DVI/VGA/TV/HDMI VIDEO DRIVER COG DOCUMENTATION (Release 0.94b)

Introduction

This single Cog P2 video driver generates graphics and/or text based video content over Parallax Propeller 2 IO pins. It can output either digital video on 8 pins (signalled as TMDS for HDMI/DVI), or on up to 24 pins (as parallel RGB), or analog video signals including VGA, HDTV, SDTV in composite, S-Video or component colour formats using the built in Propeller 2 video DACs and includes several configurable SYNC options.

The driver offers an extensive list of features including but not limited to:

- custom video resolutions and timing,
- graphics bitmap and text modes,
- multiple independent display regions on screen,
- a graphics mouse/sprite per region,
- all supported Propeller 2 colour modes,
- pixel and line doubling,
- hardware cursors,
- per region palettes,
- variable height fonts,
- per region scrolling and wrapping options,
- built-in text mode renderer,
- 16 foreground and background coloured text and configurable per character,
- a flashing text attribute option,
- fallback to monochrome text where required or requested,
- support for graphics buffers to be read from external memory such as HyperRAM & PSRAM,
- optional top/bottom/side borders with a programmable border colour,
- dynamic (updated per field) colour space adjustment in analog output modes,
- automatic interlaced output/page flipping support in graphics & text modes,
- flexible sync options including HDTV three level video sync,
- LCD DE and clock signal generation in parallel digital RGB output modes.

Driver Operation

The underlying PASM2 based driver Cog is created with a display parameter structure containing startup data which it uses to configure a given display output type with the desired operating parameters. Once initialized and operational, the driver continues to read in a subset of this same structure identifying a list of screen regions to be displayed, any optional screen borders and border colour, a global mouse sprite position, along with colour space converter and any other relevant output specific parameter(s). This is updated during every vertical blanking period, before the next field or frame begins. The driver also updates a status location stored in HUB RAM indicating what part of the frame is being displayed and can also notify other Cogs with COGATN in some situations.

DISPLAY STRUCTURE OVERVIEW

The driver's parameter structure consists of a block of 32 bit long values stored in a simple array shown below. The hub RAM address of this structure in hub RAM is provided via the PTRA register as an argument upon COG startup. The upper bits of PTRA include some startup flag option(s) that are not part of the structure itself but also influence the startup and operation of the driver.

Offset	Parameter Name	Description
+$00	STATUS	Video driver status (read only, gets written to HUB by driver)
+$04	INITIALIZATION	Used for video timing initialization and sync/pin definitions
+$08	EXTENDED_INIT	Extended initialization data
+$0C	EXT_MEMORY_MAILBOX	Nominates address of external memory mailboxes
+$10	LINE_BUFFER_1	Address of HUB RAM where even scan lines are built
+$14	LINE_BUFFER_2	Address of HUB RAM where odd scan lines are built
+$18	OUTPUT_PARAMS	Output specific parameters
+$1C	GLOBAL_MOUSE	Defines the global mouse coordinates
+$20	BORDER_SIZE	Defines top/bottom heights of optional border region
+$24	BORDER_COLOUR	Defines width and colour of optional border region
+$28	FIRST_REGION	List of displayed regions in HUB RAM
+$2C	Q_PARAMETER	Defines CQ colour modulator parameter for analog video
+$30	I_PARAMETER	Defines CI colour modulator parameter for analog video
+$34	Y_PARAMETER	Defines CY colour modulator parameter for analog video

PTRA format passed during initialization of COG:

Bits 31:24	23	22	21	20	Bits 19:0
COGATN Mask	S	V	W	F	Display structure pointer

Bit [20] Force Mono Text (F)

0 - driver generates colour text when clocking allows it but will fallback to monochrome if required
1 - driver is never allowed to generate colour text, and is always forced into monochrome mode

Bit [21] Wait for ATN [W]

0 - driver will complete configuration and begin outputting video immediately following that step
1 - driver will wait for an ATN to begin generating video - can be used to sequence Cog startups

Bit [22] VSYNC only COGATNs [V]

0 - driver issues COGATN notifications once per scan line during horizontal blanking
1 - driver issues COGATN once per field or frame output, during the vertical sync interval

Bit [23] Startup only COGATN [S]

0 - COGATN notification continues during frame output, depending on Bit 22 value
1 - COGATN notification only ever occurs once at driver startup, Bit 22 is ignored

COGATN Mask

Identifies which Cogs receive the requested notification. Each bit position that is set to 1 indicates that Cog ID will receive the ATN event. If no notification is needed, set this field to zero.

REGION STRUCTURE OVERVIEW

The driver displays a sequence of one or more screen regions to an output device. A region is a portion of the screen's vertical height, defined in terms of active scan lines. Each displayed region to be presented on the output screen is indicated using a separate region structure in hub memory.

Offset	Parameter Name	Description
+$00	NEXT_REGION	Link to next region structure in HUB memory in this display list
+$04	REGION_CONFIG	Defines size and mode of the region and attributes
+$08	SCREEN_BUFFER_1	Defines the start address of source text/graphics data
+$0C	SCREEN_BUFFER_2	Defines the start address of source text/graphics data after the region optional wraps according to SOURCE_WRAP
+$10	FONT_BUFFER	Defines the font height and font location in HUB memory
+$14	PALETTE_BUFFER	Defines the palette start address in HUB memory
+$18	TEXT_CURSOR_1 or SCREEN_BUFFER_3	Defines the first text cursor's position, colour, attributes, and for SCREEN_BUFFER_3 in any interlaced graphics modes
+$1C	TEXT_CURSOR_2 or SCREEN_BUFFER_4	Defines the second text cursor's position, colour, attributes and for SCREEN_BUFFER_4 in any interlaced graphics mode
+$20	REGION_MOUSE	Defines the region specific mouse co-ordinates
+$24	MOUSE_SPRITE	Defines the mouse sprite HUB address and its hot spots
+$28	SOURCE_WRAP	Defines a scan line to wrap around to a new source address
+$2C	SOURCE_SKEW	Defines the skew or pitch of each scan line in memory

These region structures can be chained together into linked list of multiple regions to be rendered per screen frame, and they can also be updated dynamically. Regions will be displayed in the order they appear in the linked list and the first region to be displayed (head of the list) is identified using the FIRST_REGION parameter in the display parameter structure identified above while the NEXT_REGION value in the region structure itself is the link to the next region in the list or zero at the end of the list. The format for the region structure is another array of 32 bit longs, provided below:

Display Structure Descriptions

STATUS - status of driver (written back to HUB RAM)

Once the driver is running, this 32 bit long in HUB memory is updated by the video driver to advertise the current frame/sync status to other COGs. The data that the driver writes back into HUB memory is shown below and is updated once per scan line in most cases.

