DispHSTX

version 1.00

DVI (HDMI) and VGA display driver for Pico2 RP2350, using HSTX peripheral

(c) Miroslav Nemecek February 2025

Panda38@seznam.cz , hardyplotter2@gmail.com

https://github.com/Panda381/DispHSTX

https://www.breatharian.eu/hw/disphstx/index_en.html

Contents

1. Basic description

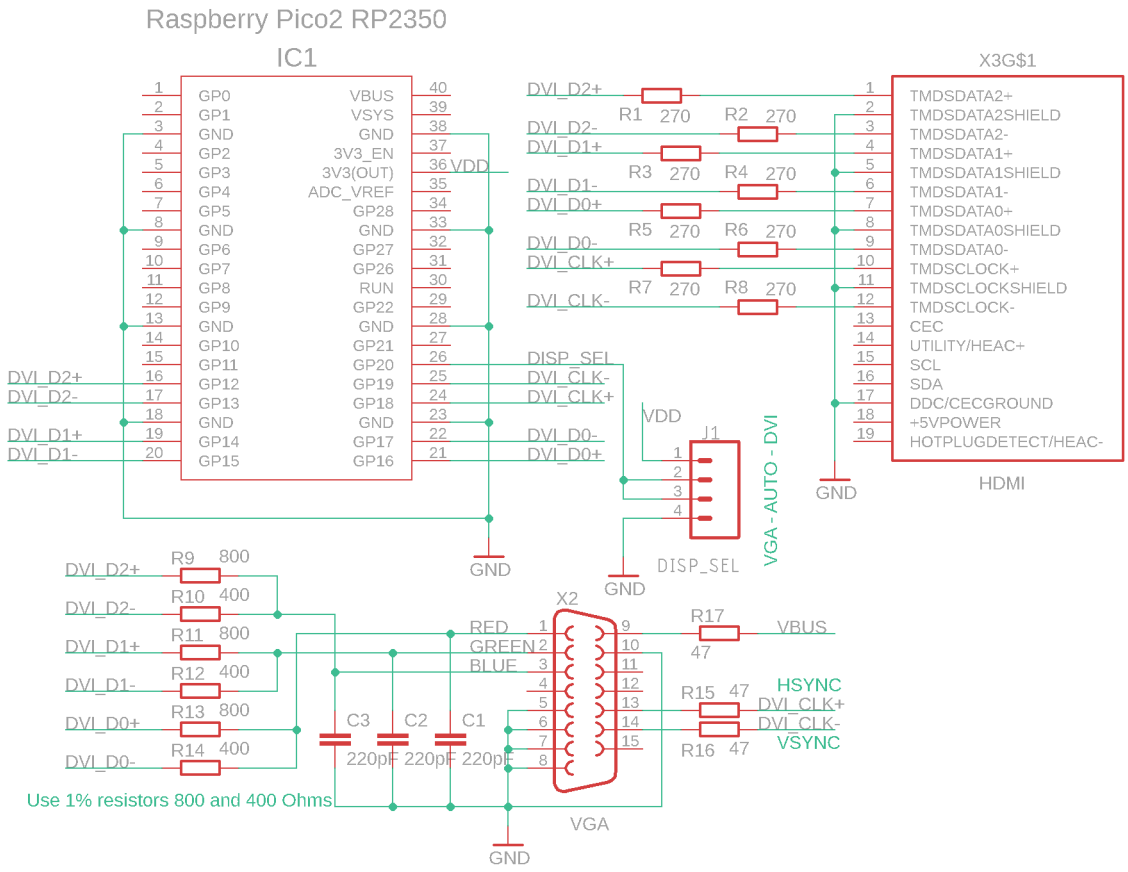
DispHSTX is a driver for Pico2 RP2350 microcontroller, both in ARMv8 and RISC-V Hazard3 mode, allowing to generate DVI (HDMI) and VGA video signal, using HSTX peripheral. It supports 40 different image formats - 9 modes of pixel graphics, 2 text modes, attribute compression, RLE and HSTX compression. Functions for drawing geometric shapes, images and texts are provided for 8 basic graphic modes. The drawing library supports a back buffer - it updates the boundaries indicating the modified "dirty area". If memory is insufficient, the back buffer can be used in strip mode. Several different image formats can be combined on one screen simultaneously by dividing the screen into strips and slots. In both modes, DVI (HDMI) and VGA, color resolution up to 16 bits per pixel is possible. Although in VGA mode the output is only via a 6-bit RGB converter, higher colour resolution is achieved by the Pulse Pattern Modulation PPM, the display is visually close to the 15-bit display. The timing of the generated video signal is limited only by the overclocking capabilities of the microcontroller used. It is usually possible to achieve a resolution of up to 1440x600 pixels.

DispHSTX is built on the PicoLibSDK library, but interface to original Raspberry SDK library is ready too.

2. License

DispHSTX driver source code is completely free to use for any purpose, including commercial use. It is possible to use and modify all or parts of the driver source code without restriction.

3. Schematic wiring diagram



The DVI (HDMI) and VGA connectors are connected to the same processor pins, GPIO 12 to GPIO 19. The selection of pins 12 to 19 is determined by the processor and cannot be changed. In the driver configuration there are 2 options for the pin order for the DVI (HDMI) output, but any other pin order can be selected after editing the code. Similarly, it is possible to select any pin order for the VGA connector, but it is recommended to keep the order used in the schematic above to allow VGA signal output from the PIO controller, which requires keeping the pin order.

The 400 and 800 ohm resistors used for the VGA output should be used within a 1% tolerance. If necessary, close values such as 390 and 820 ohms can be used, but the smoothness of the gradient transition may be degraded for multi-color modes (12 to 16 bits per pixel).

On the RGB outputs of the VGA connector 220pF capacitors are used to filter Pulse Pattern Modulation PPM. For video modes with a low system frequency, it may be beneficial to increase the value of the capacitors to reduce potential image grain. For video modes with high system frequency, it may be beneficial to increase the value of the capacitors to increase the sharpness of the image.

Any GPIO pin can be used for the switch to select VGA/DVI mode, or the switch does not need to be used. If the switch is not used or it is in the middle position, an automatic detection of the VGA monitor connection can be used to select the mode.

4. Files and building

The DispHSTX driver is primarily prepared for use with the PicoLibSDK library. The main driver files can be found in the _display/disphstx folder. The disphstx_dvi.* files provide support for the DVI (HDMI) output. The disphstx_vga.* files are used to handle the VGA output. The disphstx_vmode_time.* files contain video mode timing definitions - you can use them to define your own video mode. The disphstx_vmode_format.* files contain definitions of graphic formats. Rendering of graphic formats is done in the *_render.* files in the C codes - these functions are used only as a reference to the correct code functions. The actual rendering is done using functions optimized by the assembler, in the *.S files.

The _config.h file contains the default definitions of the driver parameter settings. If you need to change a configuration switch, specify it in the config.h file in the project folder. Default switches are used if they are not found in the project's config.h file. The main switch you will need to include in config.h is "#define USE_DISPHSTX 1". This will ensure that the DispHSTX driver functions are compiled and made available. The second important switch you may need to include is "#define USE_DRAWCAN 1". This will provide access to the DrawCan drawing library functions. For more info about projekt configuration switches, see chapter „Configuration“.

The HSTX driver is by default run in the second core of the processor. This is required by the fact that the driver operation must not be disturbed by anything to avoid dropping the generated signal. For this reason, no interrupts other than those from the HSTX driver may be running in the second core of the processor. All time-consuming operations must be performed from RAM, not from Flash - the linker must provide a "time_critical" section in RAM for this purpose.

Driver functionality is provided for both ARMv8 processor mode and RISC-V Hazard3 mode. However, it is necessary to take into account that the code for RISC-V mode is slightly slower (about 15%) and so the HSTX driver in RISC-V mode may load the processor a bit more than in ARM mode.

The driver in the basic full setup can take quite a lot of RAM for functions and auxiliary buffers (about 50 KB). You can reduce the memory requirements by limiting functionality - disabling VGA mode, reducing the maximum resolution, or disabling unused image formats. See the "Configuration" chapter for details.

5. Simple sample

```
int main()
{
	// initialize videomode 640x480
	int res = DispVMode640x480x8(0, NULL);

	// halt on memory error, blinking internal LED on Pico2 board
	while (res != DISPHSTX_ERR_OK)
		{ GPIO_Flip(LED_PIN); WaitMs(100); }

	// clear screen with blue background
	DrawClearCol(COL_BLUE);

	// draw red frame around the screen
	DrawFrameW(0, 0, DrawWidth(), DrawHeight(), COL_RED, 5);

	// draw text message
	const char msg[] = "Hello DispHSTX!";
	DrawText(msg, -1, (DrawWidth() - StrLen(msg)*8*3)/2,
		(DrawHeight() - 16*4)/2, COL_YELLOW, 3, 4);

	// delay 30 seconds
	WaitMs(30000);

	// terminate all current devices and free frame buffer
	DispHstxAllTerm();
	DispHstxFreeBuf();
}
```

For list of predefined video modes, see chapter „Predefined simple graphic video modes“. For list of drawing functions, see chapter „Drawing Canvas“.

6. Multi-slot sample

```
// halt on error
#define CHECK_ERR() while (res != DISPHSTX_ERR_OK) \
            { GPIO_Flip(LED_PIN); WaitMs(100); }

int main()
{
      // create buffers
      u8* buf_font = (u8*)malloc(sizeof(FontBold8x16));
      void* buf_text = malloc(20*1); // text screen 20 characters
      void* buf_img = malloc(Draw16Pitch(532)*400); // img screen
      void* buf_pat = malloc(56*54); // create pattern buffer
      u16* buf_pal = (u16*)malloc(256*2); // pattern palettes

      while ((buf_font == NULL) || (buf_text == NULL) ||
            (buf_img == NULL) || (buf_pat == NULL) ||
            (buf_pal == NULL)) { GPIO_Flip(LED_PIN); WaitMs(100); }

      memcpy(buf_font, FontBold8x16, sizeof(FontBold8x16));
      memcpy(buf_text, "  MultiSlot Screen  ", 20); // set text
      memcpy(buf_pat, ImgPattern, 56*54); // copy pattern to RAM
      memcpy(buf_pal, ImgPattern_Pal, 256*2); // copy pattern

      // Initialize videomode state descriptor to 640x480@60Hz
      sDispHstxVModeState* vmode = &DispHstxVMode; // descriptor
      DispHstxVModeInitTime(vmode,
            &DispHstxVModeTimeList[vmodetime_640x480_fast]);

      // add strip "0" for text screen (height 80 lines)
      int res = DispHstxVModeAddStrip(vmode, 80);
      CHECK_ERR();

      // add text screen slot "0"
      u16 pal_text[2] = { COL16_DKGREEN, COL16_YELLOW };
      u32 pal_vga[2*2];
      res = DispHstxVModeAddSlot(vmode, 4, 5, -1,
            DISPHSTX_FORMAT_MTEXT, buf_text, 20,
            pal_text, pal_vga, buf_font, 16, 0, 0);
      CHECK_ERR();

      // add strip "1" for graphics screen
      res = DispHstxVModeAddStrip(vmode, 400);
      CHECK_ERR();

      // add pattern screen slot "1-0"
      res = DispHstxVModeAddSlot(vmode, 1, 1, 54,
            DISPHSTX_FORMAT_PAT_8_PAL, buf_pat, 56,
            buf_pal, DispHstxDefVgaPal, NULL, 54, 0, 0);
      CHECK_ERR();

      // add graphics screen slot "1-1" - 532x400/16-bit
      res = DispHstxVModeAddSlot(vmode, 1, 1, 532,
            DISPHSTX_FORMAT_16, buf_img, Draw16Pitch(532),
            NULL, NULL, NULL, 0, 0, 0);
      CHECK_ERR();

      // add pattern screen slot "1-2"
      res = DispHstxVModeAddSlot(vmode, 1, 1, 54,
            DISPHSTX_FORMAT_PAT_8_PAL, buf_pat, 56, buf_pal,
            DispHstxDefVgaPal, NULL, 54, 0, 0);
      CHECK_ERR();

      // activate videomode
      DispHstxSelDispMode(DISPHSTX_DISPMODE_NONE, vmode);

      // link graphics screen "1-1" to 16-bit draw canvas
```

```
        DispHstxLinkCan(&vmode->strip[1].slot[1], &DrawCan16);

        // display image to 16-bit canvas
        Draw16Img(0, 0, Img532x400_16b, 0, 0, 532, 400, 1064);

        // delay 10 seconds
        WaitMs(10000);

        // terminate all current devices
        DispHstxAllTerm();

        // free buffers
        free(buf_font);
        free(buf_text);
        free(buf_img);
        free(buf_pat);
        free(buf_pal);
}
```

For list of driver function, see chapter „Driver functions“. There is no need for a slot separator because the slots use the same output format.

7. Multi-format sample

```
// halt on error
#define CHECK_ERR() while (res != DISPHSTX_ERR_OK) \
            { GPIO_Flip(LED_PIN); WaitMs(100); }
#define GAP 20                  // gap between slots
#define COL COL16_BLACK // color of the gap

int main()
{
      // create buffers
      u8* fnt = (u8*)malloc(sizeof(FontBold8x16)); // font buffer
      void* txt = malloc(20*1); // text screen 20 characters
      while ((fnt==NULL)||(txt==NULL))
            { GPIO_Flip(LED_PIN); WaitMs(100); }
      memcpy(fnt, FontBold8x16, sizeof(FontBold8x16)); // copy font
      memcpy(txt, " MultiFormat Screen ", 20); // set text

      // Initialize videomode state descriptor to 640x480@60Hz
      sDispHstxVModeState* v = &DispHstxVMode; // descriptor
      DispHstxVModeInitTime(v,
            &DispHstxVModeTimeList[vmodetime_640x480_fast]);

      // add strip "0" for text screen (height 30 lines)
      int res = DispHstxVModeAddStrip(v, 30);
      CHECK_ERR();

      // add text screen slot "0"
      u16 palt[2] = { COL16_DKBLUE, COL16_YELLOW };
      u32 palv[2*2];
      res = DispHstxVModeAddSlot(v, 4, 2, -1, DISPHSTX_FORMAT_MTEXT,
            txt, 20, palt, palv, fnt, 16, 0, 0);
      CHECK_ERR();

      // add strip "1" for 1st graphics row
      res = DispHstxVModeAddStrip(v, 150);
      CHECK_ERR();

      // add color format 1
      res = DispHstxVModeAddSlot(v, 1, 1, 200, DISPHSTX_FORMAT_1_PAL,
            NULL, 28, NULL, NULL, NULL, 0, COL, GAP);
      CHECK_ERR();
      DispHstxLinkCan(&v->strip[1].slot[0], &DrawCan1);
      Draw1Img(0, 0, Img1b, 0, 0, 200, 150, 28);

      // add color format 2
      res = DispHstxVModeAddSlot(v, 1, 1, 200, DISPHSTX_FORMAT_2,
            NULL, 56, NULL, NULL, NULL, 0, COL, GAP);
      CHECK_ERR();
      DispHstxLinkCan(&v->strip[1].slot[1], &DrawCan2);
      Draw2Img(0, 0, Img2b, 0, 0, 200, 150, 56);

      // add color format 3
      res = DispHstxVModeAddSlot(v, 1, 1, 200, DISPHSTX_FORMAT_3_PAL,
            NULL, 88, NULL, NULL, NULL, 0, COL, GAP);
      CHECK_ERR();
      DispHstxLinkCan(&v->strip[1].slot[2], &DrawCan3);
      Draw3Img(0, 0, Img3b, 0, 0, 200, 150, 88);

      // add strip "2" for 2nd graphics row
      res = DispHstxVModeAddStrip(v, 150);
      CHECK_ERR();

      // add color format 4
      res = DispHstxVModeAddSlot(v, 1, 1, 200, DISPHSTX_FORMAT_4_PAL,
            NULL, 108, NULL, NULL, NULL, 0, COL, GAP);
      CHECK_ERR();
```

```
	DispHstxLinkCan(&v->strip[2].slot[0], &DrawCan4);
	Draw4Img(0, 0, Img4b, 0, 0, 200, 150, 108);

	// add color format 6
	res = DispHstxVModeAddSlot(v, 1, 1, 200, DISPHSTX_FORMAT_6_PAL,
		NULL, 172, NULL, NULL, NULL, 0, COL, GAP);
	CHECK_ERR();
	DispHstxLinkCan(&v->strip[2].slot[1], &DrawCan6);
	Draw6Img(0, 0, Img6b, 0, 0, 200, 150, 172);

	// add color format 8
	res = DispHstxVModeAddSlot(v, 1, 1, 200, DISPHSTX_FORMAT_8,
		NULL, 216, NULL, NULL, NULL, 0, COL, GAP);
	CHECK_ERR();
	DispHstxLinkCan(&v->strip[2].slot[2], &DrawCan8);
	Draw8Img(0, 0, Img8b, 0, 0, 200, 150, 216);

	// add strip "3" for 3rd graphics row
	res = DispHstxVModeAddStrip(v, 150);
	CHECK_ERR();

	// add color format 12
	res = DispHstxVModeAddSlot(v, 1, 1, 200, DISPHSTX_FORMAT_12,
		NULL, 324, NULL, NULL, NULL, 0, COL, GAP);
	CHECK_ERR();
	DispHstxLinkCan(&v->strip[3].slot[0], &DrawCan12);
	Draw12Img(0, 0, Img12b, 0, 0, 200, 150, 324);

	// add color format 15
	res = DispHstxVModeAddSlot(v, 1, 1, 200, DISPHSTX_FORMAT_15,
		NULL, 428, NULL, NULL, NULL, 0, COL, GAP);
	CHECK_ERR();
	DispHstxLinkCan(&v->strip[3].slot[1], &DrawCan15);
	Draw15Img(0, 0, Img15b, 0, 0, 200, 150, 428);

	// add color format 16
	res = DispHstxVModeAddSlot(v, 1, 1, 200, DISPHSTX_FORMAT_16,
		NULL, 428, NULL, NULL, NULL, 0, COL, GAP);
	CHECK_ERR();
	DispHstxLinkCan(&v->strip[3].slot[2], &DrawCan16);
	Draw16Img(0, 0, Img16b, 0, 0, 200, 150, 428);

	// activate videomode
	DispHstxSelDispMode(DISPHSTX_DISPMODE_NONE, v);

	// delay 10 seconds
	WaitMs(10000);

	// terminate all current devices
	DispHstxAllTerm();

	// free buffers
	int i, j;
	for (i = 0; i < v->stripnum; i++)
		for (j = 0; j < v->strip[i].slotnum; j++)
			free(v->strip[i].slot[j].buf);
	free(fnt);
}
```

8. Drawing sample

```
// randomize
#define RAND() { x = RandS16MinMax(-100, 700); \
            y = RandS16MinMax(-100, 600);\
            w = RandU16MinMax(10, 100);   \
            h = RandU16MinMax(10, 100);   \
            col = COL8_RANDOM; }

int main()
{
      // initialize videomode 640x480
      int res = DispVMode640x480x8(0, NULL);

      // halt on memory error
      while (res != DISPHSTX_ERR_OK)
            { GPIO_Flip(LED_PIN); WaitMs(100); }

      // loop about 10 seconds
      int k, x, y, w, h, col;
      for (k = 7000; k > 0; k--)
      {
            // Draw rectangle
            RAND(); DrawRect(x, y, w, h, col);

            // Draw frame
            RAND(); DrawFrameW(x, y, w, h, col, RandU8MinMax(1,5));

            // Draw line
            RAND(); DrawLineW(x, y, x+w-50, y+h-50, col,
                  RandU8MinMax(1,5), 1);

            // Draw round
            RAND(); DrawRound(x, y, w, col, RandU8());
```

```
            // Draw circle
            RAND(); DrawCircle(x, y, w, col, RandU8());

            // Draw ring
            RAND(); DrawRing(x, y, w, h, col, RandU8());

            // Draw triangle
            RAND(); DrawTriangle(x, y, x+w-50, y+h-50,
                  x+h-50, y+w-50, col);

            // Draw character
            RAND(); DrawChar(RandU8(), x, y, col,
                  RandU8MinMax(1,2), RandU8MinMax(1,2));

            // Draw text
            RAND(); DrawText("DispHSTX", 8, x, y, col, 1, 1);

            // Draw ellipse
            RAND(); DrawEllipse(x, y, w, h, col, RandU8());

            // Draw filled ellipse
            RAND(); DrawFillEllipse(x, y, w, h, col, RandU8());

            // short delay
            WaitMs(1);
        }

        // terminate all current devices and free frame buffer
        DispHstxAllTerm();
        DispHstxFreeBuf();
}
```

For list of drawing functions, see chapter „Drawing Canvas“.

9. Timing modes

The DispHSTX driver is controlled by the system clock by default. This requires that the system clock is set to 5 or 10 times the pixel clock by the driver.

There is another alternative yet - by turning the DISPHSTX_USEPLL switch to 1, another variant is used. Specifying the DispHstxClockReinit() function at the beginning of the program redefines the clock control in the processor so that all peripherals and the system clock are controlled from the common PLL generator PLLUSB, while the PLLSYS generator is used only for controlling the HSTX driver. The clock generated by the PLLUSB generator should preferably have a frequency equal to the integer multiple of the USB clock, that is 48 MHz. This allows the HSTX driver timing to be controlled independently of the system clock. It is even possible to overclock the HSTX clock a little more than would be possible when driving from the system clock. If the HSTX driver malfunctions due to overclocking, there is no risk of the entire system crashing, the system can even check the correct function of the driver and adjust its clock if necessary. However, it is important to remember that the processor must be fast enough to generate the display signal in this mode, so it is recommended that only experienced users use this mode.

The disphstx_vmode_time.* files contain video mode timing definitions. The timing specifies the timing sequence of the generated video signal, which is independent of the format of the video content. It is even possible to change the video format configuration without losing video synchronization if the same timing definition is used (DispHstxExchange() function).

One of the main criteria determining the timing of video modes is the achievable processor overclocking speed. The RP2350 processor is clocked by the manufacturer at a maximum of 150 MHz as standard. Several video modes can be used with this limitation - e.g. 640x480 pixels in slow mode. However, many video modes require the processor to be overclocked to a higher frequency, outside the tolerance guaranteed by the manufacturer. In practice, it has been tested that most RP2350 processors operate reliably up to a frequency of 400 MHz, some up to 420 MHz. However, this increases the risk of processor damage and reduces its lifespan, so it is necessary to approach such exceeding of parameters with caution.

