



Application Note

BRT_AN_033

BT81X Series Programming Guide

Version 2.8

Issue Date: 20-02-2026

This application note describes the process and practice required to program BT81X Series, (BT815/6 and **BT817/8** chips).

Table of Contents

1	Introduction	11
1.1	Scope	11
1.2	Intended Audience.....	11
1.3	Conventions	11
1.4	API Reference Definitions	11
1.5	What's new in BT81X Series?.....	12
1.6	What is new in BT817/8?	13
2	Programming Model	14
2.1	Address Space	14
2.2	Data Flow Diagram	15
2.3	Read Chip Identification Code (ID).....	16
2.4	Initialization Sequence during Boot Up.....	16
2.5	Power Management	17
2.5.1	Low Power Mode.....	17
2.5.2	Normal Mode.....	18
2.6	PWM Control	18
2.7	RGB Color Signal	19
2.8	Touch Screen	19
2.9	Flash Interface	20
2.10	Audio Routines	21
2.10.1	Sound Effect	21
2.10.2	Audio Playback	21
2.11	Graphics Routines.....	22
2.11.1	Getting Started.....	22
2.11.2	Coordinate Range and Pixel Precision	23
2.11.3	Screen Rotation	23
2.11.4	Drawing Pattern.....	25

2.11.5	Bitmap Transformation Matrix	27
2.11.6	Color and Transparency	27
2.11.7	Performance.....	28
3	Register Description	29
3.1	Graphics Engine Registers	29
3.2	Audio Engine Registers	33
3.3	Flash Registers	35
3.4	Touch Screen Engine Registers	36
3.4.1	Overview	36
3.4.2	Common Registers.....	36
3.4.3	Resistive Touch Engine	38
3.4.4	Capacitive Touch Engine	41
3.4.5	Calibration	45
3.5	Coprocessor Engine Registers.....	45
3.6	Miscellaneous Registers.....	46
3.7	Special Registers	52
4	Display List Commands.....	54
4.1	Graphics State	54
4.2	Command Encoding	54
4.3	Command Groups	55
4.3.1	Setting Graphics State.....	55
4.3.2	Drawing Actions.....	55
4.3.3	Execution Control.....	55
4.4	ALPHA_FUNC	56
4.5	BEGIN	56
4.6	BITMAP_EXT_FORMAT	57
4.7	BITMAP_HANDLE	58
4.8	BITMAP_LAYOUT	59
4.9	BITMAP_LAYOUT_H	63
4.10	BITMAP_SIZE	63

4.11	BITMAP_SIZE_H	64
4.12	BITMAP_SOURCE	65
4.13	BITMAP_SWIZZLE	66
4.14	BITMAP_TRANSFORM_A	68
4.15	BITMAP_TRANSFORM_B	69
4.16	BITMAP_TRANSFORM_C	69
4.17	BITMAP_TRANSFORM_D	70
4.18	BITMAP_TRANSFORM_E	70
4.19	BITMAP_TRANSFORM_F	71
4.20	BLEND_FUNC	72
4.21	CALL	73
4.22	CELL	74
4.23	CLEAR	74
4.24	CLEAR_COLOR_A	75
4.25	CLEAR_COLOR_RGB	75
4.26	CLEAR_STENCIL	76
4.27	CLEAR_TAG	77
4.28	COLOR_A	77
4.29	COLOR_MASK	78
4.30	COLOR_RGB	79
4.31	DISPLAY	80
4.32	END	80
4.33	JUMP	80
4.34	LINE_WIDTH	81
4.35	MACRO	82
4.36	NOP	82
4.37	PALETTE_SOURCE	82
4.38	POINT_SIZE	83
4.39	RESTORE_CONTEXT	84

4.40	RETURN	84
4.41	SAVE_CONTEXT	85
4.42	SCISSOR_SIZE	85
4.43	SCISSOR_XY	86
4.44	STENCIL_FUNC	87
4.45	STENCIL_MASK.....	88
4.46	STENCIL_OP	88
4.47	TAG.....	89
4.48	TAG_MASK.....	90
4.49	VERTEX2F	90
4.50	VERTEX2II	91
4.51	VERTEX_FORMAT	92
4.52	VERTEX_TRANSLATE_X.....	92
4.53	VERTEX_TRANSLATE_Y.....	93
5	Coprocessor Engine	94
5.1	Command FIFO	94
5.2	Widgets	95
5.2.1	Common Physical Dimensions	96
5.2.2	Color Settings.....	96
5.2.3	Caveat.....	96
5.3	Interaction with RAM_DL.....	96
5.3.1	Synchronization between MCU & Coprocessor Engine	97
5.4	ROM and RAM Fonts.....	97
5.4.1	Legacy Font Metrics Block	97
5.4.2	Example to find the width of character.....	98
5.4.3	Extended Font Metrics Block.....	98
5.4.4	ROM Fonts (Built-in Fonts)	100
5.4.5	Using Custom Font.....	101
5.5	Animation support	102
5.6	String Formatting	103

5.6.1	The Flag Characters	104
5.6.2	The Field Width.....	104
5.6.3	The Precision.....	104
5.6.4	The Conversion Specifier	105
5.7	Coprocessor Faults	105
5.8	Coprocessor Graphics State	107
5.9	Parameter OPTION	107
5.10	Resources Utilization	108
5.11	Command list.....	109
5.12	Command Groups	109
5.13	CMD_APILEVEL	111
5.14	CMD_DLSTART	111
5.15	CMD_INTERRUPT	111
5.16	CMD_COLDSTART.....	112
5.17	CMD_SWAP	112
5.18	CMD_APPEND	113
5.19	CMD_REGREAD	113
5.20	CMD_MEMWRITE	114
5.21	CMD_INFLATE.....	115
5.22	CMD_INFLATE2.....	116
5.23	CMD_LOADIMAGE	116
5.24	CMD_MEDIAFIFO	118
5.25	CMD_PLAYVIDEO	119
5.26	CMD_VIDEOSTART	120
5.27	CMD_VIDEOFRAME	120
5.28	CMD_MEMCRC.....	121
5.29	CMD_MEMZERO.....	122
5.30	CMD_MEMSET	122
5.31	CMD_MEMCPY	123
5.32	CMD_BUTTON	123

5.33	CMD_CLOCK	125
5.34	CMD_FG_COLOR	127
5.35	CMD_BG_COLOR	128
5.36	CMD_GRAD_COLOR	128
5.37	CMD_GAUGE	129
5.38	CMD_GRADIENT	133
5.39	CMD_GRADIENTA	134
5.40	CMD_KEYS	135
5.41	CMD_PROGRESS	138
5.42	CMD_SCROLLBAR	139
5.43	CMD_SLIDER	141
5.44	CMD_DIAL	142
5.45	CMD_TOGGLE	144
5.46	CMD_FILLWIDTH	145
5.47	CMD_TEXT	146
5.48	CMD_SETBASE	148
5.49	CMD_NUMBER	149
5.50	CMD_NOP	150
5.51	CMD_LOADIDENTITY	150
5.52	CMD_SETMATRIX	151
5.53	CMD_GETMATRIX	151
5.54	CMD_GETPTR	152
5.55	CMD_GETPROPS	153
5.56	CMD_SCALE	154
5.57	CMD_ROTATE	154
5.58	CMD_ROTATEAROUND	155
5.59	CMD_TRANSLATE	156
5.60	CMD_CALIBRATE	157
5.61	CMD_CALIBRATESUB	158

5.62	CMD_SETROTATE	159
5.63	CMD_SPINNER	159
5.64	CMD_SCREENSAVER.....	161
5.65	CMD_SKETCH	161
5.66	CMD_STOP	163
5.67	CMD_SETFONT	163
5.68	CMD_SETFONT2	164
5.69	CMD_SETSCRATCH	165
5.70	CMD_ROMFONT.....	165
5.71	CMD_RESETFONTS	166
5.72	CMD_TRACK.....	166
5.73	CMD_SNAPSHOT	169
5.74	CMD_SNAPSHOT2	170
5.75	CMD_SETBITMAP	171
5.76	CMD_LOGO	172
5.77	CMD_FLASHERASE	173
5.78	CMD_FLASHWRITE.....	173
5.79	CMD_FLASHPROGRAM	173
5.80	CMD_FLASHREAD.....	174
5.81	CMD_APPENDF.....	174
5.82	CMD_FLASHUPDATE.....	175
5.83	CMD_FLASHDETACH.....	176
5.84	CMD_FLASHATTACH.....	176
5.85	CMD_FLASHFAST	176
5.86	CMD_FLASHSPIDESEL.....	177
5.87	CMD_FLASHSPITX.....	177
5.88	CMD_FLASHSPIRX	177
5.89	CMD_CLEARCACHE	178
5.90	CMD_FLASHSOURCE	178

5.91	CMD_VIDEOSTARTF	179
5.92	CMD_ANIMSTART	179
5.93	CMD_ANIMSTARTRAM	180
5.94	CMD_RUNANIM.....	181
5.95	CMD_ANIMSTOP	182
5.96	CMD_ANIMXY	182
5.97	CMD_ANIMDRAW.....	182
5.98	CMD_ANIMFRAME.....	183
5.99	CMD_ANIMFRAMERAM.....	184
5.100	CMD_SYNC.....	184
5.101	CMD_BITMAP_TRANSFORM	185
5.102	CMD_TESTCARD.....	186
5.103	CMD_WAIT	187
5.104	CMD_NEWLIST.....	187
5.105	CMD_ENDLIST	188
5.106	CMD_CALLLIST	189
5.107	CMD_RETURN	190
5.108	CMD_FONTCACHE	191
5.109	CMD_FONTCACHEQUERY.....	191
5.110	CMD_GETIMAGE.....	192
5.111	CMD_HSF	193
5.112	CMD_PCLKFREQ	194
6	ASTC.....	195
6.1	ASTC RAM Layout.....	195
6.2	ASTC Flash Layout	196
7	DXT1.....	197
7.1	Principle	197
7.2	Set up the bitmap handle of color layer	197
7.3	Set up the bitmap handle of gradient layer	197

7.4	Render the DXT1 bitmap	198
8	Contact Information	199
	Appendix A – References	200
	Document References	200
	Acronyms and Abbreviations.....	200
	Appendix B – List of Tables/ Figures/ Registers/ Code Snippets.....	201
	List of Tables.....	201
	List of Figures	201
	List of Registers	202
	List of Code Snippets	204
	Appendix C – Revision History	206

1 Introduction

This document captures the programming details for the **BT81X** Series chips (BT815/6, BT817/8) including graphics commands, widget commands and configurations to control **BT81X** Series chips for smooth and vibrant screen effects.

The **BT81X** Series chips are graphics controllers with add-on features such as audio playback and touch capabilities. They consist of a rich set of graphics objects that can be used for displaying various menus and screenshots for a range of products including home appliances, toys, industrial machinery, home automation, elevators, and many more.

1.1 Scope

This document will be useful to understand the command set and demonstrate the ease of usage in the examples given for each specific instruction. In addition, it also covers various power modes, audio, and touch features as well as their usage.

The descriptions in this document are applicable to both BT815/6 and **BT817/8**, unless specified otherwise.

Within this document, the endianness of commands, register values, and data in **RAM_G** are in *little-endian* format.