31	30	29	28:24	Bits 23:16	Bits 15:0
V	F	B	Status	Field Counter	Active Scan Lines Displayed

The frame/field status updated by the driver cog includes the following information:

Bit [31] VSYNC (V)
0 - driver is not generating vertical sync scan lines
1 - driver is generating vertical sync scan lines

Bit [30] Front Porch (F)
0 - driver is not generating the front porch
1 - driver is generating the front porch scan lines during blanking

Bit [29] Back Porch (B)
0 - driver is not generating the back porch
1 - driver is generating the front porch scan lines during blanking

Bits [28:24] Status
This contains a value that is updated during front and back porch times.

During the back porch the Status nibble indicates the COG ID of the driver. During the front porch it indicates whether the driver is operating in monochrome or colour text output mode by taking on one of three values below. At all other times its value is zero.

If Status = 0, the text region output supports colour mode with attributes
If Status = 1, the text region output supports monochrome without attributes
If Status = 2, the text region output supports monochrome with attributes

Bits [23:16] - Field Counter
This is an 8 bit counter that increments at the field frequency rate at vertical synchronization time.

Bits [15:0] - Active Scan Lines Displayed
This counter contains the number of active scan lines that have already been displayed in this field or frame. It resets during the vertical blanking interval.

INITIALIZATION - initializes the driver output type & timing

At driver COG startup time, the initialization long value is used for setup purposes.

31	30	29:24	Bits 23:20	Bits 19:0
D	V	Sync Outputs	Pin Group	Custom Video Timing Address

Bit [31] Display Mode (D)
Selects Analog or Digital display output mode.

0 - the display driver outputs analog video over its DACS (e.g. VGA or TV signals)
1 - the display driver outputs digital video (TMDS) on 8 P2 pins or RGB on 24 P2 pins

Bit [30] Video Output Type (V)
Indicates type of video to output.

In analog display mode:
0 - select RGB/RGBS/RGBHV output
1 - select HD/SD TV, Composite and/or Y+C (S-video), or component output

In digital display mode:
0 - select TMDS output format
1 - select 24 bit RGB output format (with optional simultaneous RBG DAC output)

Bits [29:24] Sync Outputs

In Analog mode this controls how the sync signal will be generated and how the video signals will be sent over the DACs outputs.

Use of this field when outputting RGB:

If a separate VSYNC signal is desired, then this 6 bit field indicates which P2 pin will output the VSYNC signal, the video output will be 5 pin RGBHV.

If RGBHV is not required then this field's value can be used to select other sync options. To do this map this 6 bit field's value to one of the DAC channel pin numbers already assigned to the group setup in the PinGroup field.

If mapped to DAC0 pin - send RGBS, S = VSYNC XOR HSYNC on DAC0 (75 ohms)
If mapped to DAC1 pin- send RGB/Component with sync on DAC1 pin
If mapped to DAC2 pin - send RGB/Component with sync on DAC2 pin (SoG)
If mapped to DAC3 pin - send RGB/Component with sync on DAC3 pin

Use of this field when outputting HDTV/SDTV:

Bit [29] selects HDTV tri-level (1) or standard SDTV sync (0) pulses
Bit [28] selects Interlaced (1) or Progressive (0) video frame output
Bit [27] selects SDTV colour system (this bit is reserved for HDTV, so set to 0)
    0 - NTSC colour burst generated
    1 - PAL colour burst generated (alternating per line)
Bits [26:24] defines the video output format or sync pins and depends on the mode

In SDTV sync pulse mode these 3 bits select the video output type and DAC(s) used in the pin group:
- 000 - Composite video output on DAC0 pin
- 001 - Composite video output on DAC1 pin
- 010 - Composite video output on DAC2 pin
- 011 - Composite video output on DAC3 pin
- 100 - Y+C output on DAC0(Y) & DAC1(C) (S-Video)
- 101 - Y+C output on DAC0(Y) & DAC1(C) + Composite on DAC2 pin
- 110 - Component video on DAC3(Y), DAC2(Pr), DAC1(Pb) pins
- 111 - Component video on DAC3(Pr), DAC2(Y), DAC1(Pb) pins

In HDTV sync pulse mode the 3 bits select sync pulse output(s) on DAC3, DAC2, DAC1 pins in the three pin YPrPb group, depending on the bit pattern:
- 100 - DAC3 pin enables sync pulse
- 010 - DAC2 pin enables sync pulse
- 001 - DAC1 pin enables sync pulse
- 111 - All 3 DACs enable sync pulses, etc.

In Digital mode the Sync Outputs field is currently reserved for future TTL/LCD/HDMI output options and should be configured with 0 for now.

Bits [23:20] Pin Group

This 4 bit nibble controls which P2 pins will output the DVI or analog signal and the order the signals are output on the pins in DVI mode.

In Digital display mode:

If DVI is selected the least significant 3 bits of this nibble control which group of 8 P2 pins will be used to output the DVI signal, with %000=P0-P7, %001=P8-15, etc.

The top bit of the nibble indicates which way the TMDS signals are ordered in the group of 8 allocated DVI pins, in increasing pin order below.
Bit 23 = 0 : CLK-, CLK+, BLUE-, BLUE+, GREEN-, GREEN+, RED-, RED+
Bit 23 = 1 : RED+, RED-, GREEN+, GREEN-, BLUE+, BLUE-, CLK+, CLK-

In Analog display mode:

The nibble indicates the four pin group that the video DACs will use.
Use %0000=P0-P3, %0001=P4-P7,…, %1110=P56-P59, etc.

Potentially the pins could be allocated in the following way to the colour channels to help keep the cable/connector colours consistent between component or RGB output types, but the actual colour mapping between them is also configurable using the colour space registers.

DAC3 - Red / Pr
DAC2 - Green / Y
DAC1 - Blue / Pb
DAC0 - only use as a horizontal or composite sync output with RGB

Bits [19:0] Custom Video Timing Address

When non-zero, this value represents the address of a multi-long structure containing parameters which can override the standard 640x480 VGA timing that will otherwise be attempted by default if this field is zero.

An example of the structure it could point to for a custom resolution (in this case, retro EGA) is:

```
ega_timing   'EGA resolution 640x350 @ 70Hz with 25.2MHz pixel clock

long   0      ' Optional P2 PLL clock mode settings to be used for this
              ' video mode. If non-zero the high level driver code will use
              ' this information explicitly before spawning the driver COG
              ' to change the current PLL settings. If zero the clock mode
              ' will be computed automatically based on the value in the
              ' next long parameter.

long   252000000 ' CPU clock frequency value once PLL setting is applied
              ' unless zero. If zero, no change to the PLL will be made.

long   10<<8  ' This behaves either as the P2 clock divider word or
              ' as a custom XFRQ value. If the most significant word
              ' is zero the lower word is used as 16 bit divider,
              ' otherwise the entire 32 bits becomes the XFRQ value.
              ' The 16 bit divider is comprised of an integer portion
              ' in the upper byte, and fractional portion in the LSB.

long    0     ' Custom CFRQ setting for PAL/NTSC modulator output.

      '_____Reserved_____Vsync_Polarity__Hsync_Polarity
      '     14 bits          1 bit            1 bit
word        (0<<2)   |   (SYNC_NEG<<1)  |  (SYNC_POS<<0)

      '_Breezeway_pixels   Colourburst_pixels (data is valid for PAL/NTSC)
      '     8 bits             8 bits
word       (13<<8)       |    ( 37<<0)

word    16          ' horizontal front porch pixels
word    96          ' horizontal sync pixels
word    48          ' horizontal back porch pixels
word    640/8       ' horizontal displayed columns (=total active pixels/8)

word    37          ' vertical front porch scan lines
word    2           ' vertical sync scan lines
word    60          ' vertical back porch scan lines
word    350         ' vertical displayed scan lines
```

Notes:

The PASM2 driver code does NOT modify the P2 PLL / system clock options or know what they are. Accordingly it requires that the timing data it is passed to be applicable to the current P2 system clock in effect. The high level SPIN2 initDisplay call will setup the PLL for you but if you bypass that layer and choose to spawn the code independently your code must setup the PLL based on the desired video timing information.

