

[Home](#)

ARM / EMBEDDED / STM32

14

STM32: using the LTDC display controller

BY [LUCA DAVIDIAN](#) · PUBLISHED OCTOBER 2, 2017 · UPDATED JUNE 16, 2018

The STM32 LTDC has a peripheral called **LTDC LCD TFT Display Controller** which provides a **digital parallel interface (DPI)** for a variety of LCD and TFT panels. It sends RGB data in parallel to the display and generates signals for *horizontal and vertical synchronization* (**HSYNC**, **VSYNC**), as well as *pixel clock* (**PCLK**) and *not data enable* (**DE**) signals:

LCD-TFT signals	I/O	Description
LCD_CLK	O	Clock Output
LCD_HSYNC	O	Horizontal Synchronization
LCD_VSYNC	O	Vertical Synchronization
LCD_DE	O	Not Data Enable
LCD_R[7:0]	O	Data: 8-bit Red data
LCD_G[7:0]	O	Data: 8-bit Green data
LCD_B[7:0]	O	Data: 8-bit Blue data

figure 1: LTDC RGB interface signals

RGB interface synchronization signals

LTDC synchronous timing parameters are configurable: a **synchronous timing generator block** inside the LTDC generates the horizontal and vertical synchronization signals, the pixel clock and not data enable signals. The configurable timing parameters are:

- **LTDC_SSCR** *Synchronization Size Configuration Register*, configured by programming the values *HSYNC width - 1* and *VSYNC width - 1*
- **LTDC_BPCR** *Back Porch Configuration Register*, configured by programming the accumulated values *HSYNC width + horizontal back porch - 1* and *VSYNC width + vertical back porch - 1*
- **LTDC_AWCR** *Active Width Configuration Register*, configured by programming the accumulated values *HSYNC width + horizontal back porch + active width - 1* and *VSYNC width + vertical back porch + active height - 1*

■ LTDC_TWCR *Total Width*

Configuration Register, configured by programming the accumulated values *HSYNC width + horizontal back porch + active width + horizontal front porch - 1* and *VSYNC width + vertical back porch + active height + vertical front porch - 1*

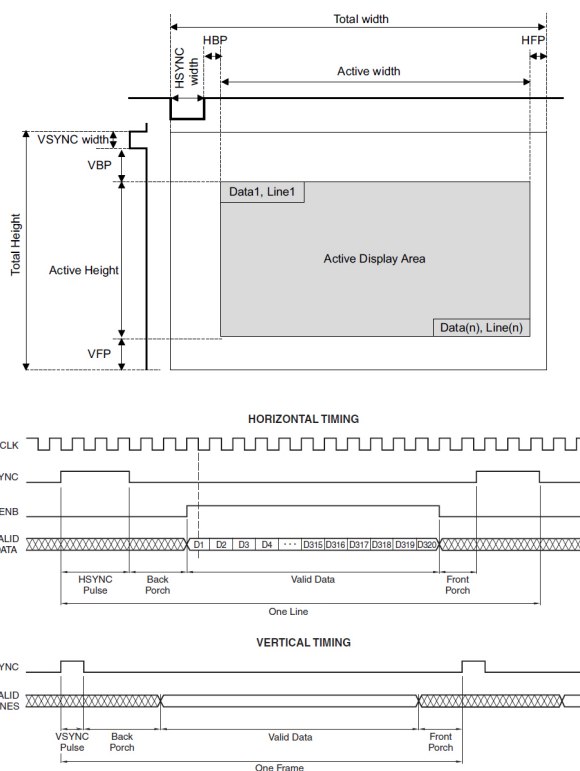


figure 2: RGB interface timing and horizontal/vertical synchronization, pixel clock and data enable signals

Horizontal timing signal widths are in **units of pixel clocks**, while vertical timing signal widths are in **units of horizontal scan lines**. The HSYNC, VSYNC, pixel clock and not data enable signal polarities can be configured to

active high or active low in the **LTDC_GCR** *Global Control Register* (not *data enable* signal must be configured **inverted** with respect to the *data enable* signal in the display datasheet). The datasheet of the panel generally provides the timing parameters for the display:

Parameters	Symbols	Condition	Min.	Typ.	Max.	Units
Horizontal Synchronization	Hsync		2	10	16	DOTCLK
Horizontal Back Porch	HBP		2	20	24	DOTCLK
Horizontal Address	HAdr		-	240	-	DOTCLK
Horizontal Front Porch	HFP		2	10	16	DOTCLK
Vertical Synchronization	Vsync		1	2	4	Line
Vertical Back Porch	VBP		1	2	-	Line
Vertical Address	VAdr		-	320	-	Line
Vertical Front Porch	VFP		3	4	-	Line

figure 3: RGB panel timing signals

Configuring the timing signals is the first thing to do to initialize the LTDC controller. The function `HAL_LTDC_MspInit()` initializes the low level details of the LTDC peripheral (clock and GPIOs):

```

1  /* initialize LTDC */
2  HAL_LTDC_MspInit(&ltdc); // initialize L
3
4  LTDC->GCR &= ~(LTDC_GCR_HSPOL | LTDC_GCR_
5  LTDC->SSCR = 9 << LTDC_SSCR_HSW_Pos | 1 <
6  LTDC->BPCR = 29 << LTDC_BPCR_AHBP_Pos | 3
7  LTDC->AWCR = 269 << LTDC_AWCR_AAW_Pos | 3
8  LTDC->TWCR = 279 << LTDC_TWCR_TOTALW_Pos

```

A constant background color can be configured in **LTDC_BCCR** *Background Color Configuration Register* (eight bits per channel are used in this register to select a solid RGB color):

```

1  LTDC->BCCR = 0x00 << LTDC_BCCR_BCRED_Pos

```

The background color is used for blending with the bottom layer.