Information on pin settings, hardware characteristics and hardware configurations can be found in the BT815/6 or **BT817/8** data sheet.

1.2 Intended Audience

The intended audience of this document are Software Programmers and System Designers who develop graphical user interface (**GUI**) applications on any processor with an **SPI** master interface.

1.3 Conventions

- All values are decimal by default.
- The values with **0x** are in hexadecimal.
- The values with **0b'** are in binary.
- Host refers to the **MCU/MPU** with SPI master interface connecting with **EVE**.
- Host commands refer to the **EVE** specific commands defined in the Serial Data Protocol section of the datasheet.

1.4 API Reference Definitions

The following table provides the functionality and nomenclature of the APIs used in this document.

wr8()	write 8 bits to intended address location
wr16()	write 16 bits to intended address location
wr32()	write 32 bits to intended address location
rd8()	read 8 bits from intended address location
rd16()	read 16 bits from intended address location
rd32()	read 32 bits from intended address location
cmd()	write 32 bits data to command FIFO i.e., RAM_CMD
cmd_*()	Write 32 bits commands with its necessary parameters to command FIFO i.e., RAM_CMD .
dl()	Write 32 bits display list command to RAM_DL .
host_command()	send host command in host command protocol

Table 1 - API Reference Definitions

1.5 What's new in BT81X Series?

Compared with the previous **FT81X** generation, the **BT81X** series introduces a range of enhanced and expanded features:

- ❖ QSPI NOR flash interface
- ❖ Adaptive Scalable Texture Compression (ASTC) format bitmap
- ❖ Unicode text display
- ❖ Animation support

The table below outline the new and updated BT81X commands associated with these features:

Coprocessor Commands	BT81X	Remarks
<i>CMD_ANIMDRAW</i> <i>CMD_ANIMFRAME</i> <i>CMD_ANIMSTART</i> <i>CMD_ANIMSTOP</i> <i>CMD_ANIMXY</i>	New	Animation features related to coprocessor commands
<i>CMD_APPENDF</i>	New	Append flash data to display list
<i>CMD_CLEARCACHE</i>	New	Clear the flash cache
<i>CMD_FLASHATTACH</i> <i>CMD_FLASHDETACH</i> <i>CMD_FLASHERASE</i> <i>CMD_FLASHFAST</i> <i>CMD_FLASHREAD</i> <i>CMD_FLASHSOURCE</i> <i>CMD_FLASHSPIDESEL</i> <i>CMD_FLASHSPIRX</i> <i>CMD_FLASHSPITX</i> <i>CMD_FLASHUPDATE</i> <i>CMD_FLASHWRITE</i>	New	Flash interface operation related to coprocessor commands
<i>CMD_FILLWIDTH</i>	New	Set the line width for the text of <i>cmd_text</i> and <i>cmd_button</i>
<i>CMD_GRADIENTA</i>	New	Draw a smooth color gradient with transparency
<i>CMD_INFLATE2</i>	New	Decompress data into memory with more options: OPT_FLASH, OPT_MEDIAFIFO
<i>CMD_RESETFONTS</i>	New	Loads a ROM font into a bitmap handle
<i>CMD_ROTATEAROUND</i>	New	Apply a rotation and scale around (x,y) for bitmap
<i>CMD_VIDEOSTARTF</i>	New	Initialize video frame decoder for the data in flash memory
<i>CMD_TEXT</i> <i>CMD_BUTTON</i> <i>CMD_TOGGLE</i>	Changed	Added option : OPT_FORMAT
<i>CMD_LOADIMAGE</i> <i>CMD_PLAYVIDEO</i> <i>CMD_VIDEOSTARTF</i>	Changed	Supports data stored in flash
Display List	BT81X	Remarks
<i>BITMAP_SOURCE</i>	Changed	Expand the address bit field to access ASTC bitmap in flash
<i>BITMAP_TRANSFORM_A</i> <i>BITMAP_TRANSFORM_B</i> <i>BITMAP_TRANSFORM_D</i> <i>BITMAP_TRANSFORM_E</i>	Changed	Added new precision control: signed fixed point 1.15
<i>BITMAP_LAYOUT</i>	Changed	Added a new valid value for format parameter: GLFORMAT
<i>BITMAP_EXT_FORMAT</i>	New	Support more bitmap formats, especially ASTC compression formats
<i>BITMAP_SWIZZLE</i>	New	Set the source for the r,g,b,a channels of a bitmap

Table 2 - Updated Commands in BT81X

1.6 What is new in BT817/8?

BT817/8 maintains backward compatibility with the BT815/6 ICs. Applications developed for the BT815/6 can run on the BT817/8 series without modification. In essence, BT817/8 is an enhanced version of BT815/6.

The BT817/8 delivers up to a **1.5x** boost in graphics engine performance over the BT815/6, along with a host new and enhanced features, including:

- ❖ Programmable timing for HSYNC and VSYNC, enabling compatibility with a wide range of displays
- ❖ Addition of a horizontal scan-out filter to support non-square-pixel LCD displays
- ❖ Adaptive Hsync mode that delays the start of the scan-out line while maintaining continuous PCLK operation
- ❖ Supports animation stored in **RAM_G**
- ❖ Enables construction of the command list in **RAM_G**
- ❖ New font caching mechanism for custom fonts with glyphs stored in flash

To support the features above, new registers and commands have been introduced for the **BT817/8**. These are indicated in this document as "**BT817/8** specific".

Furthermore, **BT817/8** introduces enhancements to the following BT815/6 commands:

- **CMD_GETPTR**
- **CMD_GETPROPS**

However, these two commands retain their original functionality for compatibility, unless **CMD_APILEVEL** is sent with parameter level 2.

2 Programming Model

EVE appears to the host MCU as a memory-mapped SPI device, with all commands and data communicated through the serial protocol defined in the datasheet.

2.1 Address Space

All memory and registers are mapped into a 22-bit address space with a 2-bit SPI command prefix: 0'b00 for read and 0'b10 for write, 0'b01 reserved for Host Commands, and 0'b11 undefined. Refer to the datasheet for details on the serial data protocol used to access these addresses. Memory space definitions are provided in the following table:

Name	Start Address	End Address	Size (bytes)	Description
RAM_G	0x000000	0x0FFFFFF	1024 Ki	General purpose graphics RAM , also called main memory in this document
ROM_FONT	0x1E0000	0x2FFFFFFB	1152 Ki	Font table and bitmap
ROM_FONTROOT	0x2FFFFC	0x2FFFFFF	4	Font table pointer address
RAM_DL	0x300000	0x301FFF	8 Ki	Display list RAM
RAM_REG	0x302000	0x302FFF	4 Ki	Registers
RAM_CMD	0x308000	0x308FFF	4 Ki	Command FIFO
RAM_ERR_REPORT	0x309800	0x3098FF	128	Coprocessor fault report area
Flash memory	0x800000	Depending on attached flash chip, up to 0x107FFFFFF	Up to 256Mi	External NOR flash memory. It can NOT be addressed by host directly. The address is used by the following commands only for rendering ASTC image only: CMD_SETBITMAP BITMAP_SOURCE

Table 3 – Memory Map

Note:

1. Addresses beyond those listed in this table are reserved and must not be read from or written to unless explicitly specified.
2. To access flash memory, host needs leverage the coprocessor commands, such as
 - **CMD_FLASHREAD**
 - **CMD_FLASHWRITE**
 - **CMD_FLASHUPDATE**
 -

These commands use **zero**-based addressing to access the flash memory blocks. See [Flash Interface](#) for more details.

2.2 Data Flow Diagram

Figure 1 illustrates the data flow between 1) **external components** (MCU and Flash) and 2) **internal components of EVE**. Note that direct writes from the **MCU** to **RAM_DL** require careful synchronization of the read/write pointers **EVE's** registers, as the coprocessor engine may simultaneously write generated display list commands into **RAM_DL**.

To simplify this process, it is recommended to write display list commands to **RAM_CMD**, allowing the coprocessor to update **RAM_DL** automatically.

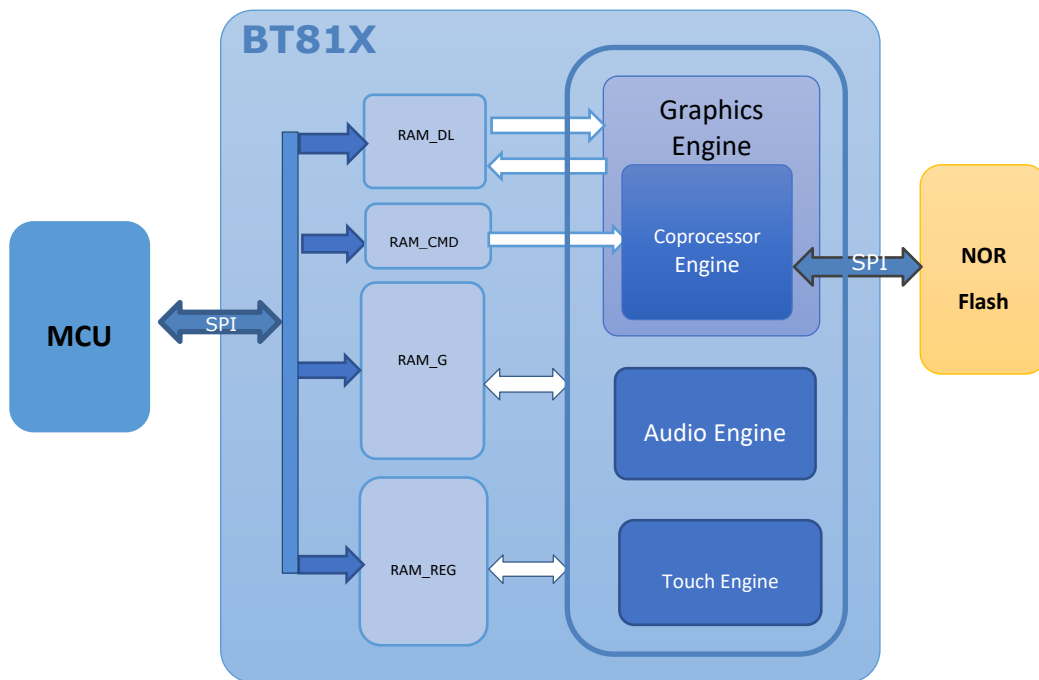


Figure 1 – BT81X data flow

The data in this context refers to the following items:

- **Display list:** Instructions for the graphics engine to render the screen
- **Coprocessor command:** Predefined commands executed by the coprocessor engine
- **Bitmap data:** Pixel data in EVE-supported formats, such as RGB565 or ASTC.
- **JPEG/PNG stream:** Image data in JPEG or PNG format, conforming to Eve requirements for co-process or decoding.
- **MJPEG stream:** Video data in MJPEG format, compatible with EVE and decoded by the coprocessor engine.
- **Audio stream:** Audio samples encoded in uLaw, ADPCM, or PCM for the audio engine to decode
- **Flash image:** Data to be programmed to or read from flash memory.
- **Register values:** Values read from or written to EVE registers.

2.3 Read Chip Identification Code (ID)

After a reset or reboot, the chip ID can be read from addresses 0xC0000 to 0xC0003.