The vertical sync timing gets overridden in SDTV interlaced modes, to comply with the PAL/NTSC vertical sync format.

The divider field is ignored and locked to 10 for DVI, as the TMDS output requires a 10x pixel clock for clocking its output data. This also (generally) implies that P2 clocks > ~250 MHz are required.

For analog modes, the pixel clock and CPU clock can have different relationships, however operating the P2 CLK lower than about 5 times the VGA pixel clock can start to cause problems running out of time in the code to complete the text mode scan lines before they display and visible artefacts or even full loss of output could then result. For this reason the code will attempt to fall back to monochrome text in this case either with or without attributes. If that has occurred the driver will make that information known at blanking time during the front porch via the status long.

Simpler graphics modes which only use the CPU lightly such as non-LUT based non-pixel doubled modes with low bit depths may operate at much lower clock ratios as low as 2x, but performance is not guaranteed. The low power 1x clock ratios are possible only in transparent mode.

If the number of columns multiplied by 8 is not divisible by the number of pixels per long then an additional write will be made in the line buffer for storing the scan line's pixels. In this case you should setup the line buffer spacing to be a value that holds the rounded up number of pixels multiplied by the bit depth accordingly otherwise corruption of the next line can occur.

Eg. if columns = 90 (720 visible pixels) and a 1bpp mode is used, then each 32 bit long holds 32 pixels. 720 is not divisible by 32 (22.5 longs), so an additional memory write of 23 longs total will occur into the line buffer. For this reason the line buffer size in this case should be configured as no less than 23*4 = 92 bytes when spawning the driver even though only 90 bytes will hold valid data. When the stored pixel data is rendered out for display all 23 longs will be read back to source the pixels, but only the data for the visible pixels in the long is actually used.

EXTENDED_INIT - expansion of configuration parameters

This long can be used to either pass in extended data value(s) for the driver's configuration or as a pointer to additional mode dependent parameters that do not fit within this 32 bit field in the display structure.

For example, it can point to some audio configuration parameters in an HDMI output mode or an extra configuration block with additional information such as pins allocated to LCD DE/CLK signals in digital RGB output mode.

EXT_MEMORY_MAILBOX - external memory interface

31	Bits 30:24	Bits 23:20	Bits 19:0
E	Spacing	Offset	External Memory Mailbox Base Address

Bit [31] Enable (E)
0 - disabled, any external access will be ignored.
1 - enabled, mailbox address is valid and external memory accesses are supported.

Spacing
Number of bytes between consecutive COGs mailboxes.

Offset
Number of longs between first and second mailbox addresses.

External Memory Mailbox Address
Configures the base mailbox address for accessing external memory by the video driver. The driver will use this mailbox address and the Bus ID along with its own COG ID and mailbox spacing information to determine the final mailbox addresses used.

When non-zero a Bus ID is a field that is used per region to identify which external memory bank will be used to access the video data, and allows more than one external memory driver to be operating at the same time. Eg. if both PSRAM and HyperRAM are fitted, each driver would be allocated its own Bus ID. Also a Bus ID of zero simply refers to HUB RAM.

In cases without external memory present the entire field can be left as zero and all accesses will occur from HUB RAM.

LINE_BUFFER_1/2 - specifies odd/even scan line buffers

Bits 31:20	Bits 19:0
Reserved (0)	Line Buffer Start Address

Line Buffer Start Address

Base address in HUB memory of a buffer that is sized to hold at least a scan line worth of graphics data at the maximum bit depth configured. Two line buffers are used, one for odd scan lines, and the other for even scan lines.

OUTPUT_PARAMS - expansion of field/frame parameters

This long can hold special parameters needed by a particular output type that can change per frame. It can also be used as a pointer to a group of parameters. This value is read in and updated per field/frame at VSYNC time by the driver COG. One current use of this parameter is to hold a 32 bit value used to XOR with the CQ parameter during phase alternating line (PAL) video.

Other examples of its use could be for updating LCD backlight brightness via PWM intensity in a digital LCD output mode, or for varying the audio volume pan/mixer levels and audio sampling rates in an HDMI output mode.

GLOBAL_MOUSE - position of global mouse on the screen

Bits 31:16	Bits 15:0
Mouse Y position	Mouse X position

Mouse X, Mouse Y position

In all modes these words are interpreted as the mouse's X,Y positions

The X,Y values are 0 based with (0,0) being top left on screen. To hide the mouse, set this long to -1 or other values outside the display range.

Note: when pixel or line doubling is enabled, the sprite will not change in size and the co-ordinates still refer to normal sized pixel positions. To compensate for this (if desired), a client of this driver could use even values for X,Y and a pixel doubled 8x8 sprite image.

BORDER_SIZE - selects optional top & bottom border sizes

Bits 31:16	Bits 15:0
Bottom Border Size	Top Border Size

Top Border Size

Number of optional border colour scan lines inserted at the top of the frame before the first region begins. Can be useful for some effects. Use zero for no top border colour scan lines.

Bottom Border Size

Number of optional border colour scan lines inserted at the bottom of the frame. It will override all region sizes. Useful for transitions. Use zero for no bottom border colour scan lines.

BORDER_COLOUR - selects colour and width of borders

Bits 31:24	Bits 23:16	Bits 15:8	Bits 7:0
Red	Green	Blue	Border Width

Red/Green/Blue

These 8 bit values change the RGB colour of any scan lines displayed before the first region begins and after the last region ends, forming an optional top and bottom border.

Border Width

When non-zero this sets the size of the borders drawn on each side of the screen in pixels. Each side will be this same width.

Note that when the P2 clock to pixel clock ratio is 1:1 there may not be sufficient CPU cycles for the streamer to generate a single pixel wide border at each side of the screen. If borders are desired at this ratio it is best to keep BorderWidths at least 2-3 pixels wide to avoid issues.