The timing definitions are designed to be supported not only by the HDMI display, which is very tolerant of non-standard formats, but also by the VGA display, which is limited to standard video modes. Therefore, the timing definitions used are derived from several of the following standard VGA video modes:

640x350@70Hz/31.5kHz ... VGA, pixel clock 25.176 MHz, H-sync POS, V-sync NEG, horizontal active 640 + front porch 16 + sync 96 + back porch 48 = total 800, hfreq 31.470kHz, vertical active 350 + front porch 37 + sync 2 + back porch 60 = total 449, vfreq 70.089Hz

640x480@60Hz/31.5kHz ... VESA DMT ID 04h, pixel clock 25.175 MHz, H-sync NEG, V-sync NEG, horizontal active 640 + front porch 16 + sync 96 + back porch 48 = total 800, hfreq 31.469kHz, vertical active 480 + front porch 10 + sync 2 + back porch 33 = total 525, vfreq 59.940Hz

720x400@70Hz/31.5kHz ... VGA, pixel clock 28.320 MHz, H-sync NEG, V-sync POS, horizontal active 720 + front porch 18 + sync 108 + back porch 54 = total 900, hfreq 31.467kHz, vertical active 400 + front porch 12 + sync 2 + back porch 35 = total 449, vfreq 70.082Hz

800x600@56Hz/35.2kHz ... VESA DMT ID 08h, pixel clock 36.000 MHz, H-sync POS, V-sync POS, horizontal active 800 + front porch 24 + sync 72 + back porch 128 = total 1024, hfreq 35.156kHz, vertical active 600 + front porch 1 + sync 2 + back porch 22 = total 625, vfreq 56.250Hz

800x600@60Hz/37.9kHz ... VESA DMT ID 09h, pixel clock 40.000 MHz, H-sync POS, V-sync POS, horizontal active 800 + front porch 40 + sync 128 + back porch 88 = total 1056, hfreq 37.879kHz, vertical active 600 + front porch 1 + sync 4 + back porch 23 = total 628, vfreq 60.317Hz

848x480@60Hz/31.0kHz ... VESA DMT ID 0Eh, pixel clock 33.750 MHz, H-sync POS, V-sync POS, horizontal active 848 + front porch 16 + sync 112 + back porch 112 = total 1088, hfreq 31.020kHz, vertical active 480 + front porch 6 + sync 8 + back porch 23 = total 517, vfreq 60.000Hz

1024x768@60Hz/48.3kHz ... VESA DMT ID 10h, pixel clock 65.000 MHz, H-sync NEG, V-sync NEG, horizontal active 1024 + front porch 24 + sync 136 + back porch 160 = total 1344, hfreq 48.363kHz, vertical active 768 + front porch 3 + sync 6 + back porch 29 = total 806, vfreq 60.004Hz

1280x720@60Hz/45kHz ... VESA DMT ID 55h, pixel clock 74.250 MHz, H-sync POS, V-sync POS, horizontal active 1280 + front porch 110 + sync 40 + back porch 220 = total 1650, hfreq 45.000kHz, vertical active 720 + front porch 5 + sync 5 + back porch 20 = total 750, vfreq 60.000Hz

Indexes of predefined signal timings that can be used in the DispHstxVModeTimeList table:

vmodetime_532x400 ... 532x400/70.2Hz/31.5kHz (EGA 4:3), pixel clock 20.9 MHz, system clock 104.571 MHz ... detected as 720x400@70Hz on VGA

vmodetime_532x400_fast ... 532x400/70.2Hz/31.5kHz (EGA 4:3), pixel clock 20.9 MHz, system clock 209.143 MHz ... detected as 720x400@70Hz on VGA

vmodetime_640x350 ... 640x350/70.2Hz/31.5kHz (EGA), pixel clock 25.2 MHz, system clock 126 MHz ... detected as 640x350@70Hz on VGA

vmodetime_640x350_fast ... 640x350/70.2Hz/31.5kHz (EGA), pixel clock 25.2 MHz, system clock 252 MHz ... detected as 640x350@70Hz on VGA

vmodetime_640x400 ... 640x400/70.2Hz/31.5kHz (EGA), pixel clock 25.2 MHz, system clock 126 MHz ... detected as 720x400@70Hz on VGA

vmodetime_640x400_fast ... 640x400/70.2Hz/31.5kHz (EGA), pixel clock 25.2 MHz, system clock 252 MHz ... detected as 720x400@70Hz on VGA

vmodetime_640x480 ... 640x480/60Hz/31.5kHz (VGA 4:3, VESA DMT ID 04h), pixel clock 25.2 MHz, system clock 126 MHz ... detected as 640x480@60Hz on VGA

vmodetime_640x480_fast ... 640x480/60Hz/31.5kHz (VGA 4:3, VESA DMT ID 04h), pixel clock 25.2 MHz, system clock 252 MHz ... detected as 640x480@60Hz on VGA

vmodetime_640x720 ... 640x720/60Hz/45kHz (half-HD), pixel clock 37.2 MHz, system clock 186 MHz ... detected as 1280x720@60Hz on VGA

vmodetime_720x400 ... 720x400/70.1Hz/31.5kHz (VGA near 16:9), pixel clock 28.32 MHz, system clock 141.6 MHz ... detected as 720x400@70Hz on VGA

vmodetime_720x400_fast ... 720x400/70.1Hz/31.5kHz (VGA near 16:9), pixel clock 28.32 MHz, system clock 283.2 MHz ... detected as 720x400@70Hz on VGA

vmodetime_720x480 ... 720x480/60Hz/31.5kHz (VGA 3:2, SD video NTSC), pixel clock 28.32 MHz, system clock 141.6 MHz ... detected as 640x480@60Hz on VGA

vmodetime_720x480_fast ... 720x480/60Hz/31.5kHz (VGA 3:2, SD video NTSC), pixel clock 28.32 MHz, system clock 283.2 MHz ... detected as 640x480@60Hz on VGA

vmodetime_800x480 ... 800x480/60Hz/31.5kHz (MAME 5:3), pixel clock 31.5 MHz, system clock 157.333 MHz ... detected as 640x480@60Hz on VGA

vmodetime_800x600 ... 800x600/60.3Hz/37.9kHz (SVGA 4:3, VESA DMT ID 09h), pixel clock 40 MHz, system clock 200 MHz ... detected as 800x600@60Hz on VGA

vmodetime_848x480 ... 848x480/60Hz/31.4kHz (DMT SVGA 16:9), pixel clock 33.33 MHz, system clock 166.666 MHz ... detected as 640x480@60Hz on VGA

vmodetime_960x720 ... 960x720/60Hz/45kHz (VESA 4:3), pixel clock 55.7 MHz, system clock 278.4 MHz ... detected as 1280x720@60Hz on VGA

vmodetime_1024x768 ... 1024x768/59.9Hz/48.3kHz (XGA 4:3, VESA DMT ID 10h), pixel clock 64.8 MHz, system clock 324 MHz ... detected as 1024x768@60Hz on VGA

vmodetime_1056x600 ... 1056x600/60Hz/37.9kHz (near 16:9), pixel clock 52.8 MHz, system clock 264 MHz ... detected as 800x600@60Hz on VGA

vmodetime_1152x600 ... 1152x600/60Hz/37.9kHz, pixel clock 57.6 MHz, system clock 288 MHz ... detected as 800x600@60Hz on VGA

vmodetime_1280x400 ... 1280x400/70Hz (16:5), pixel clock 50.4 MHz, system clock 252 MHz... detected as 720x400@70Hz on VGA

vmodetime_1280x480 ... 1280x480/60Hz (8:3), pixel clock 50.4 MHz, system clock 252 MHz ... detected as 640x480@60Hz on VGA

vmodetime_1280x600 ... 1280x600/56Hz/35.2kHz, pixel clock 57.6 MHz, system clock 288 MHz ... detected as 800x600@56Hz on VGA

vmodetime_1280x720 ... 1280x720/60.1Hz/45kHz (16:9, VESA DMT ID 55h), pixel clock 74.4 MHz, system clock 372 MHz ... detected as 1280x720@60Hz on VGA

vmodetime_1280x768 ... 1280x768/59.6Hz/47.6kHz (VESA DMT ID 17h), pixel clock 79.2 MHz, system clock 396 MHz ... detected as 1280x768@60Hz on VGA

vmodetime_1280x800 ... 1280x800/59.6Hz/49.5kHz (VESA DMT ID 1Ch), pixel clock 83.2 MHz, system clock 416 MHz ... detected as 1280x800@60Hz on VGA ... This video mode requires DISPHSTX_DISP_SEL switch to be enabled.

vmodetime_1360x768 ... 1360x768/60.1Hz/47.8kHz (VESA DMT ID 27h), pixel clock 85.6 MHz, system clock 428 MHz ... detected as 1024x768@60Hz on VGA ... This video mode requires DISPHSTX_DISP_SEL switch to be enabled.

vmodetime_1440x600 ... 1440x600/56Hz/35kHz, pixel clock 64 MHz, system clock 320 MHz ... detected as 800x600@56Hz on VGA

10. Driver functions

A brief summary of how the driver works. Before activating the driver, it is necessary to prepare the sDispHstxVModeState state descriptor, which defines the timing of the video signal and the content of the display. This state descriptor is used throughout the entire operation of the driver. In addition to the definition of the signal timing sDispHstxVModeTime, the state descriptor also contains list of strips sDispHstxVStrip of the image. The generated image can be divided into horizontal bars with different formats. The strips are the definitions of the horizontal bars of the image.

Each strip can be divided into slots withDispHstxVSlot. Slot is a separate image segment with its own image format. Slots within a strip can use different formats.

Port settings of the HSTX controller may need to be redefined between slots. For this reason, colored vertical gaps, appearing as solid-colored bars, are inserted between slots to provide the time needed to redefine ports. Gaps are not always necessary, they are only necessary if the image format of adjacent slots differs. More precisely, they are needed if the slots belong to a different DISPHSTX_GRP_* format group or if the hdbl pixel duplication settings are different.

You can use a custom structure to define the state descriptor, or you can use the default structure of the DispHstxVMode driver. The DispHstxVModeInitTime() function initializes the structure and stores the signal timing definition information sDispHstxVModeTime into the structure. You can use the prepared definitions in the DispHstxVModeTimeList table for signal timing, or you can use your own timing definition.

After initializing the status descriptor, add strips using the DispHstxVModeAddStrip() function. The function creates an empty strip of the specified height. You can add multiple strips in this way, up to the number specified by the DISPHSTX_STRIP_MAX configuration switch. If the entire height of the panel is not filled, the remaining lines are filled with black.

After each strip is inserted, add slots to the strip using the DispHstxVModeAddSlot() function. In the function parameters, you primarily specify the graphic format of the slot image and a pointer to the frame buffer that will contain the image data. The buffer for the image data should always be located in RAM. Placing it in Flash can cause image dropouts. You can create the image buffer yourself (possibly as a static array), or you can let the function allocate the buffer itself.

After defining all strips and slots, you activate the driver using the DispHstxSelDispMode() function. When activated, you select DVI (HDMI), VGA or auto-detect mode according to the mode switch or the connected connector.

You can connect slots in graphics format 1 to 16 to the DrawCan drawing library using the DispHstxLinkCan() function. The drawing library provides the necessary image cropping and updates the modified areas so that the modified image can be transferred from the back buffer to the frame buffer or to the LCD display.

The display settings can be changed without stopping the driver, thus changing the display configuration without breaking the image synchronization. The condition is that the signal timing definition must be maintained. The DispHstxExchange() function is used to change the status descriptor. However, it must be remembered that the old descriptor can still be used by the driver for one subsequent video line. The same applies to the image buffers - you can release the old buffers only after a period of several tens of microseconds, or when the

current video line is changed. If you need to discard old frame buffers before creating new ones, use an empty state handler for the temporary period, not requiring buffers. A black stripe may appear in the image, but the monitor sync will not break.

If you want to enable VGA/DVI mode change during driver operation, you can do it by calling the DispHstxAutoDispMode() function repeatedly in the main program loop. The function will detect the state of the mode selection switch and if the setting changes, it will terminate the driver and restart it for another display mode.

To terminate the driver, use the DispHstxAllTerm() function. If the driver was creating some buffers by itself, the buffers have to be cancelled manually (free() function). The addresses of the buffers can be obtained from the status descriptor.

In addition to the above procedure, the driver can also be used in a simplified way. A number of DispVMode*() functions are provided that define the state descriptor for the selected display resolution and the selected graphics mode, start the driver and connect the slot to the graphics library. In this way, you can simplistically activate the display with a single function and then immediately use the functions to draw on the display.

The driver is operated in the second core of the processor. The main program runs in the first core of the processor. The second core may not be fully loaded by the driver. You can use the remaining power of the second core to perform your own functions remotely. Remote function calls can only be used if the driver is activated and image generation is in progress. Use DispHstxCore1Exec() function to run remote function. Your function must not use interrupts (or only very briefly), and you cannot stop the driver or switch between VGA and DVI mode while your function is executing. Whether your function is running can be detected by the DispHstxCore1Busy() function.

Useful descriptors:

sDispHstxVModeTime ... videomode timings descriptor
sDispHstxVColor ... image format descriptor
sDispHstxVModeState ... videomode state descriptor
sDispHstxVStrip ... videomode strip descriptor
sDispHstxVSlot ... videomode slot descriptor

Useful variables:

DispHstxDispMode ... current selected display mode
DispHstxVModeTimeList ... list of predefined videomode timings
DispHstxVColorList ... list of image formats
DispHstxVColorCustom ... custom user format descriptor
DispHstxVMode ... default current videomode state descriptor
pDispHstxVMode ... pointer to current videomode state descriptor
DispHstxFrame ... current frame
DispHstxRenderLine ... line render buffer
DispHstxPal1b ... default palettes of 1-bit format
DispHstxPal2b ... default palettes of 2-bit format
DispHstxPal2bcol ... default color 2-bit palettes
DispHstxPal3b ... default palettes of 3-bit format
DispHstxPal4b ... default palettes of 4-bit format
DispHstxPal4bcol ... default color 4-bit palettes YRGB1111
DispHstxPal6b ... default palettes of 6-bit format
DispHstxPal8b ... default palettes of 8-bit format
DispHstxDefVgaPal ... default VGA palette buffer

DispHstxVColorCustom ... pointer to custom user format descriptor

Function error codes:

DISPHSTX_ERR_OK	all OK (= 0)
DISPHSTX_ERR_BROKEN	internal structure is broken (not initialized?)
DISPHSTX_ERR_STRIPNUM	exceeded number of strips
DISPHSTX_ERR_NOSTRIP	trying to add slot to empty videomode descriptor
DISPHSTX_ERR_SLOTNUM	exceeded number of slots
DISPHSTX_ERR_FONTH	incorrect font height in text mode
DISPHSTX_ERR_FONT	incorrect font or tile pointer (font = NULL)
DISPHSTX_ERR_BUF	memory error on frame buffer allocation
DISPHSTX_ERR_PALVGA	memory error on VGA palettes buffer allocation
DISPHSTX_ERR_ALIGN	in direct mode (no palettes) width must be at least multiply of u8 (incorrect align of modes 1-6 bits per pixel)
DISPHSTX_ERR_SUPPORT	1-6 bit modes are not supported in direct mode with double pixels (error if hdbl > 1; palnum = 0)
DISPHSTX_ERR_COLMOD	fonth in DISPHSTX_FORMAT_COL mode must be > 0 (color index modulo)
DISPHSTX_ERR_HSTXBUF	buffer must be valid in HSTX RLE format, with width equal to width of the slot
DISPHSTX_ERR_HSTXFONT	font must point to array of line offsets in HSTX RLE
DISPHSTX_ERR_PATDIM	pattern format requires dimension in pitch and fonth
DISPHSTX_ERR_ATTRBUF	memory error creating attribute buffer
DISPHSTX_ERR_PITCHALIGN	pitch must be aligned to 32-bit word

Functions:

Bool DispHstxIsVSync();

Test if an image is generated in the vertical synchronisation area.

void DispHstxWaitVSync();

Wait for start of VSync scanline.

void DispHstxVModeInitTime(sDispHstxVModeState* vmode, const sDispHstxVModeTime* vtime);

vmode ... videomode state descriptor (must not currently be used)

vtime ... videomode timings descriptor

Initialize videomode state descriptor - set timings and clear list of strips.

int DispHstxVModeAddStrip(sDispHstxVModeState* vmode, int height);

vmode ... videomode state descriptor (must not currently be used)

height ... height of the strip in number of video scanlines, can be 0, or -1=use all remaining scanlines

- final sum of heights of all strips must be equal or greater than vtime.vactive

Add empty videomode strip to videomode state descriptor. Returns error code DISPHSTX_ERR_*, 0=all OK.

int DispHstxVModeAddSlot(sDispHstxVModeState* vmode, int hdbl, int vdbl, int w, int format, void* buf, int pitch, const u16* pal, u32* palvga, const u8* font, int fonth, u16 gap_col, int gap_len);

vmode ... videomode state descriptor (must not currently be used)

hdbl ... number of video pixels per graphics pixel 1..16 (fast mode) or
1..32 (normal mode) (1=full resolution, 2=double pixels, ...)
- if hdbl > 1, pixel formats 1-6 bits are not supported
- width must be divisible by hdbl

vdbl ... number of scanlines per graphics line 1.. (1=single line, 2=double lines, ...)
- height of the slot does not have to be a multiple of vdbl

w ... width of the slot in graphics pixels (not video pixels), without color separator,
or -1=use all remaining width of the strip
- can be 0, to use only color separator (which can be combined with
any color format)
- width in video pixels = w * hdbl
- pitch must be aligned to u32, width need not to be aligned
- sum of widths + separators of all slots must be equal vtime.hactive
- in direct mode (no palettes) width must be at least multiply of u8,
or better multiply of u32 (uses faster DMA)
- in HSTX RLE compression mode the slot width must be the same
as the image width

format ... index of pixel format DISPHSTX_FORMAT_*

buf ... pointer to frame buffer (aligned to 32-bit), or NULL to create new
one with malloc() (not used in single-color modes)
- the driver does not automatically delete the buffer it created,
it must be manually deleted when the driver terminates

pitch ... length of line or row in bytes, or -1 = use default

pal ... pointer to palettes in RGB565 format (should be in RAM for faster access;
can be NULL to use default palettes)

palvga ... pointer to palettes for VGA, double-size format with time dithering
(or NULL to create new one with malloc(), or NULL if VGA is disabled)
- it must be large enough to accept palette entries in 2*u32 format
(size in bytes = palnum*8)

font ... pointer to font (width 8 pixels), or to column or row of tiles, or offsets
of lines of HSTX, or attributes (should be in RAM for faster access;
can be NULL if not text/tile format)

fonth ... height of font, or color modulo, or pitch of tile row (or 0=column),
or pitch of attributes (-1=auto)

gap_col ... color separator in RGB565 format, after this slot (not precise color, near DC balanced TMDS colors will be used)

gap_len ... number of video pixels of color separator between slots (recommended 20, 0 = do not use, or -1 = auto detection insert when needed)
- used to enable switching slot to another format (when needed to change registers)
- can be used as simple color on end of the strip, when w = 0
- if too small (less than 10 or 17), signal can drop-out

Add videomode slot to last added videomode strip. Returns error code DISPHSTX_ERR_*. DISPHSTX_ERR_OK = 0 = all OK. For a more detailed description of some entries, see the description of image formats in chapter „Graphics formats“.

The driver requires palette conversion to the internal format used in VGA mode. This applies to modes using palettes (palnum > 0), i.e. the *_PAL, *_COL, *_MTEXT and *_ATEXT formats. Therefore, the function must be passed a pointer to a buffer into which the function will generate VGA palettes in internal format. The buffer must be large enough to hold twice the number of palettes in u32 format - in bytes, this is equivalent to palnum*8. This buffer must be manually deallocated with the free() function after the driver function is finished. The exception is the case of implicit palettes (pal = NULL), in which case the function uses a predefined table in RAM instead of allocating the palvga buffer.

If there are slots with different formats next to each other in the stripe, the driver must redefine the HSTX controller registers to the new format. This takes some time. Therefore, a gap must be included between such two slots during which the driver can redefine the registers. The TMDS color sent to the output directly, bypassing the TMDS decoder, is used as the gap so that the TMDS decoder registers can be redefined during that time. The transmitted color must satisfy the DC balance condition. The gap_len entry defines the separation gap to be added after the slot, in video pixels. Typically, a value of 20 is acceptable. If too short a gap is used, the encoder operation may be disturbed and synchronization may drop out. By specifying a value of -1, the driver determines the gap itself - not all slot combinations require the use of a separator gap. The determining factor is whether the slots differ by the DISPHSTX_GRP_* format group or whether they differ by the hdbl pixel duplication entry. The gap_col entry defines the color of the gap separator. The driver does not use the exact color, but looks for a nearest TMDS code that satisfies the DC symmetry condition. Therefore, the resulting color may not exactly match the specified color.

The separation gap is also used at the end of the strip if the line length is less than the video line length. Another use is to add a color space at the beginning of the line, setting the slot width to 0 and specifying only the use of the separator gap.

int DispHstxVModeInitSimple(sDispHstxVModeState* vmode, const sDispHstxVModeTime* vtime, int hdbl, int vdbl, int format, void* buf, const u16* pal, const u8* font, int fonth);

vmode ... videomode state descriptor (must not currently be used)

vtime ... videomode timings descriptor

hdbl ... number of video pixels per graphics pixel 1..16 (fast mode) or 1..32 (normal mode) (1=full resolution, 2=double pixels, ...)
- if hdbl > 1, pixel formats 1-6 bits are not supported

- width must be divisible by hdbl

vdbl ... number of scanlines per graphics line 1.. (1=single line, 2=double lines, ...)

- height of the slot does not have to be a multiple of vdbl

format ... index of pixel format DISPHSTX_FORMAT_*

buf ... pointer to frame buffer (aligned to 32-bit), or NULL to create new one
(not used in single-color modes)

- in HSTX RLE compression mode the videomode width must be the same as the image width

- the driver does not automatically delete the buffer it created, it must be manually deleted when the driver terminates

pal ... pointer to palettes in RGB565 format (should be in RAM for faster access; can be NULL to use default palettes)

font ... pointer to font (width 8 pixels), or to column or row of tiles, or offsets of lines of HSTX, or attributes (should be in RAM for faster access; can be NULL if not text/tile format)

fonth ... height of font, or color modulo, or pitch of tile row (or 0=column), or pitch of attributes (-1=auto)

Initialize videomode state desciptor to simple videomode (uses 1 strip and 1 slot). Links video slot to DrawCan. Returns error code DISPHSTX_ERR_*, 0=all OK. This function is a simplified version of the previous functions. It is used for easy initialization of the state descriptor in case the generated image contains only one image segment. The function initializes the state descriptor, sets the signal timing, creates one strip, creates one slot in the strip, and connects the slot to the DrawCan drawing library. The function is also used in the following DispHstxVModeStartSimple() function.

As the VGA palette buffer the DispHstxDefVgaPal buffer is internally used, which is large enough for 256 palettes. It can possibly be used for other purposes if this function is not used.

int DispHstxVModeStartSimple(int dispmode, void* buf, int vmodeinx, int hdbl, int vdbl, int format);

dispmode ... display mode DISPHSTX_DISPMODE_DVI or DISPHSTX_DISPMODE_VGA. Use DISPHSTX_DISPMODE_NONE or 0 to auto-detect display mode selection.

buf ... pointer to frame buffer (aligned to 32-bit), or NULL to create new one

- the driver does not automatically delete the buffer it created, it must be manually deleted when the driver terminates

vmodeinx ... videomode timings index vmodetime_*

hdbl ... number of video pixels per graphics pixel 1..16 (fast mode) or 1..32 (normal mode) (1=full resolution, 2=double pixels, ...)

- if hdbl > 1, pixel formats 1-6 bits are not supported

- width must be divisible by hdbl

vdbl ... number of scanlines per graphics line 1.. (1=single line, 2=double lines, ...)

- height of the slot does not have to be a multiple of vdbl

format ... index of pixel format DISPHSTX_FORMAT_*

Start simple display graphics videomode. Returns error code DISPHSTX_ERR_*. DISPHSTX_ERR_OK = 0 = all OK. This function is a further simplification of the previous functions. It is used to easily activate graphical full-screen video modes. It terminates the previous driver activity, initializes the standard DispHstxVMode state descriptor, creates one strip and one slot in it, and starts the driver activity with the new descriptor. If VGA/DVI display mode is not specified, it detects the mode automatically. For list of predefined timings, see chapter „Timing modes“. For list of graphics formats, see chapter „Graphics formats“.

void DispHstxLinkCan(const sDispHstxVSlot* slot, sDrawCan* can);

slot ... display slot

can ... drawing canvas

Function links display slot to drawing canvas. Slot must be in graphics format DISPHSTX_FORMAT_1 to DISPHSTX_FORMAT_16, or DISPHSTX_FORMAT_1_PAL to DISPHSTX_FORMAT_8_PAL. This enables drawing to slot buffer by the graphics functions.

void DispHstxAllTerm();

Terminate all current devices, stop driver all activity.

void DispHstxSelDispMode(int dispmode, sDispHstxVModeState* vmode);

dispmode ... display mode DISPHSTX_DISPMODE_DVI

or DISPHSTX_DISPMODE_VGA. Use DISPHSTX_DISPMODE_NONE

or 0 to auto-detect display mode selection.

vmode ... initialized videomode state descriptor

Select and start display mode and videomode. If VGA/DVI display mode is not specified, it detects the mode automatically. This function will also reclock the system clock if needed. State descriptor is set to the pDispHstxVMode variable, which points to the currently running video mode.

void DispHstxExchange(sDispHstxVModeState* vmode);

vmode ... new initialized videomode state descriptor

This function smoothly replaces current videomode state descriptor with the new one, containing new screen layout definiton, without breaking the image synchronization. The condition is that the signal timing definition must be maintained. It must be remembered that the old descriptor can still be used by the driver for one subsequent video line. The same applies to the image buffers - you can release the old buffers only after a period of several tens of microseconds, or when the current video line is changed. If you need to discard old frame buffers before creating new ones, use an empty state handler for the temporary period, not requiring buffers. A black stripe may appear in the image, but the monitor sync will not break.

void DispHstxReSelDispMode(int dispmode);

dispmode ... display mode DISPHSTX_DISPMODE_DVI
or DISPHSTX_DISPMODE_VGA

Re-select display mode with old videomode. The function terminates the driver and restarts it in the new VGA or DVI display mode.

void DispHstxAutoDispMode();

Auto re-select current display mode by selection switch. Function reads selection switch, and if changes, re-select current VGA/DVI display mode. Function should be called periodically from the main program loop, if needed to update display mode continuously.

int DispHstxCheckDispSel();

Get current display selection switch. Display selection switch is auto-initialized. Returns current switch position DISPHSTX_DISPMODE_DVI, DISPHSTX_DISPMODE_VGA or DISPHSTX_DISPMODE_NONE if switch is in auto-detect position (or if selection switch is not used). This function can be called periodically while the driver is active to allow the VGA/DVI display to be changed by the user changing the switch.

int DispHstxAutoDispSel();

Auto-detect display selection on program start, by checking VGA monitor connection. Returns DISPHSTX_DISPMODE_DVI or DISPHSTX_DISPMODE_VGA. Function can only be used at program start or before initializing the video mode because it resets the GPIO pins. If the display selection switch is enabled, it is used in preference. The RGB pins of the VGA display are supposed to be connected to GPIO 12 to 17.

void DispHstxClockReinit(int khz);

Reconfigure system clock, to use PLLSYS only for HSTX generator. Requires DISPHSTX_USEPLL configuration switch to be set.

u8* DispHstxBuf();

Get buffer of first display slot.

void DispHstxFreeBuf();

Free buffer of first display slot.

void DispHstxCore1Exec(void (*fnc)());

Execute core 1 remote function.

11. Graphics formats

The disphstx_vmode_format.* files contain color format definitions.

Color formats are independent of the selected video mode timing. Each slot of the image can contain a different image format. If the output mode of the generated pixels differs, it is necessary to insert a separator gap between adjacent slots to allow the HSTX controller registers to be redefined. This is especially necessary when the slot format group differs, or when the pixel duplication entry hdbl differs.

All formats require buffer alignment to 32 bits (4 bytes). Formats DISPHSTX_FORMAT_1 to DISPHSTX_FORMAT_6 do not support duplicated pixels - hdbl entry must be always 1.

The PicopadImg2 program can be used to generate and create image data in the given format - see chapter „PicoPadImg2 - Image conversion“.