Layer configuration

The LTDC has two layers which can be configured, enabled and disabled independently, each with its own FIFO buffer. Layer order is fixed and layer2 is always on top of layer1. Layer can be enabled writing the *LEN Layer Enable bit* in the **LTDC_LxCR** *Layer x Control Register*. Each layer gets its data from a **framebuffer in memory** and the start address is written in **LTDC_LxCFBAR** *Layer x Color Frame Buffer Address Register*. The frame buffer contains the display frame data in **one of eight configurable pixel format:** **LTDC_LxPFCR** *Layer x Pixel Format Configuration Register* is configured to choose the pixel format used to store data into the frame buffer. The available pixel formats are:

- ARGB8888
- RGB888
- RGB565
- ARGB1555
- ARGB4444
- L8 (8 bit luminance)
- AL44 (4 bit alpha, 4 bit luminance)
- AL88 (8 bit alpha, 8 bit luminance)

The pixel data are read from the framebuffer and converted to the LTDC internal 32-bit pixel format ARGB8888:

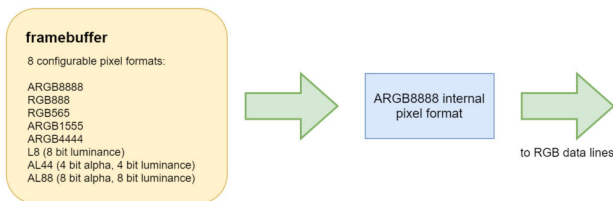


figure 4: whatever the framebuffer pixel format the LTDC converts the data into the internal 32 bit ARGB8888 pixel format

Each layer can be positioned and resized inside the active area indicating the start and stop position of the visible window in the

LTDC_LxWHPCR *Layer x Window Horizontal Position Configuration Register* and **LTDC_LxWVPCR** *Layer x Window Vertical Position Configuration Register*. These parameters select the first and last visible pixels of a line and the first and last visible lines in the window. The values must includes the timing signals (HSYNC and VSYNC) width and the back porch width programmed into **LTDC_BPCR** register. In this case the accumulated horizontal back porch is 30 – 1, so the active area starts at 30 and the image is 240 pixel wide so horizontal window stop position is $30 + 240 - 1 = 269$

(same for the vertical start and stop positions):

```
1 /* configure layer 1 */
2 LTDC_Layer1->WHPCR = 269 << LTDC_LxWHPCR;
3 LTDC_Layer1->WVPCR = 323 << LTDC_LxWVPCR;
4 LTDC_Layer1->PFCR = 0x01;
5 LTDC_Layer1->CFBAR = (uint32_t)framebuffer;
```

The frame buffer has a configurable **line length** (in bytes) in the **LTDC_LxCFBLR** *Layer x Color Frame Buffer Length Register* and a configurable **total number of lines** in the **LTDC_LxCFBLNR** *Layer x Color Frame Buffer Line Number Register*. It also has a configurable **line pitch**, which indicates the distance in bytes between the start of a line and the beginning of the next line, also configured in the **LTDC_LxCFBLR** register, and expressed in bytes. These parameters are used by the LTDC to fetch data from the frame buffer to the layer FIFO. If set to less byte than needed, a FIFO underrun interrupt will trigger (if enabled), if set to more bytes than required the rest of the data loaded into the layer's FIFO is discarded.

```
1 LTDC_Layer1->CFBLR = 240 * 3 << LTDC_LxCFBLR;
2 LTDC_Layer1->CFBLNR = 320;
```

Line length parameter is the number of bytes in a line plus three (so the total line length is *number of pixels * bits per pixel + 3*). These parameters,

together with the layer windowing settings, are useful if we want to display part of an image contained in the frame buffer, as I'll show later.

Each layer can also have a default color, configured into the

LTDC_LxDCCR *Layer x Default Color Configuration Register*, in ARGB8888 format, that is used outside the layer window or when a layer is disabled:

```
1 LTDC_Layer1->DCCR = 0xFF << LTDC_LxDCCR_0;
```

A constant alpha blending value is configured into

the **LTDC_LxCACR** *Layer x Constant Alpha Configuration Register*, and controls the alpha blending with the underlying layers. In this case the value 255 (which is divided by 255 by hardware to get a value between 0 and 1) indicates a solid color:

```
1 LTDC_Layer1->CACR = 255;
```

Blending order is fixed and if both layers are enabled, first layer 1 is blended with the background and then layer 2 is blended with the result:

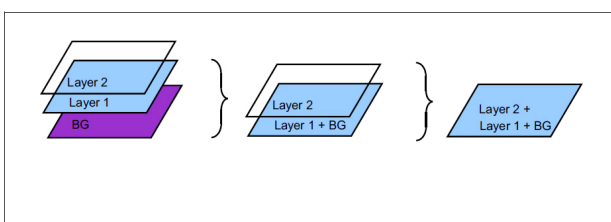


figure 5: layer blending

Then the layer is enabled by writing the LEN bit into the LTDC_LxCR LTDC Layer x Control Register:

```
1 LTDC_Layer1->CR |= LTDC_LxCR_LEN;
```

Shadow configuration registers

Some configuration registers are *shadowed*, meaning their programmed values are stored into *shadow registers* (not accessible to the programmer) and reloaded into the actual configuration registers based on the configuration of the