FIRST_REGION - link to first screen region to be displayed

Bits 31:0
First Region Pointer

First Region Pointer

Link to the first display region. This pointer is the hub RAM address of the first region's structure to be displayed on the screen. If zero, the border colour will continue all the way to the end of the screen and this is a quick way to blank out the screen with a single colour. The screen will remain blanked using the border colour until a new region list is provided to it.

Q/I/Y_PARAMETERS - colour space converter parameters

Bits 31:0
Q Parameter Value

Bits 31:0
I Parameter Value

Bits 31:0
Y Parameter Value

Q, I, Y Parameter Values

Contains the 3 values used in the SETCQ, SETCI, SETCY operations which configure the colour space converter, and only affect the analog output modes. These values are read and the registers updated once per field.

These values can be varied to affect the output brightness and colour mixing, NTSC/PAL saturation, gains, offsets etc. Care must be taken to avoid affecting the sync level in the least significant byte of each long. Modify at your own risk.

Region Structure Descriptions

NEXT_REGION - link to further region(s) after this one

Bits 31:0
Next Region Pointer

Next Region Pointer
This contains the HUB address of the next region's configuration data. Regions can be chained together in a linked list structure and the driver COG will display them all in the list sequence.

If this pointer address is 0, empty lines will continue to the end of the screen using the border colour, once this region's size has ended.

REGION_CONFIG - configuration of this screen region

Bits 31:16	Bits 15:8	Bits 7:0
Region Size	Region Flags	Colour Mode

Region Size
Number of scan lines in this region.

If zero it indicates this is the final screen region and the next region pointer value is ignored and this region will continue to the end of the displayed frame (or beginning of the bottom border).

Region Flags

Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8
Mouse Enabled	Mouse Source	Transparent Mode	Interlaced Source	Line Doubling	Pixel Doubling	Graphics Mode	Flashing Text Enable

Bit [15] Mouse Enabled
- 0 the mouse sprite is hidden in this region
- 1 the mouse sprite is displayed in this region

Bit [14] Mouse Source
- 0 the global mouse co-ordinates are used in this region
- 1 the region specific mouse co-ordinates are used in this region

Bit [13] Transparent / Sprite / pass through graphics mode (low power)

0 graphics are copied to a line buffer and optionally doubled
1 graphics are streamed out directly from the source hub memory buffer

Transparent mode is designed to work only with graphics modes using hub memory sourced frame buffer data and this flag will be ignored when the source data is read from external memory. Also if a mouse sprite is enabled in a region streaming from hub memory using transparent mode it will render the mouse into this hub memory, which might work out fine for sprite drivers which dynamically build this source data but may not be desirable for plain graphics frame buffers. In that case an independent mechanism to manage a mouse sprite would be required.

Bit [12] Interlaced Source

0 the same source is displayed on all field(s) in the frame
1 a different (interlaced) source is shown for each field of the frame

In text mode regions, this affect how fonts are displayed and when the bit is enabled, the font's scan lines will be interleaved across the odd/even fields, increasing the vertical resolution of the text.

In graphics regions, when this bit is enabled it activates an alternate source of data between odd/even fields. In this case the unused CURSOR_1 and CURSOR_2 registers are reclaimed as SCREEN_BUFFER_3 and SCREEN_BUFFER_4 which are then selected instead of SCREEN_BUFFER_1 and SCREEN_BUFFER_2 respectively in alternating fields, or alternating frames in progressive scan modes (enabling automatic double buffering of graphics without regular intervention).

Bit [11] Line Doubling

0 scan line doubling is disabled
1 scan line doubling is enabled

Bit [10] Pixel Doubling

0 pixel doubler is disabled (eg. for 80 column text / 640 pixel graphics)
1 pixel doubler is enabled (eg. for 40 column text / 320 pixel graphics)

Bit [9] Graphics Mode

0 selects text output and 16 colour mode
1 selects graphics colour mode(s) listed below

Bit [8] Flashing Text Enable

0 bit7 of text attribute byte selects background colour indexes from 8-15
1 bit7 of text attribute byte selects flashing text & background colour index range 0-7

In graphics regions this flag is unused (reserved).

Colour Mode

This 4 bit value indicates which P2 colour mode is to be used in graphics mode for this region.

Dec	Hex	Binary	Colour Mode	Bits per pixel	Colours	Pixels per long
0	$0	%0000	LUT palette	1	2/16M	32
1	$1	%0001	LUT palette	2	4/16M	16
2	$2	%0010	LUT palette	4	16/16M	8
3	$3	%0011	LUT palette	8	256/16M	4
4	$4	%0100	RGBI (3rgb + 5l)	8	8 colours x 32 levels	4
5	$5	%0101	RGB (3:3:2)	8	256 colours	4
6	$6	%0110	RGB (5:6:5)	16	64k “Hi-color”	2
7	$7	%0111	RGB (8r:8g:8b:0)	32	16M “Truecolor”	1
8	$8	%1000	LUMA orange	8	256 levels	4
9	$9	%1001	LUMA blue	8	256 levels	4
10	$A	%1010	LUMA green	8	256 levels	4
11	$B	%1011	LUMA cyan	8	256 levels	4
12	$C	%1100	LUMA red	8	256 levels	4
13	$D	%1101	LUMA magenta	8	256 levels	4
14	$E	%1110	LUMA yellow	8	256 levels	4
15	$F	%1111	LUMA white	8	256 levels (greyscale)	4

Note: Coloured text mode uses 4-bit LUT palettes, overriding the colour mode nibble. Monochrome text output use 1-bit LUT palettes. The upper 4 bits of this colour mode field are reserved and should be zeroed.

SCREEN_BUFFER_1 - video screen's source data address

Bits 31:28	Bits 27:24	Bits 23:0
Bus ID	Bank	Screen Buffer Base Address

Bus ID

When in graphics mode, the buffer base address can be set to source data from either HUB RAM, or from external memory buses. The Bus ID value should be set to 0 for HUB RAM, or to the non-zero Bus ID of the external memory bus from where the data will be requested. The mailbox address used for the external memory request for graphics data will be determined based on the Cog ID of the video driver, and the Bus ID indicated here.

Bank

Selects the bank for external memory accesses. This is passed into the mailbox during the read request for external memory data and can be used to indicate which particular physical device on a common memory bus is read from. Addresses will wrap within the same 16MB external memory bank. This field is ignored for HUB RAM accesses.

Screen Buffer Base Address

The HUB RAM or External memory start address to source pixel data from at the beginning of the region being displayed.

Notes:

In the text mode, all character data is always loaded from HUB RAM.

When external memory is being used in a graphics mode, any pixel width doubling applied is ignored and the graphics buffers in external RAM are assumed to contain standard width pixels. Scan line doubling does still work with external memory graphics buffers however.

SCREEN_BUFFER_2 - source address after region wraps

Bits 31:28	Bits 27:24	Bits 23:0
Bus ID	Bank	Screen Buffer Wrap Address

Bus ID

Refer to description for SCREEN_BUFFER_1.