Pixel format groups:

DISPHSTX_GRP_1 ... 1 bit per pixel, black & white 0..1 (8 pixels per 1 byte)
DISPHSTX_GRP_2 ... 2 bits per pixel, grayscale 0..3 (4 pixels per 1 byte)
DISPHSTX_GRP_3 ... 3 bits per pixel, format RGB111 (10 pixels per 32-bit word)
DISPHSTX_GRP_4 ... 4 bits per pixel, grayscale 0..15 (2 pixels per 1 byte)
DISPHSTX_GRP_6 ... 6 bits per pixel, format RGB222 (5 pixels per 32-bit word)
DISPHSTX_GRP_8 ... 8 bits per pixel, format RGB332 (1 pixel per 1 byte)
DISPHSTX_GRP_12 ... 12 bits per pixel, format RGB444 (8 pixels per three 32-bit words)
DISPHSTX_GRP_15 ... 15 bits per pixel, format RGB555 in bits 0..14 (1 pixel per 2 bytes)
DISPHSTX_GRP_16 ... 16 bits per pixel, format RGB565 (1 pixel per 2 bytes)

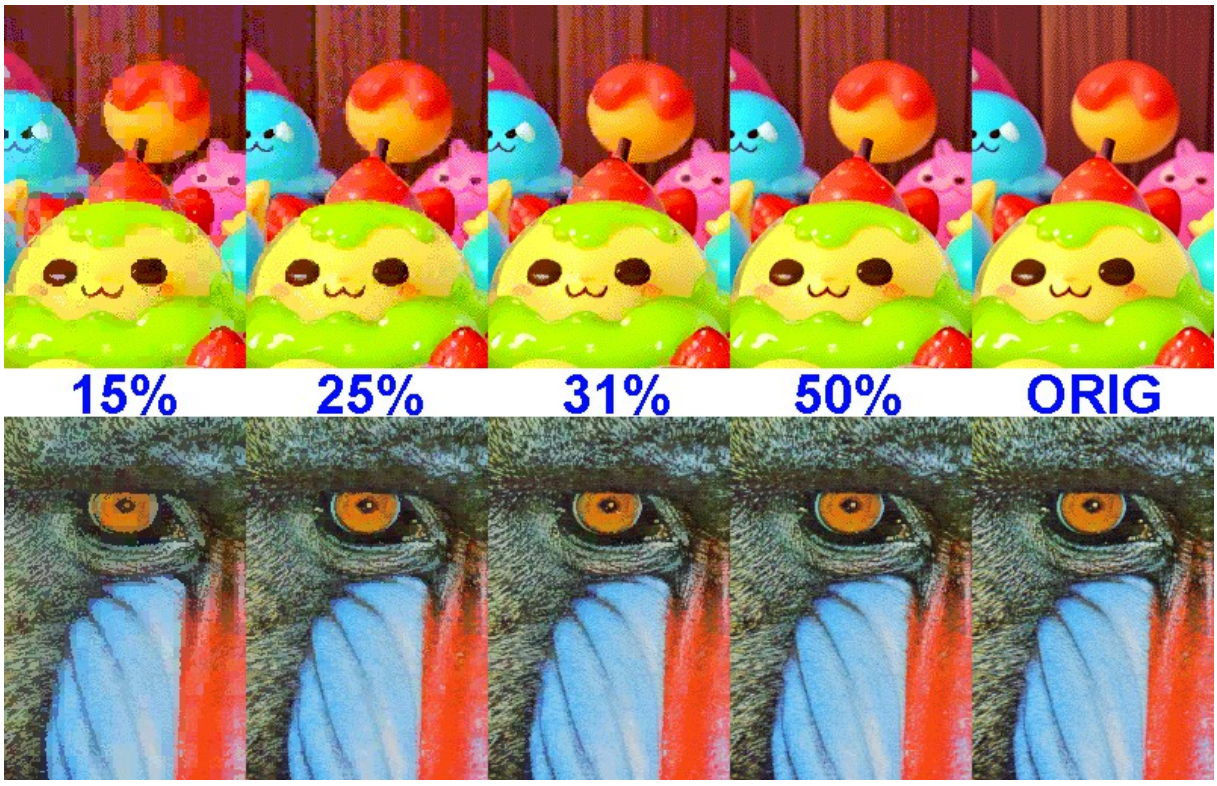
Formats with *_PAL palettes belong to the DISPHSTX_GRP_16 group independently of the image format.

Brief overview of image formats:

DISPHSTX_FORMAT_NONE	invisible slot with gap color
DISPHSTX_FORMAT_1	graphics 1 bit per pixel, black & white
DISPHSTX_FORMAT_2	graphics 2 bits per pixel, gray 4 levels
DISPHSTX_FORMAT_3	graphics 3 bits per pixel, color RGB111
DISPHSTX_FORMAT_4	graphics 4 bits per pixel, gray 16 levels
DISPHSTX_FORMAT_6	graphics 6 bits per pixel, color RGB222
DISPHSTX_FORMAT_8	graphics 8 bits per pixel, color RGB332
DISPHSTX_FORMAT_12	graphics 12 bits per pixel, color RGB444
DISPHSTX_FORMAT_15	graphics 15 bits per pixel, color RGB555
DISPHSTX_FORMAT_16	graphics 16 bits per pixel, color RGB565
DISPHSTX_FORMAT_1_PAL	graphics 1 bit per pixel, 2 paletted colors
DISPHSTX_FORMAT_2_PAL	graphics 2 bits per pixel, 4 paletted colors
DISPHSTX_FORMAT_3_PAL	graphics 3 bits per pixel, 8 paletted colors
DISPHSTX_FORMAT_4_PAL	graphics 4 bits per pixel, 16 paletted colors
DISPHSTX_FORMAT_6_PAL	graphics 6 bits per pixel, 64 paletted colors
DISPHSTX_FORMAT_8_PAL	graphics 8 bits per pixel, 256 paletted colors
DISPHSTX_FORMAT_COL	simple color, color of lines from palettes
DISPHSTX_FORMAT_MTEXT	monocolor text, color of every row from palettes
DISPHSTX_FORMAT_ATEXT	attribute text, character + attribute 2*16 colors
DISPHSTX_FORMAT_TILE4_8	tiles 4x4 pixels, 1-byte index, color RGB332

DISPHSTX_FORMAT_TILE8_8	tiles 8x8 pixels, 1-byte index, color RGB332
DISPHSTX_FORMAT_TILE16_8	tiles 16x16 pixels, 1-byte index, color RGB332
DISPHSTX_FORMAT_TILE32_8	tiles 32x32 pixels, 1-byte index, color RGB332
DISPHSTX_FORMAT_TILE4_8_PAL	tiles 4x4 pixels, 1-byte index, paletted colors
DISPHSTX_FORMAT_TILE8_8_PAL	tiles 8x8 pixels, 1-byte index, paletted colors
DISPHSTX_FORMAT_TILE16_8_PAL	tiles 16x16 pixels, 1-byte index, paletted colors
DISPHSTX_FORMAT_TILE32_8_PAL	tiles 32x32 pixels, 1-byte index, paletted colors
DISPHSTX_FORMAT_HSTX_15	HSTX RLE compression, 15 bits per pixel
DISPHSTX_FORMAT_HSTX_16	HSTX RLE compression, 16 bits per pixel
DISPHSTX_FORMAT_PAT_8	repeated pattern, color RGB332
DISPHSTX_FORMAT_PAT_8_PAL	repeated pattern, paletted colors
DISPHSTX_FORMAT_RLE8	8-bit RLE compression, color RGB332
DISPHSTX_FORMAT_RLE8_PAL	8.bit RLE compression, paletted colors
DISPHSTX_FORMAT_ATTR1_PAL	attribute compression 1, 2x 4-bit color, cell 8x8
DISPHSTX_FORMAT_ATTR2_PAL	attribute compression 2, 2x 4-bit color, cell 4x4
DISPHSTX_FORMAT_ATTR3_PAL	attribute compression 3, 4x 4-bit color, cell 8x8
DISPHSTX_FORMAT_ATTR4_PAL	attribute compression 4, 4x 4-bit color, cell 4x4
DISPHSTX_FORMAT_ATTR5_PAL	attribute compression 5, 2x 8-bit color, cell 8x8
DISPHSTX_FORMAT_ATTR6_PAL	attribute compression 6, 2x 8-bit color, cell 4x4
DISPHSTX_FORMAT_ATTR7_PAL	attribute compression 7, 4x 8-bit color, cell 8x8
DISPHSTX_FORMAT_ATTR8_PAL	attribute compression 8, 4x 8-bit color, cell 4x4
DISPHSTX_FORMAT_CUSTOM	custom user format

Comparison of attribute compression levels (formats ATTR5.. ATTR8):



DISPHSTX_FORMAT_NONE

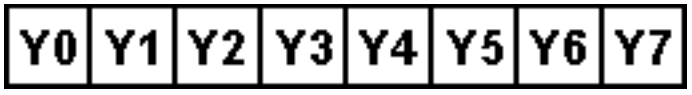
Invalid pixel format. Auxiliary format in cases of slot width 0, where only the slot gap is used.

Setup and limitations of the function DispHstxVModeAddSlot():

- **w** width of the slot set to 0
- **gap_col** set color of the gap in RGB565 format, nearest TMDS DC-balanced color will be used
- **gap_len** number of video pixels of color separator

DISPHSTX_FORMAT_1

1 bit per pixel, 8 pixels per 1 byte. Colors are black & white, with values 0..1. Width must be aligned to 8 pixels. This format does not support duplicated pixels - hdbl entry must be always 1. Format is fully hardware supported in DVI (HDMI) mode, without the need to render image to the graphics buffer. That allows very low CPU load. The disadvantage is the inability to redefine colors, the colors are fixed to black and white, while white has a lower brightness of 50%. In VGA mode the internal palette table is used, the white color has full brightness and the CPU load is higher (rendering is software into rendering buffer). Another disadvantage is that the slot width must be multiple of 8 pixels. Pixel format group is DISPHSTX_GRP_1.



u8

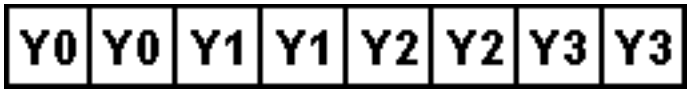
In this format, pixels in a byte are organized in order from the highest bits to the lowest bits. Bit 7 of the byte contains first pixel, bit 0 contains last pixel.

Setup and limitations of the function DispHstxVModeAddSlot():

- **hdbl** pixel repeating must be 1.
- **w** width of the slot must be multiple of 8
- **pitch** length of line in bytes must be aligned to 32-bits; pitch = ((w + 7)/8 + 3) & ~3

DISPHSTX_FORMAT_2

2 bits per pixel, 4 pixels per 1 byte. Colors are grayscale, with values 0..3. Width must be aligned to 4 pixels. This format does not support duplicated pixels - hdbl entry must be always 1. Format is fully hardware supported in DVI (HDMI) mode, without the need to render image to the graphics buffer. That allows very low CPU load. The disadvantage is the inability to redefine colors, the colors are fixed to black, dark gray, light gray and white, while white has a lower brightness of 75%. In VGA mode the internal palette table is used, the white color has full brightness and the CPU load is higher (rendering is software into rendering buffer). Another disadvantage is that the slot width must be multiple of 4 pixels. Pixel format group is DISPHSTX_GRP_2.



u8

In this format, pixels in a byte are organized in order from the highest bits to the lowest bits. Bits 7 and 6 of the byte contain first pixel, bit 1 and 0 contain last pixel.

Setup and limitations of the function DispHstxVModeAddSlot():

- **hdbl** pixel repeating must be 1.
- **w** width of the slot must be multiple of 4

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = ((w + 3)/4 + 3) & ~3

DISPHSTX_FORMAT_3

3 bits per pixel, 10 pixels per 32-bit word. Colors are in format RGB111. Width must be aligned to 10 pixels. This format does not support duplicated pixels - hdbl entry must be always 1. Format is fully hardware supported in DVI (HDMI) mode, without the need to render image to the graphics buffer. That allows very low CPU load. The disadvantage is the inability to redefine colors, the colors are fixed to RGB111 and has lower brightness of 50%. In VGA mode the internal palette table is used, colors have full brightness and the CPU load is higher (rendering is software into rendering buffer). Another disadvantage is that the slot width must be multiple of 10 pixels. Pixel format group is DISPHSTX_GRP_3.

32-bit access:

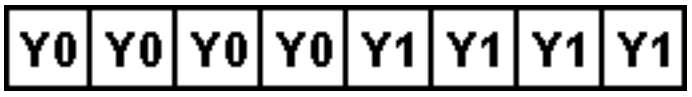
In this format, pixels in a 32-bit word are organized in order from the lowest bits to the highest bits. Bits 0, 1 and 2 of the word contain first pixel, bits 27, 28 and 29 contain last pixel. Bits 30 and 31 are not used and their contents are undefined. They should not be used for program purposes either, because the drawing functions may distort these bits.

Setup and limitations of the function DispHstxVModeAddSlot():

- **hdbl** pixel repeating must be 1.
- **w** width of the slot must be multiple of 10
- **pitch** length of line in bytes must be aligned to 32-bits; pitch = (w + 9)/10*4

DISPHSTX_FORMAT_4

4 bits per pixel, 2 pixels per 1 byte. Colors are grayscale, with values 0 to 15. Width must be aligned to 2 pixels. This format does not support duplicated pixels - hdbl entry must be always 1. Format is fully hardware supported in DVI (HDMI) mode, without the need to render image to the graphics buffer. That allows very low CPU load. The disadvantage is the inability to redefine colors, the colors are fixed to 16 gray levels. In VGA mode the internal palette table is used and the CPU load is higher (rendering is software into rendering buffer). Another disadvantage is that the slot width must be multiple of 2 pixels. Pixel format group is DISPHSTX_GRP_4.



u8

In this format, pixels in a byte are organized in order from the highest bits to the lowest bits. Bits 7 to 4 of the byte contain first pixel, bits 3 to 0 contain second pixel.

Setup and limitations of the function DispHstxVModeAddSlot():

- **hdbl** pixel repeating must be 1.
- **w** width of the slot must be multiple of 2
- **pitch** length of line in bytes must be aligned to 32-bits; pitch = ((w + 1)/2 + 3) & ~3

DISPHSTX_FORMAT_6

6 bits per pixel, 5 pixels per 32-bit word. Colors are in format RGB222. Width must be aligned to 5 pixels. This format does not support duplicated pixels - hdbl entry must be always 1. Format is fully hardware supported in DVI (HDMI) mode, without the need to render image to the graphics buffer. That allows very low CPU load. The disadvantage is the inability to redefine colors, the colors are fixed to RGB222 and has lower brightness of 75%. In VGA mode the internal palette table is used, colors have full brightness and the CPU load is higher (rendering is software into rendering buffer). Another disadvantage is that the slot width must be multiple of 5 pixels. Pixel format group is DISPHSTX_GRP_6.

32-bit access:

In this format, pixels in a 32-bit word are organized in order from the lowest bits to the highest bits. Bits 0 to 5 of the word contain first pixel, bits 24 to 29 contain last pixel. Bits 30 and 31 are not used and their contents are undefined. They should not be used for program purposes either, because the drawing functions may distort these bits.

Setup and limitations of the function DispHstxVModeAddSlot():

- **hdbl** pixel repeating must be 1.

- **w** width of the slot must be multiple of 5

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = (w + 4)/5*4

DISPHSTX_FORMAT_8

8 bits per pixel, 1 pixels per 1 byte. Colors are in format RGB332. Format is fully hardware supported in DVI (HDMI) mode, without the need to render image to the graphics buffer. That allows very low CPU load. The disadvantage is the inability to redefine colors, the colors are fixed to RGB332 format. The blue component uses 2 bits, the other components use 3 bits. The blue component in DVI mode has a slightly lower maximum brightness than the other components, which can lead to a yellowish tint to the image, compared to the palette mode. In VGA mode the internal palette table is used and the CPU load is higher (rendering is software into rendering buffer). Pixel format group is DISPHSTX_GRP_8.

u8

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = (w + 3) & ~3

DISPHSTX_FORMAT_12

12 bits per pixel, 8 pixels per three 32-bit words, or 2 pixels per 3 bytes. Colors are in format RGB444. Format is software rendered into graphics buffer. Rendering the image in DVI (HDMI) mode loads the processor to about 50% and can be rendered even at normal processor speed. In VGA mode, rendering is much more difficult (load over 100%) and requires fast CPU mode (use _fast video modes) or pixel repeating rendering (hdbl > 1). Pixel format group is DISPHSTX_GRP_12.

Byte access:

32-bit access:

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of line in bytes must be aligned to 32-bits and 8 pixels; pitch = (w + 7)/8*12

DISPHSTX_FORMAT_15

15 bits per pixel, 1 pixel per one 16-bit half-word. Colors are in format RGB555. Format is fully hardware supported in DVI (HDMI) mode, without the need to render image to the graphics buffer. That allows very low CPU load. In VGA mode, rendering is very time consuming (load over 100%) and requires fast CPU mode (use _fast video modes) or pixel repeating rendering (hdbl > 1). Pixel format group is DISPHSTX_GRP_15.

16-bit access:

This format uses 5 bits of blue component in bits 0 to 4, 5 bits of green component in bits 5 to 9 and 5 bits of red component in bits 10 to 14. Bit 15 is not used and its content is undefined. It should not be used for program purposes either, because the drawing functions may distort this bit.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = (w*2 + 3) & ~3

DISPHSTX_FORMAT_16

16 bits per pixel, 1 pixel per one 16-bit half-word. Colors are in format RGB565. Format is fully hardware supported in DVI (HDMI) mode, without the need to render image to the graphics buffer. That allows very low CPU load. In VGA mode, rendering is very time consuming (load over 100%) and requires fast CPU mode (use _fast video modes) or pixel repeating rendering (hdbl > 1). Pixel format group is DISPHSTX_GRP_16.

16-bit access:

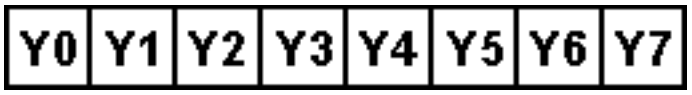
This format uses 5 bits of blue component in bits 0 to 4, 6 bits of green component in bits 5 to 10 and 5 bits of red component in bits 11 to 15.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = (w*2 + 3) & ~3

DISPHSTX_FORMAT_1_PAL

1 bit per pixel, 8 pixels per 1 byte. 2 paletted colors. Format is software rendered to the graphics buffer, using custom palettes. CPU load is about 30 to 70% and it works with normal CPU speed. If palettes are not specified (pal = NULL), the default palettes are used - black and white color. Pixel format group is DISPHSTX_GRP_16.



u8

In this format, pixels in a byte are organized in order from the highest bits to the lowest bits. Bit 7 of the byte contains first pixel, bit 0 contains last pixel.

Setup and limitations of the function DispHstxVModeAddSlot():

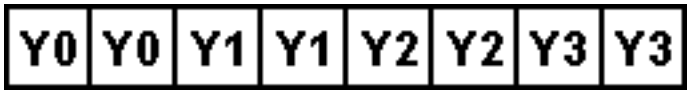
- **pitch** length of line in bytes must be aligned to 32-bits; pitch = ((w + 7)/8 + 3) & ~3

- **pal** pointer to 2 palettes u16 in RGB565 format (table size 2*2=4 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes (black & white color).

- **palvga** pointer to buffer to get 2*2 auto-generated palettes u32 for VGA (buffer size 2*2*4=16 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

DISPHSTX_FORMAT_2_PAL

2 bits per pixel, 4 pixels per 1 byte. 4 paletted colors. Format is software rendered to the graphics buffer, using custom palettes. CPU load is about 50 to 70% and it works with normal CPU speed. If palettes are not specified (pal = NULL), the default palettes are used - black, blue, red and yellow color. Pixel format group is DISPHSTX_GRP_16.



u8

In this format, pixels in a byte are organized in order from the highest bits to the lowest bits. Bits 7 and 6 of the byte contain first pixel, bit 1 and 0 contain last pixel.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = ((w + 3)/4 + 3) & ~3

- **pal** pointer to 4 palettes u16 in RGB565 format (table size 4*2=8 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes (black, blue, red and yellow color).

- **palvga** pointer to buffer to get 2*4 auto-generated palettes u32 for VGA (buffer size 2*4*4=32 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the

descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

DISPHSTX_FORMAT_3_PAL

3 bits per pixel, 10 pixels per 32-bit word. Format is software rendered to the graphics buffer, using custom palettes. CPU load is about 50 to 70% and it works with normal CPU speed. If palettes are not specified (pal = NULL), the default palettes are used - color in format RGB111. Pixel format group is DISPHSTX_GRP_16.

32-bit access:

In this format, pixels in a 32-bit word are organized in order from the lowest bits to the highest bits. Bits 0, 1 and 2 of the word contain first pixel, bits 27, 28 and 29 contain last pixel. Bits 30 and 31 are not used and their contents are undefined. They should not be used for program purposes either, because the drawing functions may distort these bits.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = (w + 9)/10*4

- **pal** pointer to 8 palettes u16 in RGB565 format (table size 8*2=16 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB111 format.

- **palvga** pointer to buffer to get 2*8 auto-generated palettes u32 for VGA (buffer size 2*8*4=64 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

DISPHSTX_FORMAT_4_PAL

4 bits per pixel, 2 pixels per 1 byte. Format is software rendered to the graphics buffer, using custom palettes. CPU load is about 50 to 70% and it works with normal CPU speed. If palettes are not specified (pal = NULL), the default palettes are used - color in format YRGB1111. Pixel format group is DISPHSTX_GRP_16.

u8

In this format, pixels in a byte are organized in order from the highest bits to the lowest bits. Bits 7 to 4 of the byte contain first pixel, bits 3 to 0 contain second pixel.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = ((w + 1)/2 + 3) & ~3

- **pal** pointer to 16 palettes u16 in RGB565 format (table size 16*2=32 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in YRGB1111 format.

- **palvga** pointer to buffer to get 2*16 auto-generated palettes u32 for VGA (buffer size 2*16*4=128 bytes). Set NULL to auto-create new one with malloc() (must be manually

deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

DISPHSTX_FORMAT_6_PAL

6 bits per pixel, 5 pixels per 32-bit word. Format is software rendered to the graphics buffer, using custom palettes. CPU load is about 50 to 70% and it works with normal CPU speed. If palettes are not specified (pal = NULL), the default palettes are used - color in format RGB222. Pixel format group is DISPHSTX_GRP_16.

32-bit access:

In this format, pixels in a 32-bit word are organized in order from the lowest bits to the highest bits. Bits 0 to 5 of the word contain first pixel, bits 24 to 29 contain last pixel. Bits 30 and 31 are not used and their contents are undefined. They should not be used for program purposes either, because the drawing functions may distort these bits.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = (w + 4)/5*4

- **pal** pointer to 64 palettes u16 in RGB565 format (table size 64*2=128 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB222 format.

- **palvga** pointer to buffer to get 2*64 auto-generated palettes u32 for VGA (buffer size 2*64*4=512 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

DISPHSTX_FORMAT_8_PAL

8 bits per pixel, 1 pixels per 1 byte. Format is software rendered to the graphics buffer, using custom palettes. CPU load is about 50 to 70% and it works with normal CPU speed. If palettes are not specified (pal = NULL), the default palettes are used - color in format RGB332. Pixel format group is DISPHSTX_GRP_16.

u8

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of line in bytes must be aligned to 32-bits; pitch = (w + 3) & ~3

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually

deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

DISPHSTX_FORMAT_COL

This format fills the graphic line with a simple color. Each graphic line can have a different color. Colors of the lines are defined in the palette table "pal". The "fonth" parameter defines the number of palettes in the table. The palette with index modulo "fonth" is used for each line. For example, if you want the entire slot to be filled with a single color, specify 1 palette in the palette table and set the "fonth" parameter to 1. If you want the line colors to alternate repeatedly between even and odd line, use 2 palettes in the palette table and set the "fonth" parameter to 2. The format is fully hardware accelerated and loads the CPU at 3%. Pixel format group is DISPHSTX_GRP_16.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pal** pointer to “fonth” palettes u16 in RGB565 format (table size fonth*2 bytes). Palettes should be in RAM for faster access.

- **palvga** pointer to buffer to get 2*fonth auto-generated palettes u32 for VGA (buffer size 2*fonth*4 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **fonth** number of palettes in color table, color modulo. Line color = pal[line % fonth].

DISPHSTX_FORMAT_MTEXT

Monocolor text. Frame buffer contains text characters. Each character occupies 1 byte. Characters on the same row have the same color. Each row can have a different foreground and background color. Colors of the rows are defined in the palette table. Even palette entries define the background color of the characters, odd palette entries define the foreground color of the characters. The number of palette entries in the table must be equal to twice the number of rows (or greater). Pixel format group is DISPHSTX_GRP_16.

The "font" parameter is a pointer to the font table. The font table is a 256-character 1-bit bitmap with a width of 256*8 = 2048 pixels (pitch is 256 bytes per line). A bit with a value of 1 is a character pixel, a bit with a value of 0 is the background. Characters in the bitmap are 8 pixels wide. The height of the characters (i.e. the number of lines) is determined by the "fonth" parameter. The font table should be in RAM. If it were in flash memory, the image synchronization could drop out.

The DispHstxVModeAddSlot() function calculates the number of rows as rows = (h + fonth-1)/fonth, where "h" is the slot height in graphic lines. It calculates the line length in bytes (if not specified) as pitch = ((w + 7)/8 + 3) & ~3, where "w" is the width of the slot in graphics pixels. The required number of palettes in the "pal" table is 2*rows, the required buffer size in bytes is rows*pitch.