To read the chip identification code, users shall read 4 bytes of data from address 0xC0000 before the application overwrites this **RAM_G** location.

The table below specifies the data to be read:

0xC0003	0xC0002	0xC0001	0xC0000
0x00	0x01	0x15 for BT815 0x16 for BT816 0x17 for BT817 0x18 for BT818	0x08

Table 4 – Read Chip Identification Code

2.4 Initialization Sequence during Boot Up

During EVE initialization, the following steps should be performed:

1. Send host command "**CLKEXT**" if the PLL input is sourced from an external crystal oscillator or clock.
2. Send host command "**CLKSEL**" to select a system clock frequency if a non-default clock is required.
By default, the system clock is set to 60MHz; however, 72MHz is recommended for optimal performance.
3. Send host command "**RST_PULSE**" to reset the **EVE** core.
4. Send host command "**ACTIVE**" to enable the clock. The self-diagnosis process may take up to 300 milliseconds, so a delay of 40 ~ 300 ms is recommended..
5. Read **REG_ID** repeatedly until the value **0x7C** is returned.
6. Read **REG_CPURESET** until **EVE** enters working status (returns zero).
7. Configure the display control timing registers, excluding **REG_PCLK**.
8. Write the first display list to **RAM_DL**.
9. Write to **REG_DLSWAP** to start the graphics engine rendering the first display list
10. Enable backlight control for the display panel.
11. Write to **REG_PCLK** to configure the display panel PCLK frequency, which initiates the output of the first display list.

```

host_command(CLKEXT); //send command "CLKEXT" to use the external clock source
host_command(CLKSEL); // Choose the system clock frequency, with an assumed value of 60MHz.
host_command(RST_PULSE); //send host command "RST_PULSE" to reset
host_command(ACTIVE); //send host command "ACTIVE" to wake up
delay(300ms);

while (0x7C != rd8(REG_ID));
while (0x0 != rd16(REG_CPURESET)); //Check if EVE is in working status.

wr32(REG_FREQUENCY, 0x3938700); //Configure the system clock to 60MHz.

/* Configure display registers - demonstration for WVGA 800x480 resolution */
wr16(REG_HCYCLE, 928);
wr16(REG_HOFFSET, 88);
wr16(REG_HSYNC0, 0);
wr16(REG_HSYNC1, 48);
wr16(REG_VCYCLE, 525);
wr16(REG_VOFFSET, 32);
wr16(REG_VSYNC0, 0);
wr16(REG_VSYNC1, 3);
wr8(REG_SWIZZLE, 0);
wr8(REG_PCLK_POL, 1);
wr8(REG_CSPREAD, 0);
wr16(REG_HSIZE, 800);
wr16(REG_VSIZE, 480);
  
```

```
/* Write first display list to display list memory RAM_DL*/
wr32(RAM_DL+0,CLEAR_COLOR_RGB(0,0,0));
wr32(RAM_DL+4,CLEAR(1,1,1));
int offset = 8;
for (int i=0; i < 16; i++)
{
    wr32(RAM_DL+offset,BITMAP_HANDLE(i));
    offset += 4;
    wr32(RAM_DL+offset,BITMAP_LAYOUT_H(0));
    offset += 4;
    wr32(RAM_DL+offset,BITMAP_SIZE_H(0));
    offset += 4;
}
wr32(RAM_DL+offset,DISPLAY());

wr8(REG_DLSWAP, DLSWAP_FRAME); //display list swap

/* Enable backlight of display panel */
#ifdef FT81X_ENABLE
    wr16(REG_GPIOX_DIR, 0xffff);
    wr16(REG_GPIOX, 0xffff);
#else
    wr8(REG_GPIO_DIR, 0xff);
    wr8(REG_GPIO, 0xff);
#endif

wr8(REG_PCLK, 2); //Configure the PCLK divisor to 2, i.e. PCLK = System CLK/2
```

Code Snippet 1 – Initialization Sequence

Note:

1. During initialization, keep the SPI clock frequency below 11MHz. After initialization, it can be increased up to 30MHz, but must not exceed *half* of the system clock frequency in single **SPI** mode. In Quad SPI mode, the maximum SPI frequency is 25MHz, again not exceeding *half* of the system clock frequency.
2. Upon boot, the bitmap handle parameters "**BITMAP_LAYOUT_H** and **BITMAP_SIZE_H**" may contain non-zero values, which can corrupt rendered bitmaps, especially if the bitmap size is less than 512 pixels. To prevent this, set these parameters to zero for bitmap handles 0-15 in the initial display list.
3. For additional details on host_command, refer to the datasheet.

2.5 Power Management

2.5.1 Low Power Mode

Low-power modes (STANDBY, SLEEP, POWERDOWN) can be activated from the ACTIVE state by sending the relevant host command.

To further minimize power usage in SLEEP state, the following registers should be configured.

```
wr16(REG_HSYNC1, 0);
wr16(REG_VSYNC1, 0);
wr8(REG_PCLK_POL, 0);
wr8(REG_PCLK, 0);
host_command(SLEEP); //send host command "SLEEP" to go to SLEEP
```

Code Snippet 2 - Setting before SLEEP

There are two ways to enter the POWERDOWN state: the host can send the "**POWERDOWN**" command to transition **EVE** from the ACTIVE state, or the host can drive the **PD_N** pin low, which forces **EVE** into POWERDOWN mode regardless of its current state.

2.5.2 Normal Mode

In the ACTIVE state, **EVE** operates normally. Waking EVE from other states into ACTIVE requires specific initialization sequences for each state.

- **Initialization Sequence from POWERDOWN (PD_N Pin Method)**

If **EVE** entered POWERDOWN state via the PD_N pin:

1. Drive the **PD_N** pin high
2. Send the host command "**ACTIVE**" to enable the system clock
3. Wait for at least 20ms
4. Send the host command "**SLEEP**"
5. Execute the "Boot-Up Initialization Sequence"

- **Initialization Sequence from POWERDOWN without PD_N Pin (Host Command Method)**

If **EVE** entered POWERDOWN state via the "**POWERDOWN**" host command:

1. Send the host command "**ACTIVE**" to enable the system clock
2. Wait for at least 20ms
3. Send the host command "**SLEEP**"
4. Execute the "Boot-up Initialization Sequence"

- **Initialization Sequence from SLEEP Mode**

1. Send the host command "**ACTIVE**" to enable the system clock
2. Wait for at least 20ms
3. Resume all registers that were configured prior to entering SLEEP

```
wr16(REG_HSYNC1, 48);  
wr16(REG_VSYNC1, 3);  
wr8(REG_PCLK_POL, 1);  
wr8(REG_PCLK, 2);
```

Code Snippet 3 - Resume registers after SLEEP

4. Resume writing display lists

- **Initialization Sequence from STANDBY Mode**

Send the host command "**ACTIVE**" to enable the system clock, after which display list updates can resume.

2.6 PWM Control

The PWM output is controlled by two registers: **REG_PWM_HZ** and **REG_PWM_DUTY**.

REG_PWM_HZ sets the PWM output frequency.

REG_PWM_DUTY sets the PWM output duty cycle.

2.7 RGB Color Signal

The RGB color signal is transmitted over 24 lines, with 8 bits each for red, green, and blue. Several registers influence the operation of these signals. The processing order in the display output system is as follows:

REG_DITHER enables color dithering. **EVE** applies a 2×2 color dither matrix to output pixels to improve the image quality. This enhances half-tone appearance, even on 1-bit displays.

REG_OUTBITS specifies the bit width of each color channel. By default, each channel uses 8 bits (value 0). Lower values reduce the number of output bits per channel and also affect the dither computation.

REG_SWIZZLE controls the arrangement of the output color pins to accommodate PCB routing with different LCD panel layouts. Bit 0 reverses the order of bits within each color channel. Bits 1-3 configure the RGB channel order. Bit 1 swaps the R and B channels. Bit 3 enables rotation; if set, the (R, G, B) channels rotate right when bit 2 = 1, or left when bit 2 = 0. For further details, refer to the BT817/8 datasheet.

2.8 Touch Screen

The raw (x, y) touch values are stored in **REG_TOUCH_RAW_XY** and range from 0-1023. If the touch screen is not pressed, both values read as 0xFFFF.

These Raw touch values are converted to screen coordinates using the transformation matrix in registers **REG_TOUCH_TRANSFORM_A-F**. The resulting coordinates are available in **REG_TOUCH_SCREEN_XY**. If no touch screen is detected, both registers read 0x8000 (-32768). Transformation values can be obtained via an on-screen calibration process. When the screen is touched, the screen coordinates are used to look up an 8-bit tag in the screen's tag buffer, stored in **REG_TOUCH_TAG**. Because tag lookup occurs once per frame and touch coordinates change continuously, the original (x, y) used for the lookup is available in **REG_TOUCH_TAG_XY**.

The Screen touch pressure is measured in **REG_TOUCH_RZ**, which reports resistance. Lower values indicate more pressure. The register's range is 0 (maximum pressure) to 32767 (no touch). The usable range depends on the screen and the touch method (stylus, fingertip, gloved finger, etc.), so calibration is recommended.

REG_TOUCH_MODE controls touch sampling frequency. **TOUCHMODE_CONTINUOUS** is continuous sampling. Writing **TOUCHMODE_ONESHOT** causes a single sample to occur. **TOUCHMODE_FRAME** causes a sample at the start of each frame. **TOUCHMODE_OFF** stops all sampling.

REG_TOUCH_ADC_MODE selects single-ended (**ADC_SINGLE_ENDED**) for lower power or differential (**ADC_DIFFERENTIAL**) for higher accuracy. Single-ended consumes less power, differential gives more accurate positioning. **REG_TOUCH_CHARGE** specifies the duration to drive the touchscreen voltage before sampling. Default value 3000 gives a delay of 0.3ms works for most screens.

REG_TOUCH_RZTHRESH specifies resistance threshold. If the measured touchscreen resistance is greater than this threshold, then no touch is reported. The default value is 65535, all touches are reported.

REG_TOUCH_SETTLE specifies voltage drive time before sampling. For screens with a large capacitance, this value should be increased. For low capacitance screens this value can be decreased to reduce "on" time and save power.

REG_TOUCH_OVERSAMPLE controls the oversampling factor used by the touchscreen system. Increase this value for better noise rejection. Decrease this value for lower power if noise is minimal.

Touch screen 32-bit register updates are atomic: all 32 bits are updated in a single cycle. Reading an XY register, for example, guarantees that both x and y values come from the same sensing cycle. Once the sensing cycle is complete, and the registers have been updated, the **INT_CONV_COMPLETE** interrupt is triggered.

In addition to the high-level registers, raw 10-bit **ADC** readings are available in **REG_TOUCH_DIRECT_XY** and **REG_TOUCH_DIRECT_Z1Z2**. The layout of these registers is as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s		X										Y																			
		Z1										Z2																			

When the S field is 0, a touch is detected and all other fields contain valid sensed values. If S is 1, no touch is detected, and all other fields should be disregarded.

2.9 Flash Interface

EVE provides dedicated registers to access an attached flash chip for high-speed read and write operations. The graphics engine can fetch these graphics assets directly from flash, bypassing the host MCU and significantly reducing its workload in supplying display content.