LTDC_SRCR Shadow Reload

Configuration Register: if this register is written with the IMR **Immediate**

Reload bit the registers are reloaded immediately (as soon as the IMR bit is set the registers are reloaded), if the

Vertical Blanking Reload bit is

written the registers are reloaded with the new values during the vertical blanking period (at the beginning of the first line after the active display area). These bits are set in software and cleared by hardware when shadow registers are reloaded:

```
1 /* reload shadow registers and enable LTDC
2 LTDC->SRCR = LTDC_SRCR_IMR; // immedi
3 LTDC->GCR |= LTDC_GCR_LTDCEN; // enable
```

The registers read the old values until they're reloaded and if a new value is written before they're reloaded the previous value is overwritten. Most of the layers' configuration registers are shadowed so they must be reloaded after being configured and before enabling the LTDC. The complete LTDC_init() function looks like this:

```

1 void LTDC_init(void)
2 {
3     /* initialize LTDC */
4     HAL_LTDC_MspInit(&ltdc);
5
6     /* configure LTDC general parameters */
7     LTDC->GCR |= ~(LTDC_GCR_HSPOL | LTDC_GCR_HSPOL);
8     LTDC->SSCR = 9 << LTDC_SSCR_HSW_Pos |
9     LTDC->BPCR = 29 << LTDC_BPCR_AHBP_Pos |
10    LTDC->AWCR = 269 << LTDC_AWCR_AAW_Pos |
11    LTDC->TWCR = 279 << LTDC_TWCR_TOTALW_Pos |
12    LTDC->BCCR = 0xFF << LTDC_BCCR_BCGREEN_Pos;
13
14    /* configure layer 1 */
15    LTDC_Layer1->WHPCR = 269 << LTDC_LxWHF_Pos |
16    LTDC_Layer1->WVPCR = 323 << LTDC_LxWVF_Pos |
17    LTDC_Layer1->PFCR = 0x01; // RGB888
18    LTDC_Layer1->DCCR = 0xFF << LTDC_LxDCC_Pos;
19    LTDC_Layer1->CFBAR = (uint32_t)image;
20    LTDC_Layer1->CFBLR = 240 * 3 << LTDC_LxCFBLR_Pos;
21    LTDC_Layer1->CFBLNR = 320;
22    LTDC_Layer1->CACR = 255;
23    LTDC_Layer1->CR |= LTDC_LxCR_LEN;
24
25    LTDC->SRCR = LTDC_SRCR_IMR;
26
27    LTDC->GCR |= LTDC_GCR_LTDCEN;
28 }

```

Using the LTDC with the ILI9341 display controller

In this example I use the display on the STM32F429-Discovery board, which is driven by the ILI9341 display controller. The ILI9341 can drive a

QVGA (Quarter VGA) 240×320 262,144 colors LCD display. The controller can be configured via SPI (or parallel interface, depending on the panel settings) to use a digital parallel 18 bit RGB interface (since only 6 lines per color channel are wired on the board to the LTDC). Since the display pixel format is less than 8 bit per channel (RGB666 in this case), the RGB display data lines are connected to the most significant bits of the LTDC controller RGB data lines:

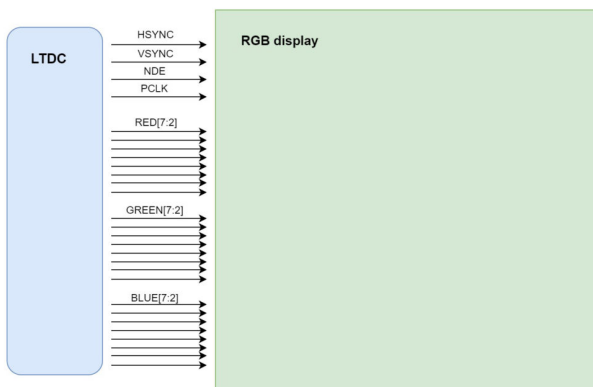


figure 5: LTDC signal lines. Only MSB are used if the display has less than 8 lines per color channel

Before enabling the LTDC we must configure the clock system. The LTDC uses a specific clock LCD_CLOCK to generate the pixel clock signal and it must be configured and enabled during the system initialization phase:

```
1 void system_clock_config(void)
2 {
3     RCC_OscInitTypeDef RCC_OscInitStruct;
```

```

4  RCC_ClkInitTypeDef RCC_ClkInitStruct;
5  RCC_PeriphCLKInitTypeDef PeriphClkInit
6
7  /* Configure the main internal regulat
8  __HAL_RCC_PWR_CLK_ENABLE();
9
10 __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_RE
11
12 /* Initialize the CPU, AHB and APB bus
13 RCC_OscInitStruct.OscillatorType = RCC
14 RCC_OscInitStruct.HSISState = RCC_HSI_C
15 RCC_OscInitStruct.HSICalibrationValue
16 RCC_OscInitStruct.PLL.PLLState = RCC_P
17 RCC_OscInitStruct.PLL.PLLSource = RCC_
18 RCC_OscInitStruct.PLL.PLLM = 8;
19 RCC_OscInitStruct.PLL.PLLN = 180;
20 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_
21 RCC_OscInitStruct.PLL.PLLQ = 7;
22 HAL_RCC_OscConfig(&RCC_OscInitStruct);
23
24 /* Activate the Over-Drive mode */
25 HAL_PWREx_EnableOverDrive();
26
27 /* Initialize the CPU, AHB and APB bus
28 RCC_ClkInitStruct.ClockType = RCC_CLOC
29 RCC_ClkInitStruct.SYSCLKSource = RCC_S
30 RCC_ClkInitStruct.AHBCLKDivider = RCC_
31 RCC_ClkInitStruct.APB1CLKDivider = RCC
32 RCC_ClkInitStruct.APB2CLKDivider = RCC
33
34 HAL_RCC_ClockConfig(&RCC_ClkInitStruct
35
36 /* initialize LTDC LCD clock */
37 PeriphClkInitStruct.PeriphClockSelecti
38 PeriphClkInitStruct.PLLSAI.PLLSAIN = 6
39 PeriphClkInitStruct.PLLSAI.PLLSAIR = 5
40 PeriphClkInitStruct.PLLSAIDivR = RCC_P
41 HAL_RCCEx_PeriphCLKConfig(&PeriphClkIr
42 }