The CPU load is about 40 to 80%, the format works even at normal CPU speed.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = ((w + 7)/8 + 3) & ~3

- **pal** pointer to rows*2 palettes u16 in RGB565 format (table size rows*2*2 bytes). Palettes should be in RAM for faster access.

- **palvga** pointer to buffer to get 2*rows*2 auto-generated palettes u32 for VGA (buffer size 2*rows*2*4 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to the font with 256 characters with width 8 pixels per pixel and height “fonth” lines. Font should be in RAM for faster access.

- **fonth** height of the font in graphics lines.

DISPHSTX_FORMAT_ATEXT

Attribute text. Frame buffer contains text characters with color attributes. Each character occupies 2 bytes. First byte is a character code. Second byte is color attribute, where lower 4 bits define foreground color, higher 4 bits define background color. Colors are taken from the palette table with 16 entries. Pixel format group is DISPHSTX_GRP_16.

The "font" parameter is a pointer to the font table. The font table is a 256-character 1-bit bitmap with a width of 256*8 = 2048 pixels (pitch is 256 bytes per line). A bit with a value of 1 is a character pixel, a bit with a value of 0 is the background. Characters in the bitmap are 8 pixels wide. The height of the characters (i.e. the number of lines) is determined by the "fonth" parameter. The font table should be in RAM. If it were in flash memory, the image synchronization could drop out.

The DispHstxVModeAddSlot() function calculates the number of rows as rows = (h + fonth-1)/fonth, where "h" is the slot height in graphic lines. It calculates the line length in bytes (if not specified) as pitch = (((w + 7)/8 * 2) + 3) & ~3, where "w" is the width of the slot in graphics pixels. The required buffer size in bytes is rows*pitch.

The CPU load is about 60 to 95%. To render a VGA image in a RISC-V core, normal CPU speed is not enough, fast mode ("_fast" timing) or pixel duplication with hdbl > 1 must be used.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = (((w + 7)/8 * 2) + 3) & ~3

- **pal** pointer to 16 palettes u16 in RGB565 format (table size 16*2=32 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in YRGB1111 format.

- **palvga** pointer to buffer to get 2*16 auto-generated palettes u32 for VGA (buffer size 2*16*4=128 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to the font with 256 characters with width 8 pixels per pixel and height “fonth” lines. Font should be in RAM for faster access.

- **fonth** height of the font in graphics lines.

DISPHSTX_FORMAT_TILE4_8

Tile graphics with 4x4 pixel tiles. Frame buffer contains 1-byte tile indices. The "font" parameter is a pointer to an image in 8-bit RGB332 format, containing a row or column of tile images. Each tile is 4x4 pixels in size. If it is a row of tiles, the "fonth" parameter contains the line length of the tile image in bytes (= pitch). The line length must be aligned to 4 bytes (32 bits). If it is a column of tiles, the "fonth" parameter has a value of zero. In this case, the line length of the tile column must be 4 bytes. Pixel format group is DISPHSTX_GRP_8.

The CPU load is about 20 to 90%. To render a VGA image in a RISC-V core, normal CPU speed is not enough, fast mode ("_fast" timing) or pixel duplication with hdbl > 1 must be used.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = ((w + 3)/4 + 3) & ~3

- **font** pointer to an image of row or column of tile images, in 8-bit format RGB332. Each tile is 4x4 pixels in size. Image should be in RAM for faster access.

- **fonth** pitch of row of tiles images, or 0 if using column of tiles images.

DISPHSTX_FORMAT_TILE8_8

Tile graphics with 8x8 pixel tiles. Frame buffer contains 1-byte tile indices. The "font" parameter is a pointer to an image in 8-bit RGB332 format, containing a row or column of tile images. Each tile is 8x8 pixels in size. If it is a row of tiles, the "fonth" parameter contains the line length of the tile image in bytes (= pitch). The line length must be aligned to 4 bytes (32 bits). If it is a column of tiles, the "fonth" parameter has a value of zero. In this case, the line length of the tile column must be 8 bytes. Pixel format group is DISPHSTX_GRP_8.

The CPU load is about 20 to 84%. Rendering speed is sufficient even at normal CPU speed.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = ((w + 7)/8 + 3) & ~3

- **font** pointer to an image of row or column of tile images, in 8-bit format RGB332. Each tile is 8x8 pixels in size. Image should be in RAM for faster access.

- **fonth** pitch of row of tiles images, or 0 if using column of tiles images.

DISPHSTX_FORMAT_TILE16_8

Tile graphics with 16x16 pixel tiles. Frame buffer contains 1-byte tile indices. The "font" parameter is a pointer to an image in 8-bit RGB332 format, containing a row or column of tile images. Each tile is 16x16 pixels in size. If it is a row of tiles, the "fonth" parameter contains the line length of the tile image in bytes (= pitch). The line length must be aligned to 4 bytes (32 bits). If it is a column of tiles, the "fonth" parameter has a value of zero. In this case, the line length of the tile column must be 16 bytes. Pixel format group is DISPHSTX_GRP_8.

The CPU load is about 15 to 80%. Rendering speed is sufficient even at normal CPU speed.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = ((w + 15)/16 + 3) & ~3

- **font** pointer to an image of row or column of tile images, in 8-bit format RGB332. Each tile is 16x16 pixels in size. Image should be in RAM for faster access.

- **fonth** pitch of row of tiles images, or 0 if using column of tiles images.

DISPHSTX_FORMAT_TILE32_8

Tile graphics with 32x32 pixel tiles. Frame buffer contains 1-byte tile indices. The "font" parameter is a pointer to an image in 8-bit RGB332 format, containing a row or column of tile images. Each tile is 32x32 pixels in size. If it is a row of tiles, the "fonth" parameter contains the line length of the tile image in bytes (= pitch). The line length must be aligned to 4 bytes (32 bits). If it is a column of tiles, the "fonth" parameter has a value of zero. In this case, the line length of the tile column must be 32 bytes. Pixel format group is DISPHSTX_GRP_8.

The CPU load is about 15 to 80%. Rendering speed is sufficient even at normal CPU speed.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = ((w + 31)/32 + 3) & ~3

- **font** pointer to an image of row or column of tile images, in 8-bit format RGB332. Each tile is 32x32 pixels in size. Image should be in RAM for faster access.

- **fonth** pitch of row of tiles images, or 0 if using column of tiles images.

DISPHSTX_FORMAT_TILE4_8_PAL

Tile graphics with 4x4 pixel tiles. Frame buffer contains 1-byte tile indices. The "font" parameter is a pointer to an image in 8-bit paletted color format, containing a row or column of tile images. Each tile is 4x4 pixels in size. If it is a row of tiles, the "fonth" parameter contains the line length of the tile image in bytes (= pitch). The line length must be aligned to 4 bytes (32 bits). If it is a column of tiles, the "fonth" parameter has a value of zero. In this case, the line length of the tile column must be 4 bytes. Pixel format group is DISPHSTX_GRP_16.

The CPU load is about 80 to 90%. To render image in a RISC-V core, normal CPU speed is not enough, fast mode ("_fast" timing) or pixel duplication with hdbl > 1 must be used.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = ((w + 3)/4 + 3) & ~3

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to an image of row or column of tile images, in 8-bit paletted color format. Each tile is 4x4 pixels in size. Image should be in RAM for faster access.

- **fonth** pitch of row of tiles images, or 0 if using column of tiles images.

DISPHSTX_FORMAT_TILE8_8_PAL

Tile graphics with 8x8 pixel tiles. Frame buffer contains 1-byte tile indices. The "font" parameter is a pointer to an image in 8-bit paletted color format, containing a row or column of tile images. Each tile is 8x8 pixels in size. If it is a row of tiles, the "fonth" parameter contains the line length of the tile image in bytes (= pitch). The line length must be aligned to 4 bytes (32 bits). If it is a column of tiles, the "fonth" parameter has a value of zero. In this case, the line length of the tile column must be 8 bytes. Pixel format group is DISPHSTX_GRP_16.

The CPU load is about 70 to 84%. Rendering speed is sufficient even at normal CPU speed.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = ((w + 7)/8 + 3) & ~3

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to an image of row or column of tile images, in 8-bit paletted color format. Each tile is 8x8 pixels in size. Image should be in RAM for faster access.

- **fonth** pitch of row of tiles images, or 0 if using column of tiles images.

DISPHSTX_FORMAT_TILE16_8_PAL

Tile graphics with 16x16 pixel tiles. Frame buffer contains 1-byte tile indices. The "font" parameter is a pointer to an image in 8-bit paletted color format, containing a row or column of tile images. Each tile is 16x16 pixels in size. If it is a row of tiles, the "fonth" parameter contains the line length of the tile image in bytes (= pitch). The line length must be aligned to 4 bytes (32 bits). If it is a column of tiles, the "fonth" parameter has a value of zero. In this case, the line length of the tile column must be 16 bytes. Pixel format group is DISPHSTX_GRP_16.

The CPU load is about 67 to 80%. Rendering speed is sufficient even at normal CPU speed.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = ((w + 15)/16 + 3) & ~3

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to an image of row or column of tile images, in 8-bit paletted color format. Each tile is 16x16 pixels in size. Image should be in RAM for faster access.

- **fonth** pitch of row of tiles images, or 0 if using column of tiles images.

DISPHSTX_FORMAT_TILE32_8_PAL

Tile graphics with 32x32 pixel tiles. Frame buffer contains 1-byte tile indices. The "font" parameter is a pointer to an image in 8-bit paletted color format, containing a row or column of tile images. Each tile is 32x32 pixels in size. If it is a row of tiles, the "fonth" parameter contains the line length of the tile image in bytes (= pitch). The line length must be aligned to 4 bytes (32 bits). If it is a column of tiles, the "fonth" parameter has a value of zero. In this case, the line length of the tile column must be 32 bytes. Pixel format group is DISPHSTX_GRP_16.

The CPU load is about 63 to 80%. Rendering speed is sufficient even at normal CPU speed.

Setup and limitations of the function DispHstxVModeAddSlot():

- **pitch** length of row in bytes must be aligned to 32-bits; pitch = ((w + 31)/32 + 3) & ~3

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to an image of row or column of tile images, in 8-bit paletted color format. Each tile is 32x32 pixels in size. Image should be in RAM for faster access.

- **fonth** pitch of row of tiles images, or 0 if using column of tiles images.

DISPHSTX_FORMAT_HSTX_15

HSTX RLE compression with 15 bits per pixel, color format RGB555. It is an image data compression format that uses the hardware functions of the HSTX controller. It is suitable for image compression in the 15-bit RGB555 color format, which contains solid color areas (typically vector drawings). To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). The compressed image can be displayed in the vertical direction in a selected range of lines, but in the horizontal direction the displayed slot width must exactly match the width of the image (the image cannot be cropped horizontally). The “buf” parameter is a pointer to the image data, the "font" parameter is a pointer to a list with line offsets in the image data (the list is generated by the PicopadImg2 program). Pixel format group is DISPHSTX_GRP_15.

Rendering in DVI mode is fully hardware supported, CPU load is 3%. However, in VGA mode, decompression is simulated in software, with a CPU load of around 120 to 150%. For this reason, it is advisable to run this format either in fast CPU mode ("_fast" timing) or use pixel repetition (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **w** width of the slot must exactly match the width of the image

- **buf** pointer to image data

- **font** pointer to list with line offsets

DISPHSTX_FORMAT_HSTX_16

HSTX RLE compression with 16 bits per pixel, color format RGB565. It is an image data compression format that uses the hardware functions of the HSTX controller. It is suitable for image compression in the 16-bit RGB565 color format, which contains solid color areas (typically vector drawings). To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). The compressed image can be displayed in the vertical direction in a selected range of lines, but in the horizontal direction the displayed slot width must exactly match the width of the image (the image cannot be cropped horizontally). The “buf” parameter is a pointer to the image data, the "font" parameter is a pointer to a list with line offsets in the image data (the list is generated by the PicopadImg2 program). Pixel format group is DISPHSTX_GRP_16.

Rendering in DVI mode is fully hardware supported, CPU load is 3%. However, in VGA mode, decompression is simulated in software, with a CPU load of around 120 to 150%. For this reason, it is advisable to run this format either in fast CPU mode ("_fast" timing) or use pixel repetition (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **w** width of the slot must exactly match the width of the image

- **buf** pointer to image data

- **font** pointer to list with line offsets

DISPHSTX_FORMAT_PAT_8

Repeated pattern with 8-bit format RGB332. This mode is similar to displaying an 8-bit image, with the difference that image repeating is provided in both horizontal and vertical directions. The format is suitable for filling an area with a pattern. The width of the image is derived from the specified line length (pitch) - it must therefore be specified and cannot be less or equal 0. The line length must also be aligned to 4 bytes as in other formats, hence the width of the sample image must be a multiple of 4. The height of the image is specified in the "fonth" parameter. Pixel format group is DISPHSTX_GRP_8.

The CPU load is usually within acceptable limits to keep rendering at normal CPU speeds. However, with small patterning sizes, rendering may become too time consuming and it may be necessary to use fast CPU mode (“_fast” timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **w** width of the slot (does not determine image width)

- **buf** pointer to image data

- **pitch** length of line or row in bytes, cannot be <= 0 and must be multiply of 4

- **fonth** height of the image

DISPHSTX_FORMAT_PAT_8_PAL

Repeated pattern with 8-bit paletted colors. This mode is similar to displaying an 8-bit paletted image, with the difference that image repeating is provided in both horizontal and vertical directions. The format is suitable for filling an area with a pattern. The width of the image is derived from the specified line length (pitch) - it must therefore be specified and cannot be less or equal 0. The line length must also be aligned to 4 bytes as in other formats, hence the width of the sample image must be a multiple of 4. The height of the image is specified in the "fonth" parameter. Pixel format group is DISPHSTX_GRP_16.

The CPU load is usually within acceptable limits to keep rendering at normal CPU speeds. However, with small patterning sizes, rendering may become too time consuming and it may be necessary to use fast CPU mode ("_fast" timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **w** width of the slot (does not determine image width)

- **buf** pointer to image data

- **pitch** length of line or row in bytes, cannot be <= 0 and must be multiply of 4

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **fonth** height of the image

DISPHSTX_FORMAT_RLE8

RLE8 compression of the image in 8-bit format RGB332. This format is suitable for image compression in the 8-bit RGB332 color format, which contains solid color areas (typically vector drawings). To create such a compressed image, you can use the PicopadImg2 program (see the chapter "PicoPadImg2 - Image conversion"). The "buf" parameter is a pointer to the image data, the "font" parameter is a pointer to a list with line offsets in the image data (the list is generated by the PicopadImg2 program). Pixel format group is DISPHSTX_GRP_8.

The CPU load is about 55 to 90%. To render a VGA image in a RISC-V core, normal CPU speed is not enough, fast mode ("_fast" timing) or pixel duplication with hdbl > 1 must be used.

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to image data

- **font** pointer to list with line offsets

DISPHSTX_FORMAT_RLE8_PAL

RLE8 compression of the image in 8-bit paletted color format. This format is suitable for image compression in the 8-bit paletted color format, which contains solid color areas (typically vector drawings). To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). The “buf” parameter is a pointer to the image data, the "font" parameter is a pointer to a list with line offsets in the image data (the list is generated by the PicopadImg2 program). Pixel format group is DISPHSTX_GRP_16.

The CPU load is about 85 to 90%. Normal CPU speed is usually not enough, fast mode ("_fast" timing) or pixel duplication with hdbl > 1 should be used.

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to image data

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to list with line offsets

DISPHSTX_FORMAT_ATTR1_PAL

Attribute compression of level 1. Compression of a 4-bit image to 28%. Attribute compression includes the color index bitmap and the color attribute array. The Level 1 compression consists of 8x8 pixel index cells, with 1-bit indices, and an array of 4-bit palette colors. Associated with each 8x8 pixel index cell are two 4-bit attributes, contained in 1 byte. The "buf" parameter is a pointer to the color index bitmap. The "font" parameter is a pointer to the attribute array. The "fonth" parameter is the line length of the attribute array. To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). Pixel format group is DISPHSTX_GRP_16.

Attribute format decompression is usually time consuming and requires the use of fast processor mode ("_fast" timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to color index bitmap

- **pal** pointer to 16 palettes u16 in RGB565 format (table size 16*2=32 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in YRGB1111 format.

- **palvga** pointer to buffer to get 2*16 auto-generated palettes u32 for VGA (buffer size 2*16*4=128 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to attribute array

- **fonth** length of line of attribute array (pitch)

DISPHSTX_FORMAT_ATTR2_PAL

Attribute compression of level 2. Compression of a 4-bit image to 37%. Attribute compression includes the color index bitmap and the color attribute array. The Level 2 compression consists of 4x4 pixel index cells, with 1-bit indices, and an array of 4-bit palette colors. Associated with each 4x4 pixel index cell are two 4-bit attributes, contained in 1 byte. The "buf" parameter is a pointer to the color index bitmap. The "font" parameter is a pointer to the attribute array. The "fonth" parameter is the line length of the attribute array. To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). Pixel format group is DISPHSTX_GRP_16.

Attribute format decompression is usually time consuming and requires the use of fast processor mode ("_fast" timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to color index bitmap

- **pal** pointer to 16 palettes u16 in RGB565 format (table size 16*2=32 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in YRGB1111 format.

- **palvga** pointer to buffer to get 2*16 auto-generated palettes u32 for VGA (buffer size 2*16*4=128 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to attribute array

- **fonth** length of line of attribute array (pitch)

DISPHSTX_FORMAT_ATTR3_PAL

Attribute compression of level 3. Compression of a 4-bit image to 56%. Attribute compression includes the color index bitmap and the color attribute array. The Level 3 compression consists of 8x8 pixel index cells, with 2-bit indices, and an array of 4-bit palette colors. Associated with each 8x8 pixel index cell are four 4-bit attributes, contained in 2 bytes. The "buf" parameter is a pointer to the color index bitmap. The "font" parameter is a pointer to the attribute array. The "fonth" parameter is the line length of the attribute array. To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). Pixel format group is DISPHSTX_GRP_16.

Attribute format decompression is usually time consuming and requires the use of fast processor mode ("_fast" timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to color index bitmap

- **pal** pointer to 16 palettes u16 in RGB565 format (table size 16*2=32 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in YRGB1111 format.

- **palvga** pointer to buffer to get 2*16 auto-generated palettes u32 for VGA (buffer size 2*16*4=128 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to attribute array

- **fonth** length of line of attribute array (pitch)

DISPHSTX_FORMAT_ATTR4_PAL

Attribute compression of level 4. Compression of a 4-bit image to 75%. Attribute compression includes the color index bitmap and the color attribute array. The Level 4 compression consists of 4x4 pixel index cells, with 2-bit indices, and an array of 4-bit palette colors. Associated with each 4x4 pixel index cell are four 4-bit attributes, contained in 2 bytes. The "buf" parameter is a pointer to the color index bitmap. The "font" parameter is a pointer to the attribute array. The "fonth" parameter is the line length of the attribute array. To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). Pixel format group is DISPHSTX_GRP_16.

Attribute format decompression is usually time consuming and requires the use of fast processor mode ("_fast" timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to color index bitmap

- **pal** pointer to 16 palettes u16 in RGB565 format (table size 16*2=32 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in YRGB1111 format.

- **palvga** pointer to buffer to get 2*16 auto-generated palettes u32 for VGA (buffer size 2*16*4=128 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to attribute array

- **fonth** length of line of attribute array (pitch)

DISPHSTX_FORMAT_ATTR5_PAL

Attribute compression of level 5. Compression of a 8-bit image to 15%. Attribute compression includes the color index bitmap and the color attribute array. The Level 5 compression consists of 8x8 pixel index cells, with 1-bit indices, and an array of 8-bit palette colors. Associated with each 8x8 pixel index cell are two 8-bit attributes, contained in 2 bytes. The "buf" parameter is a pointer to the color index bitmap. The "font" parameter is a pointer to the attribute array. The "fonth" parameter is the line length of the attribute array. To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). Pixel format group is DISPHSTX_GRP_16.

Attribute format decompression is usually time consuming and requires the use of fast processor mode ("_fast" timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to color index bitmap

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to attribute array

- **fonth** length of line of attribute array (pitch)

DISPHSTX_FORMAT_ATTR6_PAL

Attribute compression of level 6. Compression of a 8-bit image to 25%. Attribute compression includes the color index bitmap and the color attribute array. The Level 6 compression consists of 4x4 pixel index cells, with 1-bit indices, and an array of 8-bit palette colors. Associated with each 4x4 pixel index cell are two 8-bit attributes, contained in 2 bytes. The "buf" parameter is a pointer to the color index bitmap. The "font" parameter is a pointer to the attribute array. The "fonth" parameter is the line length of the attribute array. To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). Pixel format group is DISPHSTX_GRP_16.

Attribute format decompression is usually time consuming and requires the use of fast processor mode ("_fast" timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to color index bitmap

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to attribute array

- **fonth** length of line of attribute array (pitch)

DISPHSTX_FORMAT_ATTR7_PAL

Attribute compression of level 7. Compression of a 8-bit image to 31%. Attribute compression includes the color index bitmap and the color attribute array. The Level 7 compression

consists of 8x8 pixel index cells, with 2-bit indices, and an array of 8-bit palette colors. Associated with each 8x8 pixel index cell are four 8-bit attributes, contained in 4 bytes. The "buf" parameter is a pointer to the color index bitmap. The "font" parameter is a pointer to the attribute array. The "fonth" parameter is the line length of the attribute array. To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). Pixel format group is DISPHSTX_GRP_16.

Attribute format decompression is usually time consuming and requires the use of fast processor mode ("_fast" timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to color index bitmap

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to attribute array

- **fonth** length of line of attribute array (pitch)

DISPHSTX_FORMAT_ATTR8_PAL

Attribute compression of level 8. Compression of a 8-bit image to 50%. Attribute compression includes the color index bitmap and the color attribute array. The Level 8 compression consists of 4x4 pixel index cells, with 2-bit indices, and an array of 8-bit palette colors. Associated with each 4x4 pixel index cell are four 8-bit attributes, contained in 4 bytes. The "buf" parameter is a pointer to the color index bitmap. The "font" parameter is a pointer to the attribute array. The "fonth" parameter is the line length of the attribute array. To create such a compressed image, you can use the PicopadImg2 program (see the chapter “PicoPadImg2 - Image conversion”). Pixel format group is DISPHSTX_GRP_16.

Attribute format decompression is usually time consuming and requires the use of fast processor mode ("_fast" timing) or pixel duplication (hdbl > 1).

Setup and limitations of the function DispHstxVModeAddSlot():

- **buf** pointer to color index bitmap

- **pal** pointer to 256 palettes u16 in RGB565 format (table size 256*2=512 bytes). Palettes should be in RAM for faster access. Set NULL to use default palettes in RGB332 format.

- **palvga** pointer to buffer to get 2*256 auto-generated palettes u32 for VGA (buffer size 2*256*4=2048 bytes). Set NULL to auto-create new one with malloc() (must be manually deleted after the driver has been terminated). If you use default palettes (pal = NULL) and do not specify a buffer for VGA palettes (palvga = NULL), the internal table of pre-generated VGA palettes is used. In such case, do not destroy the table to which the palvga entry of the descriptor points. This entry can be NULL if the VGA output is not supported (DISPHSTX_USE_VGA = 0).

- **font** pointer to attribute array

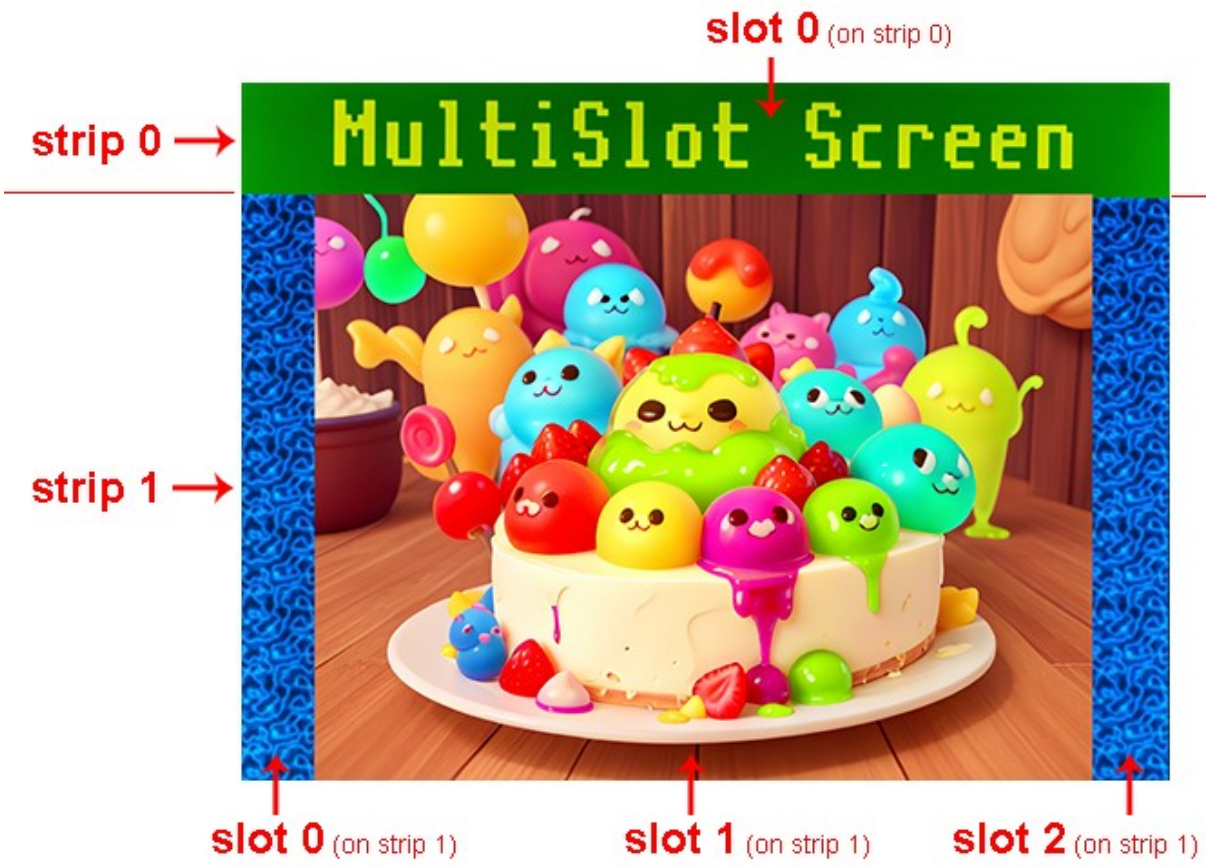
- **fonth** length of line of attribute array (pitch)

DISPHSTX_FORMAT_CUSTOM

Custom user format - set DispHstxVColorCustom to your format descriptor sDispHstxVColor.

