Device Configurator 5.50

Resource	Name(s)	Personality
Communication		
Digital		
System		
✓ Graphics	gfxss_0	Graphics Subsystem (GFXSS)-1.0
✓ Multi-Counter Watchdog Timer (MCWDT) 0	:YBSP_CM33_LPTIMER_0	Multi-counter watchdog-3.0
✓ Multi-Counter Watchdog Timer (MCWDT) 1	:YBSP_CM55_LPTIMER_1	Multi-counter watchdog-3.0

Graphics - Parameters

Name	Value
Overview	
Configuration Help	Open GFXSS Documentation
Clocks	
General	
Enable GPU	✓
Display Type	MIPI DSI Command Mode
Width	512
Height	466
Display Interface	RGB888-16bit Op1
DBI write clock (MHz)	21
DSI/DPHY configuration	
Number of DSI Lanes	1
Per Lane Mbps	700
Layer Configuration	
Graphics/Video Layer	

Ready
File saved, source code generated

Figure 12 **Command mode DSI/DPHY configuration**

8 Interrupts

The graphics subsystem exposes three independent, level-sensitive (active-high) interrupt outputs; one each from the GPU, the display controller (DC), and the MIPI DSI host. These interrupts signal composition, scan-out, and link events that are used to synchronize rendering, detect errors, and pace command-mode updates.

- GPU interrupt (`interrupt_gpu`)
  - Characteristics: One active-high, level-sensitive output; synchronous to clk_sys
  - Source: INTR.CORE from the GPU core. Typical causes include AXI error responses and other core events handled by the VGLite driver
  - Usage: Enable the GPU interrupt for "composition complete" cause so the VGLite IRQ handler can finalize operations and wake the rendering pipeline; error causes are logged and escalated
- Display controller interrupt (`interrupt_dc`)
  - Characteristics: One active-high, level-sensitive output; synchronous to clk_sys
  - Sources:
    - INTR.CORE (frame/scan-out events): Frame complete (last active pixel consumed), Display FIFO under-run (memory/bandwidth shortfall), DBI configuration error, AXI error response, software reset complete
    - INTR.ADDR0..3: Access complete for each configured input buffer address
    - INTR.RLAD_ERROR: Error while decoding compressed image data
  - Usage: Enable the DC interrupt to detect the "frame complete" event, allowing the application to schedule the next frame (performing buffer swaps and enqueuing the next composition). Additionally, monitor under-run and RLAD errors to help identify timing, bandwidth, or asset-related issues early
- MIPI DSI host interrupt (`interrupt_mipidsi`)
  - Characteristics: One active-high, level-sensitive output; synchronous to clk_sys
  - Sources:
    - INTR.CORE: DSI core status (refer to core status registers INT_ST0/INT_ST1 in the PSOC™ Edge E8x registers reference manual)
    - INTR.DBI_TE: Tearing Effect (TE) signaled by the display in DBI/command mode (per-line or per-frame), used to align partial updates and avoid tearing
    - INTR.DPI_HALT: Command transmission error in video mode—command was too large to fit blanking; pixel data was lost
  - Usage: In command mode, we can enable TE to pace HS bursts to safe points while in video mode, enable it to log/act on DPI_HALT to tighten command size/timing. Core error bits are monitored to detect link issues

The following snippet from smartwatch code example demonstrates all the relevant interrupt configurations and handling for video and command mode display.

GPU interrupt configuration and handling

```
/* GPU interrupt configuration */
const cy_stc_sysint_t gpu_int_config =
{
    .intrSrc      = GFXSS_GPU_IRQ,   /* 162 [Active] Interrupt from GPU */
    .intrPriority = GPU_INT_PRIORITY /* 3 as interrupt priority */
};

/*******************************************************************************
* Function Name: gpu_irq_handler
********************************************************************************
* Summary:
*  GPU interrupt handler which gets invoked when the GPU finishes composing
*  a frame. It clears the GPU interrupt and invokes VGLite IRQ handler.
*
* Parameters:
*  void
*
* Return:
*  void
*
*******************************************************************************/
static void gpu_irq_handler(void)
{
    Cy_GFXSS_Clear_GPU_Interrupt(base, &gfx_context);
    vg_lite_IRQHandler();
}

/* Intialize GPU interrupt */
result = (cy_rslt_t)Cy_SysInt_Init(&gpu_int_config, gpu_irq_handler);
if (CY_RSLT_SUCCESS != result)
{
    process_error(result, "GPU interrupt registration failed. STOP.");
}

/* Enable GPU interrupt */
Cy_GFXSS_Enable_GPU_Interrupt(base);

/* Enable GPU interrupt in NVIC */
NVIC_EnableIRQ(gpu_int_config.intrSrc);

/* Clear pending GPU interrupt in NVIC for first time */
NVIC_ClearPendingIRQ(gpu_int_config.intrSrc);
```

In case of video mode display, the user needs to sync the next frame transfer based on frame complete interrupt from DC while in case of command mode the next frame can be submitted to DC once DBI transfer to DSI host is complete. The following snippet shows the DC interrupt configuration and handling for the 4.3-inch video mode display use case.

DC interrupt configuration and handling

```
/* Display controller interrupt configuration */
const cy_stc_sysint_t dc_int_config =
{
    .intrSrc      = GFXSS_DC_IRQ,    /* 163 [Active] Interrupt from DC */
    .intrPriority = DC_INT_PRIORITY /* 3 as interrupt priority */
};

#if defined(W4P3INCH_DISP)
/*******************************************************************************
* Function Name: dc_irq_handler
********************************************************************************
* Summary:
*  Display Controller interrupt handler which gets invoked when the DC finishes
*  utilizing the current frame buffer.
*
* Parameters:
*  void
*
* Return:
*  void
*
*******************************************************************************/
static void dc_irq_handler(void)
{
    fb_pending = false;
    Cy_GFXSS_Clear_DC_Interrupt(base, &gfx_context);
}
#endif /* W4P3INCH_DISP */

/* Intialize Display Controller (DC) interrupt */
result = (cy_rslt_t)Cy_SysInt_Init(&dc_int_config, dc_irq_handler);
if (CY_RSLT_SUCCESS != result)
{
    process_error(result, "DC interrupt registration failed. STOP.");
}

/* Enable DC interrupt in NVIC */
NVIC_EnableIRQ(dc_int_config.intrSrc);

/* Clear pending DC interrupt in NVIC */
NVIC_ClearPendingIRQ(dc_int_config.intrSrc);
```

9 Functional analysis

This application note effectively demonstrates the end-to-end graphics pipeline for both high-performance and low-power operations using the smartwatch code example of the PSOC™ Edge E84 Evaluation Kit. The code example showcases a smartwatch GUI, utilizing a custom 1.43-inch round AMOLED (466×466) in MIPI DSI command mode. Additionally, the same use case is emulated on an out-of-box available 4.3-inch MIPI DSI video mode Raspberry Pi DSI display in absence of access to the 1.43-inch display. Due to differences in interface protocol and display hardware capabilities, the firmware flow has some variations, along with distinct application states and KPI measurements.

The firmware follows a dual-CPU, three-project structure: CM33 runs split secure (SPE) and non-secure (NSPE) projects, while CM55 executes the main graphics workload; all the three are placed in external QSPI flash and execute in XIP mode. To minimize idle power, CM55 uses `MCWDT1` as a low-power timer, enabling FreeRTOS tickless idle and allowing the device to enter Deep Sleep when the CPU is not actively rendering or servicing display events.

The smartwatch application operates in three states - high-performance, low-power, and ultra-low power. For the 1.43-inch round display, the application state is changed from high-performance to low-power and finally to ultra-low power after 30 seconds of inactivity. The application state can be brought back to high-performance by a simple touch activity. For the 4.3-inch display, the state transition is done by pressing the `USER BTN1 (SW2)` on the kit.

The high-performance and low-power optimization strategies outlined in the later sections are effectively implemented in this smartwatch application, followed by the performance evaluations across its different application states.

9.1 High-performance graphics optimization

This section summarizes the practical and measurable steps to raise frame rate and responsiveness while keeping power low. The focus will be on using the 2.5D GPU for blits/vector paths, placing double buffers in the fastest on-chip memory, preparing assets offline (pre-multiplied alpha and stride alignment), leveraging cache and ITCM, and exploiting partial rendering in command mode.

9.1.1 Use the 2.5D GPU effectively

The 2.5D GPU, driven through the VGLite library, accelerates raster operations (fills, blits, scaling) and vector primitives (paths, strokes) by executing a compact stream of GPU commands recorded and submitted by CPU. The user prepare surfaces and paths, issue `vg_lite_draw` (for vector paths) and `vg_lite_blit` (for image compositing), then flush/submit, allowing the GPU runs asynchronously while the DC scans the front buffer. With double buffering and interrupt-driven swaps, the CPU mostly orchestrates work and feeds commands rather than pushing pixels itself, yielding high frame rate at low CPU load.