Bank

Selects the bank for external memory accesses. Addresses will wrap within the same 16MB external memory bank.

Screen Buffer Wrap Address

Hub or External memory address to use if the screen data has wrapped in the region according to the settings in the SOURCE WRAP long. Source data wrapping can only optionally occur once per region.

NOTE: The Bus Id values can be set independently in both SCREEN_BUFFER_1 and SCREEN_BUFFER_2, enabling some of the displayed graphics in the region to be sourced from HUB and some from external RAM after the wrap occurs.

SCREEN_BUFFER_3/4 - interlaced graphics output modes

Replaces CURSOR_1 and CURSOR_2 registers when interlaced graphics regions are used. SCREEN_BUFFER_3 and SCREEN_BUFFER_4 have the same format as the previously defined SCREEN_BUFFER_1 and SCREEN_BUFFER_2 respectively. See the description of how these get used in the REGION_CONFIG information.

SCREEN_BUFFER_3

Bits 31:28	Bits 27:24	Bits 23:0
Bus ID	Bank	Screen Buffer Base Address

SCREEN_BUFFER_4

Bits 31:28	Bits 27:24	Bits 23:0
Bus ID	Bank	Screen Buffer Wrap Address

FONT_BUFFER - selects which font is used in text mode

Bits 31:24	Bits 23:0
Last Font Line	Font Table Base Hub Address

Font Table Base Hub Address

The address of the font data in HUB memory.

Last Font Line

The font's height in pixels minus 1.

A font is required in text mode, but this field is ignored for graphics.

Font data format

Fonts are required to be formatted as 256 byte blocks for each scan line in the font (requiring 8 pixel wide data per character). These 256 byte blocks are then repeated by the height of the font in pixels.

The least significant bits of the font data bytes get displayed first.

Bits set in the font data represent foreground coloured pixels for each character while zeroed bits are used for background coloured pixels.

For an “N” scan line sized font, the font’s data byte for each of the 256 characters are stored in HUB RAM in the following order at increasing HUB addresses from left to right, top to bottom in this list:

char0, char1, char2, … char255 data for scan line 1 (256 bytes)
char0, char1, char2, … char255 data for scan line 2 (256 bytes)
char0, char1, char2, … char255 data for scan line 3 (256 bytes)
char0, char1, char2, … char255 data for scan line 4 (256 bytes)
…
char0, char1, char2, … char255 data for scan line N (256 bytes)

PALETTE_BUFFER - configures the palette

Bits 31:24	Bits 23:0
Reserved (0)	Palette Base Hub Address

Palette Base Hub Address

The address of the palette's base address in HUB RAM. The palette is only required in text and LUT based graphics modes.

The reserved field is currently unused. It could be used by a calling API to track the current text foreground & background colour in this region or to count the number of active colours and free entries which might be dynamically created sometime later, etc.

Palette format

The palette is an array of N sequential longs in HUB RAM where N=2/4/16/256, containing 24 bit RGB data in the (8:8:8:0) format. One of these longs is indexed by the source data bit/nits/nibbles/ bytes in LUT modes to select the colour to output for the pixel.

Bits 31:24	Bits 23:16	Bits 15:8	Bits 7:0
Red 0	Green 0	Blue 0	0
Red 1	Green 1	Blue 1	0
…			0
Red N-1	Green N-1	Blue N-1	0

NOTE: Some care should be taken to ensure bit 1 of each palette entry is 0 or a TMDS control symbol could be sent instead of RGB pixels. Bit 0 should also be zeroed in order to allow the analog output from this driver to operate correctly without affecting the HSYNC state.

CURSOR_1/2 - sets position, type & colour of text cursors

Bits 31:24	Bits 23:16	Bits 15:8	Bits 7:0
Cursor Row	Cursor Column	Cursor Attributes	Colour Indexes

Cursor Row & Cursor Column

In the text mode these values represents the cursor's co-ordinates. Both values are 0 based, with (0,0) being top left in the region. Cursor 2 is rendered last and can overwrite cursor 1. If the cursor's row or column is out of the region's display range, it won't be visible, or you can disable it separately.

Cursor Attributes

Bit 15	Bit 14	Bit 13	Bit 12	Bits 11:8
Enable	Style	Direction	Blink Phase	Cursor Scan Line Height

Bit [15] Enable

0 - cursor is not displayed
1 - cursor is displayed

Bit [14] Style

0 - cursor blinks
1 - cursor is fixed

Bit [13] Direction

0 - underline style cursor drawn upwards from the ending scan line of the character's cell
1 - underline style cursor drawn downwards from the starting scan line of the character's cell

Bit [12] Blink Phase

0 - cursor uses blink phase 1
1 - cursor uses blink phase 2 (opposite to phase 1)

Cursor Scan Line Height

0 = Full cell block cursor displayed
1-15 = Underscore cursor height in scan lines

Colour Indexes

The two nibbles define the cursors colour(s). Each forms a 16 colour palette index value used to lookup the colour. The two colours are alternating per pixel in the cursor displayed. When both values are the same, the cursor will be rendered in a single colour, but they can also be set differently for different effects, useful for indicating states such as insert/overwrite or to show CAPS lock is on etc.

REGION_MOUSE - sets position of a region specific mouse

Bits 31:16	Bits 15:0
Mouse Y position	Mouse X position

Mouse X, Mouse Y positions

In all modes these words are interpreted as the mouse's X,Y positions

The X,Y values are 0 based with (0,0) being top left in the region. To hide the mouse, set this long to -1 or other values outside the display or disable it in the region configuration flags.

Note: these mouse co-ordinates will only be used if the region specific mouse select bit is enabled in the region configuration flags.

This setting allows each region to control it's own independent mouse position, or use the global mouse position, or have no mouse displayed. The same mouse sprite image (defined below) is used in either case.

MOUSE_SPRITE - sets mouse sprite image data & hotspots

Bits 31:28	Bits 27:24	Bits 23:20	Bits 19:0
Y hotspot	X hotspot	0	Mouse Sprite HUB Address

X, Y hot spots

These 4 bit offsets in the 16x16 sprite are centered at the mouse's X, Y screen co-ordinates.

Mouse Sprite Hub Address

The address of the mouse mask and sprite image data in scan line order.

BitDepth Mouse sprite data layout
1 bpp - 1 mask long then 1 long of pixel source data x 16 scan lines
2 bpp - 1 mask long then 1 long of pixel source data x 16 scan lines
4 bpp - 1 mask long then 2 longs of pixel source data x 16 scan lines
8 bpp - 1 mask long then 4 longs of pixel source data x 16 scan lines
16 bpp - 1 mask long then 8 longs of pixel source data x 16 scan lines
24 bpp - 1 mask long then 16 longs of pixel source data x 16 scan lines

The 16 bit mask (right aligned) indicates whether the corresponding mouse pixel in the source data long(s) will be drawn over the screen pixels. The bits in the mask are 1 for an active mouse pixel, 0 for transparent.