```

The HAL_LTDC_MspInit() function, called at the beginning of LTDC_init() enables the LTDC peripheral clock and takes care of the low level hardware initialization:

```

1  void HAL_LTDC_MspInit(LTDC_HandleTypeDef
2  {
3      GPIO_InitTypeDef GPIO_InitStructure;
4
5      /* Enable the LTDC clock */
6      __HAL_RCC_LTDC_CLK_ENABLE();
7
8      /* Enable GPIOs clock */
9      __HAL_RCC_GPIOA_CLK_ENABLE();
10     __HAL_RCC_GPIOB_CLK_ENABLE();
11     __HAL_RCC_GPIOC_CLK_ENABLE();
12     __HAL_RCC_GPIOD_CLK_ENABLE();
13     __HAL_RCC_GPIOF_CLK_ENABLE();
14     __HAL_RCC_GPIOG_CLK_ENABLE();
15
16     /* GPIOs Configuration */
17     /*

```

```

18      +-----+-----+
19      +                               LCD pins
20      +-----+-----+
21      | LCD_TFT R2 <--> PC.10 | LCD_T
22      | LCD_TFT R3 <--> PB.00 | LCD_T
23      | LCD_TFT R4 <--> PA.11 | LCD_T
24      | LCD_TFT R5 <--> PA.12 | LCD_T
25      | LCD_TFT R6 <--> PB.01 | LCD_T
26      | LCD_TFT R7 <--> PG.06 | LCD_T
27      +-----+-----+
28      | LCD_TFT HSYNC <--> PC.
29      | LCD_TFT CLK  <--> PG.
30      +-----+-----+
31      */
32
33      /* GPIOA configuration */
34      GPIO_InitStructure.Pin = GPIO_PIN_3 |
35      GPIO_InitStructure.Mode = GPIO_MODE_AF
36      GPIO_InitStructure.Pull = GPIO_NOPULL;
37      GPIO_InitStructure.Speed = GPIO_SPEED_
38      GPIO_InitStructure.Alternate= GPIO_AF1
39      HAL_GPIO_Init(GPIOA, &GPIO_InitStructu
40
41      /* GPIOB configuration */
42      GPIO_InitStructure.Pin = GPIO_PIN_8 |
43      HAL_GPIO_Init(GPIOB, &GPIO_InitStructu
44
45      /* GPIOC configuration */
46      GPIO_InitStructure.Pin = GPIO_PIN_6 |
47      HAL_GPIO_Init(GPIOC, &GPIO_InitStructu
48
49      /* GPIOD configuration */
50      GPIO_InitStructure.Pin = GPIO_PIN_3 |
51      HAL_GPIO_Init(GPIOD, &GPIO_InitStructu
52
53      /* GPIOF configuration */
54      GPIO_InitStructure.Pin = GPIO_PIN_10;
55      HAL_GPIO_Init(GPIOF, &GPIO_InitStructu
56
57      /* GPIOG configuration */
58      GPIO_InitStructure.Pin = GPIO_PIN_6 |
59      HAL_GPIO_Init(GPIOG, &GPIO_InitStructu
60
61      /* GPIOB configuration */
62      GPIO_InitStructure.Pin = GPIO_PIN_0 |
63      GPIO_InitStructure.Alternate= GPIO_AF9
64      HAL_GPIO_Init(GPIOB, &GPIO_InitStructu
65
66      /* GPIOG configuration */
67      GPIO_InitStructure.Pin = GPIO_PIN_10 |
68      HAL_GPIO_Init(GPIOG, &GPIO_InitStructu
69  }

```

To display an image we must convert an image file to an array (possibly a const one, so it can be stored in flash memory) of bytes. To do this I used **LCD image converter**, a simple but powerful application that can convert

a file to a variety of different pixel formats:

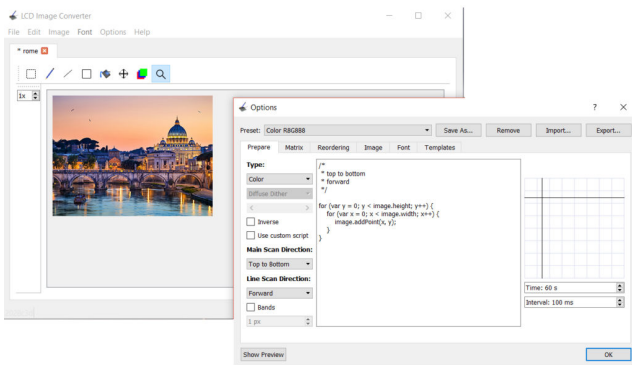


figure 6: LTDC image converter is used to generate a RGB888 image array

Once the image is converted to a byte array the generated header file is included and the array address can be used as the frame buffer starting address in the **LTDC_LxCFBAR** register. Layer window parameters are configured according to the image size (240 x 320, I rotated the image to fit the display in portrait mode).