Key points for efficient GPU usage with VGLite

- Command flow and synchronization
  - Initialize once, reuse resources; record commands per frame, then `vg_lite_flush` (non-blocking). Avoid `vg_lite_finish` in the hot path; rely on GPU/DC interrupts to swap the front and back frame buffers, rather than polling or blocking
  - Keep command lists small and batched (group similar draws/blits to reduce state changes such as blend or matrix updates)
- Blit (`vg_lite_blit`) best practices
  - Use pre-multiplied ARGB assets; select a blend mode like SRC_OVER that assumes pre-multiplication to avoid extra math and color fringes
  - Prefer formats that avoid conversion (e.g., RGB565 for UI, ARGB8888 for effects). Avoid per-draw format changes

  - Pack sprites into atlases to improve locality and reduce command/state churn; use scissor to limit writes to the damaged area
  - Choose filtering wisely: nearest for UI (speed), linear only when quality requires it
- Vector draw (`vg_lite_draw`) best practices
  - Pre-create and reuse paths/paints; avoid rebuilding complex paths every frame
  - Reuse transformation matrices; minimize per-draw matrix changes
  - Use scissor/clip rectangles to constrain rasterization to the damaged region

  Considering smartwatch code example, the GPU is enabled in "High-performance state". It can be enabled via Device Configurator or using PDL APIs as shown in section GPU configuration. Since this code example is leveraging LVGL graphics framework, to enable GPU support in the framework set `LV_USE_DRAW_VG_LITE` macro in lvgl configuration file (`lv_conf.h`) and additionally patch following platform dependent three files to use GPU effectively using VGLite APIs

  - `lv_draw_vg_lite_img.c`: Provides implementation of LVGL functions to perform GPU accelerated image rendering via VGLite driver
  - `lv_draw_vg_lite.c`: Provides implementation of LVGL functions to use GPU draw unit for raster and vector operations
  - `lv_vg_lite_utils.c`: Provides implementation of LVGL utility APIs ported to VGLite library
- Use alpha pre-multiplied ARGB images

  As pre-multiply operation is not supported by GPU, it is performed by CPU which is time consuming. So pre-multiply image offline and then use in the application. LVGL image converter python utility can be used to generate alpha pre-multiplied image assets.

  Following command shows an example of generating alpha pre-multiplied image using png source.

```
python LVGLImage.py --ofmt C --cf ARGB8888 --premultiply --align 64 -o ./output --name
ui_img_brightness_png brightness.png
```

- Convert images to expected stride alignments

  Converting images to expected stride alignment before hand saves additional calculations and memcpy at the time of opening/decoding the image. For example, btn_bg_3 image asset has 30x41 dimension with 32-bit depth. So, the actual stride = width * bpp = 30 * 4 = 120 (not 64 byte aligned). The following example shows the command generating stride aligned image asset. For more information on the image stride alignment, see section 5.4.2 Image buffer alignment requirement in VGLite API reference manual.

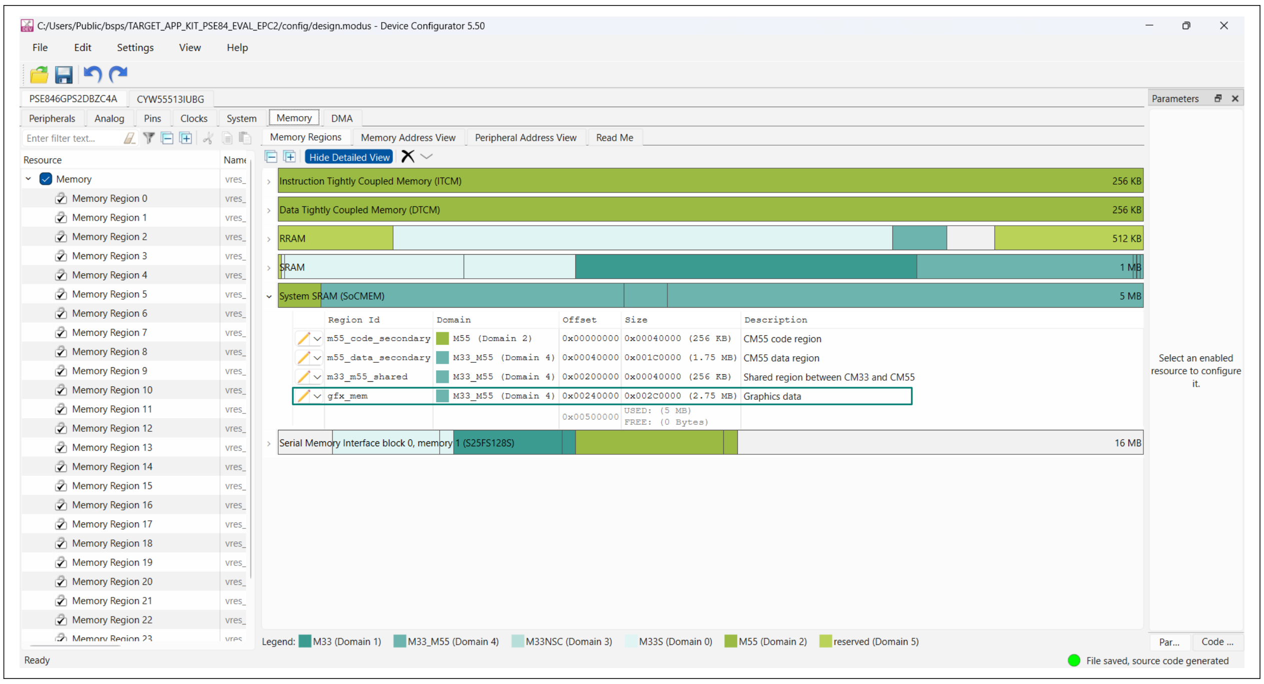
```
python LVGLImage.py --ofmt C --cf ARGB8888 --align 64 -o ./output --name
ui_img_btn_bg_3_png btn_bg_3.png

Resultant stride is 128
```

9.1.2 Memory strategy for graphics

For high-frame-rate, low-latency graphics, where the display controller must fetch pixels continuously while the GPU renders, memory placement and buffer policy are as critical as GPU usage. The core recommendation for the placement of frame buffer, code, and data memory is as follows:

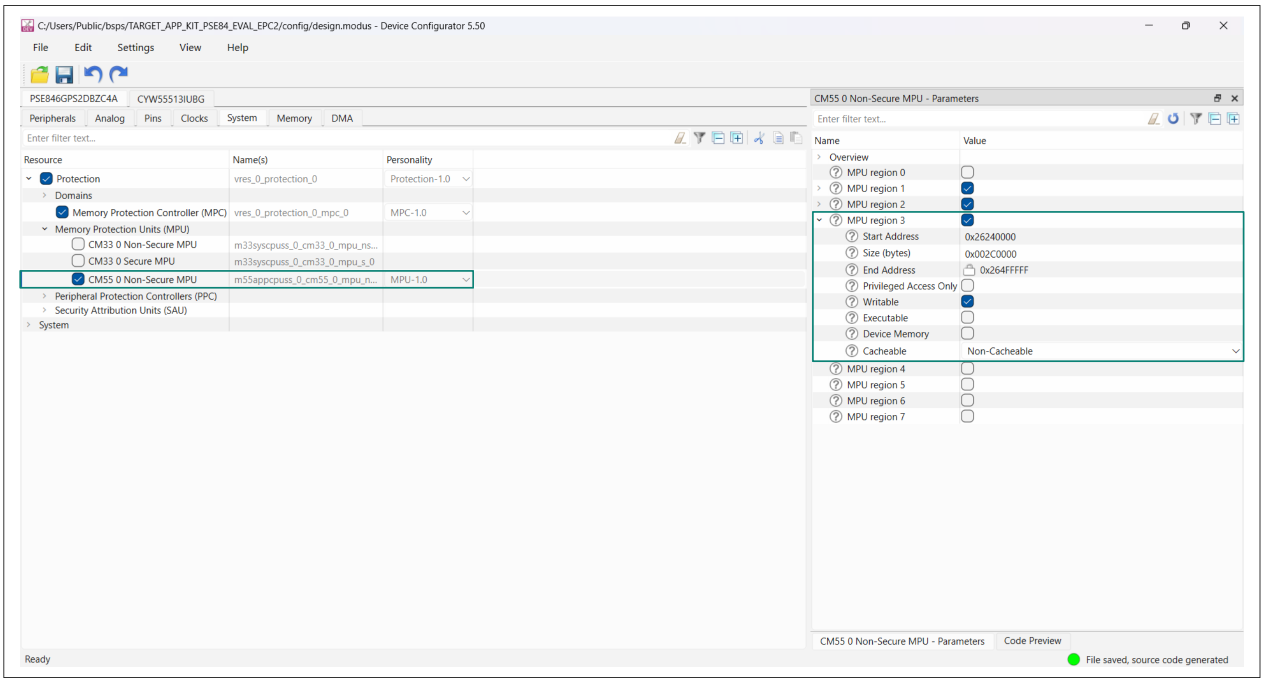
- Allocate two full-screen buffers (double buffering) in the fastest on-chip RAM (e.g., system SRAM) compared to external RAM (e.g. PSRAM) based on the memory availability. This minimizes read latency for the display controller and write latency for the GPU/CPU

- Run the zero wait state system SRAM at its highest specifications using any of the available PLLs to drive CLK_HF2 at 300 MHz
- Allocate frame buffers and the GPU memory in non-cacheable system SRAM region as shown in Figure 13 and Figure 14.