SOURCE_WRAP - sets region wrap options

Bits 31:16	Bits 15:0
Scan Line Offset/Rewrap	Source Wrap Scan Lines

Scan line Offset/Rewrap

This field has two uses, one for text regions, the other for graphics.

In text regions this value is the initial scan line offset in the first row of text displayed in the region, enabling very fine (single pixel) vertical scrolling of text.

Set to 0 if not used, or up to the font's height minus 1.

In graphics regions it is used after the first wrap occurs, and is the value reloaded into the wrap counter. Set it to zero for only one wrap, or to the number of scan lines until the second and future wrap occurs. This is used in external sprite modes where you might have (say) 3 COGs rendering into 3 different scan line buffers, and you need to continuously repeat the display of these 3 scan line buffers throughout the entire region or some later part of the region.

Source Wrap Scan lines

Number of scan lines to display after which the region's screen buffer source pointer will change to the second value in SCREEN BUFFER 2.

This allows infinite vertical scrolling effects with finite graphics frame buffers, or to be used with text modes to support a screen buffer that does not require its text data to be memory copied between rows to scroll it vertically.

Use a value of zero to prevent any buffer wrap occurring.

Wrapping could also be conveniently used when the number of scan lines in a region is not perfectly divisible by the font height. By setting the source wrap scan line position after the last completed row ends in the region, a second single row screen buffer could be used to display some extra blanks in the desired background colour to hide this until the region ends.

SOURCE_SKEW - controls skew per source scan line or row

Bits 31:0
Scan Line Skew (or Pitch)

Scan Line Skew

Optional extra bytes added between the end of the source graphics scan line or text row data in memory and the beginning of the next.

A typical value of zero keeps the source data fully packed, while non-zero values will allow some amount of horizontal offset or skew in memory.

By changing the screen buffer address and using non-zero skew values you can create some useful horizontal window panning/scrolling effects in addition to basic vertical scrolling.

Skew works well in text modes too and is row based. In text mode cases it should be kept a multiple of two to account for the 16 bit screen content.

Negative skew values can be used to reverse bitmaps, or replicate a single scan line in a region (solid colour/stripe fill) when the negative skew applied equals the scan line's source data width in bytes.

Scan Line Pitch

Bytes between a scan line's data and the next scan line's data.

In transparent pass through mode, the scan line skew works differently and becomes the exact byte spacing (pitch) between source scan lines instead of indicating the bytes between the end of one scan line's data and the beginning of the next (skew). In normal pass through cases it should be set to a non-zero value, because a zero value would simple repeat the same source data on every line otherwise (which may possibly be desirable for some simpler rapid fill effect perhaps).

Video Driver SPIN2 API

The P2 video driver SPIN2 API is briefly summarized below in groups of functionally related methods. See the driver source code for more information and parameter descriptions, etc. Note: In all of the APIs below, when a display or region parameter is indicated as an argument, it's always the start address of the structure in memory that is passed.

Initialization related methods

The driver has been developed to support multiple independent displays and multiple regions per display. Different displays can have different regions or even share regions if required.

Displays are initialized using ***initDisplay***. This method lets you nominate what type of video output to generate, which pins are used, sync options, memory allocation for line buffers, and a list of display regions, resolution timing and optionally a COG ID for the driver. An appropriate PASM driver COG is spawned during this step. This also lets you nominate a mailbox for a memory driver to access pixel data if the video content is to be sourced from an external memory fitted to the P2.

Regions are initialized using ***initRegion***. This method lets you specify the type of text/graphics mode a region will generate and its size, font, palette, memory source and other parameters. It also lets you chain regions together in a list.

You can either choose to initialize one of more regions before spawning the driver and passing them into initDisplay or you can apply a region to a display later. If a region is not applied to a display when the driver is created the driver will still generate a video signal but the display will remain blank (in the border colour) until a region is linked to the display, allowing a clean startup to a blank screen.

The P2 PLL timing and other values needed for a given resolution can be determined by calling ***getTiming***. By default a VGA (640x480) resolution will be used, or something applicable to the selected TV output modes such as NTSC or PAL SDTV resolution.

initRegion(…) - sets up a new display region
initDisplay(…) - sets up the display and starts the driver
getTiming(resolution) - retrieves the timing structure for a given resolution
shutdown(display) - shuts down the driver generating a given display

Display related methods

These API's control what is present on the entire display and manage the displayed list of regions.

setDisplayRegions(display, region) - select the region display list
setDisplayBorderSizes(display, top, bottom, width) - setup top/bottom/side borders
setDisplayBorderColour(display, RGBcolour) - setup border colour/width
setYIQ(display, Y, I, Q) - setup P2 colour space converter parameters
linkRegion(region1, region2) - link two regions together in a display list
getRegion(display, regionOffset) - get a region in a display list by its position in the list from the head
getNextRegion(region) - traverse the region display list

Reporting methods

The state of the display output can be read back with a range of reporting functions, and you can use this to help synchronize client code with different portions of the display being generated:

getScanLine(display) - get the current active scan line being output on a display
getFieldCount(display) - get a counter value updated once per field for a display
getBlankingState(display) - gets the state of blanking for a display
waitForVsync(display) - waits until the start of the *next* vsync interval is indicated
waitForBlanking(display) - waits until the start of the *next* blanking interval is indicated
waitForBackPorch(display) - waits until the start of the *next* back porch is indicated
isTextMonoOnly(display) - reports if driver is capable of coloured text, or monochrome only

The following reporting methods help collect information about the screen dimensions and some memory related region settings.

getActiveLines(display) - returns the number of active scan lines in the display
getActivePixels(display) - returns the number of active pixels on each scan line in the display
getCurrentRows(display, region) - get the number of rows in a region based on font size/height
getCurrentColumns(display, region) - get the number of region columns based on pixel width
getFontHeight(region) - get height of a region's current font in scan lines
getBitDepth(region) - get the number of memory bits per pixel used by a region's colour mode

Region configuration related methods

All the per region settings can be modified or read back using the following methods.

setFlags(region, flags) - setup control flags in a region
getFlags(region) - get control flags applied to a region
setSize(region, size) - setup the size of a region
getSize(region) - get the current size of a region
setMode(region, mode) - set the mode of a region
getMode(region) - get the current mode of a region
setSource(region, address) - setup the source data address used for a region
getSource(region) - get current source of the data used for a region
setWrapSource(region, address) - setup region data source if region wraps
getWrapSource(region) - get region data source if region wraps
setAltSource(region, address) - set alternate field source data address
getAltSource(region) - get alternate field source data address
setAltWrapSource(region, address) - get source address of alternate field if wrapping occurs
getAltWrapSource(region) - get alternate field source data address if wrapping occurs
setWrap(region, firstwrap, rewrap) - setup wrap and rewrap scan line settings applied to a region
getWrap(region) - get wrap settings currently applied to a region
setSkew(region, skew) - setup the skew/pitch applied to a region
getSkew(region) - read skew/pitch applied to a region
setFont(region, fontaddress, fontsize) - setup font used by a text region
getFont(region, font) - get font address in use by a region
setPalette(region, paletteAddress) - setup palette in use by a region

setTextScanStart(region, startscan) - setup a scan line offset for the first row of text in a region
setTextFlash(region, enable) - controls text flashing mode
setDoubleWide(region, enable) - controls double width text
setDoubleHigh(region, enable) - controls double height text