The register **REG_FLASH_STATUS** reflects the current state of the flash subsystem. During boot, the flash state is **FLASH_STATE_INIT**. Once flash detection is complete, the state transitions either **FLASH_STATE_DETACHED** or **FLASH_STATE_BASIC**, depending on whether a flash device is detected. If no device is found, all SPI output signals are driven low. The host MCU can issue the **CMD_FLASHFAST** command to put the flash system into full-speed mode, setting the state to **FLASH_STATE_FULL**. Direct rendering of ASTC-based bitmaps from flash is only possible in this full-speed state. At any time, the host can call **CMD_FLASHDETACH** to disable the flash communication, which tri-states all flash signals and allows an external MCU to drive the flash directly. While in the detached state, the SPI bus can be controlled using **CMD_FLASHSPIDSEL**, **CMD_FLASHSPITX** and **CMD_FLASHSPIRX**, and communication can be restored with **CMD_FLASHATTACH**. After modifying flash contents, the host MCU should clear the on-chip bitmap cache by calling **CMD_CLEARCACHE** to ensure proper rendering.

Command	DETACHED	BASIC	FULL	Operation
CMD_FLASHERASE		✓	✓	Erase all of flash
CMD_FLASHWRITE		✓	✓	Write data from RAM_CMD to blank flash
CMD_FLASHUPDATE		✓	✓	Read the flash and update to flash if different
CMD_FLASHPROGRAM			✓	Write data from RAM_G to blank flash
CMD_FLASHREAD		✓	✓	Reads data from flash to main memory
CMD_FLASHDETACH		✓	✓	Detach from flash
CMD_FLASHATTACH	✓			Attach to flash
CMD_FLASHFAST		✓		Enter full-speed(fast) mode
CMD_FLASHSPIDSEL	✓			SPI bus: deselect device
CMD_FLASHSPITX	✓			SPI bus: write bytes
CMD_FLASHSPIRX	✓			SPI bus: read bytes

Table 5 – Flash Interface states and commands

To support SPI NOR flash chips from different vendors, the first block 4KB (4096 bytes) of the flash is reserved for the flash driver, provided by **Bridgetek** as a **BLOB** file. This **BLOB** file must be programmed first, as it allows the flash subsystem to operate in full-speed (fast) mode. For more information, refer to the BT81x datasheet.

2.10 Audio Routines

The audio engine provides two main functions: it can synthesize built-in sound effects at selected pitches, and it can play back audio data stored in **RAM_G**.

2.10.1 Sound Effect

The audio engine includes a variety of built-in sound data that allows it to function as a sound synthesizer. The following sample code can be used to play the note C8 on the xylophone:

```
wr8(REG_VOL_SOUND,0xFF); //set the volume to maximum
wr16(REG_SOUND, (0x6C<< 8) | 0x41); // C8 MIDI note on xylophone
wr8(REG_PLAY, 1); // play the sound
```

Code Snippet 4 – Play C8 on the Xylophone

The following sample code can be used to stop sound play:

```
wr16(REG_SOUND,0x0); //configure "silence" sound to be played
wr8(REG_PLAY,1); //play sound
Sound_status = rd8(REG_PLAY); //1-play is going on, 0-play has finished
```

Code snippet 5 – Stop Playing Sound

To prevent a pop sound during reset or power-state changes, the audio engine can first play a "mute" sound and wait for it to finish (i.e., until **REG_PLAY** reads 0). This action sets the audio output pin to zero voltage levels. On reboot, the audio engine then plays an "unmute" sound to restore normal audio output.

```
wr16(REG_SOUND,0x60); //configure "mute" sound to be played
wr8(REG_PLAY,1); //play sound
Sound_status = rd8(REG_PLAY); //1-play is going on, 0-play has finished
```

Code snippet 6 – Avoid Pop Sound

Note: Refer to BT817/8 datasheet for more information on the sound synthesizer and audio playback.

2.10.2 Audio Playback

The audio engine supports audio playback feature for data stored in **RAM_G**. To play back audio, the start address specified in **REG_PLAYBACK_START** must be aligned to an 8-byte boundary. Additionally, the length of the audio data, defined by **REG_PLAYBACK_LENGTH**, must also be multiple of 8 bytes.

The audio engines support three audio formats: 4 bit IMA ADPCM, 8-bit signed PCM, and 8-bit u-Law. For the IMA ADPCM format, the byte order requires attention: each byte contains two 4-bit samples with the first sample occupying bits 0 to 3 and the second sample occupying bits 4 to 7.

To learn how to play back the audio data, please check the sample code below:

```
wr8(REG_VOL_PB,0xFF); //configure audio playback volume
wr32(REG_PLAYBACK_START,0); //configure audio buffer starting address
wr32(REG_PLAYBACK_LENGTH,100*1024); //configure audio buffer length
wr16(REG_PLAYBACK_FREQ,44100); //configure audio sampling frequency
wr8(REG_PLAYBACK_FORMAT,ULAW_SAMPLES); //configure audio format
wr8(REG_PLAYBACK_LOOP,0); //configure once or continuous playback
wr8(REG_PLAYBACK_PLAY,1); //start the audio playback
```

Code Snippet 7 – Audio Playback

```
AudioPlay_Status = rd8(REG_PLAYBACK_PLAY); //1-audio playback is going on, 0-audio playback has finished
```

Code Snippet 8 – Check the status of Audio Playback

```
wr32 (REG_PLAYBACK_LENGTH,0); //configure the playback length to 0
wr8 (REG_PLAYBACK_PLAY,1); //start audio playback
```

Code Snippet 9 – Stop the Audio Playback

2.11 Graphics Routines

This section covers the graphics features and provides several examples. Please note that the included code is intended to illustrate how Display Lists operate. In a typical application, commands are sent through the **command FIFO (RAM_CMD)** rather than being directly to **RAM_DL**.

2.11.1 Getting Started

The following example generates a screen displaying the text "TEXT" along with a red dot.

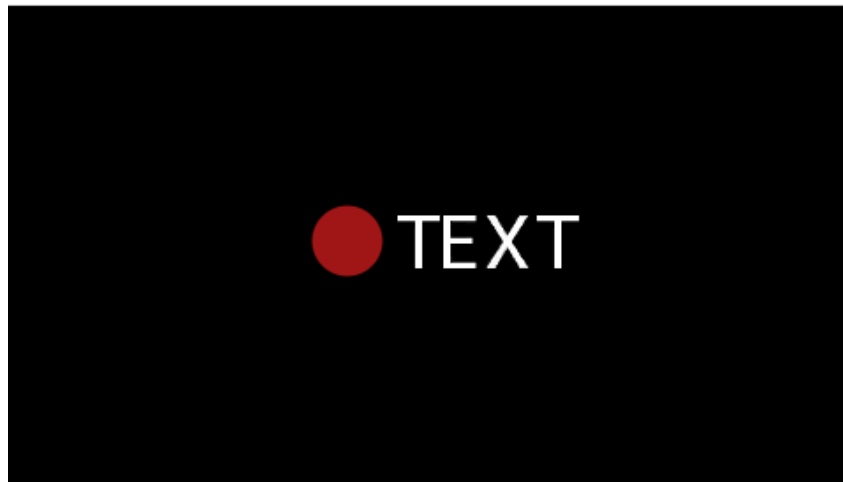


Figure 2 – Getting Started Example

The code to draw the screen is:

```
wr32 (RAM_DL + 0, CLEAR(1, 1, 1)); // clear screen
wr32 (RAM_DL + 4, BEGIN(BITMAPS)); // start drawing bitmaps
wr32 (RAM_DL + 8, VERTEX2II(220, 110, 31, 'T')); // ASCII T in font 31
wr32 (RAM_DL + 12, VERTEX2II(244, 110, 31, 'E')); // ASCII E in font 31
wr32 (RAM_DL + 16, VERTEX2II(270, 110, 31, 'X')); // ASCII X in font 31
wr32 (RAM_DL + 20, VERTEX2II(299, 110, 31, 'T')); // ASCII T in font 31
wr32 (RAM_DL + 24, END());
wr32 (RAM_DL + 28, COLOR_RGB(160, 22, 22)); // change colour to red
wr32 (RAM_DL + 32, POINT_SIZE(320)); // set point size to 20 pixels in radius
wr32 (RAM_DL + 36, BEGIN(POINTS)); // start drawing points
wr32 (RAM_DL + 40, VERTEX2II(192, 133, 0, 0)); // red point
wr32 (RAM_DL + 44, END());
wr32 (RAM_DL + 48, DISPLAY()); // display the image
```

Code Snippet 10 – Getting Started

After the drawing commands are loaded into **RAM_DL**, the register **REG_DLSWAP** must be set to **0x02** to activate the new display list on the next frame refresh.

Note:

- The display list always begins at address **RAM_DL**.
- The address always increments by 4 bytes for each command, since all display list commands are 32 bits wide.
- It is recommended to issue the **CLEAR** command before any other drawing operations to place the graphics engine in a known state. The display list is always terminated with the **DISPLAY** command.

2.11.2 Coordinate Range and Pixel Precision

In addition to single pixel precision, **EVE** supports several fractional pixel precision modes, each resulting in a different coordinate range. Users can trade coordinate range for higher pixel precision as needed. See **VERTEX_FORMAT** for more details.

Please note that the maximum screen resolution **EVE** can render is limited to 2048 x 2048 pixels, regardless of the pixel precision selected.

VERTEX2F and **VERTEX_FORMAT** commands enable drawing operations to span the full coordinate plane. In contrast, the **VERTEX2II** command supports only positive screen coordinates, while **VERTEX2F** command allows negative coordinates. When a bitmap is partially off-screen, for example during screen scrolling, negative screen coordinates must be used.

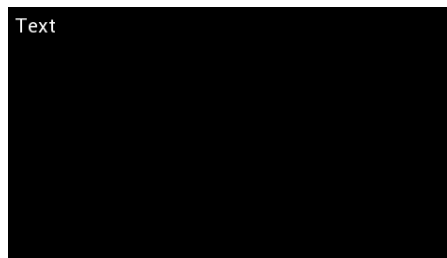
2.11.3 Screen Rotation

The **REG_ROTATE** controls the screen orientation. Changing its value immediately updates the screen's orientation. At the same time, the coordinate system is adjusted so that all display and coprocessor commands operate within the rotated coordinate system.

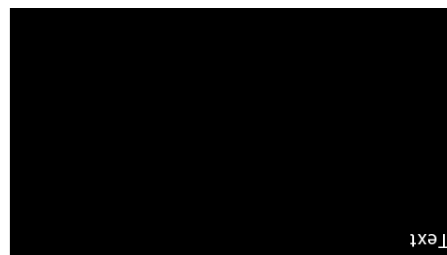
Note: The touch transformation matrix is not affected by setting **REG_ROTATE**.

To adjust the touch screen accordingly, it is recommended to use the [CMD_SETROTATE](#) command rather than directly modifying **REG_ROTATE**.