C:/Users/Public/bsps/TARGET_APP_KIT_PSE84_EVAL_EPC2/config/design.modus - Device Configurator 5.50
File
Edit
Settings
View
Help
PSE846GPS2DBZC4A
CYW55513IUBG
Peripherals
Analog
Pins
Clocks
System
Memory
DMA
Memory Regions
Memory Address View
Peripheral Address View
Read Me
Hide Detailed View
Memory
Memory Region 0 … Memory Region 23
Instruction Tightly Coupled Memory (ITCM) 256 KB
Data Tightly Coupled Memory (DTCM) 256 KB
RRAM 512 KB
SRAM 1 MB
System SRAM (SoCMEM) 5 MB
Serial Memory Interface block 0, memory 1 (S25FS128S) 16 MB
Legend: M33 (Domain 1) M33_M55 (Domain 4) M33NSC (Domain 3) M33S (Domain 0) M55 (Domain 2) reserved (Domain 5)
Parameters
Select an enabled resource to configure it.
Ready
File saved, source code generated

Region Id	Domain	Offset	Size	Description
m55_code_secondary	M55 (Domain 2)	0x00000000	0x00040000 (256 KB)	CM55 code region
m55_data_secondary	M33_M55 (Domain 4)	0x00040000	0x001C0000 (1.75 MB)	CM55 data region
m33_m55_shared	M33_M55 (Domain 4)	0x00200000	0x00040000 (256 KB)	Shared region between CM33 and CM55
gfx_mem	M33_M55 (Domain 4)	0x00240000	0x002C0000 (2.75 MB)	Graphics data
		0x00500000	USED: (5 MB) FREE: (0 Bytes)	

Figure 13 **Frame buffer and graphics memory configuration**


C:/Users/Public/bsps/TARGET_APP_KIT_PSE84_EVAL_EPC2/config/design.modus - Device Configurator 5.50
File
Edit
Settings
View
Help
PSE846GPS2DBZC4A
CYW55513IUBG
Peripherals
Analog
Pins
Clocks
System
Memory
DMA
CM55 0 Non-Secure MPU - Parameters
Code Preview
Ready
File saved, source code generated

Resource	Name(s)	Personality
Protection ✓	vres_0_protection_0	Protection-1.0
Domains		
Memory Protection Controller (MPC) ✓	vres_0_protection_0_mpc_0	MPC-1.0
Memory Protection Units (MPU)		
CM33 0 Non-Secure MPU	m33syscpuss_0_cm33_0_mpu_ns...	
CM33 0 Secure MPU	m33syscpuss_0_cm33_0_mpu_s_0	
CM55 0 Non-Secure MPU ✓	m55appcpuss_0_cm55_0_mpu_n...	MPU-1.0
Peripheral Protection Controllers (PPC)		
Security Attribution Units (SAU)		
System		

CM55 0 Non-Secure MPU - Parameters

Name	Value
Overview	
MPU region 0	☐
MPU region 1	✓
MPU region 2	✓
MPU region 3	✓
Start Address	0x26240000
Size (bytes)	0x002C0000
End Address	0x264FFFFF
Privileged Access Only	☐
Writable	✓
Executable	☐
Device Memory	☐
Cacheable	Non-Cacheable
MPU region 4	☐
MPU region 5	☐
MPU region 6	☐
MPU region 7	☐

Figure 14 **MPU configuration for graphics memory region**

The following snippet from the smartwatch code example illustrates the VGLite heap memory region designated for GPU and frame buffer allocation.

```
/* Heap memory for VGLite to allocate memory for buffers, command, and tessellation buffers
located in non-cacheable region */
CY_SECTION(".cy_gpu_buf") uint8_t vglite_mem[VGLITE_HEAP_SIZE] = { 0xFF };

/* Frame buffers located in non-cacheable region */
CY_SECTION(".cy_gpu_buf") LV_ATTRIBUTE_MEM_ALIGN uint8_t disp_buf1[MY_DISP_HOR_RES *
MY_DISP_VER_RES * 2];
CY_SECTION(".cy_gpu_buf") LV_ATTRIBUTE_MEM_ALIGN uint8_t disp_buf2[MY_DISP_HOR_RES *
MY_DISP_VER_RES * 2];
```

***Note**:* *Frame buffers can be in the cacheable system SRAM region. Invalidate the cache after rendering frames with the `SCB_CleanDCache()` API to slightly improve CPU usage*

- Align the stride and base address of each frame buffer to a 128-byte boundary to match AXI burst sizes and cache lines. For more information, refer to the section 35.4.3.4 Alignments and maximum read OTs in the PSOC™ Edge E8x architecture reference manual
- Place performance-critical routines (e.g., ISR handlers, blit/dispatch glue) in ITCM to minimize latency and instruction fetch stalls, while frequently used image assets should be placed in system SRAM for faster fetching subject to memory availability

  The following snippet shows designated macros used in LVGL to relocate performance critical sections in low latency memories:

```
/* Following macros are defined in lv_conf.h */

/* Compiler prefix for a big array declaration in RAM */
#define LV_ATTRIBUTE_LARGE_RAM_ARRAY CY_SECTION(".cy_socmem_data")

/* Place performance critical functions into a faster memory (e.g RAM) */
#define LV_ATTRIBUTE_FAST_MEM CY_SECTION(".cy_itcm")


/* IMAGE DATA: assets/brightness.png located in system SRAM (SoCMEM) */
LV_ATTRIBUTE_LARGE_RAM_ARRAY LV_ATTRIBUTE_MEM_ALIGN uint8_t ui_img_brightness_png_data[] =
{...}

/* disp_flush function located in ITCM ensuring periodic refresh/rendering of the screen */
void LV_ATTRIBUTE_FAST_MEM disp_flush(lv_display_t *disp, const lv_area_t *area, uint8_t
*px_map)
{
}
```

- Utilize RLAD encoded pre-rendered image assets in system SRAM that can be decoded directly into a display layer, thereby reducing storage and bus bandwidth, eliminating runtime repacks, and decreasing CPU load for smoother, higher-FPS rendering

  With 3rd party graphics framework such as in case of LVGL, we can plugin image decoder to stream LZ4-compressed assets from flash directly into stride-aligned system SRAM buffers (RGB565/ARGB8888, pre-multiplied if needed), optionally via DMA, reducing storage and bus traffic while accelerating load time and on-screen updates

9.2 Low-power graphics optimization

Low-power graphics optimization focuses on reducing energy per frame by adapting the pipeline to the content and exploiting MIPI DSI + display-controller features. This section outlines practical techniques to minimize dynamic power without compromising the user experience. The overall strategy is to decrease the clock rates and duty cycles, avoid unnecessary pixels, leverage MIPI DSI low-power features, and adapt the rendering path to content.

Recommended key strategies for low-power graphics rendering includes:

- Choosing lower bpp (RGB565)
- Switch system clocks to ULP mode (50 MHz @ 0.7 core voltage), when the display timing permits. Directly use IHO and disable PLLs
- Lower refresh rate where acceptable (for example, 60 → 1 Hz in case of always-on screen); enable LP return in blanking if the panel supports non-continuous clock
- Skip redraws when content is unchanged; rely on stable front buffer
- For small UI changes, render with CPU and DMA fills/blits, and keep the GPU clock-gated
- System can enter deep sleep state between two refresh cycles when the lower refresh state is set in case of command mode displays. Use tickless idle to enter deep sleep promptly
  - In CPU sleep mode, DC as an AXI bus master can continue fetching data from system SRAM, octal-SPI PSRAM and drive the display
- Use pre-computed assets and avoid per-frame tessellation in low-power scenes
- Use on-chip system SRAM for frame buffer rather than PSRAM can save power in VDDD domain. In addition, the frame rate is limited by the bandwidth of PSRAM
- Enable DSI ULPM mode for both clock and data lanes during extended idle/static screens. Also, activate this mode during deep sleep
- Reduce screen brightness to an acceptable level to save power

10 Performance evaluation

Power consumption and frame rate are crucial factors to consider when assessing performance during frame rendering. These two parameters will be measured across high-performance and low-power graphics use cases demonstrated in the smartwatch code example.