Mouse sprite related methods

An optional mouse sprite (selectable per region or a global one) can be controlled with the following methods:

setRegionMouse(region, x, y) - sets a region mouse's X, Y pixel co-ordinates
setGlobalMouse(display, x, y) - sets the global mouse's X, Y pixel co-ordinates
setMouseImage(region, image, x, y) - setup mouse sprite image address and hotspot pixel offset
showMouse(region, regionMouse) - displays a mouse sprite in a region (as region/global mouse)
hideMouse(region) - hide the mouse sprite in a region

Text cursor related methods

Each text region supports two independent cursors managed entirely by the driver. They are indexed by a cursor ID value as either 0, or 1. These text cursors can be controlled with the following methods:

setCursor(…) - sets up a cursor's attributes for use including colour, position and style
showCursor(region, cursorid) - show a cursor on the screen if is within the region's bounds
hideCursor(region, cursorid) - hide the cursor from view
setCursorPos(region, cursorid, row, column) - set the cursor's row and column position
setCursorFlags(region, cursorid, flags) - setup the cursor's flags
getCursorPos(region, cursorid) - get the cursor's row and column position
getCursorRow(region, cursorid) - get the cursor's row only
getCursorCol(region, cursorid) - get the cursor's column only
setCursorColour(region, cursorid, colour) - setup cursor's colour
setCursorColours(region, cursorid, colour1, colour2) - setup a two colour cursor
setCursorHeight(region, cursorid, height) - setup a cursor's height in scan lines

Text output related methods

The driver uses the concept of a text context to determine where the printed text will go. Each Cog gets its own currently active text context maintained by the driver. A context links to a given display and region for output and maintains the current printing foreground and background colours, and one of the region cursor's positions. Each Cog can then print text in its own text region independent of the other Cogs at it's own position on screen.

Before any text is generated you first need to call ***initTextOutput.*** This will fill in a context structure in HUB RAM that holds the data associated with the context provided by the caller. If you wish to change to a different display and/or region you can preserve the original and create a new context. You can then flip between several different contexts using ***setTextOutput***. The currently associated context for a Cog can be obtained by calling ***getTextOutput***.

initTextOutput(context, display, region, fg, bg, hwcursor, clearscreen) - setup a text printing context
setTextOutput(context, singleCursor) - change to new printing context and if a single cursor follows it
getTextOutput() - get current printing context
setTextColours(fgcolour, bgcolour) - setup foreground and background colours to use for printing
getTextColours() - get current foreground and background printing colour
setTextPos(curs_row, curs_col) - set cursor position in current printing context
getTextPos() - get cursor position in current printing context

One the context is setup then text output can then be directed to the associated region using the following methods either by calling them directly or by assigning SPIN2's SEND function to the address of ***out, tx, tx16, tx32, txraw*** or ***print*** as a method pointer:

out(char) - output a single character and interpret control characters
tx(char) - alias for out
txraw(char) - output a single character raw without BS/TAB/LF/FF/CR handling
tx16(char) - experimental 16 pixel wide font characters
tx32(char) - experimental 32 pixel wide font characters
nl() - output a new line and scrolls the displayed region as required
printStr(str) - display a string of text characters
printStr16(str) - prints a string using 16 pixel wide font characters
printStr32(str) - prints a string using 32 pixel wide font characters
print(char_str) - prints a single char or string (char if value is less than 256, string address otherwise)
printLn(char_str) - prints a single char or string followed by newline
dec(value) - print signal decimal value
hex(value, digits) - print hexadecimal value digits
bin(value, digits) - print binary value digits
clear() - clears out the entire text region used by printing context with the background colour

Other output specific APIs

For HDMI:

initHdmiConfig(…) - initialize HDMI specific configuration data for initDisplay to use
setHdmiVolumePercent(display, left, right) - sets volume level using logarithmic ramp (0-100)
setHdmiVolume(display, left, right) - sets absolute volume multipliers per channel (0-$4000)
fadeHdmiVolume(display, initial, final, durationMs) - fades the HDMI volume over time
setSampleRate(display, rate) - changes HDMI output rate (FIFO mode)

For digital RGB/LCD:

initRgbPinInfo(…) - initialize extra RGB pin data parameters for initDisplay to use
setBacklight(display, pwm_duty) - changes backlight PWM pin duty cycle for LCD displays

Advanced features

While this video driver has many capabilities there are some particular built in features that are worth explaining further and identifying their potential uses.

Multiple Regions

In many cases a displayed screen with just a single region is sufficient, however having multiple regions can become useful as it allows the screen contents to be altered dynamically without impacting client applications which continue to write to the same screen addresses. For example a UI control Cog can use it display an online help screen or a debug/status region over the top of a portion of the screen on demand and a secondary Cog's application can continue presenting data unaffected. Nothing has to be modified within the secondary Cog's application in order for that to work. Having multiple regions allows these popup regions to come and go without affecting memory layout of other regions. This is particularly useful when independent Cogs want to write to different portions of the display and you'd otherwise have to co-ordinate a feature like this with different Cogs.

You could also use this feature to maintain one or two lines of a text region at the bottom of the screen for a command line to interact with, without building a full GUI while the rest of the screen can remain in graphics output mode displaying graphical content for example. Or you could quickly flip entire screens between two different applications requiring a screen output with a single update to the display list. Nothing needs to be redrawn if each application uses its own dedicated memory area for its region(s).

Interlaced Source

The driver can be configured to automatically flip source address for a region's video data after each video field occurs in time. This enables a couple of possible applications:

When interlaced video output is being generated it lets a common frame buffer which is being displayed fully once per frame to be interleaved into two different fields by setting a skew value between scan lines equal to one extra scan line width in memory. No need for the two buffers to be managed separately in memory, the entire frame in can remain contiguous which simplifies write access into it for graphics drawing for example.

For fixed output rate applications such as some real-time games, it can also automatically swap screen buffers for you, no need to wait to update the source address manually at the correct time (although you still can). As long as you render the next frame in time the flip will happen on its own automatically, providing very smooth animations at the field/frame rate. This way you can work on one frame while the other is being output.

Skew

The display can be configured with skew added per scan line in any region. This lets you setup wider than currently displayed frame buffers, which are useful for rapid horizontal scrolling into other off-screen regions. It's only a finite scroll effect however, and dependent on how wide you make

your scan line offsets in memory. Nonetheless it can be extremely useful. It also works in text regions which might be helpful for editors that wish to scroll sideways showing wider lines that were off screen.