12. Divided screen

The DispHSTX driver allows the screen to be divided into several segments, where each segment can be used as a separate screen, independent of the other segments. In the vertical direction, the screen can be divided into strips, which are bars of a given height. Each strip can be further divided into multiple slots, which are separate screen segments of a given width and with their own format.



The following procedure is used to define a divided screen:

- The DispHstxVModeInitTime() function initializes the sDispHstxVModeState descriptor according to the selected sDispHstxVModeTime signal generation timing descriptor.

- The DispHstxVModeAddStrip function adds a new strip of the specified height to the state descriptor.

- The DispHstxVModeAddSlot function adds a new slot of the specified width to the last added strip.

You can repeatedly add additional slots to the last strip, or add another strip and slots to it. The maximum number of strips and the maximum number of slots in a strip are determined by the DISPHSTX_STRIP_MAX and DISPHSTX_SLOT_MAX configuration switches.

After defining the screen content, the video mode is activated by the DispHstxSelDispMode function.

Slots in graphic format can be connected to the drawing canvas library by the DispHstxLinkCan() function. The video node can be terminated with the DispHstxAllTerm() function. After the video mode is terminated, it may be necessary to discard buffers that were automatically created. While the driver is running, the screen definition can be exchanged for a different configuration seamless without disrupting video synchronization using the DispHstxExchange() function.

If slots with different configurations are adjacent to each other, it may be necessary to include a separation gap between the slots to ensure sufficient time for the driver to redefine registers in the HSTX controller. The separator gap is a one-color field with a selectable color that is generated in a different way than the normal image (it does not use the TMDS encoder), allowing the driver to redefine the controller registers.

The separation gap must be used under the following conditions:

- If the color format group of the pixels DISPHSTX_GRP_* of the slots is different,

- or if the pixel duplication item hdbl differs,

- or if the slot data alignment differs.

The slot data alignment is varying according to the displayed slot width - if possible, faster sending of image data as 32-bit u32 entries is preferred. If the width does not correspond to an integer multiple of u32, data is sent in 16-bit or 8-bit entries.

If you don't know whether a slot separator is necessary, specify a slot separator width of -1 and the DispHstxVModeAddSlot() function will determine whether to use the slot separator gap. The function determines the separation gap uniformly for both VGA and DVI mode, although in some cases a gap would not be necessary in VGA mode, in order to ensure a uniform image after the mode switch.

The recommended width of the separation gap is 20 video points. If the separation gap is too short, the data being sent will be distorted and this will result in either image corruption or synchronization dropout.

The colour of the separating gap can be selected. The gap will not appear in the exact color specified, the color may vary slightly because the driver will use the closest possible color that meets the DC balance requirement in the TMDS code. Otherwise, the DC level would be drifted and the colors of the image would be deformed.

13. Drawing Canvas

The "Drawing Canvas" library is used to draw graphics into the frame buffer in the DISPHSTX_FORMAT_1 to DISPHSTX_FORMAT_16, or DISPHSTX_FORMAT_1_PAL to DISPHSTX_FORMAT_8_PAL graphic format. In order to use the library, the USE_DRAWCAN compilation switch must be set to 1. The sDrawCan descriptor contains the information needed for the drawing functions to work - such as the frame buffer address, line length, dimensions and image format. After the video mode is initialized, the DispHstxLinkCan() function can be used to link a slot in the graphics format to the drawing canvas descriptor. The function fills the descriptor entries as needed. Linking to the default drawing canvas is also done automatically by functions to simplify video mode initialization.

The library contains both separate sections for individual graphics formats from 1 to 16 bits per pixel, and a common section allowing the use of functions with automatic selection of bit depth according to the active format.

Each separate section, serving a specific color depth of 1 to 16 bits per pixel, contains a default drawing canvas descriptor DrawCan1 to DrawCan16, a pointer to the current drawing canvas pDrawCan1 to pDrawCan16, and a set of functions Draw1* to Draw16* or DrawCan1* to DrawCan16*. You can attach an image slot directly to a specific descriptor according to the corresponding bit depth, and call directly the functions corresponding to the format used. Thus, you can operate simultaneously drawing canvases with different formats.

Another option is to use a common canvas. The descriptor for the common canvas is DrawCan and the pointer to the currently selected canvas is pDrawCan. You connect the canvas to the image slot again with the DispHstxLinkCan() function. The difference is that the function itself distinguishes which image format is used and selects the appropriate function set accordingly. If you then call functions labeled Draw* or DrawCan* (not containing the format number in the name), the appropriate corresponding functions will be called according to the function address table. This method of use is a little slower, but allows more universal access to the drawing functions.

The "**sDrawCan* can**" parameter in the following drawing functions indicates a pointer to the drawing canvas.

A brief overview of the main functions for the common canvas

void DrawSelFont8x8(); Select font 8x8

void DrawSelFont8x14(); Select font 8x14

void DrawSelFont8x16(); Select font 8x16

void DrawClearCol(u16 col); Clear canvas color

void DrawClear(); Clear canvas black

void DrawPoint(int x, int y, u16 col); Draw point

u16 DrawGetPoint(int x, int y); Get point

void DrawRect(int x, int y, int w, int h, u16 col); Draw rectangle

void DrawHLine(int x, int y, int w, u16 col); Draw horizontal line

void DrawVLine(int x, int y, int h, u16 col); Draw vertical line

void DrawFrame(int x, int y, int w, int h, u16 col_light, u16 col_dark); Draw 3D frame

void DrawFrameW(int x, int y, int w, int h, u16 col, int thick); Draw thick frame

void DrawLine(int x1, int y1, int x2, int y2, u16 col); Draw line

void DrawLineW(int x1, int y1, int x2, int y2, u16 col, int thick, Bool round);
Draw thick line

void DrawRound(int x, int y, int d, u16 col, u8 mask); Draw round

void DrawCircle(int x, int y, int d, u16 col, u8 mask); Draw circle or arc

void DrawRing(int x, int y, int d, int din, u16 col, u8 mask); Draw ring

void DrawTriangle(int x1, int y1, int x2, int y2, int x3, int y3, u16 col); Draw triangle

void DrawFill(int x, int y, u16 col); Draw fill area

void DrawChar(char ch, int x, int y, u16 col, int scalex, int scaley);
Draw character transparent

void DrawCharBg(char ch, int x, int y, u16 col, u16 bgcol, int scalex, int scaley);
Draw character with background

void DrawText(const char* text, int len, int x, int y, u16 col, int scalex, int scaley);
Draw text transparent

void DrawTextBg(const char* text, int len, int x, int y, u16 col, u16 bgcol, int scalex, int scaley); Draw text with background

void DrawEllipse(int x, int y, int dx, int dy, u16 col, u8 mask); Draw ellipse

void DrawFillEllipse(int x, int y, int dx, int dy, u16 col, u8 mask); Draw filled ellipse

void DrawImg(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs);
Draw image

void DrawBlit(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs, u16 col);
Draw transparent image

void DrawBlitSubs(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs, u16 col, u16 fnd, u16 subs); Draw image with substitute color

void DrawGetImg(int xs, int ys, int w, int h, void* dst, int xd, int yd, int wbd);
Get image from canvas to buffer

A set of functions usable independently of bit depth

void DrawCanDirtyAll(sDrawCan* can);

Set dirty all frame buffer.

void DrawCanDirtyNone(sDrawCan* can);

Set dirty none of frame buffer.

void DrawCanDirtyPoint_Fast(sDrawCan* can, int x, int y);

x ... X coordinate

y ... Y coordinate

Update dirty area by point. Does not check clipping limits.

void DrawCanDirtyPoint(sDrawCan* can, int x, int y);

x ... X coordinate

y ... Y coordinate

Update dirty area by point. Does check clipping limits.

void DrawCanDirtyRect_Fast(sDrawCan* can, int x, int y, int w, int h);

x ... X coordinate

y ... Y coordinate

w ... width of the rectangle

h ... height of the rectangle

Update dirty area by rectangle. Does not check clipping limits.

void DrawCanDirtyRect(sDrawCan* can, int x, int y, int w, int h);

x ... X coordinate

y ... Y coordinate

w ... width of the rectangle

h ... height of the rectangle

Update dirty area by rectangle. Does check clipping limits.

Bool DrawCanRectClipped(sDrawCan* can, int x, int y, int w, int h);

x ... X coordinate

y ... Y coordinate

w ... width of the rectangle

h ... height of the rectangle

Check if rectangle is clipped and if it is safe to use fast variant of the function.

void DrawCanSelFont(sDrawCan* can, const u8* font, u8 fontw, u8 fonth);

font ... pointer to the font (256 characters in cells of width 8 pixels, 1-bit format)

fontw ... font width (number of pixels, max. 8)

fonth ... font height (number of lines)

Select font.

void DrawCanSelFont8x8(sDrawCan* can);

Select sans-serif bold font, height 8 lines.

void DrawCanSelFont8x14(sDrawCan* can);

Select sans-serif bold font, height 14 lines.

void DrawCanSelFont8x16(sDrawCan* can);

Select sans-serif bold font, height 16 lines.

void DrawCanSelFont8x14Serif(sDrawCan* can);

Select serif bold font, height 14 lines.

void DrawCanSelFont8x16Serif(sDrawCan* can);

Select serif bold font, height 16 lines.

void DrawCanSelFont6x8(sDrawCan* can);

Select condensed font, width 6 pixels, height 8 lines.

void DrawCanSelFont8x8Game(sDrawCan* can);

Select game font, height 8 lines.

void DrawCanSelFont8x8Ibm(sDrawCan* can);

Select IBM charset font, height 8 lines.

void DrawCanSelFont8x14Ibm(sDrawCan* can);

Select IBM charset font, height 14 lines.

void DrawCanSelFont8x16Ibm(sDrawCan* can);

Select IBM charset font, height 16 lines.

void DrawCanSelFont8x8IbmThin(sDrawCan* can);

Select IBM charset thin font, height 8 lines.

void DrawCanSelFont8x8Italic(sDrawCan* can);

Select italic font, height 8 lines.

void DrawCanSelFont8x8Thin(sDrawCan* can);

Select thin font, height 8 lines.

void DrawCanSelFont5x8(sDrawCan* can);

Select tiny font, width 5 pixels, height 8 lines.

Colour constants

The color specified as a function parameter is always in the format corresponding to the bit depth of the format. For example, for a 3 bits per pixel format, the color is a 3-bit number in RGB111 format. If you are using functions for a specific bit depth, you can use the corresponding COL?_* constants. For example, yellow color in 8-bit format has the name COL8_YELLOW.

For universal use, a set of constants with automatic selection according to the bit format used is also available. These colors are named COL_* and are loaded from the currently selected canvas pDrawCan.

COL_RGB(r,g,b)	convert RGB888 color to pixel color
COL_RANDOM	random pixel color
COL_BLACK	black color
COL_BLUE	blue color
COL_GREEN	green color
COL_CYAN	cyan color
COL_RED	red color
COL_MAGENTA	magenta color
COL_YELLOW	yellow color
COL_WHITE	white color
COL_GRAY	gray color
COL_DKBLUE	dark blue clor
COL_DKGREEN	dark green color
COL_DKCYAN	dark cyan color
COL_DKRED	dark red color
COL_DKMAGENTA	dark magenta color
COL_DKYELLOW	dark yellow color
COL_DKWHITE	dark white color
COL_DKGRAY	dark gray color
COL_LTBLUE	light blue color
COL_LTGREEN	light green color
COL_LTCYAN	light cyan color
COL_LTRED	light red color
COL_LTMAGENTA	light magenta color
COL_LTYELLOW	light yellow color
COL_LTGRAY	light gray color
COL_AZURE	azure blue color
COL_ORANGE	orange color
COL_BROWN	brown color

Bit-depth specific functions

The functions listed here occur both in the bit depth specific version, in which case they contain the number 1 to 16 in their name, and in the universal version for the common drawing canvas, in which case they do not contain a number in their name. For simplicity, the functions are listed here with the "?" character, instead of which the number 1 to 16 is used, or they will be without a number.

For example, the function for clearing the surface is given here as Draw?Clear(), and occurs in the versions Draw1Clear, Draw2Clear, Draw3Clear, Draw4Clear, Draw6Clear, Draw8Clear, Draw12Clear, Draw15Clear, Draw16Clear and DrawClear. These functions

refer to the current canvas pDrawCan1, pDrawCan2, pDrawCan3, pDrawCan4, pDrawCan6, pDrawCan8, pDrawCan12, pDrawCan15, pDrawCan16 and pDrawCan.

Most of the functions occur in multiple variants. If the suffix "_Fast" is specified, it is a fast variant of the function that does not check canvas boundaries (including the Y boundary of the strip mode of the back buffer) and that does not update dirty boundaries. Functions named DrawCan?* refer to the canvas specified as a function parameter. Functions named Draw?* refer to the current canvas set in the pDrawCan? pointer.

In addition to the basic drawing functions that use colour, there are also inverse functions, marked "Inv" in the name. Inverse functions invert the pixel color. These functions are used, for example, to display a selection cursor. The first use of the function highlights the object on the screen, the second use of the function returns the original screen content without the need to keep the original content. For example, the Draw?Rect() function draws a rectangle with the specified color. The Draw?RectInv() function inverts the rectangle, and a second use undoes the drawn rectangle and returns the original background.

void SetDrawCan?(sDrawCan* can);

Set default drawing canvas for this format.

int Draw?Width();

Get current width of current canvas.

int Draw?Height();

Get current height of current canvas.

u8* Draw?Buf();

Get current frame buffer of current canvas.

u16 Draw?ColRgb(u8 r, u8 g, u8 b);

- **r** ... red component 0..255
- **g** ... green component 0..255
- **b** ... blue component 0..255

Convert RGB888 color to this pixel color format.

COLOR?(r,g,b)

- **r** ... red component 0..255
- **g** ... green component 0..255
- **b** ... blue component 0..255

Convert RGB888 color to this pixel color format.

u16 Draw?ColRand();

Random color.

int Draw?Pitch(int w);

w ... width of the image

Calculate pitch of graphics line from image width (get length of line in bytes, rounded up to 4-byte word).

int Draw?MaxWidth(int pitch);

pitch ... pitch of the image (length of the line, in bytes)

Calculate max. width of image from the pitch (pitch is length of line in bytes, rounded up to 4-byte word).

Clear canvas

void DrawCan?ClearCol(sDrawCan* can, u16 col);
void Draw?ClearCol(u16 col);

col ... color

Clear canvas with color.

void DrawCan?Clear(sDrawCan* can);
void Draw?Clear();

Clear canvas with black color.

Draw point

void DrawCan?Point_Fast(sDrawCan* can, int x, int y, u16 col);
void DrawCan?Point(sDrawCan* can, int x, int y, u16 col);
void DrawCan?PointInv_Fast(sDrawCan* can, int x, int y);
void DrawCan?PointInv(sDrawCan* can, int x, int y);
void Draw?Point_Fast(int x, int y, u16 col);
void Draw?PointInv_Fast(int x, int y);
void Draw?PointInv(int x, int y);
void Draw?Point(int x, int y, u16 col);

x ... X coordinate

y ... Y coordinate

col ... color

Draw point.

Get point

u16 DrawCan?GetPoint_Fast(sDrawCan* can, int x, int y);
u16 Draw?GetPoint_Fast(int x, int y);
u16 DrawCan?GetPoint(sDrawCan* can, int x, int y);
u16 Draw?GetPoint(int x, int y);

x ... X coordinate

y ... Y coordinate

Get point.

Draw rectangle

void DrawCan?Rect_Fast(sDrawCan* can, int x, int y, int w, int h, u16 col);
void DrawCan?Rect(sDrawCan* can, int x, int y, int w, int h, u16 col);
void DrawCan?RectInv_Fast(sDrawCan* can, int x, int y, int w, int h);
void DrawCan?RectInv(sDrawCan* can, int x, int y, int w, int h);
void Draw?Rect_Fast(int x, int y, int w, int h, u16 col);
void Draw?RectInv_Fast(int x, int y, int w, int h);
void Draw?RectInv(int x, int y, int w, int h);
void Draw?Rect(int x, int y, int w, int h, u16 col);

x ... X coordinate

y ... Y coordinate

w ... width

h ... height

col ... color

Draw rectangle. Slow version - negative dimensions flip rectangle. Fast version - dimensions must be > 0.

Draw horizontal line

void DrawCan?HLine_Fast(sDrawCan* can, int x, int y, int w, u16 col);
void DrawCan?HLine(sDrawCan* can, int x, int y, int w, u16 col);
void DrawCan?HLineInv_Fast(sDrawCan* can, int x, int y, int w);
void DrawCan?HLineInv(sDrawCan* can, int x, int y, int w);
void Draw?HLine_Fast(int x, int y, int w, u16 col);
void Draw?HLineInv_Fast(int x, int y, int w);
void Draw?HLineInv(int x, int y, int w);
void Draw?HLine(int x, int y, int w, u16 col);

x ... X coordinate

y ... Y coordinate

w ... width of the line

col ... color

Draw horizontal line of the height 1 pixel. Slow version - negative width flip line. Fast version - width must be > 0.

Draw vertical line

void DrawCan?VLine_Fast(sDrawCan* can, int x, int y, int h, u16 col);
void DrawCan?VLine(sDrawCan* can, int x, int y, int h, u16 col);
void DrawCan?VLineInv_Fast(sDrawCan* can, int x, int y, int h);
void DrawCan?VLineInv(sDrawCan* can, int x, int y, int h);
void Draw?VLine_Fast(int x, int y, int h, u16 col);
void Draw?VLineInv_Fast(int x, int y, int h);
void Draw?VLineInv(int x, int y, int h);
void Draw?VLine(int x, int y, int h, u16 col);

x ... X coordinate

y ... Y coordinate

h ... height of the line

col ... color

Draw vertical line of the width 1 pixel. Slow version - negative height flip line. Fast version - height must be > 0.

Draw frame

void DrawCan?Frame_Fast(sDrawCan* can, int x, int y, int w, int h, u16 col_light, u16 col_dark);
void DrawCan?Frame(sDrawCan* can, int x, int y, int w, int h, u16 col_light, u16 col_dark);
void DrawCan?FrameInv_Fast(sDrawCan* can, int x, int y, int w, int h);
void DrawCan?FrameInv(sDrawCan* can, int x, int y, int w, int h);
void Draw?Frame_Fast(int x, int y, int w, int h, u16 col_light, u16 col_dark);
void Draw?FrameInv_Fast(int x, int y, int w, int h);
void Draw?FrameInv(int x, int y, int w, int h);
void Draw?Frame(int x, int y, int w, int h, u16 col_light, u16 col_dark);

x ... X coordinate

y ... Y coordinate

w ... width

h ... height

col ... color

Draw 1-pixel frame. Slow version - negative dimensions flip frame. Fast version - dimensions must be > 0.

void DrawCan?FrameW_Fast(sDrawCan* can, int x, int y, int w, int h, u16 col, int thick);
void DrawCan?FrameW(sDrawCan* can, int x, int y, int w, int h, u16 col, int thick);
void DrawCan?FrameWInv_Fast(sDrawCan* can, int x, int y, int w, int h, int thick);
void DrawCan?FrameWInv(sDrawCan* can, int x, int y, int w, int h, int thick);
void Draw?FrameW_Fast(int x, int y, int w, int h, u16 col, int thick);
void Draw?FrameWInv_Fast(int x, int y, int w, int h, int thick);
void Draw?FrameWInv(int x, int y, int w, int h, int thick);
void Draw?FrameW(int x, int y, int w, int h, u16 col, int thick);

x ... X coordinate

y ... Y coordinate

w ... width

h ... height

col ... color

thick ... thick of the frame

Draw thick frame. Slow version - negative dimensions flip frame. Fast version - dimensions and thick must be > 0.

Draw line

void DrawCan?LineOver_Fast(sDrawCan* can, int x1, int y1, int x2, int y2, u16 col, Bool over);
void DrawCan?LineOver(sDrawCan* can, int x1, int y1, int x2, int y2, u16 col, Bool over);
void DrawCan?LineOverInv_Fast(sDrawCan* can, int x1, int y1, int x2, int y2, Bool over);
void DrawCan?LineOverInv(sDrawCan* can, int x1, int y1, int x2, int y2, Bool over);
void Draw?LineOver_Fast(int x1, int y1, int x2, int y2, u16 col, Bool over);
void Draw?LineOverInv_Fast(int x1, int y1, int x2, int y2, Bool over);
void Draw?LineOverInv(int x1, int y1, int x2, int y2, Bool over);
void Draw?LineOver(int x1, int y1, int x2, int y2, u16 col, Bool over);

x1 ... X coordinate of 1st end point

y1 ... Y coordinate of 1st end point

x2 ... X coordinate of 2nd end point

y2 ... Y coordinate of 2nd end point

col ... color

over ... flag whether to draw overlapped pixels

Draw line with overlapped pixels. This variant of the function is not intended for normal use. It is used internally in the following 2 functions. The "over" flag determines whether overlay pixels - which are additional pixels drawn at line break points - will be drawn in addition to the regular line pixels.

void DrawCan?Line_Fast(sDrawCan* can, int x1, int y1, int x2, int y2, u16 col);
void DrawCan?Line(sDrawCan* can, int x1, int y1, int x2, int y2, u16 col);
void DrawCan?LineInv_Fast(sDrawCan* can, int x1, int y1, int x2, int y2);
void DrawCan?LineInv(sDrawCan* can, int x1, int y1, int x2, int y2);
void Draw?Line_Fast(int x1, int y1, int x2, int y2, u16 col);
void Draw?LineInv_Fast(int x1, int y1, int x2, int y2);
void Draw?LineInv(int x1, int y1, int x2, int y2);
void Draw?Line(int x1, int y1, int x2, int y2, u16 col);

x1 ... X coordinate of 1st end point

y1 ... Y coordinate of 1st end point

x2 ... X coordinate of 2nd end point

y2 ... Y coordinate of 2nd end point

col ... color

Draw line of the 1-pixel thick.

void DrawCan?LineW_Fast(sDrawCan* can, int x1, int y1, int x2, int y2, u16 col, int thick, Bool round);
void DrawCan?LineW(sDrawCan* can, int x1, int y1, int x2, int y2, u16 col, int thick, Bool round);
void DrawCan?LineWInv_Fast(sDrawCan* can, int x1, int y1, int x2, int y2, int thick);
void DrawCan?LineWInv(sDrawCan* can, int x1, int y1, int x2, int y2, int thick);
void Draw?LineW_Fast(int x1, int y1, int x2, int y2, u16 col, int thick, Bool round);
void Draw?LineWInv_Fast(int x1, int y1, int x2, int y2, int thick);
void Draw?LineWInv(int x1, int y1, int x2, int y2, int thick);
void Draw?LineW(int x1, int y1, int x2, int y2, u16 col, int thick, Bool round);

x1 ... X coordinate of 1st end point

y1 ... Y coordinate of 1st end point

x2 ... X coordinate of 2nd end point

y2 ... Y coordinate of 2nd end point

col ... color

thick ... thick of line in pixels

round ... flag whether to draw round ends

Draw thick line with round ends. Inverse variant does not support round ends.

Draw round (filled circle)

void DrawCan?Round_Fast(sDrawCan* can, int x, int y, int d, u16 col, u8 mask);
void DrawCan?Round(sDrawCan* can, int x, int y, int d, u16 col, u8 mask);
void DrawCan?RoundInv_Fast(sDrawCan* can, int x, int y, int d, u8 mask);
void DrawCan?RoundInv(sDrawCan* can, int x, int y, int d, u8 mask);
void Draw?Round_Fast(int x, int y, int d, u16 col, u8 mask);
void Draw?RoundInv_Fast(int x, int y, int d, u8 mask);
void Draw?RoundInv(int x, int y, int d, u8 mask);
void Draw?Round(int x, int y, int d, u16 col, u8 mask);

x ... X coordinate of the centre of the round

y ... Y coordinate of the centre of the round

d ... diameter of the round (radius = d/2)

mask ... hide parts of the round with DRAWCAN_ROUND_* (or their combination);

or use DRAWCAN_ROUND_ALL or 0 to draw whole round

Draw round. Fast version - diameter must be > 0. Note that the parameter 'd' means the diameter of the round and not the radius, as is usually the case. This is in order to be able to draw a round with an odd diameter, which would otherwise not be possible when specifying the radius as an integer.