10.1 Power domains and current measurement jumpers

The PSOC™ Edge MCU supports two supply configurations and three core performance modes. It can operate either from a regulated power supply (VDDD = 1.8 V ±5%) or with an external battery power supply (VBAT = 3.3 V ±5% with VDDD = 1.8 V ±5%). The core logic voltage is dynamically managed by an on-chip buck regulator with three modes: Ultra low power (ULP = 0.7 V), Low power (LP = 0.8 V), and High performance (HP = 0.9 V), allowing the applications to trade off performance against energy.

The PSOC™ Edge MCU uses four distinct power domains. On the KIT_PSE84_EVAL board, jumpers are allocated for power measurement across these domains, enabling to place a digital multimeter in series in place of jumpers. Table 5 section outlines these power domains along with the corresponding jumpers on the board used for power measurement.

Table 5 Power domains and current measurement jumpers

Power domain	Power measurement jumper	Description
VDDD/VDDIO_1V8	J26	1.8 V power rail used to power the core and majority of I/O domains. This domain is not used by default in the PSOC™ Edge E84 Evaluation Kit IDDD current includes current from the following pins: VDDD, VDDQ, VDD.BAT, VDD.DSI, VDDA, and VDD.1P8 (supply powered only)
VDDD/VDDIO_1V8_3V3	J24	1.8 V/3.3 V configurable (using J23) power rail used to power the Port 16 and Port 17 I/O domains
VDDUSB_3V3	J18	3.3 V power rail used to power the VDDUSB domain
VBAT_MCU	J25	2.7 V/3.3 V/4.2 V configurable (using J22) power rail used to power the VBAT domain of the PSOC™ Edge E84. This domain is used by default in the PSOC™ Edge E84 Evaluation Kit

There are three more power rails (VBAT_RADIO_3V3, VDD_MEM_1V8 and VDD/VDDIO_PERI_1V8) to power the other peripherals on the processor SoM (AIROC™ CYW55513 radio, M.2 memory module and CAPSENSE™ co-processor). Each rail is connected to its respective regulator outputs on the kit through a 0 Ω resistors (R415, R414, and R304 respectively). These resistors can be removed to measure current in the respective domains; however, they are excluded from measurement as these are unused in the application. For more information, see KIT_PSE84_EVAL PSOC™ Edge E84 Evaluation Kit guide.

***Note**: These jumpers (J26, J24, J18, J25) are not recommended to change or remove while the board is powered on. For accurate current measurements and to avoid leakage current, the PSOC™ Edge E84 Evaluation Kit may require hardware rework described in Rework for PSOC™ Edge E84 MCU low power current measurement section of the* KIT_PSE84_EVAL PSOC™ Edge E84 Evaluation Kit guide*; without this rework, readings can be up to ~100 µA higher in HP mode. Because the rework affects the functionality of Wi-Fi and Bluetooth® analog microphone interface, on-board KitProg3 JTAG interface and potentiometer interface, its not taken into account for smartwatch code example's power measurements.*

10.2 Power consumption and frame rate

To better analyze the performance, measure the power consumption of each power domain and the corresponding frame rate in below use cases/application states:

- High-performance application state: Complex watch screens rendered using GPU (shown in Figure 15 and Figure 16)

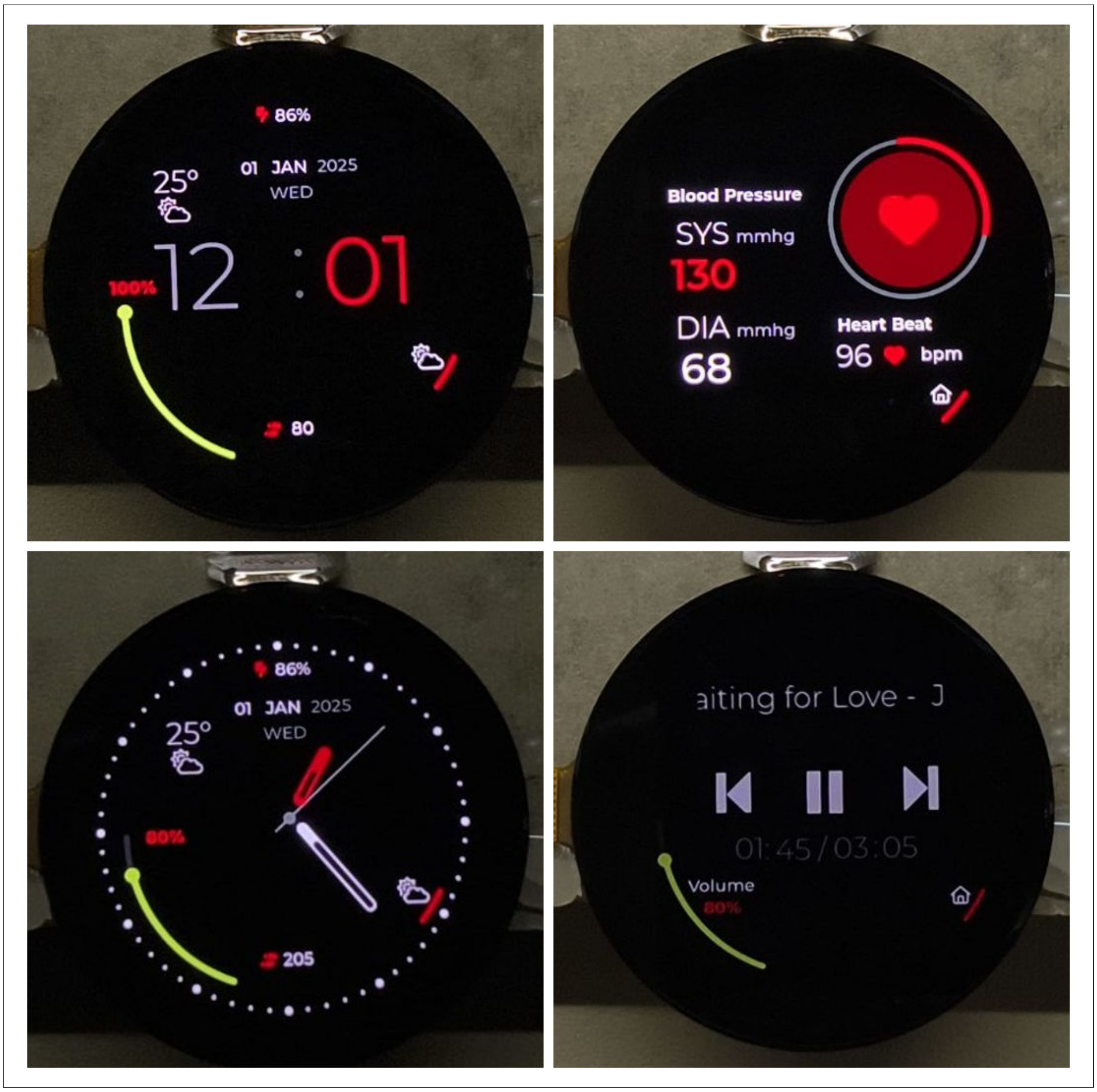


Figure 15 High-performance screens for 1.43-inch display

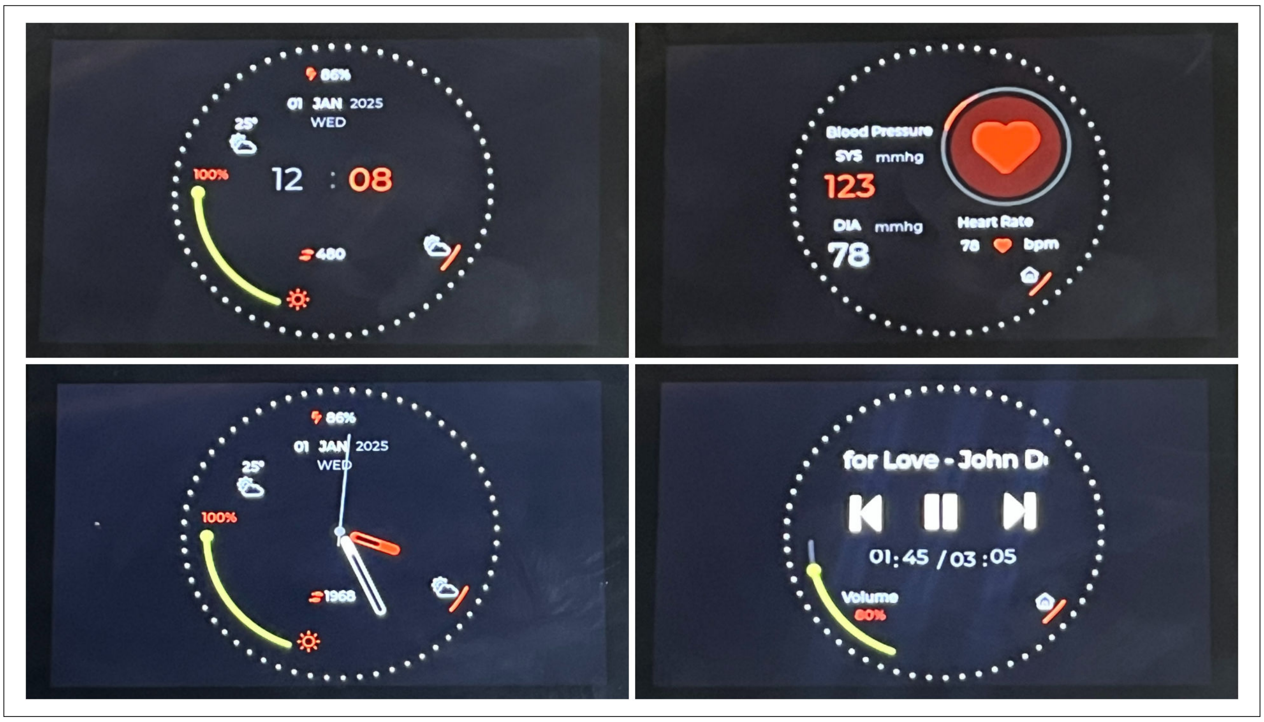


Figure 16 **High-performance screens for 4.3-inch display**

- Low-power application state: Simple always-on watch face rendered using CPU with periodic system deep sleep (shown in Figure 17 and Figure 18)