Negative skews allow graphics buffers to be read in a reverse order from a high memory to low memory - can be useful for some reverse bitmap formats or to flip the screen upside down. You can also set it to the negative scan line size in memory to have each scan line display the same contents as the previous line for a particular rapid fill/clear effect which could be useful if the actual clear is still taking place at some slower rate, improving apparent responsiveness.

Wrap and rewrap

A region can be configured to jump to some different source address to obtain its data once some number of scan lines have been generated in the region. A region always begins with data sourced from the address in SCREEN_BUFFER_1/2. However if a non-zero wrap value is configured the driver can wrap back to another address identified in SCREEN_BUFFER_2/4 for scrolling purposes after that many scan lines are generated. The other rewrap parameter is the number of scan lines that are output *after* the first/next wrap before it again jumps to the same wrap source address identified in SCREEN_BUFFER_2/4. This cycle continues on until the region ends.

Having this can be useful for sprite drivers that render into a small number of scan line buffers, (e.g. 4) and you want to scroll one line at a time so you can begin the region displaying any scan line in the rendered group. An example follows:

Normally you might want to display some scan lines in a region from data sourced at rendered scan line buffer numbers 0,1,2,3,0,1,2,3,0,1,2…and repeating like this.

Then after some scrolling operation you now want to offset by one scan line and output content in this order from the start of the region: 1,2,3,0,1,2,3,0,1,2,3…and repeating like this.

To achieve the first case you can setup wrap and rewrap parameters both to 4 scan lines each and have SCREEN_BUFFER_1 and SCREEN_BUFFER_2, both pointing to data at an address for scan line buffer number 0.

To do the second case, you setup the wrap to 3 scan lines and rewrap to 4 scan lines and setup SCREEN_BUFFER_1 to point to scan line buffer number 1 data, keep SCREEN_BUFFER_2 pointing back to scan line 0 data.

Transparent Mode

This mode sends data directly from the source data buffer address identified in a region to the pins without any local read into a temporary scan line buffer first. This helps lowers the clocking requirements of the P2 down in low powered graphics applications but it would prevent pixel doubling for DVI/HDMI which requires software based doubling solutions using the temporary scan line buffers and it cannot be done for external memory based graphics regions or text regions.

Fine Text Scrolling

The video driver has an the ability to offset the first row of a text region by some number of scan lines. By making use of this it allows the text to be finely scrolled per scan line instead of per row. This can give a smother scroll effect when updated synchronously with vertical sync. It does require some extra client intervention to co-ordinate this however.

Larger Font Widths

By splitting a 8 bit font defined for 256 characters in half or one quarter, then wider 16 or 32 pixel fonts can be displayed, which can be useful if you want larger sized characters on screen for titles, banner displays etc. Doing this will sacrifice the number of simultaneous characters possible per region from 256 down to 128 or 64 character glyphs. Having 128 is still useful for ASCII font, but 64 will require some sacrifices, eg. remove lower case.

Monochrome Text Output

Text is fairly resource intensive to implement in colour so during startup the driver examines whether the P2 clock and pixel rate timing allows for colour text to be rendered in time for each scan line. If it can, it will be allowed but if it cannot it will try to fall back to a reduced monochrome mode if possible, one with attribute processing for flashing text and one without for even lower requirements. The final text capability selected by the driver is notified back to the client via the status long. In some applications for consistency of operation having this varying possible of coloured text or not may not be desirable, so there exists the ability to force the driver to consistently operate with monochrome text by using a startup flag passed in PTRA when the COG is spawned.

Region/Global Mouse Sprite

Each region can nominate a mouse sprite to be displayed in it. The sprite can be defined as global (which uses the same global XY co-ordinates from the top left of the screen to bottom right. These co-ordinates are updated in the display parameters. Or it can be defined as a region specific sprite in which case it will use XY co-ordinates relative to the start of the region.

The idea here is for multiple region applications you may want the mouse sprite constrained within a single region, or you may want it to span all the regions. Each region can also have a different colour mode/bit depth so each region has its own settings for the image sprite data which needs to be selectable according to the mode used by the region. Having different colour modes and bit depths with multiple regions using a global mouse is possible, but the mouse colour may change as it travels across a region boundary depending on palette and selected colour modes etc.

Dual cursors

With two seperate cursors per region, one can be used for showing the current text editing/output position and can automatically update during printing while the second one can be use for other purposes. One application of the second cursor is a block based mouse cursor as used by some text user interfaces prior to full GUIs, or as a way to draw your attention to something else in the region.

External memory support

With a limited amount of HUB RAM present on the P2, not all resolutions possible will have sufficient room to store all their video data in HUB RAM, especially at high colour depth like 16bpp and 24bpp. The driver has been developed to support external memory drivers via a mailbox scheme that follows the P2 memory driver document, released separately. This allows the video driver to access much larger amounts of memory - 16-32MB for example with HyperRAM or PSRAM add on devices. At higher P2 speeds this allows full 1080p60 at 8 bits per pixel over VGA or component HDTV, or 1024x768 at 24 bits per pixel for example, neither of which are remotely possible with full frame buffers when using HUB RAM alone.

To achieve rock solid artefact free video to be sustained from the external RAMs care must be taking when setting up quality of service (QoS) parameters for memory drivers. This includes making the video driver the highest priority served Cog with locked bursts, as well as limiting the burst size allowed for lower priority Cogs that share the same memory. Doing this will ensure that the video driver doesn't have to wait a long time for its chance to read the scan line data it requires from external memory, and when multiple Cogs also have requests pending along with the video driver after some Cog completes its transfer the high priority video driver will always go first.

Setting the optimal maximum burst size for lower priority Cogs depends on the horizontal scan line frequency and video driver's own pixel data rate requirements per scan line, and how much spare time exists at the end of the scan line once the video driver has performed its own read. Setting a burst size too high would cause problems for the video driver, while setting the burst too low is less efficient in some cases when transferring larger amounts of data.

COGATN notifications

The driver has a few other capabilities regarding synchronization.

Firstly, it can be setup to optionally wait for a notification at startup time after it configures everything before it starts generating video. This may allow multiple video drivers to be spawned sequentially and then start all at once which may be useful if you want them to share a common display structure, or it some other work is needed before it makes sense to begin the video output, such as starting an external memory driver only after the PLL has been changed to run at a new clock rate.

Secondly it can generate a COGATN notification when it begins, this may be useful to synchronize some other COGs so they get some reference pulse when the video actually begins and can remain locked onto the output pixel rate by simple computations based on the reading the local P2 counter periodically.

Finally the COG can be configured to output notifications via COGATN to a group of COGs interested in accurately tracking what part of the frame or field is being output without continuously polling the status an possibly incurring hub window jitter. This notification can be setup to happen only when the vertical sync occurs, in order to trigger new frame rendering or to wait until it is safe to update certain parameters in the region or display structure with new values. Alternatively it can be setup to happen on every scan line (during horizontal sync) so a group of sprite driver Cogs could know when another scan line has completed being sent and can start then next scan line's render process.