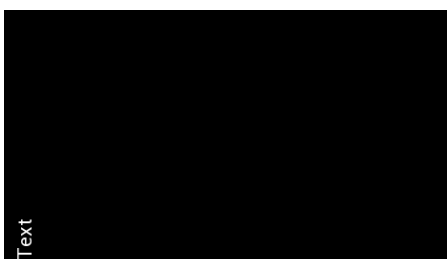
REG_ROTATE = 0 is the default landscape orientation:



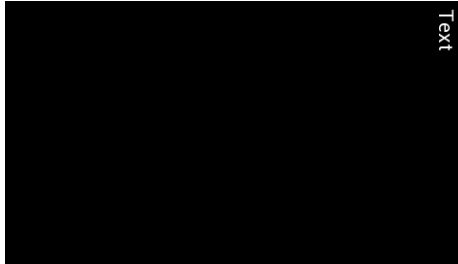
REG_ROTATE = 1 is inverted landscape:



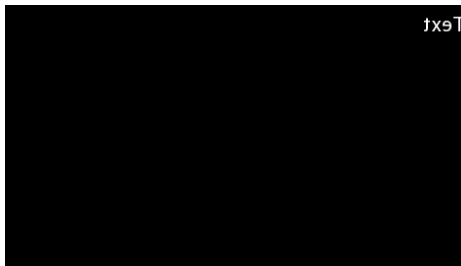
REG_ROTATE = 2 is portrait:



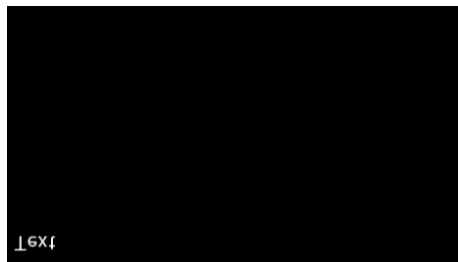
REG_ROTATE = 3 is inverted portrait:



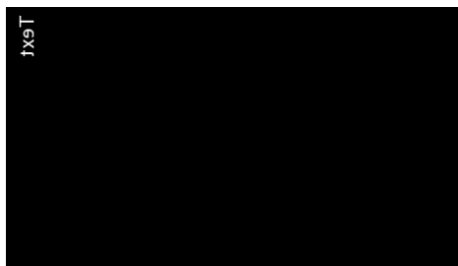
REG_ROTATE = 4 is mirrored landscape:



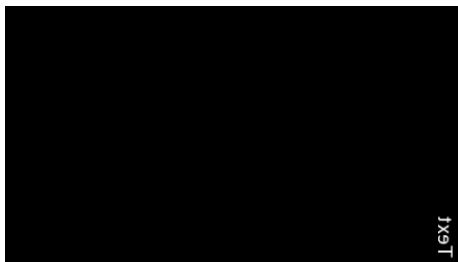
REG_ROTATE = 5 is mirrored inverted landscape:



REG_ROTATE = 6 is mirrored portrait:



REG_ROTATE = 7 is mirrored inverted portrait:



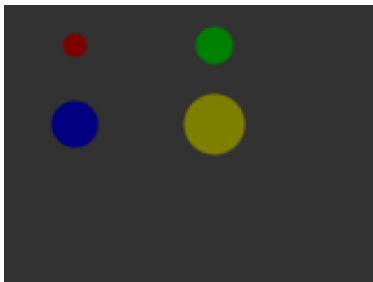
2.11.4 Drawing Pattern

The general drawing process is controlled through display list commands:

- Start with a **BEGIN** command to specify the primitive type
- Provide one or more vertices using “**VERTEX2II**” or “**VERTEX2F**” to define the primitive’s placement on the screen
- Finish with an **END** command to complete the primitive.

Examples

Draw points with varying radius from 5 to 13 pixels, each displayed in a different color:



```
//The VERTEX2F command gives the location of the circle center
dl( COLOR_RGB(128, 0, 0) );
dl( POINT_SIZE(5 * 16) );
dl( BEGIN(POINTS) );
dl( VERTEX2F(30 * 16, 17 * 16) );
dl( COLOR_RGB(0, 128, 0) );
dl( POINT_SIZE(8 * 16) );
dl( VERTEX2F(90 * 16, 17 * 16) );
dl( COLOR_RGB(0, 0, 128) );
dl( POINT_SIZE(10 * 16) );
dl( VERTEX2F(30 * 16, 51 * 16) );
dl( COLOR_RGB(128, 128, 0) );
dl( POINT_SIZE(13 * 16) );
dl( VERTEX2F(90 * 16, 51 * 16) );
```

Draw lines with varying widths, ranging from 2 to 6 pixels, each in a different color (the line width is measured from the center of the line to its edge):



```
//The VERTEX2F commands are in pairs to define the start and finish point of the line.
dl( COLOR_RGB(128, 0, 0) );
dl( LINE_WIDTH(2 * 16) );
dl( BEGIN(LINES) );
dl( VERTEX2F(30 * 16, 38 * 16) );
dl( VERTEX2F(30 * 16, 63 * 16) );
dl( COLOR_RGB(0, 128, 0) );
dl( LINE_WIDTH(4 * 16) );
dl( VERTEX2F(60 * 16, 25 * 16) );
dl( VERTEX2F(60 * 16, 63 * 16) );
dl( COLOR_RGB(128, 128, 0) );
dl( LINE_WIDTH(6 * 16) );
dl( VERTEX2F(90 * 16, 13 * 16) );
dl( VERTEX2F(90 * 16, 63 * 16) );
```

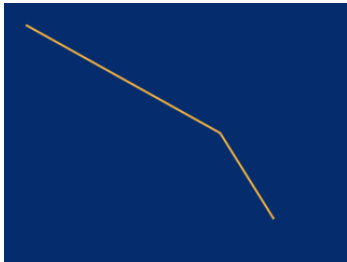
Draw rectangles with dimensions of 5x25, 10x38 and 15x50 pixels:

(The line width influences the corner curvature, with LINE_WIDTH pixels added in both directions beyond the rectangle's specified dimensions):



```
//The VERTEX2F commands are in pairs to define the top left
and bottom right corners of the rectangle.
dl( COLOR_RGB(128, 0, 0) );
dl( LINE_WIDTH(1 * 16) );
dl( BEGIN(RECTS) );
dl( VERTEX2F(28 * 16, 38 * 16) );
dl( VERTEX2F(33 * 16, 63 * 16) );
dl( COLOR_RGB(0, 128, 0) );
dl( LINE_WIDTH(5 * 16) );
dl( VERTEX2F(50 * 16, 25 * 16) );
dl( VERTEX2F(60 * 16, 63 * 16) );
dl( COLOR_RGB(128, 128, 0) );
dl( LINE_WIDTH(10 * 16) );
dl( VERTEX2F(83 * 16, 13 * 16) );
dl( VERTEX2F(98 * 16, 63 * 16) );
```

Draw line strips for sets of coordinates:



```
dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1, 1, 1) );
dl( BEGIN(LINE_STRIP) );
dl( VERTEX2F(5 * 16, 5 * 16) );
dl( VERTEX2F(50 * 16, 30 * 16) );
dl( VERTEX2F(63 * 16, 50 * 16) );
```

Draw Edge strips for above:



```
dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1, 1, 1) );
dl( BEGIN(EDGE_STRIP_A) );
dl( VERTEX2F(5 * 16, 5 * 16) );
dl( VERTEX2F(50 * 16, 30 * 16) );
dl( VERTEX2F(63 * 16, 50 * 16) );
```

Draw Edge strips below:



```
dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1, 1, 1) );
dl( BEGIN(EDGE_STRIP_B) );
dl( VERTEX2F(5 * 16, 5 * 16) );
dl( VERTEX2F(50 * 16, 30 * 16) );
dl( VERTEX2F(63 * 16, 50 * 16) );
```

Draw Edge strips for right:



```
dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1, 1, 1) );
dl( BEGIN(EDGE_STRIP_R) );
dl( VERTEX2F(5 * 16, 5 * 16) );
dl( VERTEX2F(50 * 16, 30 * 16) );
dl( VERTEX2F(63 * 16, 50 * 16) );
```

Draw Edge strips for left:



```
dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1, 1, 1) );
dl( BEGIN(EDGE_STRIP_L) );
dl( VERTEX2F(5 * 16, 5 * 16) );
dl( VERTEX2F(50 * 16, 30 * 16) );
dl( VERTEX2F(63 * 16, 50 * 16) );
```

2.11.5 Bitmap Transformation Matrix

To perform the bitmap transformation, the following bitmap transform matrix is defined and denoted as m :

$$m = \begin{bmatrix} \text{BITMAP_TRANSFORM_A} & \text{BITMAP_TRANSFORM_B} & \text{BITMAP_TRANSFORM_C} \\ \text{BITMAP_TRANSFORM_D} & \text{BITMAP_TRANSFORM_E} & \text{BITMAP_TRANSFORM_F} \end{bmatrix}$$

by default, $m = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix}$, which is named as the **identity matrix**.

The coordinates x' y' after transforming are calculated in the following equation:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = m \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

i.e.:

$$\begin{aligned} x' &= x * A + y * B + C \\ y' &= x * D + y * E + F \end{aligned}$$

Where A, B, C, D, E, F stands for the values assigned by commands $\text{BITMAP_TRANSFORM_A-F}$.

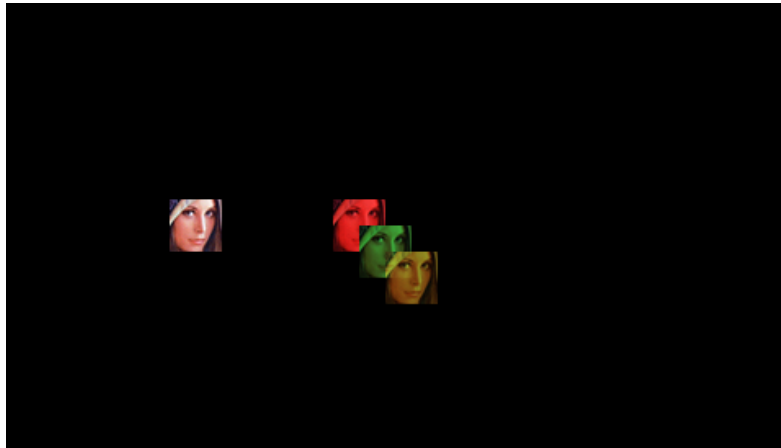
2.11.6 Color and Transparency

The same bitmap can be rendered in multiple locations on the screen, with varying colors and levels of transparency:

```
dl(COLOR_RGB(255, 64, 64)); // red at (200, 120)
dl(VERTEX2II(200, 120, 0, 0));
dl(COLOR_RGB(64, 180, 64)); // green at (216, 136)
dl(VERTEX2II(216, 136, 0, 0));
dl(COLOR_RGB(255, 255, 64)); // transparent yellow at (232, 152)
dl(COLOR_A(150));
dl(VERTEX2II(232, 152, 0, 0));
```

Code Snippet 11 – Color and Transparency

The **COLOR_RGB** command sets the current drawing color, which tints the bitmap. If this command is not used, the default color (255,255,255) is applied, rendering the bitmap in its original colors. The **COLOR_A** command adjusts the current drawing alpha, controlling the transparency of the rendering. An alpha value of 0 makes the drawing fully transparent, while 255 makes it fully opaque; for example, a value of 150 produces a partially transparent effect.