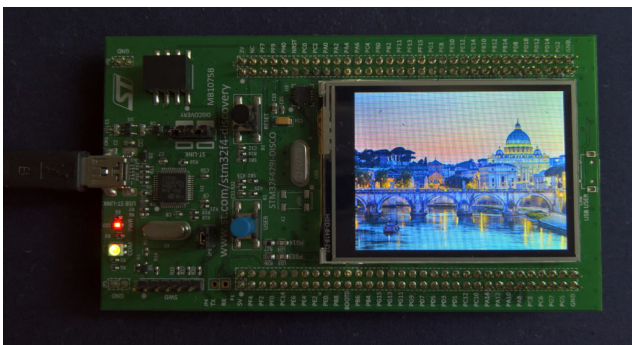


figure 7: a 240 x 320 (rotated) image displayed

The second layer can be enabled as well and its contents drawn on top of layer 1. LTDC can manage transparency using the values in the

LTDC_LxCACR *Layer x Constant Alpha*

Configuration Register and

LTDC_LxBFCR *Layer x Blending Factor*

Configuration Register: here I used a constant alpha of 255 to obtain a 100% opacity (the value in the constant alpha register is divided by 255 by hardware so for example a value of 128 represents an alpha value of 0.5). Since the layer window is smaller than the display area the default layer background color is set to a transparent black (otherwise the default layer background color is used if the layer window is smaller than the display). The image is 110 x 110 pixels and the pixel format is ARGB8888 (the alpha channel is used to draw transparent pixels). Note that the

LTDC_LxCBLR and

LTDC_LxCBLNR registers are configured according to the image size: the LTDC always starts fetching data from the address defined in the **LTDC_LxCFBAR** register. I added the following lines of code to the LTDC_init() function to configure and enable layer 2:

```
1  /* configure layer 2 */
2  LTDC_Layer2->WHPCR = 139 + 20 << LTDC_Lx
3  LTDC_Layer2->WVPCR = 113 + 200 << LTDC_L
4  LTDC_Layer2->PFCR = 0x00;
5  LTDC_Layer2->DCCR = 0x00 << LTDC_LxDCCR
6  LTDC_Layer2->CFBAR = (uint32_t)stamp;
7  LTDC_Layer2->CFBLR = 110 * 4 << LTDC_LxC
8  LTDC_Layer2->CFBLNR = 110;
```

```

9  LTDC_Layer2->CACR = 255;
10 LTDC_Layer2->CR  |= LTDC_LxCR_LEN;

```



figure 8: the layer window must be inside the active display area. layer2 image is in ARGB8888 format, allowing transparent pixels to show through

If we want to display portion of an image, we must configure

LTDC_LxCBLR and **LTDC_LxCBLNR** accordingly:

```

1  /* configure layer 1 */
2  LTDC_Layer1->WHPCR = 129 + 50 << LTDC_LxW
3  LTDC_Layer1->WVPCR = 103 + 50 << LTDC_LxW
4  LTDC_Layer1->PFCR = 0x01; // RGB888 pix
5  LTDC_Layer1->DCCR = 0xFF << LTDC_LxDCCR_D
6  LTDC_Layer1->CFBAR = (uint32_t)image;
7  LTDC_Layer1->CFBLR = 240 * 3 << LTDC_LxCF
8  LTDC_Layer1->CFBLNR = 100;
9  LTDC_Layer1->CACR = 255;

```

Now I'm just showing 100 x 100 pixels of the layer 1 image so I configured the color buffer line length as 100 and the color buffer number of lines as 100. The line pitch value indicates that a framebuffer line is still 240 * 3 bytes long so the controller knows how to fetch bytes from the frame buffer correctly. I also moved the start of the

window adding an offset of 50 pixels and 50 scan lines. The default background color is used where the layer isn't used (the layer background color is a solid green):

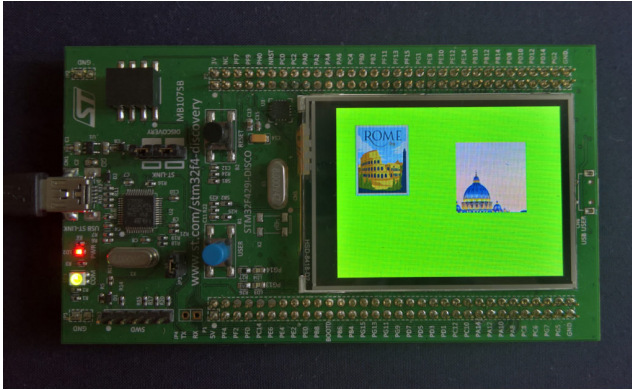


figure 9: layer1 window is resized and repositioned inside active display area

Using two layers and playing with the layer window size and position allows to create simple animations by simply moving the layer window around the frame:

```

1  #define SIZE                110
2  #define DISPLAY_WIDTH      240
3  #define DISPLAY_HEIGHT     320
4  #define ACTIVE_AREA_START_X 30
5  #define ACTIVE_AREA_START_Y 4
6
7  int main(void)
8  {
9      HAL_Init();
10     system_clock_config();
11     ILI9341_init();
12     LTDC_init();
13
14     double x_offset = 10;
15     double y_offset = 10;
16     double vx = 0.1;
17     double vy = 0.1;
18
19     for(;;)
20     {
21         if (x_offset <= 0 || x_offset + SIZE
22             vx = -vx;
23         if (y_offset <= 0 || y_offset + SIZE
24             vy = -vy;
25         x_offset += vx;
26         y_offset += vy;