The 'mask' parameter is used to hide parts of the round. It is a value or ORed combination of values:

DRAWCAN_ROUND_NOTOP = 1	hide top part of the round
DRAWCAN_ROUND_NOBOTTOM = 2	hide bottom part of the round
DRAWCAN_ROUND_NOLEFT = 4	hide left part of the round
DRAWCAN_ROUND_NORIGHT = 8	hide right part of the round

Or other alternatives of names of the values:

DRAWCAN_ROUND_ALL = 0	draw whole round
DRAWCAN_ROUND_TOP = DRAWCAN_ROUND_NOBOTTOM	... draw top half-round
DRAWCAN_ROUND_BOTTOM = DRAWCAN_ROUND_NOTOP	... draw bottom half-round
DRAWCAN_ROUND_LEFT = DRAWCAN_ROUND_NORIGHT	... draw left half-round
DRAWCAN_ROUND_RIGHT = DRAWCAN_ROUND_NOLEFT	... draw right half-round

Draw circle

void DrawCan?Circle_Fast(sDrawCan* can, int x, int y, int d, u16 col, u8 mask);
void DrawCan?Circle(sDrawCan* can, int x, int y, int d, u16 col, u8 mask);
void DrawCan?CircleInv_Fast(sDrawCan* can, int x, int y, int d, u8 mask);
void DrawCan?CircleInv(sDrawCan* can, int x, int y, int d, u8 mask);
void Draw?Circle_Fast(int x, int y, int d, u16 col, u8 mask);
void Draw?CircleInv_Fast(int x, int y, int d, u8 mask);
void Draw?CircleInv(int x, int y, int d, u8 mask);
void Draw?Circle(int x, int y, int d, u16 col, u8 mask);

x ... X coordinate of the centre of the circle

y ... Y coordinate of the centre of the circle

d ... diameter of the circle (radius = d/2)

mask ... draw circle arcs, use combination of DRAWCAN_CIRCLE_*;

or use DRAWCAN_CIRCLE_ALL or 0xFF to draw whole circle

Draw circle or arc. Note that the parameter 'd' means the diameter of the circle and not the radius, as is usually the case. This is in order to be able to draw a circle with an odd diameter, which would otherwise not be possible when specifying the radius as an integer.

The 'mask' parameter is used to display parts of the circle. It is a value or ORed combination of values:

DRAWCAN_CIRCLE_ARC0 = 1	draw arc 0..45 deg
DRAWCAN_CIRCLE_ARC1 = 2	draw arc 45..90 deg
DRAWCAN_CIRCLE_ARC2 = 4	draw arc 90..135 deg
DRAWCAN_CIRCLE_ARC3 = 8	draw arc 135..180 deg
DRAWCAN_CIRCLE_ARC4 = 16	draw arc 180..225 deg
DRAWCAN_CIRCLE_ARC5 = 32	draw arc 225..270 deg
DRAWCAN_CIRCLE_ARC6 = 64	draw arc 270..315 deg
DRAWCAN_CIRCLE_ARC7 = 128	draw arc 315..360 deg

Or other alternatives of the values:

DRAWCAN_CIRCLE_ALL = 0xFF	draw whole circle
DRAWCAN_CIRCLE_TOP	draw top half-circle
DRAWCAN_CIRCLE_BOTTOM	draw bottom half-circle
DRAWCAN_CIRCLE_LEFT	draw left half-circle
DRAWCAN_CIRCLE_RIGHT	draw right half-circle

Draw ring

void DrawCan?Ring_Fast(sDrawCan* can, int x, int y, int d, int din, u16 col, u8 mask);
void DrawCan?Ring(sDrawCan* can, int x, int y, int d, int din, u16 col, u8 mask);
void DrawCan?RingInv_Fast(sDrawCan* can, int x, int y, int d, int din, u8 mask);
void DrawCan?RingInv(sDrawCan* can, int x, int y, int d, int din, u8 mask);
void Draw?Ring_Fast(int x, int y, int d, int din, u16 col, u8 mask);
void Draw?RingInv_Fast(int x, int y, int d, int din, u8 mask);
void Draw?RingInv(int x, int y, int d, int din, u8 mask);
void Draw?Ring(int x, int y, int d, int din, u16 col, u8 mask);

x ... X coordinate of the centre of the circle

y ... Y coordinate of the centre of the circle

d ... outer diameter of the ring (outer radius = d/2)

din ... inner diameter of the ring (inner radius = din/2), must be din < d

mask ... hide parts of the ring with DRAWCAN_ROUND_* (or their combination);

or use DRAWCAN_ROUND_ALL or 0 to draw whole ring

Draw ring. Fast version - outer diameter must be > 0. Note that the parameters 'd' and 'din' mean the diameters of the ring and not the radius, as is usually the case. This is in order to

be able to draw a ring with an odd diameter, which would otherwise not be possible when specifying the radius as an integer.

The 'mask' parameter is used to hide parts of the ring. It is a value or ORed combination of values:

DRAWCAN_ROUND_NOTOP = 1	hide top part of the ring
DRAWCAN_ROUND_NOBOTTOM = 2	hide bottom part of the ring
DRAWCAN_ROUND_NOLEFT = 4	hide left part of the ring
DRAWCAN_ROUND_NORIGHT = 8	hide right part of the ring

Or other alternatives of names of the values:

DRAWCAN_ROUND_ALL = 0	draw whole ring
DRAWCAN_ROUND_TOP = DRAWCAN_ROUND_NOBOTTOM	... draw top half-ring
DRAWCAN_ROUND_BOTTOM = DRAWCAN_ROUND_NOTOP	... draw bottom half-ring
DRAWCAN_ROUND_LEFT = DRAWCAN_ROUND_NORIGHT	... draw left half-ring
DRAWCAN_ROUND_RIGHT = DRAWCAN_ROUND_NOLEFT	... draw right half-ring

Draw triangle

void DrawCan?Triangle_Fast(sDrawCan* can, int x1, int y1, int x2, int y2, int x3, int y3, u16 col);
void DrawCan?Triangle(sDrawCan* can, int x1, int y1, int x2, int y2, int x3, int y3, u16 col);
void DrawCan?TriangleInv_Fast(sDrawCan* can, int x1, int y1, int x2, int y2, int x3, int y3);
void DrawCan?TriangleInv(sDrawCan* can, int x1, int y1, int x2, int y2, int x3, int y3);
void Draw?Triangle_Fast(int x1, int y1, int x2, int y2, int x3, int y3, u16 col);
void Draw?TriangleInv_Fast(int x1, int y1, int x2, int y2, int x3, int y3);
void Draw?TriangleInv(int x1, int y1, int x2, int y2, int x3, int y3);
void Draw?Triangle(int x1, int y1, int x2, int y2, int x3, int y3, u16 col);

x1 ... X coordinate of 1st vertex

y1 ... Y coordinate of 1st vertex

x2 ... X coordinate of 2nd vertex

y2 ... Y coordinate of 2nd vertex

x3 ... X coordinate of 3rd vertex

y3 ... Y coordinate of 3rd vertex

col ... color

Draw filled triangle.

Draw fill area

void DrawCan?Fill(sDrawCan* can, int x, int y, u16 col);
void Draw?Fill(int x, int y, u16 col);

x ... X coordinate, where to start filling

y ... Y coordinate, where tostart filling

col ... color

Draw fill area. The function fills a solid color continuous area with the specified color. The set clipping borders are applied during filling.

Draw character

void Draw?SelFont(const u8* font, u8 fontw, u8 fonth);

font ... pointer to the font (256 characters in cells of width 8 pixels, 1-bit format)

fontw ... font width (number of pixels, max. 8)

fonth ... font height (number of lines)

Select current font.

void Draw?SelFont8x8();

Select sans-serif bold font, height 8 lines.

void Draw?SelFont8x14();

Select sans-serif bold font, height 14 lines.

void Draw?SelFont8x16();

Select sans-serif bold font, height 16 lines.

void Draw?SelFont8x14Serif();

Select serif bold font, height 14 lines.

void Draw?SelFont8x16Serif();

Select serif bold font, height 16 lines.

void Draw?SelFont6x8();

Select condensed font, width 6 pixels, height 8 lines.

void Draw?SelFont8x8Game();

Select game font, height 8 lines.

void Draw?SelFont8x8Ibm();

Select IBM charset font, height 8 lines.

void Draw?SelFont8x14Ibm();

Select IBM charset font, height 14 lines.

void Draw?SelFont8x16Ibm();

Select IBM charset font, height 16 lines.

void Draw?SelFont8x8IbmThin();

Select IBM charset thin font, height 8 lines.

void Draw?SelFont8x8Italic();

Select italic font, height 8 lines.

void Draw?SelFont8x8Thin();

Select thin font, height 8 lines.

void Draw?SelFont5x8();

Select tiny font, width 5 pixels, height 8 lines.

void DrawCan?Char_Fast(sDrawCan* can, char ch, int x, int y, u16 col, int scalex, int scaley);
void DrawCan?Char(sDrawCan* can, char ch, int x, int y, u16 col, int scalex, int scaley);
void DrawCan?CharInv_Fast(sDrawCan* can, char ch, int x, int y, int scalex, int scaley);
void DrawCan?CharInv(sDrawCan* can, char ch, int x, int y, int scalex, int scaley);
void Draw?Char_Fast(char ch, int x, int y, u16 col, int scalex, int scaley);
void Draw?CharInv_Fast(char ch, int x, int y, int scalex, int scaley);
void Draw?CharInv(char ch, int x, int y, int scalex, int scaley);
void Draw?Char(char ch, int x, int y, u16 col, int scalex, int scaley);

ch ... character to draw

x ... X coordinate

y ... Y coordinate

col ... color

scalex ... scale size in X direction (> 0)

scaley ... scale size in Y direction (> 0)

Draw character with transparent background.

void DrawCan?CharBg_Fast(sDrawCan* can, char ch, int x, int y, u16 col, u16 bgcol, int scalex, int scaley);
void DrawCan?CharBg(sDrawCan* can, char ch, int x, int y, u16 col, u16 bgcol, int scalex, int scaley);
void Draw?CharBg_Fast(char ch, int x, int y, u16 col, u16 bgcol, int scalex, int scaley);
void Draw?CharBg(char ch, int x, int y, u16 col, u16 bgcol, int scalex, int scaley);

ch ... character to draw

x ... X coordinate

y ... Y coordinate

col ... color

bgcol ... background color

scalex ... scale size in X direction (> 0)

scaley ... scale size in Y direction (> 0)

Draw character with background.

Draw text

void DrawCan?Text_Fast(sDrawCan* can, const char* text, int len, int x, int y, u16 col, int scalex, int scaley);
void DrawCan?Text(sDrawCan* can, const char* text, int len, int x, int y, u16 col, int scalex, int scaley);
void DrawCan?TextInv_Fast(sDrawCan* can, const char* text, int len, int x, int y, int scalex, int scaley);
void DrawCan?TextInv(sDrawCan* can, const char* text, int len, int x, int y, int scalex, int scaley);
void Draw?Text_Fast(const char* text, int len, int x, int y, u16 col, int scalex, int scaley);
void Draw?TextInv_Fast(const char* text, int len, int x, int y, int scalex, int scaley);
void Draw?TextInv(const char* text, int len, int x, int y, int scalex, int scaley);
void Draw?Text(const char* text, int len, int x, int y, u16 col, int scalex, int scaley);

text ... pointer to ASCII text

len ... text length in number of characters, or -1 to detect ASCIIZ text length

x ... X coordinate

y ... Y coordinate

col ... color

scalex ... scale size in X direction (> 0)

scaley ... scale size in Y direction (> 0)

Draw text with transparent background.

void DrawCan?TextBg_Fast(sDrawCan* can, const char* text, int len, int x, int y, u16 col, u16 bgcol, int scalex, int scaley);
void DrawCan?TextBg(sDrawCan* can, const char* text, int len, int x, int y, u16 col, u16 bgcol, int scalex, int scaley);
void Draw?TextBg_Fast(const char* text, int len, int x, int y, u16 col, u16 bgcol, int scalex, int scaley);
void Draw?TextBg(const char* text, int len, int x, int y, u16 col, u16 bgcol, int scalex, int scaley);

text ... pointer to ASCII text

len ... text length in number of characters, or -1 to detect ASCIIZ text length

x ... X coordinate

y ... Y coordinate

col ... color

bgcol ... background color

scalex ... scale size in X direction (> 0)

scaley ... scale size in Y direction (> 0)

Draw text with background.

Draw ellipse

void DrawCan?Ellipse_Fast(sDrawCan* can, int x, int y, int dx, int dy, u16 col, u8 mask);
void DrawCan?Ellipse(sDrawCan* can, int x, int y, int dx, int dy, u16 col, u8 mask);
void DrawCan?EllipseInv_Fast(sDrawCan* can, int x, int y, int dx, int dy, u8 mask);
void DrawCan?EllipseInv(sDrawCan* can, int x, int y, int dx, int dy, u8 mask);
void Draw?Ellipse_Fast(int x, int y, int dx, int dy, u16 col, u8 mask);
void Draw?EllipseInv_Fast(int x, int y, int dx, int dy, u8 mask);
void Draw?EllipseInv(int x, int y, int dx, int dy, u8 mask);
void Draw?Ellipse(int x, int y, int dx, int dy, u16 col, u8 mask);

x ... X coordinate of the centre of the ellipse

y ... Y coordinate of the centre of the ellipse

dx ... diameter of the ellipse in X direction, range 1..430 (X radius = dx/2)

dy ... diameter of ellipse in Y direction, range 1..430 (Y radius = dy/2)

mask ... draw ellipse arcs, use combination of DRAWCAN_ELLIPSE_*;
or use DRAWCAN_ELLIPSE_ALL or 0x0F to draw whole ellipse

Draw ellipse or arc. Note that the parameters 'dx' and 'dy' mean the diameters of the ellipse and not the radius, as is usually the case. This is in order to be able to draw an ellipse with an odd diameter, which would otherwise not be possible when specifying the radius as an integer. Note that the diameters dx and dy cannot be greater than 430.

The 'mask' parameter is used to display parts of the ellipse. It is a value or ORed combination of values:

DRAWCAN_ELLIPSE_ARC0 = 1	draw arc 0..90 deg
DRAWCAN_ELLIPSE_ARC1 = 2	draw arc 90..180 deg
DRAWCAN_ELLIPSE_ARC2 = 4	draw arc 180..270 deg
DRAWCAN_ELLIPSE_ARC3 = 8	draw arc 270..360 deg

Or other alternatives of the values:

DRAWCAN_ELLIPSE_ALL = 0x0F	draw whole ellipse
DRAWCAN_ELLIPSE_TOP	draw top half-ellipse
DRAWCAN_ELLIPSE_BOTTOM	draw bottom half-ellipse
DRAWCAN_ELLIPSE_LEFT	draw left half-ellipse
DRAWCAN_ELLIPSE_RIGHT	draw right half-ellipse

Draw filled ellipse

void DrawCan?FillEllipse_Fast(sDrawCan* can, int x, int y, int dx, int dy, u16 col, u8 mask);
void DrawCan?FillEllipse(sDrawCan* can, int x, int y, int dx, int dy, u16 col, u8 mask);
void DrawCan?FillEllipseInv_Fast(sDrawCan* can, int x, int y, int dx, int dy, u8 mask);
void DrawCan?FillEllipseInv(sDrawCan* can, int x, int y, int dx, int dy, u8 mask);
void Draw?FillEllipse_Fast(int x, int y, int dx, int dy, u16 col, u8 mask);
void Draw?FillEllipseInv_Fast(int x, int y, int dx, int dy, u8 mask);
void Draw?FillEllipseInv(int x, int y, int dx, int dy, u8 mask);
void Draw?FillEllipse(int x, int y, int dx, int dy, u16 col, u8 mask);

x ... X coordinate of the centre of the ellipse

y ... Y coordinate of the centre of the ellipse

dx ... diameter of the ellipse in X direction, range 1..430 (X radius = dx/2)

dy ... diameter of ellipse in Y direction, range 1..430 (Y radius = dy/2)

mask ... hide parts of the ellipse with DRAWCAN_ROUND_* (or their combination);
or use DRAWCAN_ROUND_ALL or 0 to draw whole ellipse

Draw filled ellipse. . Note that the parameters 'dx' and 'dy' mean the diameters of the ellipse and not the radius, as is usually the case. This is in order to be able to draw an ellipse with an odd diameter, which would otherwise not be possible when specifying the radius as an integer. Note that the diameters dx and dy cannot be greater than 430.

The 'mask' parameter is used to hide parts of the ellipse. It is a value or ORed combination of values:

DRAWCAN_ROUND_NOTOP = 1	hide top part of the ellipse
DRAWCAN_ROUND_NOBOTTOM = 2	hide bottom part of the ellipse
DRAWCAN_ROUND_NOLEFT = 4	hide left part of the ellipse
DRAWCAN_ROUND_NORIGHT = 8	hide right part of the ellipse

Or other alternatives of names of the values:

DRAWCAN_ROUND_ALL = 0	draw whole ellipse
DRAWCAN_ROUND_TOP = DRAWCAN_ROUND_NOBOTTOM	... draw top half-ellipse
DRAWCAN_ROUND_BOTTOM = DRAWCAN_ROUND_NOTOP	... draw bottom half-ellipse
DRAWCAN_ROUND_LEFT = DRAWCAN_ROUND_NORIGHT	... draw left half-ellipse
DRAWCAN_ROUND_RIGHT = DRAWCAN_ROUND_NOLEFT	... draw right half-ellipse

Draw image

void DrawCan?Img(sDrawCan* can, int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs);
void DrawCan?ImgInv(sDrawCan* can, int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs);
void Draw?ImgInv(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs);
void Draw?Img(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs);

xd ... destination X coordinate

yd ... destination Y coordinate

src ... pointer to data of the source image

xs ... source X coordinate

ys ... source Y coordinate

w ... width to draw

h ... height to draw

wbs ... pitch of the source image (length of line in bytes)

Draw image with the same format as destination.

void DrawCan?Blit(sDrawCan* can, int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs, u16 col);

void DrawCan?BlitInv(sDrawCan* can, int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs, u16 col);

void Draw?BlitInv(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs, u16 col);

void Draw?Blit(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs, u16 col);

xd ... destination X coordinate

yd ... destination Y coordinate

src ... pointer to data of the source image

xs ... source X coordinate

ys ... source Y coordinate

w ... width to draw

h ... height to draw

wbs ... pitch of the source image (length of line in bytes)

col ... key transparent color

Draw transparent image with the same format as destination

void DrawCan?BlitSubs(sDrawCan* can, int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs, u16 col, u16 fnd, u16 subs);

void Draw?BlitSubs(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs, u16 col, u16 fnd, u16 subs);

xd ... destination X coordinate

yd ... destination Y coordinate

src ... pointer to data of the source image

xs ... source X coordinate

ys ... source Y coordinate

w ... width to draw

h ... height to draw

wbs ... pitch of the source image (length of line in bytes)

col ... key transparent color

fnd ... color to find for

subs ... color to substitute with

Draw transparent image with the same format as destination, with substitute color.

void DrawCan?GetImg(const sDrawCan* can, int xs, int ys, int w, int h, void* dst, int xd, int yd, int wbd);

void Draw?GetImg(int xs, int ys, int w, int h, void* dst, int xd, int yd, int wbd);

can ... source canvas

xs ... source X coordinate

ys ... source Y coordinate

w ... width

h ... height

dst ... destination buffer

xd ... destination X coordinate

yd ... destination Y coordinate

wbd ... pitch of destination buffer (length of line in bytes)

Get image from canvas to buffer.

Draw image specific only to 15-bit canvas

void DrawCan15Img12(sDrawCan* can, int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs);

void DrawCan15Img8(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan15Img6(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan15Img4(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan15Img3(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan15Img2(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan15Img1(sDrawCan* can, int xd, int yd, const void* src, u16 col, u16 bgcol, int xs, int ys, int w, int h, int wbs);

void Draw15Img12(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs);

void Draw15Img8(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw15Img6(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw15Img4(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw15Img3(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw15Img2(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw15Img1(int xd, int yd, const void* src, u16 bgcol, u16 col, int xs, int ys, int w, int h, int wbs);

can ... destination canvas

xd ... destination X coordinate

yd ... destination Y coordinate

src ... source image

pal ... pointer to palettes of the image

xs ... source X coordinate

ys ... source Y coordinate

w ... width to draw

h ... height to draw

wbs ... pitch of source image (length of line in bytes)

bgcol ... color of pixel with value '0' (1-bit image)

col ... color of pixel with value '1' (1-bit image)

Draw 12/8/6/4/3/2/1-bit image to 15-bit destination canvas. For 8-bit to 2-bit images, image palettes are specified. For a 1-bit image, 2 colors are specified - the foreground and background color.

void DrawCan15Blit12(sDrawCan* can, int xd, int yd, const void* src, u16 col, int xs, int ys, int w, int h, int wbs);

void DrawCan15Blit8(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan15Blit6(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan15Blit4(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan15Blit3(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan15Blit2(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan15Blit1(sDrawCan* can, int xd, int yd, const void* src, u16 col, int xs, int ys, int w, int h, int wbs);

void Draw15Blit12(int xd, int yd, const void* src, u16 col, int xs, int ys, int w, int h, int wbs);

void Draw15Blit8(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw15Blit6(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw15Blit4(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw15Blit3(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw15Blit2(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw15Blit1(int xd, int yd, const void* src, u16 col, int xs, int ys, int w, int h, int wbs);

can ... destination canvas

xd ... destination X coordinate

yd ... destination Y coordinate

src ... source image

pal ... pointer to palettes of the image

xs ... source X coordinate

ys ... source Y coordinate

w ... width to draw

h ... height to draw

wbs ... pitch of source image (length of line in bytes)

col ... key transparent color (12 to 2-bit image) or color of pixel '1' (1-bit image)

Draw 12/8/6/4/3/2/1-bit image transparent to 15-bit destination canvas. For 8-bit to 2-bit images, image palettes and key transparent color are specified. For a 1-bit image, the color is specified for a pixel with a value of '1'; pixels with a value of '0' are not drawn.

Draw image specific only to 16-bit canvas

void DrawCan16Img12(sDrawCan* can, int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs);

void DrawCan16Img8(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan16Img6(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan16Img4(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan16Img3(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan16Img2(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void DrawCan16Img1(sDrawCan* can, int xd, int yd, const void* src, u16 col, u16 bgcol, int xs, int ys, int w, int h, int wbs);

void Draw16Img12(int xd, int yd, const void* src, int xs, int ys, int w, int h, int wbs);

void Draw16Img8(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw16Img6(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw16Img4(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw16Img3(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw16Img2(int xd, int yd, const void* src, const u16* pal, int xs, int ys, int w, int h, int wbs);

void Draw16Img1(int xd, int yd, const void* src, u16 bgcol, u16 col, int xs, int ys, int w, int h, int wbs);

can ... destination canvas

xd ... destination X coordinate

yd ... destination Y coordinate

src ... source image

pal ... pointer to palettes of the image

xs ... source X coordinate

ys ... source Y coordinate

w ... width to draw

h ... height to draw

wbs ... pitch of source image (length of line in bytes)

bgcol ... color of pixel with value '0' (1-bit image)

col ... color of pixel with value '1' (1-bit image)

Draw 12/8/6/4/3/2/1-bit image to 16-bit destination canvas. For 8-bit to 2-bit images, image palettes are specified. For a 1-bit image, 2 colors are specified - the foreground and background color.

void DrawCan16Blit12(sDrawCan* can, int xd, int yd, const void* src, u16 col, int xs, int ys, int w, int h, int wbs);

void DrawCan16Blit8(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan16Blit6(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan16Blit4(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan16Blit3(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan16Blit2(sDrawCan* can, int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void DrawCan16Blit1(sDrawCan* can, int xd, int yd, const void* src, u16 col, int xs, int ys, int w, int h, int wbs);

void Draw16Blit12(int xd, int yd, const void* src, u16 col, int xs, int ys, int w, int h, int wbs);

void Draw16Blit8(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw16Blit6(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw16Blit4(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw16Blit3(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw16Blit2(int xd, int yd, const void* src, const u16* pal, u8 col, int xs, int ys, int w, int h, int wbs);

void Draw16Blit1(int xd, int yd, const void* src, u16 col, int xs, int ys, int w, int h, int wbs);

can ... destination canvas

xd ... destination X coordinate

yd ... destination Y coordinate

src ... source image

pal ... pointer to palettes of the image

xs ... source X coordinate

ys ... source Y coordinate

w ... width to draw

h ... height to draw

wbs ... pitch of source image (length of line in bytes)

col ... key transparent color (12 to 2-bit image) or color of pixel '1' (1-bit image)

Draw 12/8/6/4/3/2/1-bit image transparent to 16-bit destination canvas. For 8-bit to 2-bit images, image palettes and key transparent color are specified. For a 1-bit image, the color is specified for a pixel with a value of '1'; pixels with a value of '0' are not drawn.

14. Configuration

Configuration of the DispHSTX driver is stored in the _config.h file. The file contains default parameter settings to be used if the parameter is not defined in the project configuration file. To change a parameter, set its value in the project configuration file config.h, which is loaded before the compiler loads the default driver file _config.h. For example, if you want to use the switch to select the display on GPIO pin 20, add this line in config.h: #define DISPHSTX_DISP_SEL 20.