2.11.7 Performance

The graphics engine does not use a frame buffer; instead, it employs a dynamic compositing method to generate each display line during scan out. As a result,, there is a limited amount of time available to draw each line. This time is determined by the scan-out parameters, set by **REG_PCLK** and **REG_HCYCLE**, but it is never less than 2048 internal clock cycles.

Some performance limits:

- The display list length must be under 2048 instructions, as the graphics engine fetches one display list command per clock.
- Pixel rendering typically occurs at 16 pixels per clock in **NEAREST** filter mode, except for the following formats:
 - **TEXT8X8,**
 - **TEXTVGA**
 - **PALETTED4444/565**

which renders 8 pixels per clock.
- For **BILINEAR** filtered pixels, the drawing rate will be reduced to ¼.

To summarize:

Filter Modes	Bitmap Formats	Drawing Rate
NEAREST	TEXT8X8, TEXTVGA, PALETTED4444/565	8 pixel per clock
NEAREST	The remaining formats not listed in the row above	16 pixel per clock
BILINEAR	TEXT8X8, TEXTVGA, PALETTED4444/565	2 pixel per clock
BILINEAR	The remaining formats not listed in the row above	4 pixel per clock

Table 6 – Bitmap Rendering Performance

3 Register Description

The registers are organized into the following groups based on their functionality:

- Graphics Engine Registers,
- Audio Engine Registers,
- Touch Engine Registers,
- Coprocessor Engine Registers,
- Special Registers,
- Miscellaneous Registers.

The detailed definitions for each register are listed below. Most registers are **32** bits wide, with special noted separately. Reserved bits must always be read as zero and written as **zero**.

- Bit fields marked **r/o** are read-only.
- Bit fields marked **w/o** are write-only.
- Bit fields marked **r/w** are read-write.

Register offsets are based on the **RAM_REG** address.

3.1 Graphics Engine Registers

REG_TAG Definition			
31		8	7
	Reserved		r/o
Offset: 0x7C		Reset Value: 0x0	
Bit 31 – 8: Reserved bits			
Bit 7 – 0: These bits are updated with the tag value, which corresponds to the touching point coordinates provided in REG_TAG_X and REG_TAG_Y.			
Note: Please note the difference between REG_TAG and REG_TOUCH_TAG.			
REG_TAG is updated based on the X, Y given by REG_TAG_X and REG_TAG_Y.			
REG_TOUCH_TAG is updated based on the current touching point captured from touch screen.			

Register Definition 1 – REG_TAG Definition

REG_TAG_Y Definition			
31		11	10
	Reserved		r/w
Offset: 0x78		Reset Value: 0x0	
Bit 31 – 11: Reserved Bits			
Bit 10 – 0: These bits are set by the host as the Y coordinate of the touching point, which will enable the host to query the tag value. This register shall be used together with REG_TAG_X and REG_TAG. Normally, in the case where the host has already captured the touching point’s coordinate; this register can be updated to query the tag value of respective touching point.			

Register Definition 2 – REG_TAG_Y Definition

REG_TAG_X Definition			
31		11	10
	Reserved		r/w
Offset: 0x74		Reset Value: 0x0	
Bit 31 – 11: Reserved Bits			
Bit 10 – 0: These bits are set by the host as the X coordinate of the touching point, which will enable the host to query the tag value. This register shall be used together with REG_TAG_Y and REG_TAG. Normally, in the case where the host has already captured the touching point’s coordinate; this register can be updated to query the tag value of the respective touching point.			

Register Definition 3 – REG_TAG_X Definition

REG_PCLK Definition		
31	8	7 0
Reserved		r/w
Offset: 0x70		Reset Value: 0x0
Bit 31 – 8: Reserved bits		
Bit 7 – 0: These bits are set to divide the main clock for PCLK . If the main clock is 60Mhz and the value of these bits are set to 5, the PCLK will be set to 12 MHz If these bits are set to zero, it means there is no PCLK output.		

Register Definition 4 – REG_PCLK Definition

REG_PCLK_POL Definition		
31		1 0
reserved		r/w
Offset: 0x6C		Reset Value: 0x0
Bit 31 – 1: Reserved bits		
Bit 0: This bit controls the polarity of PCLK . If it is set to zero, PCLK polarity is on the rising edge. If it is set to one, PCLK polarity is on the falling edge		

Register Definition 5 – REG_PCLK_POL Definition

REG_CSPREAD Definition		
31		1 0
reserved		r/w
Offset: 0x68		Reset Value: 0x1
Bit 31 – 1: Reserved bits		
Bit 0: This bit controls the transition of RGB signals with PCLK active clock edge, which helps reduce the system noise. When it is zero, all the color signals are updated at the same time. When it is one, all the color signal timings are adjusted slightly so that fewer signals change simultaneously.		

Register Definition 6 – REG_CSPREAD Definition

REG_SWIZZLE Definition		
31	4	3 0
Reserved		r/w
Offset: 0x64		Reset Value: 0x0
Bit 31 – 4: Reserved bits		
Bit 3 – 0: These bits are set to control the arrangement of output RGB pins, which help support different LCD panels. Please see the datasheet for the exact definitions.		

Register Definition 7 – REG_SWIZZLE Definition

REG_DITHER Definition		
31		1 0
reserved		r/w
Offset: 0x60		Reset Value: 0x1
Bit 31 – 1: Reserved bits		
Bit 0: Set to 1 to enable dithering feature on RGB signals output. Set to 0 to disable dithering feature. Reading 1 from this bit means dithering feature is enabled. Reading 0 from this bit means dithering feature is disabled.		

Register Definition 8 – REG_DITHER Definition

REG_OUTBITS Definition		
31	9	8 0
Reserved		r/w
Offset: 0x5C		Reset Value: 0x0
Bit 31 – 9: Reserved Bits		
Bit 8 – 0: These 9 bits are split into 3 groups for Red, Green and Blue color output signals: Bit 8 – 6: Red Color signal lines number. Value zero means 8 output signals. Bit 5 – 3: Green Color signal lines number. Value zero means 8 output signals. Bit 2 – 0: Blue color signal lines number. Value zero means 8 output signals. Host can write these bits to control the number of output signals for each color.		

For 6/6/6 bits configuration, pins R2-R7, G2-G7 and B2-B7 will be used, while pins R0, R1, G0, G1, B0, B1 can be left un-connected. Please refer to the datasheet for more details.

Register Definition 9 – REG_OUTBITS Definition

REG_ROTATE Definition			
31		3	2
Reserved			r/w
Offset: 0x58		Reset Value: 0x0	
Bit 31 – 3: Reserved bits			
Bit 2 – 0: screen rotation control bits.			
0b'000: Default landscape orientation			
0b'001: Inverted landscape orientation			
0b'010: Portrait orientation			
0b'011: Inverted portrait orientation			
0b'100: Mirrored landscape orientation			
0b'101: Mirrored invert landscape orientation			
0b'110: Mirrored portrait orientation			
0b'111: Mirrored inverted portrait orientation			
Note: Setting this register will NOT affect touch transform matrix.			

Register Definition 10 – REG_ROTATE Definition

REG_DLSWAP Definition			
31		2	1
Reserved			r/w
Offset: 0x54		Reset Value: 0x0	
Bit 31 – 2: Reserved bits			
Bit 1 – 0: These bits can be set by the host to validate the display list buffer. The graphics engine will determine when to render the screen, depending on how these bits are set:			
0b'01: Graphics engine will render the screen immediately after current line is scanned out. It may cause tearing effect.			
0b'10: Graphics engine will render the screen immediately after current frame is scanned out.			
0b'00: Do not write this value into this register.			
0b'11: Do not write this value into this register.			
These bits can also be read by the host to check the availability of the display list buffer. If the value is read as zero, the display list buffer is safe and ready to write. Otherwise, the host needs to wait till it becomes zero.			

Register Definition 11 – REG_DLSWAP Definition

REG_VSYNC1 Definition			
31		12	11
Reserved			r/w
Offset: 0x50		Reset Value: 0xA	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: These bits specify how many lines of signal VSYNC1 takes at the start of a new frame			

Register Definition 12 – REG_VSYNC1 Definition

REG_VSYNC0 Definition			
31		12	11
Reserved			r/w
Offset: 0x4C		Reset Value: 0x0	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: The value of these bits specifies how many lines of signal VSYNC0 takes at the start of a new frame			

Register Definition 13 – REG_VSYNC0 Definition

REG_VSIZE Definition			
31		12	11
	Reserved		r/w
Offset: 0x48		Reset Value: 0x110	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: The value of these bits specifies how many lines of pixels in one frame. The valid range is from 0 to 2047.			

Register Definition 14 – REG_VSIZE Definition

REG_VOFFSET Definition			
31		12	11
	Reserved		r/w
Offset: 0x44		Reset Value: 0xC	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: The value of these bits specifies how many lines taken after the start of a new frame.			

Register Definition 15 – REG_VOFFSET Definition

REG_VCYCLE Definition			
31		12	11
	Reserved		r/w
Offset: 0x40		Reset Value: 0x124	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: The value of these bits specifies how many lines in one frame.			

Register Definition 16 – REG_VCYCLE Definition

REG_HSYNC1 Definition			
31		12	11
	Reserved		r/w
Offset: 0x3C		Reset Value: 0x29	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: The value of these bits specifies how many PCLK cycles for HSYNC1 during start of line.			

Register Definition 17 – REG_HSYNC1 Definition

REG_HSYNC0 Definition			
31		12	11
	Reserved		r/w
Offset: 0x38		Reset Value: 0x0	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: The value of these bits specifies how many PCLK cycles for HSYNC0 during start of line.			

Register Definition 18 – REG_HSYNC0 Definition

REG_HSIZE Definition			
31		12	11
	Reserved		r/w
Offset: 0x34		Reset Value: 0x1E0	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: These bits are used to specify the number of PCLK cycles per horizontal line.			

Register Definition 19 – REG_HSIZE Definition

REG_HOFFSET Definition			
31		12	11
	Reserved		r/w
Offset: 0x30		Reset Value: 0x2B	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: These bits are used to specify the number of PCLK cycles before pixels are scanned out.			

Register Definition 20 – REG_HOFFSET Definition

REG_HCYCLE Definition			
31		12	11
	Reserved		r/w
Offset: 0x2C		Reset Value: 0x224	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: These bits are the number of total PCLK cycles per horizontal line scan.			

Register Definition 21 – REG_HCYCLE Definition

3.2 Audio Engine Registers

REG_PLAY Definition			
31		1	0
	reserved		r/w
Offset: 0x8C		Reset Value: 0x0	
Bit 31 – 1: Reserved bits			
Bit 0: A write to this bit triggers the play of the synthesized sound effect specified in REG_SOUND . Reading value 1 in this bit means the sound effect is playing. To stop the sound effect, the host needs to select the silence sound effect by setting up REG_SOUND and set this register to play.			

Register Definition 22 – REG_PLAY Definition

REG_SOUND Definition			
31		16	15
	Reserved		r/w
Offset: 0x88		Reset Value: 0x0	
Bit 31 – 16: Reserved bits			
Bit 15 – 0: These bits are used to select the synthesized sound effect. They are split into two groups: Bit 15 – 8 and Bit 7 – 0.			
Bit 15 – 8: The MIDI note for the sound effect defined in Bits 0 – 7.			
Bit 7 – 0: These bits define the sound effect. Some of them are pitch adjustable and the pitch is defined in Bits 8 – 15. Some of them are not pitch adjustable and the Bits 8 – 15 will be ignored.			
Note: Please refer to the section "Sound Synthesizer" in BT81X datasheet for details of this register.			