```

```

27     LTDC_Layer2->WHPCR = ACTIVE_AREA_STA
28     LTDC_Layer2->WVPCR = ACTIVE_AREA_STA
29     LTDC->SRCR = LTDC_SRCR_VBR;
30     while ((LTDC->CDSR & LTDC_CDSR_VSYNCR) != 0)
31         ;
32     while ((LTDC->CDSR & LTDC_CDSR_VSYNCR) != 0)
33         ;
34 }
35 }

```

Shadow configuration registers are reloaded each vertical blanking period (after the last line has been drawn) and the code waits for the next frame by polling the **VSYNCS** flag of the **LTDC_CDSR** *Current Display Status Register*, whose bits contain the state of the synchronization signals (high if they're asserted, no matter the polarity configured). Running the code we get a nice smooth animation:



LTDC interrupts

The LTDC controller has four interrupts logically OR-ed into two interrupt request lines:

- **Register Reload Interrupt**, generated as soon as the shadow registers are reloaded

- **Line Interrupt**, generated when a line number (programmed into **LTDC_LIPCR** *Line Interrupt Position Control Register*) is reached
- **Transfer Error Interrupt**, generated when an AHB bus error occurs during a transfer
- **FIFO underrun Interrupt**, generated when a pixel is requested from an empty layer FIFO

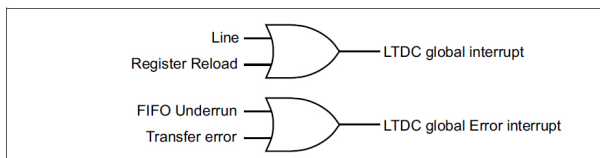


figure 10: LTDC interrupts and IRQ lines

The Line and Register Reload interrupts are useful to synchronize the code with the controller.

Using double buffering

Double buffer is used when we want the code to write on a frame buffer while another buffer is being read by the LTDC. This avoids corrupting the data being displayed on the screen. The buffers are switched during the vertical blanking period using polling or interrupts.

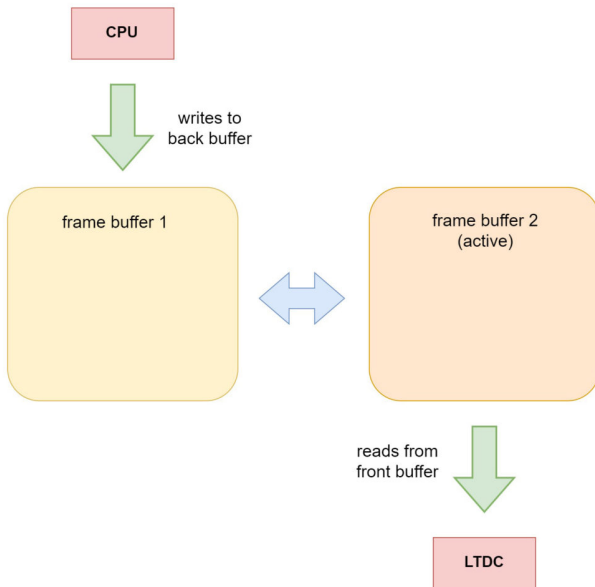


figure n: while the code writes to the back buffer the LTDC fetches data from the front (active) buffer. Framebuffers are switched during vertical blanking period.

In this example the framebuffers have a RGB888 color depth and for a 240×320 display that makes 225 KiB of memory for each buffer (3 bytes per pixel x 240 x 320 pixels) so they must be stored in external SRAM (the STM32F429I-DISCOVERY has a 64Mbit external SRAM so we're good). The **FMC Flexible Memory Controller** has to be initialized and the address of the two frame buffers has to be configured. Drawing on the framebuffer is a matter of writing the right bytes in order to change the color. Once all pixels are drawn (bytes are written) the buffers are switched and the code can draw the next frame:

```

1  #define SDRAM_ADDR
2
3  /* double buffering (RGB888 frame buffer) */
4  #define FRAMEBUFFER_SIZE
5  #define FRAMEBUFFER1_ADDR
6  #define FRAMEBUFFER2_ADDR
7
8  enum framebuffer
9  {
10     FRAMEBUFFER1, FRAMEBUFFER2
11 };
12
13 static enum framebuffer active = FRAMEBU
14
15 void LTDC_init(void)
16 {
17     /* initialize SDRAM */
18     SDRAM_init();
19
20     /* fill framebuffers with black */
21     for (int i = 0 ; i < DISPLAY_WIDTH * D
22         ((int8_t*)FRAMEBUFFER1_ADDR)[i] = 0x
23     for (int i = 0 ; i < DISPLAY_WIDTH * D
24         ((int8_t*)FRAMEBUFFER2_ADDR)[i] = 0x
25
26     /* LTDC initialization code */
27     /* ..... */
28
29     /* layer1 initialization code */
30     /* ..... */
31     LTDC_Layer1->CFBAR = FRAMEBUFFER1_ADDR;
32     active = FRAMEBUFFER1;
33
34     /* other LTDC initialization code */
35     /* ..... */
36 }
37
38 void LTDC_switch_framebuffer(void)
39 {
40     if (active == FRAMEBUFFER1)
41     {
42         LTDC_Layer1->CFBAR = FRAMEBUFFER2_AD
43         active = FRAMEBUFFER2;
44     }
45     else
46     {
47         LTDC_Layer1->CFBAR = FRAMEBUFFER1_AD
48         active = FRAMEBUFFER1;
49     }
50     LTDC->SRCR = LTDC_SRCR_VBR;
51     while ((LTDC->CDSR & LTDC_CDSR_VSYNCS)
52         ;
53 }
54
55 uint8_t *LTDC_get_backbuffer_address(voi
56 {
57     if (active == FRAMEBUFFER1)
58         return (int8_t*)FRAMEBUFFER2_ADDR;
59     else
60         return (int8_t*)FRAMEBUFFER1_ADDR;
61 }

```

Now as soon as a frame is done with,
calling LTDC_switch_framebuffer()
waits for the vertical synchronization

period and swaps the buffers. If the code is faster than the display refresh rate (70Hz in our case) it waits for the LTDC to complete drawing the frame.