The following is a detailed list of parameters, along with their default value.

#define **DISPHSTX_PICOSDK** 0

The DispHSTX driver is primarily designed for the PicoLibSDK library. Setting this switch to 1 will modify the code to use the functions of the original Raspberry SDK library PicoSDK.

#define **USE_DISPHSTX** 1

This compilation switch should always be set to 1 if you want to use the DispHSTX driver.

#define **USE_DRAWCAN** 1

Set this compilation switch to 1, if you want to use drawing library DrawCan.

#define **DISPHSTX_USE_DVI** 1

This parameter enables DVI (HDMI) support. If you are using only the VGA output and not the DVI (HDMI) output, set this switch to 0. This will reduce the RAM requirements by 15 KB.

#define **DISPHSTX_USE_VGA** 1

This parameter enables VGA support. If you are using only the DVI (HDMI) output and not the VGA output, set this switch to 0. This will reduce the RAM requirements by 30 KB.

#define **DISPHSTX_DVI_PINOUT** 0

This parameter selects predefined pinout layout of the DVI (HDMI) output. Value 0 selects pinout of the DVI breakout board, value 1 selects pinout of order D2+...CLK-, value 2 selects pinout of order CLK-..D2+. You can also set the pinouts for the DVI (HDMI) output in the following parameters separately on a pin-by-pin basis. In this case, the setting of this parameter does not matter.

#define **DISPHSTX_ARM_ASM** 1

This parameter specifies whether to use C code functions (parameter value 0) or optimized assembly language functions (parameter value 1) to render the image in ARMv8 processor mode. The parameter should always remain at a value of 1. The C code functions are slower and only serve as a reference function during driver development.

#define **DISPHSTX_RISCV_ASM** 1

This parameter specifies whether to use C code functions (parameter value 0) or optimized assembly language functions (parameter value 1) to render the image in RISC-V Hazard3 processor mode. The parameter should always remain at a value of 1. The C code functions are slower and only serve as a reference function during driver development.

#define **DISPHSTX_USEPLL** 0

In the normal parameter setting (value 0), the CPU system clock is set to 5 or 10 times the pixel clock value of the generated image signal. In the special case, setting the parameter to value 1 allows you to select an operation mode where the timing of the system clock and all peripherals is derived from the PLLUSB generator, and the PLLSYS generator is reserved only for timing the HSTX peripherals. This allows system timing independent of the generated image. The system clock can be set by the DispHstxClockReinit() function. Its value is preferably to be an integer multiple of the 48 MHz USB clock and should be sufficient for the processor to keep up with rendering the image signal.

#define **DISPHSTX_CHECK_LOAD** 0

This parameter is for debugging purposes only. Setting it to 1 will measure the processor load - see the "Processor load" chapter for more details.

#define **DISPHSTX_CHECK_LEDIRQ** -1

Setting the parameter to the GPIO number with the LED connected activates an auxiliary debug indication to test if an interrupt is in progress to service the video output. The LED flashes at approximately 1-second intervals. The internal LED on the Pico board is connected to GPIO25 (cannot be used if the board contains a WiFi module). A default value of -1 will ensure that the function is disabled.

#define **DISPHSTX_DISP_SEL** -1

The parameter contains the GPIO pin number to which the switch is connected to select the display mode VGA or DVI (HDMI). The switch has 3 positions - connecting to GND selects the VGA monitor, connecting to +3V3 selects the DVI (HDMI) monitor. Leaving the pin free without connection will use automatic detection of the connected VGA monitor. Setting the

parameter to -1 means that the switch is not used - the display selection is done by program or by autodetection.

#define **DISPHSTX_DMA_DATA** 15

#define **DISPHSTX_DMA_CMD** 14

The parameters select the numbers of the two DMA channels that the driver requires for its operation. The DMA channel number can be in the range of 0 to 15. The selected DMA channels must not be used for other purposes during driver operation. The driver uses the IRQ_DMA_1 interrupt along with the DMA channels.

#define **DISPHSTX_WIDTHMAX** 1440

This parameter specifies the maximum horizontal resolution of the video mode used. There are preset modes in the driver up to a horizontal resolution of 1440 pixels. The parameter is mainly used to limit the size of the render buffer of the video line. At the default value of 1440, the video line render buffer occupies over 11 KB of RAM. By reducing the parameter according to the resolution used, the memory requirements can be reduced.

#define **DISPHSTX_STRIP_MAX** 4

This parameter specifies the maximum number of strips possible when splitting the screen (horizontal bands). An excessively high number may unnecessarily increase RAM requirements.

#define **DISPHSTX_SLOT_MAX** 4

This parameter specifies the maximum number of slots (image segments) possible in the strip when splitting the screen. An excessively high number may unnecessarily increase RAM requirements.

#define **DISPHSTX_DVI_D2P** 12

Selects the GPIO pin for the D2+ DVI output signal. The pins GPIO12 to GPIO19 can be selected. If the parameter is not specified in the config.h project file, the default pin setting is used, according to the configuration given by the DISPHSTX_DVI_PINOUT switch.

#define **DISPHSTX_DVI_D2M** 13

Selects the GPIO pin for the D2- DVI output signal. The pins GPIO12 to GPIO19 can be selected. If the parameter is not specified in the config.h project file, the default pin setting is used, according to the configuration given by the DISPHSTX_DVI_PINOUT switch.

#define **DISPHSTX_DVI_D1P** 14

Selects the GPIO pin for the D1+ DVI output signal. The pins GPIO12 to GPIO19 can be selected. If the parameter is not specified in the config.h project file, the default pin setting is used, according to the configuration given by the DISPHSTX_DVI_PINOUT switch.

#define **DISPHSTX_DVI_D1M** 15

Selects the GPIO pin for the D1- DVI output signal. The pins GPIO12 to GPIO19 can be selected. If the parameter is not specified in the config.h project file, the default pin setting is used, according to the configuration given by the DISPHSTX_DVI_PINOUT switch.

#define **DISPHSTX_DVI_D0P** 16

Selects the GPIO pin for the D0+ DVI output signal. The pins GPIO12 to GPIO19 can be selected. If the parameter is not specified in the config.h project file, the default pin setting is used, according to the configuration given by the DISPHSTX_DVI_PINOUT switch.

#define **DISPHSTX_DVI_D0M** 17

Selects the GPIO pin for the D0- DVI output signal. The pins GPIO12 to GPIO19 can be selected. If the parameter is not specified in the config.h project file, the default pin setting is used, according to the configuration given by the DISPHSTX_DVI_PINOUT switch.

#define **DISPHSTX_DVI_CLKP** 18

Selects the GPIO pin for the CLK+ DVI output signal. The pins GPIO12 to GPIO19 can be selected. If the parameter is not specified in the config.h project file, the default pin setting is used, according to the configuration given by the DISPHSTX_DVI_PINOUT switch.

#define **DISPHSTX_DVI_CLKM** 19

Selects the GPIO pin for the CLK- DVI output signal. The pins GPIO12 to GPIO19 can be selected. If the parameter is not specified in the config.h project file, the default pin setting is used, according to the configuration given by the DISPHSTX_DVI_PINOUT switch.

#define **DISPHSTX_VGA_B0** 12

Selects the GPIO pin for the B0 VGA output signal. The pins GPIO12 to GPIO19 can be selected. It is recommended to keep the default pin assignment of B0-B1-G0-G1-R0-R1-HSYNC-VSYNC, for the possibility of outputting from the PIO peripheral.

#define **DISPHSTX_VGA_B1** 13

Selects the GPIO pin for the B1 VGA output signal. The pins GPIO12 to GPIO19 can be selected. It is recommended to keep the default pin assignment of B0-B1-G0-G1-R0-R1-HSYNC-VSYNC, for the possibility of outputting from the PIO peripheral.

#define **DISPHSTX_VGA_G0** 14

Selects the GPIO pin for the G0 VGA output signal. The pins GPIO12 to GPIO19 can be selected. It is recommended to keep the default pin assignment of B0-B1-G0-G1-R0-R1-HSYNC-VSYNC, for the possibility of outputting from the PIO peripheral.

#define **DISPHSTX_VGA_G1** 15

Selects the GPIO pin for the G1 VGA output signal. The pins GPIO12 to GPIO19 can be selected. It is recommended to keep the default pin assignment of B0-B1-G0-G1-R0-R1-HSYNC-VSYNC, for the possibility of outputting from the PIO peripheral.

#define **DISPHSTX_VGA_R0** 16

Selects the GPIO pin for the R0 VGA output signal. The pins GPIO12 to GPIO19 can be selected. It is recommended to keep the default pin assignment of B0-B1-G0-G1-R0-R1-HSYNC-VSYNC, for the possibility of outputting from the PIO peripheral.

#define **DISPHSTX_VGA_R1** 17

Selects the GPIO pin for the R1 VGA output signal. The pins GPIO12 to GPIO19 can be selected. It is recommended to keep the default pin assignment of B0-B1-G0-G1-R0-R1-HSYNC-VSYNC, for the possibility of outputting from the PIO peripheral.

#define **DISPHSTX_VGA_HSYNC** 18

Selects the GPIO pin for the HSYNC VGA output signal. The pins GPIO12 to GPIO19 can be selected. It is recommended to keep the default pin assignment of B0-B1-G0-G1-R0-R1-HSYNC-VSYNC, for the possibility of outputting from the PIO peripheral.

#define **DISPHSTX_VGA_VSYNC** 19

Selects the GPIO pin for the VSYNC VGA output signal. The pins GPIO12 to GPIO19 can be selected. It is recommended to keep the default pin assignment of B0-B1-G0-G1-R0-R1-HSYNC-VSYNC, for the possibility of outputting from the PIO peripheral.

#define **DISPHSTX_USE_FORMAT_1** 1
#define **DISPHSTX_USE_FORMAT_2** 1
#define **DISPHSTX_USE_FORMAT_3** 1
#define **DISPHSTX_USE_FORMAT_4** 1
#define **DISPHSTX_USE_FORMAT_6** 1
#define **DISPHSTX_USE_FORMAT_8** 1
#define **DISPHSTX_USE_FORMAT_12** 1
#define **DISPHSTX_USE_FORMAT_15** 1
#define **DISPHSTX_USE_FORMAT_16** 1
#define **DISPHSTX_USE_FORMAT_1_PAL** 1
#define **DISPHSTX_USE_FORMAT_2_PAL** 1
#define **DISPHSTX_USE_FORMAT_3_PAL** 1
#define **DISPHSTX_USE_FORMAT_4_PAL** 1
#define **DISPHSTX_USE_FORMAT_6_PAL** 1
#define **DISPHSTX_USE_FORMAT_8_PAL** 1
#define **DISPHSTX_USE_FORMAT_COL** 1
#define **DISPHSTX_USE_FORMAT_MTEXT** 1
#define **DISPHSTX_USE_FORMAT_ATEXT** 1
#define **DISPHSTX_USE_FORMAT_TILE4_8** 1
#define **DISPHSTX_USE_FORMAT_TILE8_8** 1
#define **DISPHSTX_USE_FORMAT_TILE16_8** 1
#define **DISPHSTX_USE_FORMAT_TILE32_8** 1
#define **DISPHSTX_USE_FORMAT_TILE4_8_PAL** 1
#define **DISPHSTX_USE_FORMAT_TILE8_8_PAL** 1
#define **DISPHSTX_USE_FORMAT_TILE16_8_PAL** 1
#define **DISPHSTX_USE_FORMAT_TILE32_8_PAL** 1
#define **DISPHSTX_USE_FORMAT_HSTX_15** 1
#define **DISPHSTX_USE_FORMAT_HSTX_16** 1
#define **DISPHSTX_USE_FORMAT_PAT_8** 1
#define **DISPHSTX_USE_FORMAT_PAT_8_PAL** 1
#define **DISPHSTX_USE_FORMAT_RLE8** 1
#define **DISPHSTX_USE_FORMAT_RLE8_PAL** 1

#define **DISPHSTX_USE_FORMAT_ATTR1_PAL** 1
#define **DISPHSTX_USE_FORMAT_ATTR2_PAL** 1
#define **DISPHSTX_USE_FORMAT_ATTR3_PAL** 1
#define **DISPHSTX_USE_FORMAT_ATTR4_PAL** 1
#define **DISPHSTX_USE_FORMAT_ATTR5_PAL** 1
#define **DISPHSTX_USE_FORMAT_ATTR6_PAL** 1
#define **DISPHSTX_USE_FORMAT_ATTR7_PAL** 1
#define **DISPHSTX_USE_FORMAT_ATTR8_PAL** 1

The switches enable the use of the corresponding image formats. If the format is not used by the program, setting the parameter to 0 can disable the format and thus save some RAM.

15. Processor load

The DISPHSTX_CHECK_LOAD configuration switch allows you to enable CPU load measurement during image rendering. The function measures the time spent in the rendering function and the time spent out of the interrupt handler. The following code can be used to evaluate the measurements and it is called repeatedly in the main loop of the program (with a period of e.g. 1 second). The text is printed out on the USB console (the USE_USB_STDIO switch must be enabled).

The measurement is only illustrative - the concurrence of variable modification with the interrupt is not handled and the time required to service the interrupt is not measured. In practice, it is found that rendering the image is safe up to a measured CPU load of 85%. For higher CPU loads, image synchronization may already drop out. In this case, either a fast CPU mode (timing "_fast") or pixel duplication (hdbl > 1) must be used.

The following table shows the measured CPU load values for all graphics modes, for both ARMv8 and RISC-V Hazard3 processors, for both VGA and DVI (HDMI) display modes. The measurements were performed at 640x480 video mode. In the case of image synchronization dropout (CPU load above 85%), the fast CPU mode "_fast" was used (instead of the system clock of 126 MHz, 252 MHz was used) and the measured load data was shown in the table 2x higher. Thus, the load value from the table e.g. 140% means the measured CPU load is 70% in fast mode.

```
#if DISPHSTX_CHECK_LOAD
      int n = DispHstxTimeNum;
      double in = (double)DispHstxTimeIn/n
                  *1000000/ClockGetHz(CLK_SYS);
      double out = (double)DispHstxTimeOut/n
                  *1000000/ClockGetHz(CLK_SYS);
      printf("in=%.1fus out=%.1fus all=%.1fus CPU load=%.2f%%\n",
            (double)DispHstxTimeInMax*1000000/ClockGetHz(CLK_SYS),
            (double)DispHstxTimeOutMin*1000000/ClockGetHz(CLK_SYS),
            in+out, in/(in+out)*100);
      DispHstxTimeNum = 0;
      DispHstxTimeIn = 0;
      DispHstxTimeOut = 0;
      DispHstxTimeInMax = 0;
      DispHstxTimeOutMin = 10000000;
#endif
```

126 MHz CPU loads (640x480@60Hz)	DVI ARM	VGA ARM	DVI RISC-V	VGA RISC-V
DISPHSTX_FORMAT_1	2%	55%	2%	67%
DISPHSTX_FORMAT_2	2%	57%	2%	69%
DISPHSTX_FORMAT_3	2%	61%	2%	71%
DISPHSTX_FORMAT_4	2%	61%	2%	73%
DISPHSTX_FORMAT_6	2%	59%	2%	71%
DISPHSTX_FORMAT_8	2%	60%	2%	73%
DISPHSTX_FORMAT_12	47%	112%	50%	142%
DISPHSTX_FORMAT_15	2%	110%	2%	152%
DISPHSTX_FORMAT_16	2%	110%	2%	141%
DISPHSTX_FORMAT_1_PAL	28%	55%	34%	67%
DISPHSTX_FORMAT_2_PAL	56%	57%	71%	69%
DISPHSTX_FORMAT_3_PAL	55%	61%	73%	71%
DISPHSTX_FORMAT_4_PAL	61%	61%	76%	73%
DISPHSTX_FORMAT_6_PAL	58%	59%	75%	71%
DISPHSTX_FORMAT_8_PAL	58%	60%	74%	73%
DISPHSTX_FORMAT_COL	3%	3%	3%	3%
DISPHSTX_FORMAT_MTEXT	41%	66%	48%	81%
DISPHSTX_FORMAT_ATEXT	60%	74%	74%	95%
DISPHSTX_FORMAT_TILE4_8	28%	82%	26%	90%
DISPHSTX_FORMAT_TILE8_8	20%	73%	20%	84%
DISPHSTX_FORMAT_TILE16_8	15%	68%	16%	80%
DISPHSTX_FORMAT_TILE32_8	13%	64%	13%	79%
DISPHSTX_FORMAT_TILE4_8_PAL	79%	82%	90%	90%
DISPHSTX_FORMAT_TILE8_8_PAL	70%	73%	82%	84%
DISPHSTX_FORMAT_TILE16_8_PAL	67%	68%	80%	80%
DISPHSTX_FORMAT_TILE32_8_PAL	63%	64%	78%	79%
DISPHSTX_FORMAT_HSTX_15	3%	128%	3%	152%
DISPHSTX_FORMAT_HSTX_16	3%	128%	3%	144%
DISPHSTX_FORMAT_PAT_8	19%	64%	17%	79%
DISPHSTX_FORMAT_PAT_8_PAL	63%	64%	81%	79%
DISPHSTX_FORMAT_RLE8	55%	83%	71%	90%
DISPHSTX_FORMAT_RLE8_PAL	86%	83%	92%	90%
DISPHSTX_FORMAT_ATTR1_PAL	76%	78%	108%	105%
DISPHSTX_FORMAT_ATTR2_PAL	87%	89%	120%	116%
DISPHSTX_FORMAT_ATTR3_PAL	87%	88%	120%	115%
DISPHSTX_FORMAT_ATTR4_PAL	91%	91%	135%	130%
DISPHSTX_FORMAT_ATTR5_PAL	74%	74%	105%	102%
DISPHSTX_FORMAT_ATTR6_PAL	77%	77%	110%	106%
DISPHSTX_FORMAT_ATTR7_PAL	75%	79%	108%	104%
DISPHSTX_FORMAT_ATTR8_PAL	76%	77%	110%	105%

Formats with load > 85% requires "_fast" mode, or hdbl > 1, or some borders.

16. Select display mode

The DispHSTX driver supports output to both VGA monitor and DVI (HDMI) monitor, via the 8 output pins of the HSTX peripheral. If either output is not required, it can be disabled with the DISPHSTX_USE_DVI and DISPHSTX_USE_VGA configuration switches to save some RAM.

The following constants can be used when selecting an output:

DISPHSTX_DISPMODE_DVI ... output to a DVI (HDMI) display
DISPHSTX_DISPMODE_VGA ... output to VGA display
DISPHSTX_DISPMODE_NONE ... automatic display selection

A configuration jumper, set by the user, can be used to select the output display. By setting the DISPHSTX_DISP_SEL configuration parameter to the GPIO pin number of the configuration jumper, the jumper settings can be read by the program and the desired display mode can be selected accordingly.

If the configuration jumper is in the lower position (GND), the VGA monitor is used. If the configuration jumper is in the upper position (+3V3), a DVI (HDMI) monitor is used. If it is in the middle position (unconnected input), or if the configuration jumper is not used (DISPHSTX_DISP_SEL parameter is set to -1), the VGA monitor connection is automatically detected and the VGA or DVI mode is selected accordingly.

The automatic detection of the VGA monitor connection can only be performed at program start or before activating the video mode, as it requires modification of the GPIO pins. The detection takes advantage of the fact that the DVI (HDMI) monitor input is connected via decoupling capacitors, while the VGA monitor has terminating resistors on the input. During the test, the pull-up resistors are activated - the VGA monitor pulls down the input of the pins to 0, while the DVI (HDMI) monitor leaves the inputs at 1.

The display mode can also be switched while the program is running if the user switches the configuration jumper. For this purpose, the DispHstxAutoDispMode() function can be called repeatedly in the program, which reads the configuration switch. If it detects a change in the switcher setting, it terminates the currently running video mode and restarts the driver with the same settings but for a different selected display mode.

17. VGA Pulse Pattern Modulation PPM

The output to the VGA display is via the same pins GPIO12 to GPIO19 as the output to the DVI (HDMI) display. The 6 output pins are used to output color signals. There are 2 output bits for each of the 3 RGB color components. A resistor network of two resistors, 800 and 400 ohms, is used to control the output voltage.

With the above configuration, only 4 output voltage levels could normally be achieved. Nevertheless, it is possible to display a higher resolution color image on a VGA display by using Pulse Pattern Modulation PPM to generate the VGA image.

Calculation of the output signal. Input resistance of the VGA signal is 75 ohms. Output voltage of the GPIO pin is 0V or 3.2V. Input voltage of the VGA signal should be in range 0 ro 0.7V. The following table shows the calculated relationship between the bit settings of one channel and the output voltage.

R2	R1	U [V]	relative
0	0	0,000	**0,000**
0	1	0,234	**0,334**
1	0	0,468	**0,669**
1	1	0,702	**1,003**

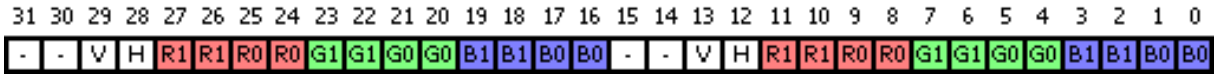
In this way, a 2-bit output can be generated.

The color depth can be increased by using the HSTX shift register functionality. What this does is that on the rising edge of the HSTX clock it outputs the state of one bit, and on the falling edge the state of the other bit. Thus, the states of the output bits alternate at the output at high speed. So we use 4 bits in the output word. As an example, we will give here the output to the blue component. Bits 0 and 1 of the output word are switched at output B0 (lower weight output, 800 ohm resistor). Bits 2 and 3 of the output word are switched at output B1 (higher weight output, 400 ohm resistor). We will switch the state between 2 adjacent voltage values. In this way we extend the 4 states of the output voltage to 7 states, which is already almost a 3-bit output.

	B1-1	B1-0	B0-1	B0-0	code	relative	delta	error	
0	0	0	0	0	0x00	**0,000**			
1	0	0	0	1	0x01	**0,167**	0,167	0,0%	also 0x02
2	0	0	1	1	0x03	**0,334**	0,167	0,0%	
3	1	0	0	1	0x09	**0,502**	0,167	0,0%	also 0x06
4	1	1	0	0	0x0C	**0,669**	0,167	0,0%	
5	1	1	0	1	0x0D	**0,836**	0,167	0,0%	also 0x0E
6	1	1	1	1	0x0F	**1,003**	0,167	0,0%	
						average:	0,167	0,0%	

As can be seen from the table, we achieve 7 voltage levels with uniform distribution (relative interval 0.167).The 4-bit codes for each level are 0x00, 0x01/0x02, 0x03, 0x09/0x06, 0x0C, 0x0D/0x0E and 0x0F.

To further refine the color depth, we take advantage of the other functionality of the HSTX peripheral. The HSTX shift register works by outputting one set of bits on the rising edge of the clock, a second set of bits on the falling edge, and then rotating the output word by the specified number of bits. The word rotates cyclically in the register, the bits from the output go back to the input. We spread the bits in the output word as follows.



The output word for the HSTX shift register is a 32-bit word. We split the word into two 16-bit half-words with similar content. In each half-word, we store the output bits for the blue channel in bits 0 to 3, the bits for the green channel in bits 4 to 7, the bits for the red channel in bits 8 to 11, the bit for the HSYNC signal in bit 12, and the bit for the VSYNC signal in bit 13. Bits 14 and 15 remain unused in each half-word.

We could now cyclically switch between 2 close states of the output signals and extend the bit depth to 13 voltage levels. Instead, we will use similar functionality as for the DVI signal. In DVI mode, the output TMDS words are 10 bits. In the shift register, 2 bits are sent to the output for the odd and even edges, then a 2 bit shift of the output register is made. This makes 5 shifts, which at 2 bits per clock will ensure all 10 bits of the TMDS word are sent. Hence, the system clock must be 5 times higher than the pixel output frequency.

For VGA mode, we also use 5 output register shifts, which ensures similar timing conditions for VGA and DVI mode. We send 2 bits in one clock cycle of the HSTX clock, then perform a shift of 16 bits for a total of 5 times. This will mean that the lower word is sent to the output 3 times and the higher word is sent 2 times. This will give us a higher fineness of output levels. The differences between the levels are not uniform, but that doesn't bother us yet.

	codeH (2x)	codeL (3x)	relative	delta	error
0	0x00	0x00	0,000		
	0x01	0x00	0,067		
1	0x00	0x01	0,100	0,100	50,0%
2	0x01	0x01	0,167	0,067	0,0%
3	0x03	0x01	0,234	0,067	0,0%
4	0x01	0x03	0,268	0,033	50,0%
5	0x03	0x03	0,334	0,067	0,0%
6	0x09	0x03	0,401	0,067	0,0%
7	0x03	0x09	0,435	0,033	50,0%
8	0x09	0x09	0,502	0,067	0,0%
9	0x0C	0x09	0,569	0,067	0,0%
10	0x09	0x0C	0,602	0,033	50,0%
11	0x0C	0x0C	0,669	0,067	0,0%
12	0x0D	0x0C	0,736	0,067	0,0%
13	0x0C	0x0D	0,769	0,033	50,0%
14	0x0D	0x0D	0,836	0,067	0,0%
	0x0F	0x0D	0,903		
	0x0D	0x0F	0,937		
15	0x0F	0x0F	1,003	0,167	150,0%
			average:	0,067	26,7%

The table above contains the values we use to fill the blue channel at the top and bottom of the word. The top half-word is repeated 2 times for each pixel, the bottom half-word is repeated 3 times. The 3 combinations that we omit from the total of 19 items to reach 16 levels are marked in grey. In purple are marked "clean" codes from the original table of 7 levels, which do not require level switching and can be better filtered in the output. In this way we have increased the color resolution to 4 bits. However, this means that to display the data on the VGA output we need a conversion table in which we prepare 32-bit codes for 16 output levels.