Register Definition 23 – REG_SOUND Definition

REG_VOL_SOUND Definition			
31		8	7
	Reserved		r/w
Offset: 0x84		Reset Value: 0xFF	
Bit 31 – 8: Reserved bits			
Bit 7 – 0: These bits control the volume of the synthesizer sound. The default value 0xFF is highest volume. Value zero means mute.			

Register Definition 24 – REG_VOL_SOUND Definition

REG_VOL_PB Definition			
31		8	7
	Reserved		r/w
Offset: 0x80		Reset Value: 0xFF	
Bit 31 – 8: Reserved bits			
Bit 7 – 0: These bits control the volume of the audio file playback. The default value 0xFF is highest volume. Value zero means mute.			

Register Definition 25 – REG_VOL_PB Definition

REG_PLAYBACK_PLAY Definition		
31		1 0
Reserved		r/w
Offset: 0xCC Reset Value: 0x0		
Bit 31 – 1: Reserved bits		
Bit 0: A write to this bit triggers the start of audio playback, regardless of writing 0 or 1. It will read back 1 when playback is on-going, and 0 when playback completes.		
Note: Please refer to the section "Audio Playback" in BT81X datasheet for details of this register.		

Register Definition 26 – REG_PLAYBACK_PLAY Definition

REG_PLAYBACK_LOOP Definition		
31		1 0
Reserved		r/w
Offset: 0xC8 Reset Value: 0x0		
Bit 31 – 1: Reserved bits		
Bit 0: this bit controls the audio engine to play back the audio data in RAM_G from the start address once it consumes all the data. A value of 1 means LOOP is enabled; a value of 0 means LOOP is disabled.		
Note: Please refer to the section "Audio Playback" in BT81X datasheet for details of this register.		

Register Definition 27 – REG_PLAYBACK_LOOP Definition

REG_PLAYBACK_FORMAT Definition		
31		2 1 0
Reserved		r/w
Offset: 0xC4 Reset Value: 0x0		
Bit 31 – 2: Reserved bits		
Bit 1 – 0: These bits define the format of the audio data in RAM_G .		
0b'00: Linear Sample format		
0b'01: uLaw Sample format		
0b'10: 4-bit IMA ADPCM Sample format		
0b'11: Undefined.		
Note: Please refer to the section "Audio Playback" in BT81X datasheet for details of this register.		

Register Definition 28 – REG_PLAYBACK_FORMAT Definition

REG_PLAYBACK_FREQ Definition		
31	16 15	0
Reserved		r/w
Offset: 0xC0 Reset Value: 0x1F40		
Bit 31 – 16: Reserved bits		
Bit 15 – 0: These bits specify the sampling frequency of audio playback data. Unit is in Hz.		
Note: Please refer to the section "Audio Playback" in BT81X datasheet for details of this register.		

Register Definition 29 – REG_PLAYBACK_FREQ Definition

REG_PLAYBACK_READPTR Definition		
31	20 19	0
reserved		r/w
Offset: 0xBC Reset Value: 0x0		
Bit 31 – 20: Reserved bits		
Bit 19 – 0: These bits are updated by the audio engine while playing audio data from RAM_G. It is the current audio data address which is playing back. The host can read this register to check if the audio engine has consumed all the audio data.		
Note: Please refer to the section "Audio Playback" in BT81X datasheet for details of this register.		

Register Definition 30 – REG_PLAYBACK_READPTR Definition

REG_PLAYBACK_LENGTH Definition		
31	20	19
reserved		r/w
Offset: 0xB8		Reset Value: 0x0
Bit 31 – 20: Reserved bits		
Bit 19 – 0: These bits specify the length of audio data in RAM_G to playback, starting from the address specified in REG_PLAYBACK_START register.		
Note: Please refer to the section "Audio Playback" in BT81X datasheet for details of this register.		

Register Definition 31 – REG_PLAYBACK_LENGTH Definition

REG_PLAYBACK_START Definition		
31	20	19
reserved		r/w
Offset: 0xB4		Reset Value: 0x0
Bit 31 – 20: Reserved bits		
Bit 19 – 0: These bits specify the start address of audio data in RAM_G to playback.		
Note: Please refer to the section "Audio Playback" in BT81X datasheet for details of this register.		

Register Definition 32 – REG_PLAYBACK_START Definition

REG_PLAYBACK_PAUSE Definition		
7		0
reserved		r/w
Offset: 0x5EC		Reset Value: 0x0
Bit 7 – 1: Reserved bits		
Bit 0: Audio playback control bit. Writing 1 to pause the playback, writing 0 to start the playback.		
Note: Please refer to the section "Audio Playback" in BT81X datasheet for details of this register.		

Register Definition 33 – REG_PLAYBACK_PAUSE Definition

3.3 Flash Registers

REG_FLASH_STATUS Definition		
7	2	1
reserved		r/o
Offset: 0x5F0		Reset Value: 0x0
Bit 7 – 2: Reserved bits		
Bit 1 – 0: These bits reflect the current status of attached flash. 0b'00: FLASH_STATUS_INIT 0b'01: FLASH_STATUS_DETACHED 0b'10: FLASH_STATUS_BASIC 0b'11: FLASH_STATUS_FULL		
Note: Please refer to the section "SPI NOR Flash Interface" in BT817/8 datasheet for details.		

Register Definition 34 – REG_FLASH_STATUS Definition

REG_FLASH_SIZE Definition		
31		0
r/o		
Offset: 0x7024		Reset Value: 0x0
Bit 31 – 0: The value indicates the capacity of attached flash, in Mbytes.		
Note: Please refer to the section "SPI NOR Flash Interface" in BT817/8 datasheet for details		

Register Definition 35 – REG_FLASH_SIZE Definition

3.4 Touch Screen Engine Registers

3.4.1 Overview

EVE provides touchscreen functionality through either **Resistive Touch Engine (RTE)** or a **Capacitive Touch Screen Engine (CTSE)**. The BT815/BT817 models include a built-in **CTSE**, while BT816/BT818 models feature a built-in **RTE**.

3.4.2 Common Registers

This chapter covers the common registers that are applicable to both **RTE** and **CTSE**.

Offset	Register Name	Description
0x150 – 0x164	REG_TOUCH_TRANSFORM_A~F	Transform coefficient matrix coefficient
0x168	REG_TOUCH_CONFIG	Configuration register

Table 7 – Common Registers Summary

REG_TOUCH_CONFIG Definition												
31	16	15	14	13	12	11	10	4	3	2	1	0
reserved			r/o	r/w	rsvp	r/w	r/w	r/w		rsvp	r/w	r/w
Offset: 0x168						Reset Value: 0x8381 (BT816/818) or 0x381 (BT815/817)						
Bit 31 – 16: Reserved bits												
Bit 15: Working mode of touch engine. 0: capacitive 1: resistive												
Bit 14: 1: Enable the host mode. 0: Normal mode												
Bit 13: Reserved bit												
Bit 12: Ignore short-circuit protection. For resistive touch screen only.												
Bit 11: Enable low-power mode (for FocalTech only)												
Bit 10 – 4: I2C address of capacitive touch screen module: 0b'0111000 for FocalTech/Hycontex 0b'1011101 for Goodix												
Bit 3: Reserved bit												
Bit 2: Suppress 300ms startup (for FocalTech only)												
Bit 1 – 0: Sampling clocks (for resistive touch screen only)												

Register Definition 36 – REG_TOUCH_CONFIG Definition

REG_TOUCH_TRANSFORM_F Definition				
31	30	16	15	0
r/w	r/w		r/w	
Offset: 0x164			Reset Value: 0x0	
Bit 31: The sign bit for fixed point number				
Bit 30 – 16: The value of these bits represents the integer part of the fixed-point number.				
Bit 15 – 0: The value of these bits represents the fractional part of the fixed-point number.				
Note: This register represents a fixed-point number, and the default value is +0.0 after reset.				

Register Definition 37 – REG_TOUCH_TRANSFORM_F Definition

REG_TOUCH_TRANSFORM_E Definition			
31	30	16	15
r/w		r/w	
Offset: 0x160			Reset Value: 0x10000
Bit 31: The sign bit for fixed point number			
Bit 30 – 16: The value of these bits represents the integer part of the fixed-point number.			
Bit 15 – 0: The value of these bits represents the fractional part of the fixed-point number.			
Note: This register represents a fixed-point number, and the default value is +1.0 after reset.			

Register Definition 38 – REG_TOUCH_TRANSFORM_E Definition

REG_TOUCH_TRANSFORM_D Definition			
31	30	16	15
r/w		r/w	
Offset: 0x15C			Reset Value: 0x0
Bit 31: The sign bit for fixed point number			
Bit 30 – 16: The value of these bits represents the integer part of the fixed-point number.			
Bit 15 – 0: The value of these bits represents the fractional part of the fixed-point number.			
Note: This register represents a fixed-point number, and the default value is +0.0 after reset.			

Register Definition 39 – REG_TOUCH_TRANSFORM_D Definition

REG_TOUCH_TRANSFORM_C Definition			
31	30	16	15
r/w		r/w	
Offset: 0x158			Reset Value: 0x0
Bit 31: The sign bit for fixed point number			
Bit 30 – 16: The value of these bits represents the integer part of the fixed-point number.			
Bit 15 – 0: The value of these bits represents the fractional part of the fixed-point number.			
Note: This register represents fixed point number, and the default value is +0.0 after reset.			

Register Definition 40 – REG_TOUCH_TRANSFORM_C Definition

REG_TOUCH_TRANSFORM_B Definition			
31	30	16	15
r/w		r/w	
Offset: 0x154			Reset Value: 0x0
Bit 31: The sign bit for fixed point number			
Bit 30 – 16: The value of these bits represents the integer part of the fixed-point number.			
Bit 15 – 0: The value of these bits represents the fractional part of the fixed-point number.			
Note: This register represents a fixed-point number, and the default value is +0.0 after reset.			

Register Definition 41 – REG_TOUCH_TRANSFORM_B Definition

REG_TOUCH_TRANSFORM_A Definition			
31	30	16	15
r/w		r/w	
Offset: 0x150			Reset Value: 0x10000
Bit 31: The sign bit for fixed point number			
Bit 30 – 16: The value of these bits represents the integer part of the fixed-point number.			
Bit 15 – 0: The value of these bits represents the fractional part of the fixed-point number.			
Note: This register represents a fixed-point number, and the default value is +1.0 after reset.			

Register Definition 42 – REG_TOUCH_TRANSFORM_A Definition

3.4.3 Resistive Touch Engine

All registers in the **RTE** are identical to those in the FT810.