In the next post I'm going to use the double buffer technique to draw and animate sprites.

[source code](#)


[LTDC datasheet](#)

14 RESPONSES

 **Comments** 14

 **Pingbacks** 0



Sunnipaul  November 14, 2017 at 8:05 am

Hello,

Thank you for your sharing.

I copied and ran your code. But it only works when starting sing step debug from here:

```
ILI9341_write_command(ILI9341_DISPLAY_FUNCTION_CONTROL);
```

until the end of the function ILI9341_Init.
Otherwise it will show white screen.

Could you please help me figure out why?

Reply



Luca Davidian

 November 14, 2017 at 9:49 am

if it works when you single step during debugging must be a timing issue in the init sequence

for the display. Do you initialize the clock system correctly in your main? Have you tried modifying the HAL_Delay() call at the end of the sequence?

Reply



Sunnipaul ⌚ November 15, 2017 at 12:56 am

Thank you for your suggestion.

I indeed added several HAL_Delays (also made delay longer) among the sequence. It gave the same white screen result. I asked someone else for help. He added delays after every ili9341_send_command and ili9341_send_data function then it worked. He explained to me that F429 is too fast for the screen to respond.

Reply



Luca Davidian

⌚ November 15, 2017 at 11:49 am

Speed of the micro is not an issue, as long as you respect the timings provided in the datasheet of the device you are driving. I experimented quite a bit with the initialization sequence and came up with a working sequence of commands. Are you using the same STM32 discovery board (and microcontroller) as I am? Did you double-check the clock system initialization routine?

Reply



sunnipaul

⌚ November 17, 2017 at 1:22 am

I've remove the delay function calls and changed Spi_transmit to

HAL_SPI_Transmit

instead. It worked then.

However the

HAL_SPI_Transmit has

more than 100 lines of

code. Which is also a

considerable delay.

Here is the system_config

I used:

```
void
system_clock_config(void)
{
    RCC_OscInitTypeDef
    RCC_OscInitStruct;
    RCC_ClkInitTypeDef
    RCC_ClkInitStruct;
    RCC_PeriphCLKInitTypeDef
    PeriphClkInitStruct;

    /**Configure the main
    internal regulator output
    voltage
    */
    __HAL_RCC_PWR_CLK_ENABLE();

    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /**Initializes the CPU,
    AHB and APB busses clocks
    */
    RCC_OscInitStruct.OscillatorType
    = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSISState
    = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue
    = 16;
    RCC_OscInitStruct.PLL.PLLState
    = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource
    = RCC_PLLSOURCE_HSI;
```



```
RCC_OscInitStruct.PLL.PLLM
= 8;

RCC_OscInitStruct.PLL.PLLN
= 180;

RCC_OscInitStruct.PLL.PLLP
= RCC_PLLP_DIV2;

RCC_OscInitStruct.PLL.PLLQ
= 7;

HAL_RCC_OscConfig(&RCC_OscInitStruct);

/**Activate the Over-Drive
mode
*/
HAL_PWREx_EnableOverDrive();

/**Initializes the CPU,
AHB and APB busses clocks
*/
RCC_ClkInitStruct.ClockType
=
RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK|RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource
= RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider
= RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider
= RCC_HCLK_DIV4;
RCC_ClkInitStruct.APB2CLKDivider
= RCC_HCLK_DIV2;

HAL_RCC_ClockConfig(&RCC_ClkInitStruct,
FLASH_LATENCY_5);

PeriphClkInitStruct.PeriphClockSelection
= RCC_PERIPHCLK_LTDC;

PeriphClkInitStruct.PLLSAI.PLLSAIN
= 60;

PeriphClkInitStruct.PLLSAI.PLLSAIR
= 5;

PeriphClkInitStruct.PLLSAIDivR
= RCC_PLLSAIDIVR_4;

HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct);
}
```

Reply



Luca Davidian

🕒 November 17,
2017 at 4:21 pm

I downloaded the source code from the page and created a new project with those same files. I added the relevant STM32CubeHAL and CMSIS files and it works on my STM32F429Discovery. Which IDE are you using? Are you using the same board or are you on a different micro? Did you check the display connections?

Reply



Bernd 🕒 January 24, 2018 at 11:42 am

Hey Luca,

Thanks for sharing this. It helped me to understand the configuration of the LTDC and the layer concept.

I still have one problem and wonder if you can point me in the right direction.

My demo project is from another source but the basic things are set up and the display is working.

A background image (in display resolution) is shown on layer 1 and tried to only show a part of it as you did in

figure 9.

Only the part with the cathedral dome is visible. If I try to setup something similar, I see the upper left corner of the full image (in the wanted size of the partial viewing area) shifted to the wanted position. I have absolutely no clue what I do wrong. From your description I understood that I have to set line length and number of lines to fit my wanted partial viewing area size. I set up the X and Y offset of the partial viewing area which works as expected. The link to the framebuffer still points to the beginning of the image array and the pitch still fits to the full image width. So with this information the LTDC should pick the correct part from the image array, but in my case it always picks the part starting from upper left corner which is (0/0). Any idea?