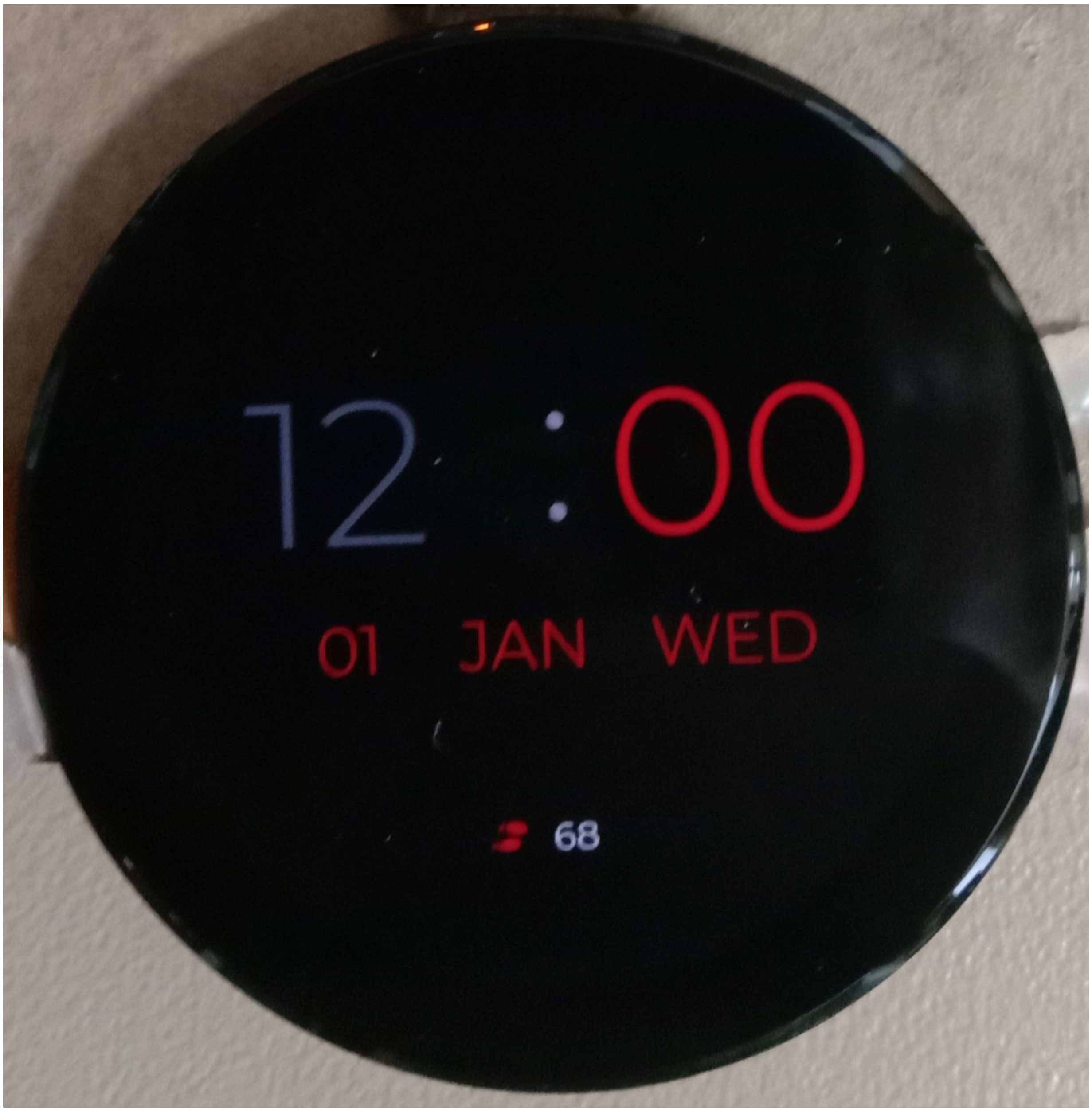


Figure 17 Low-power always-on screen for 1.43-inch display

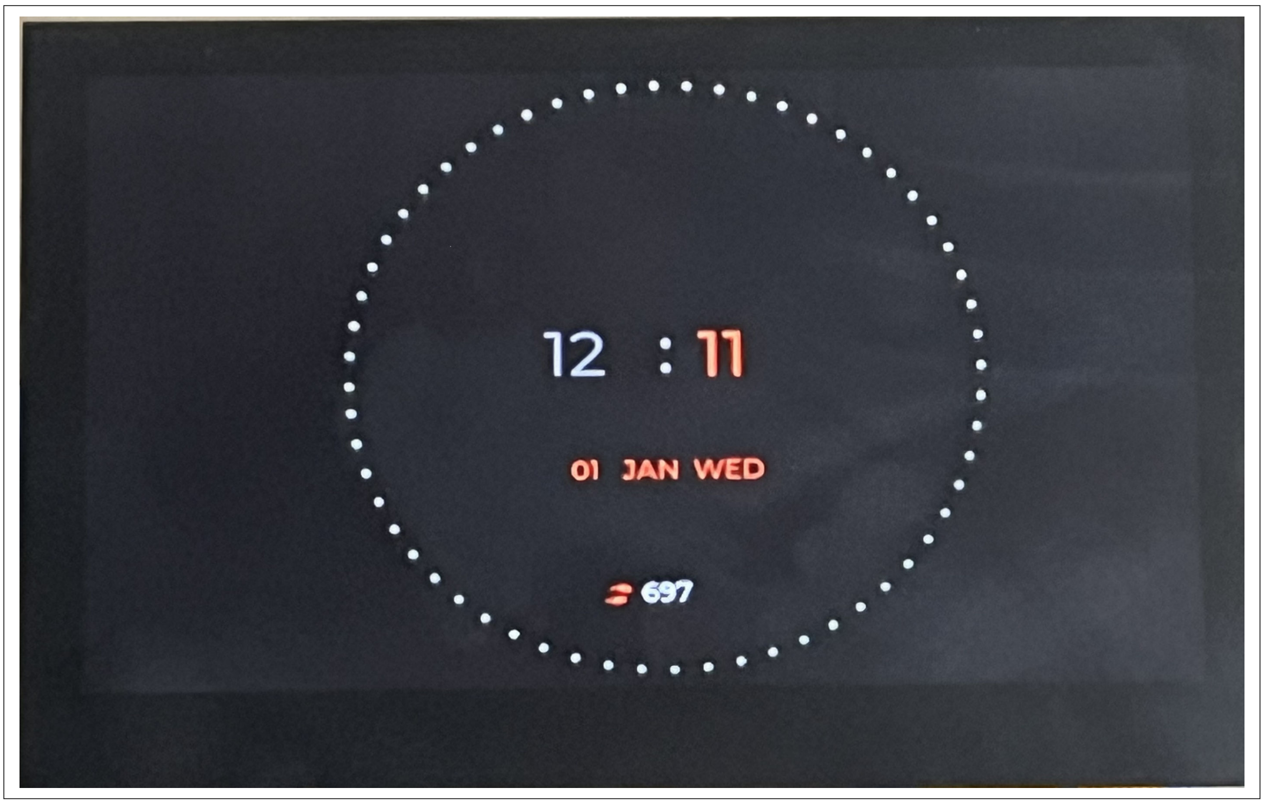


Figure 18 **Low-power always-on screen for 4.3-inch display**

- Ultra-low power application state: Display OFF, DSI ULPM with system deep sleep

10.2.1 Application state: High performance

The high-performance state of the smartwatch application configures the clock and system active power profile as outlined in Table 6.