The last "enhancement" to the VGA color display is time dithering. In the odd and even frames of the VGA image we will alternate the color for both the normal 4-bit output and the color with the added correction in the 5th bit. We will thus switch colors between the 2 levels with 5-bit resolution, increasing the color depth to 5 bits. This partially compensates for the uneven distribution of levels in the 4-bit table. We don't alternate colors equally in even and odd frames - such an image would jitter and flicker. We also alternate even and odd in even and odd lines. This eliminates the flickering effect and the image looks quite good. Of course, the display quality is a bit worse than the full 15-bit output or DVI output, but overall this display mode is usable. The partial degradation in output quality is also due to the filter capacitors on the VGA output, which filter out the pulse modulation of the VGA output and cause a slight blurring of the image.

From the above principle of VGA signal generation it follows that for the driver to work we need to have prepared VGA palettes in double amount compared to DVI palettes (different palettes for even and odd frame are used) and in larger size (32 bits for VGA compared to 16 bits for DVI). The VGA buffer for the palettes must therefore be "8*palnum" bytes in size.

18. PicoPadImg2 - Image conversion

The PicoPadImg version 2.0 tool is used to convert images in BMP format to a format suitable for the DispHSTX driver. The input is Windows BMP format images, in 24-bit, 15-bit (marked as 16-bit format in the editor), 8-bit, 4-bit or 1-bit bit depth. Compression must not be used in the image. When saving, you must either enable the "flip row order" switch or flip the image vertically. The output of the program is source code in C++ format.

The program takes 3 or 4 parameters in the command line. Syntax:

input.bmp output.cpp name format [test.bmp]

'**input.bmp**' input image in BMP format

'**output.cpp**' output file as C++ source

'**name**' name of data array

'**format**' specifies required output format:

1 ... 1-bit Y1 (8 pixels per byte), requires 1-bit input image
2 ... 2-bit Y2 (4 pixels per byte), requires 4-bit input image with 4 palettes
3 ... 3-bit RGB111 (10 pixels per 32-bit), requires 4-bit input image with 8 palettes
4 ... 4-bit YRGB1111 (2 pixels per byte), requires 4-bit input image
6 ... 6-bit RGB222 (5 pixels per 32-bit), requires 8-bit input image
8 ... 8-bit RGB332 (1 pixel per byte), requires 8-bit input image
12 ... 12-bit RGB444 (8 pixels per three 32-bit), requires 15-bit input image
12d ... 12-bit RGB444 with dithering, requires 15-bit input image
15 ... 15-bit RGB555 (1 pixel per 2 bytes), requires 15-bit input image
16 ... 16-bit RGB565 (1 pixel per 2 bytes), requires 24-bit input image
r4 ... 4-bit compression RLE4, requires 4-bit input image
r8 ... 8-bit compression RLE8, requires 8-bit input image
x15 ... 15-bit HSTX compression, requires 15-bit input image
x16 ... 16-bit HSTX compression, requires 24-bit input image
a1 ... attribute compression 1, cell 8x8, 2 colors, requires 4-bit input image
a2 ... attribute compression 2, cell 4x4, 2 colors, requires 4-bit input image
a3 ... attribute compression 3, cell 8x8, 4 colors, requires 4-bit input image
a4 ... attribute compression 4, cell 4x4, 4 colors, requires 4-bit input image
a5 ... attribute compression 5, cell 8x8, 2 colors, requires 8-bit input image
a6 ... attribute compression 6, cell 4x4, 2 colors, requires 8-bit input image
a7 ... attribute compression 7, cell 8x8, 4 colors, requires 8-bit input image
a8 ... attribute compression 8, cell 4x4, 4 colors, requires 8-bit input image

'**test.bmp**' output test image of attribute compression

19. Predefined simple graphic video modes

The following group of functions are used to quickly and easily activate a video mode in the graphic format supported by the DrawCan drawing library. It is sufficient to call the function at the beginning of the main program and then call the Draw* functions.

Each function has 2 parameters. The first parameter is the display mode, which determines whether to output to a VGA monitor or a DVI (HDMI) monitor. Use the constant DISPHSTX_DISPMODE_VGA or DISPHSTX_DISPMODE_DVI. If you specify the DISPHSTX_DISPMODE_NONE or 0 constant, the display mode selection will be performed automatically, either by reading the display selection switch or by detecting the connected monitor.

The second parameter is a pointer to the frame buffer. You can create the buffer by allocating with malloc() function or as a static buffer. The required buffer size is specified with the functions, and is equal to the length of the graphics line, aligned to a 32-bit word, multiplied by the number of lines. You can determine the required graphic line length using the Draw1Pitch() to Draw16Pitch() functions. The address of the buffer must be aligned to a 32-bit word (for a static buffer, use the ALIGNED attribute). If you specify a NULL constant instead of a pointer, the function allocates the required buffer automatically, using the malloc() function. Use the DispHstxAllTerm() function to deactivate the video mode. Note that the frame buffer is not automatically deallocated when the driver function terminates; it must be deallocated manually - function DispHstxFreeBuf() can be used after DispHstxAllTerm().

The predefined video modes are prepared to work in most cases - functionality on VGA and HDMI monitor, ARMv8 and RISC-V Hazard3 core, RAM buffer size up to max 450 KB, CPU overclock up to max 400 MHz, CPU second core load in the range of 50 to 70%.

Using the VGA/HDMI driver requires setting the system clock frequency. The system clock frequency corresponds to a multiple of 5x or 10x the pixel frequency. This usually requires overclocking the processor to a higher frequency than the frequency specified by the manufacturer. Most tested RP2350-A2 processors operate reliably up to overclocking to 300 MHz, some up to 336 MHz. However, proper processor functionality is not guaranteed with overclocking and may reduce processor life. It is therefore necessary to choose overclocking options very carefully.

Functions return error code DISPHSTX_ERR_*, or DISPHSTX_ERR_OK = 0 if all OK.

The recommended most used video modes are:

DispVMode320x240x16 ... resolution 320x240 pixels, 16 bits per pixel, system clock 126 MHz, required buffer size 153600 bytes, aspect ratio 4:3. It uses standard 640x480 pixel video mode and does not require CPU overclocking. Can be used for compatibility with 320x240 LCD frame buffer.

DispVMode532x400x16_Fast ... resolution 532x400 pixels, 16 bits per pixel, system clock 210 MHz, required buffer size 425600 bytes, aspect ratio 4:3. Can be used as the highest resolution for 16-bit color mode.

DispVMode640x480x8 ... resolution 640x480 pixels, 8 bits per pixel, system clock 126 MHz, required buffer size 307200 bytes, aspect ratio 4:3. It uses standard 640x480 pixel video mode and does not require CPU overclocking.

DispVMode720x400x8 ... resolution 720x400 pixels, 8 bits per pixel, system clock 141.6 MHz, required buffer size 288000 bytes, aspect ratio 16:9. Can be used as a wide-screen alternative to 640x480 mode. Does not require CPU overclocking.

DispVMode800x600x6 ... resolution 800x600 pixels, 6 bits per pixel, system clock 200 MHz, required buffer size 384000 bytes, aspect ratio 4:3. It uses another standard video mode, 800x600 pixels, but it requires overclocking to 200 MHz and 6-bit mode can be slower to handle.

DispVMode848x480x8 ... resolution 848x480 pixels, 8 bits per pixel, system clock 166.666 MHz, required buffer size 407040 bytes, aspect ratio 16:9. Can be used as a wide-screen alternative to 800x600 mode, with only a little overclocking.

DispVMode960x720x4 ... resolution 960x720 pixels, 4 bits per pixel, system clock 278.4 MHz, required buffer size 345600 bytes, aspect ratio 4:3. It can be used as a compromise

between 800x600 and 1024x768 modes - it has higher resolution than 800x600 mode and does not require as much overclocking as 1024x768 mode. A VGA monitor will display the image in widescreen because it considers it to be a 1280x720 mode.

DispVMode1024x768x4 ... resolution 1024x768 pixels, 4 bits per pixel, system clock 324 MHz, required buffer size 393216 bytes, aspect ratio 4:3. It requires overclocking close to the limits of the processor's capabilities, plus the frequency used is slightly lower than the norm, so some VGA monitors may crop the margins of the image slightly.

DispVMode1056x600x4 ... resolution 1056x600 pixels, 4 bits per pixel, system clock 264 MHz, required buffer size 316800 bytes, aspect ratio 16:9. Can be used as a wide-screen alternative to 1024x768 mode. A VGA monitor can display the image as non-widescreen because it considers it to be in 800x600 mode.

DispVMode1280x600x4 ... resolution 1280x600 pixels, 4 bits per pixel, system clock 288 MHz, required buffer size 384000 bytes. High-resolution mode. A VGA monitor can display the image as non-widescreen because it considers it to be in 800x600 mode.

DispVMode1440x600x4 ... resolution 1440x600 pixels, 4 bits per pixel, system clock 320 MHz, required buffer size 432000 bytes. Next high-resolution mode. A VGA monitor can display the image as non-widescreen because it considers it to be in 800x600 mode. This mode is already at the limit of the processor's capabilities.

Example of use: **DispVMode640x480x8(0, NULL);** initializes video mode with resolution 640x480 pixels, 8 bits per pixel, and creates frame buffer of the size 307200 bytes.

The following tables list the predefined functions for initializing a simplified video node. The table contains horizontal resolution XRES, vertical resolution YRES, and columns representing the number of BITS per pixel that contain the frame buffer size in bytes. If the frame buffer size is not specified, the appropriate combination of timing, resolution, and color depth is not supported - either due to lack of RAM or because the driver would not render the data fast enough.

The function to initialize the video mode is of the form DispVModeXRESxYRESxBITS(), where XRES is used to specify the horizontal resolution of the video mode, YRES is used to specify the vertical resolution, and BITS is used to specify the number of bits per pixel. For example, in the "Timings 640x480" table, there is a row with the entries 640, 480, and in column 8 there is the entry 307200. This means that you can use the DispVMode640x480x8(0, NULL) function to initialize a video mode with a resolution of 640x480 pixels, in 8-bit color depth, and it creates a frame buffer of 307200 bytes.

The frame buffer size is not specified in column 16, which means that this combination is not supported because there is not enough RAM for the frame buffer. Similarly, the frame buffer size for 640x240 resolution and a column of 16 bits per pixel is not given, even though there would be enough memory for the frame buffer, but the processor would not be able to render this mode fast enough in VGA mode. This mode is only possible at the next timing, 640x480 fast, where the processor can render the image due to the higher frequency.

If "Fast system clock" is specified for the timing, add suffix "_Fast" to the function name to indicate that the fast system clock option will be used. For example, you can use the 640x480x8 resolution in two variants: DispVMode640x380x8() for 126 MHz, and DispVMode640x380x8_Fast() for 252 MHz.

Timings 532x400/70.2Hz/31.5kHz (EGA 4:3)

Pixel clock 20.9 MHz, system clock 104.571 MHz, detected as 720x400@70Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
266	**200**	106400	106400	81600	53600	43200	27200	21600	13600	7200
266	**400**	212800	212800	163200	107200	86400	54400	43200	27200	14400
532	**200**	-	-	-	106400	85600	53600	43200	27200	13600
532	**400**	-	-	-	212800	171200	107200	86400	54400	27200

Recommended example: DispVMode532x400x8(0, NULL);

Timings 532x400/70.2Hz/31.5kHz (EGA 4:3), fast system clock

Pixel clock 20.9 MHz, **Fast** system clock 209.143 MHz, detected as 720x400@70Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
266	**200**	106400	106400	81600	53600	43200	27200	21600	13600	7200
266	**400**	212800	212800	163200	107200	86400	54400	43200	27200	14400
532	**200**	212800	212800	160800	106400	85600	53600	43200	27200	13600
532	**400**	425600	425600	321600	212800	171200	107200	86400	54400	27200

Recommended example: DispVMode532x400x16_Fast(0, NULL);

Timings 640x350/70.2Hz/31.5kHz

Pixel clock 25.2 MHz, system clock 126 MHz, detected as 640x350@70Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
320	**175**	112000	112000	84000	56000	44800	28000	22400	14000	7000
320	**350**	224000	224000	168000	112000	89600	56000	44800	28000	14000
640	**175**	-	-	-	112000	89600	56000	44800	28000	14000
640	**350**	-	-	-	224000	179200	112000	89600	56000	28000

Recommended example: DispVMode640x350x8(0, NULL);

Timings 640x350/70.2Hz/31.5kHz, fast system clock

Pixel clock 25.2 MHz, **Fast** system clock 252 MHz, detected as 640x350@70Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
320	**175**	112000	112000	84000	56000	44800	28000	22400	14000	7000
320	**350**	224000	224000	168000	112000	89600	56000	44800	28000	14000
640	**175**	224000	224000	168000	112000	89600	56000	44800	28000	14000
640	**350**	448000	448000	336000	224000	179200	112000	89600	56000	28000

Recommended example: DispVMode640x350x16_Fast(0, NULL);

Timings 640x400/70.2Hz/31.5kHz

Pixel clock 25.2 MHz, system clock 126 MHz, detected as 720x400@70Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
320	**200**	128000	128000	96000	64000	51200	32000	25600	16000	8000
320	**400**	256000	256000	192000	128000	102400	64000	51200	32000	16000
640	**200**	-	-	-	128000	102400	64000	51200	32000	16000
640	**400**	-	-	-	256000	204800	128000	102400	64000	32000

Recommended example: DispVMode640x400x8(0, NULL);

Timings 640x400/70.2Hz/31.5kHz, fast system clock

Pixel clock 25.2 MHz, **Fast** system clock 252 MHz, detected as 720x400@70Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
320	**200**	128000	128000	96000	64000	51200	32000	25600	16000	8000
320	**400**	256000	256000	192000	128000	102400	64000	51200	32000	16000
640	**200**	256000	256000	192000	128000	102400	64000	51200	32000	16000
640	**400**	-	-	384000	256000	204800	128000	102400	64000	32000

Recommended example: DispVMode640x400x12_Fast(0, NULL);

Timings 640x480/60Hz/31.5kHz (VGA 4:3, VESA DMT ID 04h)

Pixel clock 25.2 MHz, system clock 126 MHz, detected as 640x480@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
320	**240**	153600	153600	115200	76800	61440	38400	30720	19200	9600
320	**480**	307200	307200	230400	153600	122880	76800	61440	38400	19200
640	**240**	-	-	-	153600	122880	76800	61440	38400	19200
640	**480**	-	-	-	307200	245760	153600	122880	76800	38400

Recommended example: DispVMode640x480x8(0, NULL);

Timings 640x480/60Hz/31.5kHz (VGA 4:3, VESA DMT ID 04h), fast clock

Pixel clock 25.2 MHz, **Fast** system clock 252 MHz, detected as 640x480@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
320	**240**	153600	153600	115200	76800	61440	38400	30720	19200	9600
320	**480**	307200	307200	230400	153600	122880	76800	61440	38400	19200
640	**240**	307200	307200	230400	153600	122880	76800	61440	38400	19200
640	**480**	-	-	-	307200	245760	153600	122880	76800	38400

Recommended example: DispVMode640x480x8_Fast(0, NULL);

Timings 640x720/60Hz/45kHz (half-HD)

Pixel clock 37.2 MHz, system clock 186 MHz, detected as 1280x720@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
320	**360**	230400	230400	172800	115200	92160	57600	46080	28800	14400
320	**720**	-	-	345600	230400	184320	115200	92160	57600	28800
640	**360**	-	-	-	230400	184320	115200	92160	57600	28800
640	**720**	-	-	-	-	368640	230400	184320	115200	57600

Recommended example: DispVMode640x360x8(0, NULL);

Timings 720x400/70.1Hz/31.5kHz

Pixel clock 28.32 MHz, system clock 141.6 MHz, detected as 720x400@70Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
360	**200**	144000	144000	108000	72000	57600	36000	28800	18400	9600
360	**400**	288000	288000	216000	144000	115200	72000	57600	36800	19200
720	**200**	-	-	-	144000	115200	72000	57600	36000	18400
720	**400**	-	-	-	288000	230400	144000	115200	72000	36800

Recommended example: DispVMode720x400x8(0, NULL);

Timings 720x400/70.1Hz/31.5kHz, fast system clock

Pixel clock 28.32 MHz, **Fast** system clock 283.2 MHz, detected as 720x400@70Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
360	**200**	144000	144000	108000	72000	57600	36000	28800	18400	9600
360	**400**	288000	288000	216000	144000	115200	72000	57600	36800	19200
720	**200**	288000	288000	216000	144000	115200	72000	57600	36000	18400
720	**400**	-	-	-	288000	230400	144000	115200	72000	36800

Recommended example: DispVMode720x400x8_Fast(0, NULL);

Timings 720x480/60Hz/31.5kHz (VGA 3:2, SD video NTSC)

Pixel clock 28.32 MHz, system clock 141.6 MHz, detected as 640x480@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
360	**240**	172800	172800	129600	86400	69120	43200	34560	22080	11520
360	**480**	345600	345600	259200	172800	138240	86400	69120	44160	23040
720	**240**	-	-	-	172800	138240	86400	69120	43200	22080
720	**480**	-	-	-	345600	276480	172800	138240	86400	44160

Recommended example: DispVMode720x480x8(0, NULL);

Timings 720x480/60Hz/31.5kHz (VGA 3:2, SD video NTSC), fast clock

Pixel clock 28.32 MHz, **Fast** system clock 283.2 MHz, detected as 640x480@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
360	**240**	172800	172800	129600	86400	69120	43200	34560	22080	11520
360	**480**	345600	345600	259200	172800	138240	86400	69120	44160	23040
720	**240**	345600	345600	259200	172800	138240	86400	69120	43200	22080
720	**480**	-	-	-	345600	276480	172800	138240	86400	44160

Recommended example: DispVMode720x480x8_Fast(0, NULL);

Timings 800x480/60Hz/31.5kHz (MAME 5:3)

Pixel clock 31.5 MHz, system clock 157.333 MHz, detected as 640x480@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
400	**240**	192000	192000	144000	96000	76800	48000	38400	24000	12480
400	**480**	384000	384000	288000	192000	153600	96000	76800	48000	24960
800	**240**	-	-	-	192000	153600	96000	76800	48000	24000
800	**480**	-	-	-	384000	307200	192000	153600	96000	48000

Recommended example: DispVMode800x480x8(0, NULL);

Timings 800x600/60.3Hz/37.9kHz (SVGA 4:3, VESA DMT ID 09h)

Pixel clock 40 MHz, system clock 200 MHz, detected as 800x600@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
400	**300**	240000	240000	180000	120000	96000	60000	48000	30000	15600
400	**600**	-	-	360000	240000	192000	120000	96000	60000	31200
800	**300**	-	-	-	240000	192000	120000	96000	60000	31200
800	**600**	-	-	-	-	384000	240000	192000	120000	60000

Recommended example: DispVMode800x600x6(0, NULL);

Timings 848x480/60Hz/31.4kHz (DMT SVGA 16:9)

Pixel clock 33.33 MHz, system clock 166.667 MHz, detected as 640x480@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
424	**240**	203520	203520	152640	101760	81600	50880	41280	25920	13440
424	**480**	407040	407040	305280	203520	163200	101760	82560	51840	26880
848	**240**	-	-	-	203520	163200	101760	81600	50880	25920
848	**480**	-	-	-	407040	326400	203520	163200	101760	51840

Recommended example: DispVMode848x480x8(0, NULL);

Timings 960x720/60Hz/45kHz (VESA 4:3)

Pixel clock 55.7 MHz, system clock 278.4 MHz, detected as 1280x720@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
480	**360**	345600	345600	259200	172800	138240	86400	69120	43200	21600
480	**720**	-	-	-	345600	276480	172800	138240	86400	43200
960	**360**	-	-	-	345600	276480	172800	138240	86400	43200
960	**720**	-	-	-	-	-	345600	276480	172800	86400

Recommended example: DispVMode960x720x4(0, NULL);

Timings 1024x768/59.9Hz/48.3kHz (XGA 4:3, VESA DMT ID 10h)

Pixel clock 64.8 MHz, system clock 324 MHz, detected as 1024x768@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
512	**384**	393216	393216	294912	196608	158208	98304	79872	49152	24576
512	**768**	-	-	-	393216	316416	196608	159744	98304	49152
1024	**384**	-	-	-	393216	316416	196608	159744	98304	49152
1024	**768**	-	-	-	-	-	393216	316416	196608	98304

Recommended example: DispVMode1024x768x4(0, NULL);

Timings 1056x600/60Hz/37.9kHz (near 16:9)

Pixel clock 52.8 MHz, system clock 264 MHz, detected as 800x600@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
528	**300**	316800	316800	237600	158400	127200	79200	63600	39600	20400
528	**600**	-	-	-	316800	254400	158400	127200	79200	40800
1056	**300**	-	-	-	316800	254400	158400	127200	79200	39600
1056	**600**	-	-	-	-	-	316800	254400	158400	79200

Recommended example: DispVMode1056x600x4(0, NULL);

Timings 1152x600/60Hz/37.9kHz

Pixel clock 57.6 MHz, system clock 288 MHz, detected as 800x600@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
576	**300**	345600	345600	259200	172800	139200	86400	69600	43200	21600
576	**600**	-	-	-	345600	278400	172800	139200	86400	43200
1152	**300**	-	-	-	345600	277200	172800	139200	86400	43200
1152	**600**	-	-	-	-	-	345600	278400	172800	86400

Recommended example: DispVMode1152x600x4(0, NULL);

Timings 1280x400/70Hz/31.5kHz

Pixel clock 50.4 MHz, system clock 252 MHz, detected as 720x400@70Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
1280	**200**	-	-	-	256000	204800	128000	102400	64000	32000
1280	**400**	-	-	-	-	409600	256000	204800	128000	64000

Recommended example: DispVMode1280x400x6(0, NULL);

Timings 1280x480/60Hz/31.5kHz

Pixel clock 50.4 MHz, system clock 252 MHz, detected as 640x480@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
1280	**240**	-	-	-	307200	245760	153600	122880	76800	38400
1280	**480**	-	-	-	-	-	307200	245760	153600	76800

Recommended example: DispVMode1280x480x4(0, NULL);

Timings 1280x600/56Hz/35.2kHz

Pixel clock 57.6 MHz, system clock 288 MHz, detected as 800x600@56Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
640	**300**	384000	384000	288000	192000	153600	96000	76800	48000	24000
640	**600**	-	-	-	384000	307200	192000	153600	96000	48000
1280	**300**	-	-	-	384000	307200	192000	153600	96000	48000
1280	**600**	-	-	-	-	-	384000	307200	192000	96000

Recommended example: DispVMode1280x600x4(0, NULL);

Timings 1280x720/60.1Hz/45kHz

Pixel clock 74.4 MHz, system clock 372 MHz, detected as 1280x720@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
640	**360**	-	-	345600	-	-	-	-	-	-
1280	**360**	-	-	-	-	368640	230400	184320	115200	57600
1280	**720**	-	-	-	-	-	-	368640	230400	115200

Recommended example: DispVMode1280x720x3(0, NULL);

Timings 1280x768/59.6Hz/47.6kHz

Pixel clock 79.2 MHz, system clock 396 MHz, detected as 1280x768@60Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
640	**384**	-	-	368640	245760	196608	122880	98304	61440	30720
640	**768**	-	-	-	-	393216	245760	196600	122880	61440
1280	**384**	-	-	-	-	393216	245760	196608	122880	61440
1280	**768**	-	-	-	-	-	-	393216	245760	122880

Recommended example: DispVMode1280x768x3(0, NULL);

Timings 1280x800/59.6Hz/49.5kHz

Pixel clock 83.2 MHz, system clock 416 MHz, detected as 1280x800@60Hz.

This video mode requires DISPHSTX_DISP_SEL switch to be enabled.

XRES	YRES	16	15	12	8	6	4	3	2	1
640	**400**	-	-	384000	-	-	-	-	-	-
640	**800**	-	-	-	-	409600	256000	204800	128000	64000
1280	**800**	-	-	-	-	-	-	409600	256000	128000

Recommended example: DispVMode1280x800x3(0, NULL);

Timings 1360x768/60.1Hz/47.8kHz

Pixel clock 85.6 MHz, system clock 428 MHz, detected as 1024x768@60Hz.

This video mode requires DISPHSTX_DISP_SEL switch to be enabled.

XRES	YRES	16	15	12	8	6	4	3	2	1
680	**384**	-	-	391680	261120	208896	130560	104448	66048	33792
680	**768**	-	-	-	-	417792	261120	208896	132096	67584
1360	**384**	-	-	-	-	417792	261120	208896	130560	66048
1360	**768**	-	-	-	-	-	-	417792	261120	132096

Recommended example: DispVMode1360x768x3(0, NULL);

Timings 1440x600/56Hz/35kHz

Pixel clock 64 MHz, system clock 320 MHz, detected as 800x600@56Hz.

XRES	YRES	16	15	12	8	6	4	3	2	1
720	**300**	432000	432000	324000	216000	172800	108000	86400	54000	27600
720	**600**	-	-	-	432000	345600	216000	172800	108000	55200
1440	**300**	-	-	-	432000	345600	216000	172800	108000	54000
1440	**600**	-	-	-	-	-	432000	345600	216000	108000

Recommended example: DispVMode1440x600x4(0, NULL);