Reply



Luca Davidian

🕒 February 2, 2018 at 7:11 pm

Hi Bernd. The window horizontal and vertical start and stop positions represent respectively the first and the last pixels and the first and the last lines of the image that are displayed on the screen inside the active area of the display (that's why you take into consideration the horizontal and vertical back porch timing parameters). The LTDC always starts fetching data from the beginning of the frame buffer base address in LTDC_LxCFBAR register. The line length, line pitch and number of lines parameters are used by the LTDC

to know how many bytes to fetch from the buffer, so if you want to display a portion of the image you have to program the line length and the number of lines accordingly (the line pitch is always the number of bytes in one line). Note that the parameters in LTDC_LxCFBLR register are in units of bytes, so you must take into account the color format of the image pixels (bytes per pixel, 3 for a RGB image). So if you're displaying a partial image in a window it'll always show the upper left corner of the image. To display an actual window (a kind of "viewport") inside your image you have to shift the base address of the frame buffer in the following way:

$$\text{base address} + ((\text{start_x} + \text{start_y} * \text{DISPLAY_WIDTH}) * \text{BPP})$$

Where BPP stands for bytes per pixels and depends on the color format. In this case you'll get a window inside the image as if the the entire image was displayed on the whole screen, but only a portion is shown. Try this snippet, it should display the lower right corner of the image:

```
/* configure layer 1 */
LTDC_Layer1->WHPCR = 269 <<
LTDC_LxWHPCR_WHSPP0S_Pos | 30 +
200 << LTDC_LxWHPCR_WHSTPOS_Pos;
// window horizontal start/stop
positions LTDC_Layer1->WVPCR = 323
<< LTDC_LxWVPCR_WVSPP0S_Pos | 4 +
```

```

200 << LTDC_LxWVPCR_WVSTPOS_Pos;

// window vertical start/stop

positions LTDC_Layer1->PFCR =

0x01; // RGB888 pixel format

LTDC_Layer1->DCCR = 0xFF < <

LTDC_LxDCCR_DCALPHA_Pos | 0xFF <<

LTDC_LxDCCR_DCGREEN_Pos; // layer

default color LTDC_Layer1->CFBAR =

(uint32_t)(0 + (200 + 200 *

240) * 3); // frame buffer start

address (SHIFTED)

LTDC_Layer1->CFBLR = 240 * 3 < <

LTDC_LxCFBLR_CFBP_Pos | 40 * 3 + 3

<< LTDC_LxCFBLR_CFBLL_Pos; //

frame buffer line length and pitch

LTDC_Layer1->CFBLNR = 120; //

frame buffer line number

LTDC_Layer1->CACR = 255; //

constant alpha

LTDC_Layer1->CR |= LTDC_LxCR_LEN;

// enable Layer1

```

Reply



Bernd

🕒 February 7, 2018 at 4:49 pm

Thank you very much!

Adding the offset to the
frambuffer base address
works as expected:
base address + ((start_x +
start_y * DISPLAY_WIDTH)
* BPP)

Reply



gilloup 🕒 January 31, 2018 at 8:39 am

Excellent tutorial. It works also for me.
But, as Sunnipaul, I need to add delays :
with “Hal_delay(10);” after the code line

“SPI_transmit(&data, 1);” (inside the functions write_data and write_command) , it is OK. Without, it doesn't work. ...

Reply



andreaahmed ⌚ June 8, 2018 at 12:33 am

Would you help me with that problem please

<https://electronics.stackexchange.com/questions/378714/ltcd-stmf32-double-buffering-problem>

Reply



Luca Davidian

⌚ June 16, 2018 at 4:04 pm

You shouldn't initialize the two layers of LTDC for double buffering. As I explained in the article under “Using double buffering”, you only use one layer and change the base address of the layer (layer 1 in this case). What you do is the following: you set up two buffers in memory (generally external memory, in order to fit the entire display size in them), and initialize the LTDC base address with the starting address of one of them (this is your initial front buffer). The LTDC fetches data continuously from the buffer starting from the configured base address and when you're done updating the off-screen buffer you switch the addresses. So the front buffer becomes the off-screen back buffer (where you can update the scene) and the former back buffer becomes the new front buffer and is “presented” to the screen.

Using two layers is used for overlays and other effects, since the two layers are both presented to the screen at the same time.

Reply



Neil March 20, 2019 at 12:57 am

Hi Luca,

Thank you very much for the tutorial. I haven't tried this code yet, but I'm confused about one thing:

As far as I understand, the TFT glass uses the RGB+Vsync+Hsync interface. The ILI9341 connects to that interface and presents the user with a simpler parallel or SPI interface.

At the very beginning, you configure the LTDC to work directly with the TFT glass, correct? Why then are you then connecting to the ILI9341... is this just an alternate way to drive the LCD display, or is this in addition to the LTDC driving the LCD glass?

I'm confused about the intention here.

Thanks.

Reply



redquartista April 13, 2019 at 5:29 pm

Hello, excellent article.

How can I print characters on LCD via LTDC on ST32F429i discovery board?

Thank you!

Reply

LEAVE A REPLY

Comment

Name *

Email *

Website

Post Comment

“People who think they know everything really annoy those of us who know we don’t.”

— *Bjarne Stroustrup*

Next quote »



hello world © 2019. All Rights Reserved.