Table 6 **Clock and system active power profile in HP state**

Parameters	Video and command mode display
CLK_HF1 (Core clock for CM55, graphics sub-system)	400 MHz
clk_2d (GPU core clock derived from CLK_HF1/2)	200 MHz
CLK_HF2 (Input clock for system SRAM/SoCMEM)	300 MHz
CLK_HF3 (Input clock for QSPI0 XIP flash)	200 MHz
MIPI DSI reference clock	24 MHz
System active power profile	HP mode (VCCD = 0.9 V and VCCSRAM = 0.9 V)

Frame generation and rendering on video and command mode displays

1. **4.3-inch video mode display**

   The application uses the default LVGL refresh behavior, where the dirty (or invalid) areas are checked and redrawn in every `LV_DEF_REFR_PERIOD` milliseconds (set in *lv_conf.h*). This happens as a result of a refresh timer created for a display instance, and is executed at that interval.

Following code snippet from the application shows display instance creation:

```
void lv_port_disp_init(void)
{

    /*-------------------------
     * Initialize your display
     * -----------------------*/
    disp_init();

    reset_frame_buffer();

    /* Create display instance and configure it */
    disp_gpu = lv_display_create(MY_DISP_HOR_RES, MY_DISP_VER_RES);
    if (disp_gpu == NULL) return;

    lv_display_set_flush_cb(disp_gpu, disp_flush);

    /* Provide LVGL a millisecond tick */
    lv_tick_set_cb(get_tick_ms);

    /* Double buffers, full-screen, full render mode */
    lv_display_set_buffers(disp_gpu,
                           disp_buf1,
                           disp_buf2,
                           sizeof(disp_buf1),
                           LV_DISPLAY_RENDER_MODE_FULL);

    /* 16bpp RGB565 to fit gfx_mem and match panel */
    lv_display_set_color_format(disp_gpu, LV_COLOR_FORMAT_RGB565);
    lv_display_set_render_mode(disp_gpu, LV_DISPLAY_RENDER_MODE_FULL);

    /* Make it the default display */
    lv_display_set_default(disp_gpu);

    Cy_GFXSS_Clear_DC_Interrupt(base, &gfx_context);
}
```

The application calls the `refresh_screen()` function every `HIGH_PERF_REFRESH_MIN_TIME_MS` (set to 10 milliseconds) or time till it needs to be run next (in ms) in synchronization with the 60 Hz display refresh rate in a thread-safe manner. The `refresh_screen()` function, in turn calls the `lv_timer_handler()` API to drive LVGL time-related tasks ensuring invalidated areas of the screen are checked and redrawn every `LV_DEF_REFR_PERIOD`.

```
__STATIC_INLINE uint32_t refresh_screen(void)
{
    lv_display_t *disp      = NULL;
    lv_timer_t *anim_timer = NULL;
    uint32_t ret            = RESET_VAL;

    if (xSemaphoreTake(lvgl_mutex, portMAX_DELAY) == pdTRUE)
    {
        ret = lv_timer_handler();

        if ((LOW_POWER_STATE == active_state) && state_change_complete)
        {
            disp = lv_display_get_default();

            lv_timer_t *refr_timer = lv_display_get_refr_timer(disp);
            if (refr_timer)
            {
                lv_timer_set_period(refr_timer, LVGL_REFRESH_TIME_MS);
            }

            anim_timer = lv_anim_get_timer();
            lv_timer_set_period(anim_timer, LVGL_REFRESH_TIME_MS);

            state_change_complete = false;

        }
        else if (state_change_complete)
        {
            state_change_complete = false;
        }
        xSemaphoreGive(lvgl_mutex);
    }

    return ret;
}
```

In *lv_port_disp.c*, LVGL's `disp_flush()` callback submits the GPU-composed back buffer to the DC. On the DC frame-complete interrupt, front and back buffers are swapped; so the previously displayed buffer becomes the new composition target, while the DC streams the current front buffer to the panel over MIPI DSI.

```
void LV_ATTRIBUTE_FAST_MEM disp_flush(lv_display_t *disp, const lv_area_t *area,
        uint8_t *px_map)
{
    CY_UNUSED_PARAMETER(area);

    /* Wait until frame is transmitted to display */
#if defined(MTB_DISPLAY_CO5300)
    while (!frame_tx_done)
#elif defined(W4P3INCH_DISP)
    while (fb_pending)
#endif
    {
        vTaskDelay(pdMS_TO_TICKS(FRAME_TX_WAIT_MS));
    }

    Cy_GFXSS_Set_FrameBuffer(base, (uint32_t*) px_map, &gfx_context);

    /* Kick the transfer task (non-blocking flush) */
#if defined(MTB_DISPLAY_CO5300)
    extern TaskHandle_t rtos_frame_tx_task_handle;
    if (rtos_frame_tx_task_handle != NULL)
    {
        xTaskNotify(rtos_frame_tx_task_handle, 0, eNoAction);
    }
#endif

    /* Count frames for performance monitor (FPS) */
#if defined(USE_PERFORMANCE_MONITOR)
    frame_count++;
#endif

    /* Complete the flush for LVGL immediately (non-blocking) */
    lv_display_flush_ready(disp);
#if defined(MTB_DISPLAY_CO5300)
    frame_tx_done = false;
#elif defined(W4P3INCH_DISP)
    fb_pending = true;
#endif
}
```

2. **1.43-inch command mode display**

Similar to video mode display, the application calls the `refresh_screen()` function every `SCREEN_REFRESH_TIME_MS`(30 ms), which in turn uses `lv_timer_handler()` to invalidate and redraw the invalidated areas of screen at every `LV_DEF_REFR_PERIOD`. This use case also incorporates the `disp_flush()` function, which sets the frame buffer address, allowing the DC to render the frame from it. Upon completion, `disp_flush()` sends a notification to the *Frame Tx Task*, which then utilizes the

`Cy_GFXSS_Transfer_Frame()` API to initiate DBI transfers from the DC to the MIPI DSI host to render the frame buffer onto the display.

```
#if defined(MTB_DISPLAY_CO5300)
/*******************************************************************************
* Function Name: frame_transfer_task
********************************************************************************
* Summary:
*  This freeRTOS task handles transferring rendered frames to display panel from
*  display controller.
*
* Parameters:
*  *arg: Not used
*
* Return:
*  void
*
*******************************************************************************/
static void frame_transfer_task(void *arg)
{
    CY_UNUSED_PARAMETER(arg);

    frame_tx_done = true;

    for (;;)
    {
        if (pdPASS == xTaskNotifyWait(RESET_VAL, RESET_VAL, NULL, portMAX_DELAY))
        {
            Cy_GFXSS_Transfer_Frame((GFXSS_Type*) GFXSS, &gfx_context);

            frame_tx_done = true;
        }
    }
}
#endif /* MTB_DISPLAY_CO5300 */
```

In the high-performance state, the application composes complex watch screens using GPU by leveraging double frame buffers located in on-chip system SRAM.

The measurement results for 4.3-inch video mode display are listed in Table 7 and Table 8

Table 7 **Current measurement result in HP state for 4.3-inch video mode display**

Power domain	Power measurement jumper	Current
VDDD/VDDIO_1V8	J26	MIN: 10.8 mA MAX: 16.8 mA AVG: 12.8 mA
VDDD/VDDIO_1V8_3V3	J24	MIN: 60.5 μA MAX: 60.9 μA AVG: 60.7 μA

(table continues...)

Table 7 (continued) Current measurement result in HP state for 4.3-inch video mode display

Power domain	Power measurement jumper	Current
VDDUSB_3V3	J18	MIN: 118.0 µA MAX: 118.2 µA AVG: 118.1 µA
VBAT_MCU	J25	MIN: 17.2 mA MAX: 27.3 mA AVG: 20.5 mA

Table 8 Frame rate measurement result in HP state for 4.3-inch video mode display

Screen name	Frame rate
Start screen	36 fps
Analog watch	30 fps
Heart rate	36 fps
Music screen	34 fps
Weather Screen	32 fps

The measurement result for 1.43-inch command mode display are listed in Table 9 and Table 10

Table 9 Current measurement result in HP state for 1.43-inch command mode display

Power domain	Power measurement jumper	Current
VDDD/VDDIO_1V8	J26	MIN: 7.0 mA MAX: 14.7 mA AVG: 11.8 mA
VDDD/VDDIO_1V8_3V3	J24	MIN: 75.6 µA MAX: 76.6 µA AVG: 76.0 µA
VDDUSB_3V3	J18	MIN: 116.2 µA MAX: 117.2 µA AVG: 116.5 µA
VBAT_MCU	J25	MIN: 15.7 mA MAX: 29.7 mA AVG: 20.8 mA

Table 10 Frame rate measurement result in HP state for 1.43-inch command mode display

Screen name	Frame rate
Start screen	24 fps
Analog watch	24 fps
Heart rate	24 fps
Music screen	22 fps

(table continues...)

Table 10 (continued) Frame rate measurement result in HP state for 1.43-inch command mode display

Screen name	Frame rate
Weather Screen	23 fps

10.2.2 Application state: Low power

The low-power state of the smartwatch application configures the clock and system active power profile, as outlined in Table 11.

Table 11 Clock and system active power profile in LP state

Parameters	Video mode display	Command mode display
CLK_HF1 (Core clock for CM55, graphics sub-system)	140 MHz	50 MHz
clk_2d (GPU core clock derived from CLK_HF1/2)	NA (GPU disabled)	NA (GPU disabled)
CLK_HF2 (Input clock for system SRAM/SoCMEM)	110 MHz	50 MHz
CLK_HF3 (Input clock for QSPI0 XIP flash)	70 MHz	50 MHz
MIPI DSI reference clock	24 MHz	24 MHz
System active power profile	LP mode (VCCD = 0.8 V and VCCSRAM = 0.8 V)	ULP mode (VCCD = 0.7 V and VCCSRAM = 0.8 V)

In case of low-power state, same frame generation and rendering path is followed as described in HP state for both of the displays except the below points:

- The always-on screen with minimal data is composed using CPU while GPU stays disabled
- System clock is switched to ULP mode (50 MHz @ 0.7 core voltage)
- Display backlight PWM duty cycle is reduced to lowest possible value
- Using tickless idle mode, the system enters deep sleep between two refresh cycles when CPU is idle in case of command mode display

Following snippet from the code example showcases switching to low power state and render always-on screen for both display use case:

```
/* Low power always-on graphics powered by CPU in LP/ULP mode. CM55 @140/50 MHz */

            case LOW_POWER_STATE:
                /* Suspend performance monitor in LP depending on display */
#if defined(USE_PERFORMANCE_MONITOR)
#if defined(MTB_DISPLAY_CO5300)
                /* ROUND display */
                performance_monitor_suspend();
#endif
#endif /* USE_PERFORMANCE_MONITOR */
                stat = vg_lite_close();
                if (VG_LITE_SUCCESS != stat)
                {
                    process_error((cy_rslt_t)stat, "VGLite close failed. STOP.");
                }
                gpu_enable = false;
                /* Disable GPU interrupt */
                Cy_GFXSS_Disable_GPU_Interrupt(base);

                /* Disable GPU interrupt in NVIC */
                NVIC_DisableIRQ(GFXSS_GPU_IRQ);
#if defined(W4P3INCH_DISP)
                /* Disable DC interrupt in NVIC */
                NVIC_DisableIRQ(GFXSS_DC_IRQ);
#endif
                result = Cy_GFXSS_DeInit(base, &gfx_context);
                if (CY_GFX_SUCCESS != result)
                {
                    process_error((cy_rslt_t)result, "Gfxss deinitialization failed.
STOP.");
                }
#if defined(MTB_DISPLAY_CO5300)
                /* Set ULP as System Active Power Profile */
                dpll_lp0_set_freq(DPLL_LP_OUTPUT_FREQ_ULP_HZ);
                dpll_lp1_set_freq(DPLL_LP_OUTPUT_FREQ_ULP_HZ);
                status= Cy_SysPm_SystemEnterUlp();
                if (CY_SYSPM_SUCCESS != status)
                {
                    process_error((cy_rslt_t)result, "System enter ULP failed. STOP.");
                }

                /** Check if the system successfully entered ULP mode. */
                if (Cy_SysPm_ReadStatus() & CY_SYSPM_STATUS_SYSTEM_ULP)
                {
                    /* Set the RRAM to ULP voltage mode for lower power
                     * consumption.
                     */
                    Cy_RRAM_SetVoltageMode(RRAMC0, CY_RRAM_VMODE_ULP);

                    /** Set the high-frequency clock (CLKHF) to no divide */
                    Cy_SysClk_ClkHfSetDivider(CY_CFG_SYSCLK_CLKHF0,
CY_SYSCLK_CLKHF_NO_DIVIDE);

                    /** Set the peripheral clock divider for the debug UART */

Cy_SysClk_PeriPclkSetDivider((en_clk_dst_t)CYBSP_DEBUG_UART_CLK_DIV_GRP_NUM,
                            CY_SYSCLK_DIV_16_BIT, 1U, UART_ULP_DIV);
                }

#elif defined(W4P3INCH_DISP)
                /* Set LP as System Active Power Profile.
                 * Note: On 4.3 inch display with 50 MHz core clock in ULP mode,
                 * the pixel clock limitation (<= 25 MHz) prevents graphics
                 * rendering.
                 */
                dpll_lp0_set_freq(DPLL_LP0_OUTPUT_FREQ_LP_HZ);
                dpll_lp1_set_freq(DPLL_LP1_OUTPUT_FREQ_LP_HZ);
                Cy_SysPm_SystemEnterLp();

                /** Check if the system successfully entered LP mode. */
                if (Cy_SysPm_ReadStatus() & CY_SYSPM_STATUS_SYSTEM_LP)
                {
                    /** Set the RRAM to LP voltage mode for lower power
                        * consumption. */
                    Cy_RRAM_SetVoltageMode(RRAMC0, CY_RRAM_VMODE_LP);

                    Cy_SysClk_ClkHfSetDivider(CY_CFG_SYSCLK_CLKHF0,
CY_SYSCLK_CLKHF_DIVIDE_BY_2);

                    /** Set the peripheral clock divider for the debug UART */

Cy_SysClk_PeriPclkSetDivider((en_clk_dst_t)CYBSP_DEBUG_UART_CLK_DIV_GRP_NUM,
                            CY_SYSCLK_DIV_16_BIT, 1U, UART_LP_DIV);
                }
#endif
                SystemCoreClockUpdate();
                Cy_SysTick_Disable();
                Cy_SysTick_SetReload((configCPU_CLOCK_HZ/configTICK_RATE_HZ ) - SET_VALUE);
                Cy_SysTick_Clear();
                Cy_SysTick_Enable();

#if defined(W4P3INCH_DISP)
                /* Update CLK divider for Debug UART as per core frequency */

Cy_SysClk_PeriPclkDisableDivider((en_clk_dst_t)CYBSP_DEBUG_UART_CLK_DIV_GRP_NUM,
CY_SYSCLK_DIV_16_BIT, DEBUG_UART_DIVIDER_NUM);

Cy_SysClk_PeriPclkSetDivider((en_clk_dst_t)CYBSP_DEBUG_UART_CLK_DIV_GRP_NUM,
CY_SYSCLK_DIV_16_BIT, DEBUG_UART_DIVIDER_NUM, DEBUG_UART_LP_DIVIDER_VAL);

Cy_SysClk_PeriPclkEnableDivider((en_clk_dst_t)CYBSP_DEBUG_UART_CLK_DIV_GRP_NUM,
CY_SYSCLK_DIV_16_BIT, DEBUG_UART_DIVIDER_NUM);
#elif defined(MTB_DISPLAY_CO5300)
```

```
                /* Update CLK divider for Debug UART as per core frequency */

Cy_SysClk_PeriPclkDisableDivider((en_clk_dst_t)CYBSP_DEBUG_UART_CLK_DIV_GRP_NUM,
CY_SYSCLK_DIV_16_BIT, DEBUG_UART_DIVIDER_NUM);

Cy_SysClk_PeriPclkSetDivider((en_clk_dst_t)CYBSP_DEBUG_UART_CLK_DIV_GRP_NUM,
CY_SYSCLK_DIV_16_BIT, DEBUG_UART_DIVIDER_NUM, DEBUG_UART_ULP_DIVIDER_VAL);

Cy_SysClk_PeriPclkEnableDivider((en_clk_dst_t)CYBSP_DEBUG_UART_CLK_DIV_GRP_NUM,
CY_SYSCLK_DIV_16_BIT, DEBUG_UART_DIVIDER_NUM);
#endif
                /* Re-initialize the graphics subsystem as per updated clock */
                GFXSS_config.clockHz = Cy_SysClk_ClkHfGetFrequency(CY_CFG_SYSCLK_CLKHF1);
                GFXSS_config.gpu_cfg->enable = false;

                /* Reset frame buffers before switching to different application
                 * state for smoother UI transition.
                 */
                reset_frame_buffer();

                result = Cy_GFXSS_Init(base, &GFXSS_config, &gfx_context);
                if (CY_GFX_SUCCESS != result)
                {
                    process_error((cy_rslt_t)result, "Gfxss re-initialization failed.
STOP.");
                }

#if defined(W4P3INCH_DISP)
                /* Enable DC interrupt in NVIC to synchronize frame transfers
                 * with the completion interrupt of frame buffer transfers from
                 * DC.
                 */
                NVIC_EnableIRQ(GFXSS_DC_IRQ);
#endif

                /* Display ui_LPScreen */
                brightness = MIN_BRIGHTNESS_PERCENT;
#if defined(MTB_DISPLAY_CO5300)
                mtb_display_co5300_set_brightness(&base->GFXSS_MIPIDSI,
SET_BRIGHTNESS(brightness));
#elif defined(W4P3INCH_DISP)
                mtb_disp_waveshare_4p3_set_brightness(CYBSP_I2C_CONTROLLER_HW,
&i2c_context,brightness );
#endif
                _ui_screen_change(&ui_LPScreen, LV_SCR_LOAD_ANIM_FADE_ON, RESET_VALUE,
RESET_VALUE, &ui_LPScreen_screen_init);

#if defined(MTB_DISPLAY_CO5300)
                /* Reset/restart the input_inactivity_timer */
                xTimerStart(input_inactivity_timer, RESET_VALUE);
                /* System Domain Idle Power Mode Configuration */
                Cy_SysPm_SetDeepSleepMode(CY_SYSPM_MODE_DEEPSLEEP);
                /* System SRAM (SoCMEM) Idle Power Mode Configuration */
```

```
                    Cy_SysPm_SetSOCMEMDeepSleepMode(CY_SYSPM_MODE_DEEPSLEEP);
                    /* Allow CPU to go DeepSleep */
                    mtb_hal_syspm_unlock_deepsleep();
                    xTimerStart(lp_task_timer, RESET_VALUE);
#endif /*defined(MTB_DISPLAY_CO5300)*/
                    break;
```

- To achieve lowest possible refresh rate, the LVGL's display refresh timer period (`refr_timer`) is set to 1 second for video mode and 9 seconds for command mode display

```
#if defined(MTB_DISPLAY_CO5300)
#define LVGL_REFRESH_TIME_MS                (9000U)
#elif defined(W4P3INCH_DISP)
#define LVGL_REFRESH_TIME_MS                (1000U)
#endif

if ((LOW_POWER_STATE == active_state) && state_change_complete)
{
    disp = lv_display_get_default();

    lv_timer_t *refr_timer = lv_display_get_refr_timer(disp);
    if (refr_timer)
    {
        lv_timer_set_period(refr_timer, LVGL_REFRESH_TIME_MS);
    }

    anim_timer = lv_anim_get_timer();
    lv_timer_set_period(anim_timer, LVGL_REFRESH_TIME_MS);

    state_change_complete = false;
}
```

***Note**: In low-power state, the system active power profile is set to LP mode (VCCD = 0.8 V and VCCSRAM = 0.8 V) for 4.3-inch video mode display to maintain the required pixel clock (<= 25 MHz) and correct video timings as in ULP mode this pixel clock cannot be sustained, so the panel cannot be driven reliably.*

In low-power state, the application composes simple always-on screen by leveraging double frame buffers located in on-chip system SRAM.

The measurement result for both the displays are listed in Table 12 and Table 13.

Table 12 Measurement result in LP state for 4.3-inch video mode display

Power domain	Power measurement jumper	Current	Frame rate
VDDD/VDDIO_1V8	J26	MIN: 9.8 mA MAX: 12.9 mA AVG: 11.1 mA	1 fps

(table continues...)

Table 12 (continued) Measurement result in LP state for 4.3-inch video mode display

Power domain	Power measurement jumper	Current	Frame rate
VDDD/VDDIO_1V8_3V3	J24	MIN: 60.4 µA MAX: 60.7 µA AVG: 60.5 µA	
VDDUSB_3V3	J18	MIN: 117.9 µA MAX: 118.1 µA AVG: 118.0 µA	
VBAT_MCU	J25	MIN: 4.0 mA MAX: 7.2 mA AVG: 4.9 mA	

Table 13 Measurement result in LP state for 1.43-inch command mode display

Power domain	Power measurement jumper	Current	Frame rate
VDDD/VDDIO_1V8	J26	MIN: NA MAX: 11.1 mA AVG: 307.8 µA	<= 1 fps
VDDD/VDDIO_1V8_3V3	J24	MIN: 5.6 µA MAX: 7.7 µA AVG: 7.1 µA	
VDDUSB_3V3	J18	MIN: 53.3 µA MAX: 54.1 µA AVG: 52.3 µA	
VBAT_MCU	J25	MIN: 91.3 µA MAX: 2.7 mA AVG: 119.2 µA	

10.2.3 Application state: Ultra-low power

In ultra-low power state, the application suspends all screen updates, display OFF, enters DSI ULPM mode for both clock and data lines and allows system to go to a deep sleep in tickless idle mode.

```
case ULTRA_LOW_POWER_STATE:
                /* Always suspend performance monitor in ULP */
#if defined(USE_PERFORMANCE_MONITOR)
                performance_monitor_suspend();
#endif /* USE_PERFORMANCE_MONITOR */
                vTaskSuspend(rtos_date_time_task_handle);
                vTaskSuspend(rtos_step_count_task_handle);

                /* Reset frame buffers before switching to different application
                 * state for smoother UI transition.
                 */
                reset_frame_buffer();

#if defined(MTB_DISPLAY_CO5300)
                mtb_display_co5300_off(&base->GFXSS_MIPIDSI);
#elif defined(W4P3INCH_DISP)

                /* System Domain Idle Power Mode Configuration */
                Cy_SysPm_SetDeepSleepMode(CY_SYSPM_MODE_DEEPSLEEP);
                /* System SRAM (SoCMEM) Idle Power Mode Configuration */
                Cy_SysPm_SetSOCMEMDeepSleepMode(CY_SYSPM_MODE_DEEPSLEEP);

                /* Allow CPU to go DeepSleep */
                mtb_hal_syspm_unlock_deepsleep();

                /* Disable DC interrupt in NVIC */
                NVIC_DisableIRQ(GFXSS_DC_IRQ);
                vTaskSuspend(rtos_app_task_handle);
#endif

                result = (cy_en_gfx_status_t)Cy_MIPIDSI_EnterULPM(&base->GFXSS_MIPIDSI);
                if (CY_GFX_SUCCESS != result)
                {
                    process_error((cy_rslt_t)result, "Entering ULPS mode failed. STOP.");
                }
                break;
```

The measurement result for both the displays are listed in Table 14 and Table 15.

Table 14 Measurement result in ULP state for 4.3-inch video mode display

Power domain	Power measurement jumper	Current	SRAM retention in deep sleep
VDDD/VDDIO_1V8	J26	198.4 µA	Both SRAM and system SRAM are fully retained
VDDD/VDDIO_1V8_3V3	J24	5 µA	
VDDUSB_3V3	J18	52.6 µA	

(table continues...)

Table 14 (continued) Measurement result in ULP state for 4.3-inch video mode display

Power domain	Power measurement jumper	Current	SRAM retention in deep sleep
VBAT_MCU	J25	67.1 µA	

Table 15 Measurement result in ULP state for 1.43-inch command mode display

Power domain	Power measurement jumper	Current	SRAM retention in deep sleep
VDDD/VDDIO_1V8	J26	255 µA[1)]	Both SRAM and system SRAM are fully retained
VDDD/VDDIO_1V8_3V3	J24	7.7 µA	
VDDUSB_3V3	J18	53.8 µA	
VBAT_MCU	J25	96.8 µA	

1) Large spikes of current are observed in MCU current @ VDDD = 1.8 V (J26) due to kit rework to interface 1.43-inch display.

Summary

This application note kicks off by guiding users in identifying the requirements for their target graphics application, specifically regarding the display module. It helps in choosing the right hardware interface and color format based on memory needs. After finalizing the display module, it details the configuration of the clock, DC, GPU, DSI host, and interrupts tailored to the application requirements. The application note showcases an end-to-end graphics pipeline, from frame generation to rendering, using a smartwatch use case, and enables the users to evaluate and optimize their application's performance. In summary, this application note equips the users to develop high-performance graphics applications that leverage the GPU with low-power optimizations on the PSOC™ Edge MCU, utilizing the ModusToolbox™ ecosystem.

References

Reference manuals

- PSOC™ Edge E8x registers reference manual
- PSOC™ Edge E8x architecture reference manual
- VGLite API reference manual

Kit hardware

- PSOC™ Edge E84 Evaluation Kit (KIT_PSE84_EVAL)
- KIT_PSE84_EVAL PSOC™ Edge E84 Evaluation Kit guide

Application notes

- AN239191 Getting started with graphics on PSOC™ Edge MCU
- AN239774: Selecting and configuring memories for power and performance in PSOC™ Edge MCU

Code examples

- CE242216: PSOC™ Edge MCU: Smartwatch Demo using LVGL
- CE239203: PSOC™ Edge MCU: Graphics using RLAD

Revision history

Document revision	Date	Description of changes
**	2025-10-23	Initial release
*A	2025-12-09	Minor edits throughout the document

Trademarks

PSOC™, formerly known as PSoC™, is a trademark of Infineon Technologies. Any references to PSoC™ in this document or others shall be deemed to refer to PSOC™.

Edition 2025-12-09

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about any aspect of this document?

Email: erratum@infineon.com

Document reference
IFX-stq1756725728329