Offset	Register Name	Description
0x104	REG_TOUCH_MODE	Touch screen sampling Mode
0x108	REG_TOUCH_ADC_MODE	Select ADC working mode
0x10C	REG_TOUCH_CHARGE	Touch screen charge time, unit of 6 clocks
0x110	REG_TOUCH_SETTLE	Touch screen settle time, unit of 6 clocks
0x114	REG_TOUCH_OVERSAMPLE	Touch screen oversample factor
0x118	REG_TOUCH_RZTHRESH	Touch screen resistance threshold
0x11C	REG_TOUCH_RAW_XY	Touch screen raw x,y(16,16)
0x120	REG_TOUCH_RZ	Touch screen resistance
0x124	REG_TOUCH_SCREEN_XY	Touch screen x,y(16,16)
0x128	REG_TOUCH_TAG_XY	coordinate used to calculate the tag of touch point
0x12C	REG_TOUCH_TAG	Touch screen Tag result

Table 8 – RTE Registers Summary

REG_TOUCH_TAG Definition		
31	8	7
Reserved		r/o
Offset: 0x12C		Reset Value: 0x0
Bit 31 – 8: Reserved Bits		
Bit 7 – 0: These bits are set as the tag value of the specific graphics object on the screen which is being touched. These bits are updated once when all the lines of the current frame are scanned out to the screen.		
Note: The valid tag value is from 1 to 255 and the default value of this register is zero, meaning there is no touch by default.		

Register Definition 43 – REG_TOUCH_TAG Definition

REG_TOUCH_TAG_XY Definition		
31	16	15
r/o		r/o
Offset: 0x128		Reset Value: 0x0
Bit 31 – 16: The value of these bits are X coordinates of the touch screen to look up the tag result.		
Bit 15 – 0: The value of these bits are the Y coordinates of the touch screen to look up the tag result.		
Note: Host can read this register to check the coordinates used by the touch engine to update the tag register REG_TOUCH_TAG .		

Register Definition 44 – REG_TOUCH_TAG_XY Definition

REG_TOUCH_SCREEN_XY Definition		
31	16	15
r/o		r/o
Offset: 0x124		Reset Value: 0x80008000
Bit 31 – 16: The value of these bits is the X coordinates of the touch screen. After calibration, it shall be within the width of the screen size. If the touch screen is not being touched, it shall be 0x8000.		
Bit 15 – 0: The value of these bits is the Y coordinates of the touch screen. After calibration, it shall be within the width of the screen size. If the touch screen is not touched, it shall be 0x8000.		

Register Definition 45 – REG_TOUCH_SCREEN_XY Definition

REG_TOUCH_DIRECT_Z1Z2 Definition			
31	26	25	0
reserved	r/o	reserved	r/o
Offset: 0x190			Reset Value: NA
Bit 31 – 26: Reserved Bits			
Bit 25 – 16: 10-bit ADC value for touch screen resistance Z1			
Bit 15 – 10: Reserved Bits			
Bit 9 – 0: 10-bit ADC value for touch screen resistance Z2			
Note: To know if it is touched or not, please check the 31 st bit of REG_TOUCH_DIRECT_XY . Touch engine will do the post-processing for these Z1 and Z2 values and update the result in REG_TOUCH_RZ .			

Register Definition 46 – REG_TOUCH_DIRECT_Z1Z2 Definition

REG_TOUCH_DIRECT_XY Definition			
31	30	26	0
r/o	reserved	r/o	r/o
Offset: 0x18C			Reset Value: 0x0
Bit 31: If this bit is zero, it means a touch is being sensed and the two fields above contain the sensed data. If this bit is one, it means no touch is being sensed and the data in the two fields above shall be ignored.			
Bit 30 – 26: Reserved Bits			
Bit 25 – 16: 10-bit ADC value for touch screen resistance Z1			
Bit 15 – 10: Reserved Bits			
Bit 9 – 0: 10-bit ADC value for touch screen resistance Z2			

Register Definition 47 – REG_TOUCH_DIRECT_XY

REG_TOUCH_RZ Definition	
31	0
Reserved	r/o
Offset: 0x120	
Reset Value: 0x7FFF	
Bit 31 – 16: Reserved Bits	
Bit 15 – 0: These bits measure the touching pressure on the touch screen. The valid value is from 0 to 0x7FFF. The highest value(0x7FFF) means no touch and the lowest value (0) means the maximum touching pressure.	

Register Definition 48 – REG_TOUCH_RZ Definition

REG_TOUCH_RAW_XY Definition	
31	0
r/o	r/o
Offset: 0x11C	
Reset Value: 0xFFFFFFFF	
Bit 31 – 16: These bits are the raw X coordinates before going through calibration process. The valid range is from 0 to 1023. If there is no touch on screen, the value shall be 0xFFFF.	
Bit 15 – 0: These bits are the raw Y coordinates of the touch screen before going through calibration process. The valid range is from 0 to 1023. If there is no touch on screen, the value shall be 0xFFFF.	
Note: The coordinates in this register have not mapped into the screen coordinates. To get the screen coordinates, please refer to REG_TOUCH_SCREEN_XY .	

Register Definition 49 – REG_TOUCH_RAW_XY Definition

REG_TOUCH_RZTHRESH Definition	
31	0
Reserved	r/w
Offset: 0x118	
Reset Value: 0xFFFF	
Bit 31 – 16: Reserved Bits	
Bit 15 - 0: These bits control the touch screen resistance threshold. The host can adjust the touch screen touching sensitivity by setting this register. The default value after reset is 0xFFFF and it means the lightest touch will be accepted by the RTE. The host can set this register by doing experiments. The typical value is 1200.	

Register Definition 50 – REG_TOUCH_RZTHRESH Definition

REG_TOUCH_OVERSAMPLE Definition			
31		4	3 0
reserved		r/w	
Offset: 0x114		Reset Value: 0x7	
Bit 31 – 4: Reserved Bits			
Bit 3 – 0: These bits control the touch screen oversample factor. The higher value of this register causes more accuracy with more power consumption but may not be necessary. The valid range is from 1 to 15.			

Register Definition 51 – REG_TOUCH_OVERSAMPLE Definition

REG_TOUCH_SETTLE Definition			
31		4	3 0
reserved		r/w	
Offset: 0x110		Reset Value: 0x3	
Bit 31 – 4: Reserved Bits			
Bit 3 – 0: These bits control the touch screen settle time, in the unit of 6 clocks. The default value is 3, meaning the settle time is 18 (3*6) system clock cycles.			

Register Definition 52 – REG_TOUCH_SETTLE Definition

REG_TOUCH_CHARGE Definition			
31		16	15 0
Reserved		r/w	
Offset: 0x10C		Reset Value: 0x1770	
Bit 31 – 16: Reserved Bits			
Bit 15 – 0: These bits control the touch screen charge time, in the unit of 6 clocks. The default value is 6000, meaning the charge time is (6000*6) system clock cycles.			

Register Definition 53 – REG_TOUCH_CHARGE Definition

REG_TOUCH_ADC_MODE Definition			
31		2	1 0
reserved		r/w	
Offset: 0x108		Reset Value: 0x1	
Bit 31 – 1: Reserved bits			
Bit 0: The host can set this bit to control the ADC sampling mode, as per: 0: Single Ended mode. It causes lower power consumption but with less accuracy. 1: Differential Mode. It causes higher power consumption but with more accuracy.			

Register Definition 54 – REG_TOUCH_ADC_MODE Definition

REG_TOUCH_MODE Definition			
31		2	1 0
reserved		r/w	
Offset: 0x104		Reset Value: 0x3	
Bit 31 – 2: Reserved bits			
Bit 1 – 0: The host can set these two bits to control the touch screen sampling mode of touch engine, as per: 0b'00: Off mode. No sampling happens. RTE stops working. 0b'01: Single mode. Cause one single sample to occur. 0b'10: Frame mode. Cause a sample at the start of each frame. 0b'11: Continuous mode. Up to 1000 times per seconds. Default mode after reset.			

Register Definition 55 – REG_TOUCH_MODE Definition

3.4.4 Capacitive Touch Engine

Capacitive Touch Screen Engine (**CTSE**) offers the following features:

- I²C interface to the **Capacitive Touch Panel Module (CTPM)**
- Detects up to 5 simultaneous touch points
- Supports CTPM modules with FocalTech and Goodix touch controller.
- Supports touch host mode; see the datasheet for details.
- Provides a compatibility mode for single touch and an extended mode for multi-touch points.

After a reset or boot-up, **CTSE** operates in compatibility mode, detecting only single touch point. In extended mode, it can detect up to **five** touch points simultaneously.

CTSE uses the same registers set, **REG_TOUCH_TRANSFORM_A~F**, to convert raw touch coordinates into calibrated screen coordinate, regardless of whether it is in compatibility or extended mode.

Note: The touchscreen calibration process should be carried out only in compatibility mode.

Offset	Register Name	Description
0x104	REG_CTOUCH_MODE	Touch screen sampling Mode
0x108	REG_CTOUCH_EXTENDED	Select ADC working mode
0x11C	REG_CTOUCH_TOUCH1_XY	Coordinate of second touch point
0x120	REG_CTOUCH_TOUCH4_Y	Y coordinate of fifth touch point
0x124	REG_CTOUCH_TOUCH_XY	Coordinate of first touch point
0x128	REG_CTOUCH_TAG_XY	coordinate used to calculate the tag of first touch point
0x12C	REG_CTOUCH_TAG	Touch screen Tag result of fist touch point
0x130	REG_CTOUCH_TAG1_XY	XY used to tag of second touch point
0x134	REG_CTOUCH_TAG1	Tag result of second touch point
0x138	REG_CTOUCH_TAG2_XY	XY used to tag of third touch point
0x13C	REG_CTOUCH_TAG2	Tag result of third touch point
0x140	REG_CTOUCH_TAG3_XY	XY used to tag of fourth touch point
0x144	REG_CTOUCH_TAG3	Tag result of fourth touch point
0x148	REG_CTOUCH_TAG4_XY	XY used to tag of fifth touch point
0x14C	REG_CTOUCH_TAG4	Tag result of fifth touch point
0x16C	REG_CTOUCH_TOUCH4_X	X coordinate of fifth touch point
0x18C	REG_CTOUCH_TOUCH2_XY	Third touch point coordinate
0x190	REG_CTOUCH_TOUCH3_XY	Fourth touch point coordinate

Table 9 – CTSE Registers Summary

The following tables define the registers provided by **CTSE**:

REG_CTOUCH_MODE Definition		
31		2
	Reserved	1
		0
Offset: 0x104		Reset Value: 0x3
Bit 31 – 2: Reserved bits		
Bit 1 – 0: The host can set these two bits to control the touch screen sampling mode of the touch engine, as per:		
0b'00: Off mode. No sampling happens. CTSE stops working.		
0b'01: Not defined.		
0b'10: Not defined.		
0b'11: On mode.		

Register Definition 56 – REG_CTOUCH_MODE Definition

REG_CTOUCH_EXTEND Definition	
31	1 0
reserved	
Offset: 0x108	Reset Value: 0x1
Bit 31 - 1: Reserved bits	
Bit 0: This bit controls the detection mode of the touch engine, as per:	
0: Extended mode, multi-touch detection mode	
1: Compatibility mode, single touch detection mode	

Register Definition 57 – REG_CTOUCH_EXTENDED Definition

REG_CTOUCH_TOUCH_XY Definition	
31	16 15 0
r/o	r/o
Offset: 0x124	Reset Value: 0x80008000
Bit 31 - 16: The value of these bits is X coordinate of the first touch point	
Bit 15 - 0: The value of these bits is Y coordinate of the first touch point.	
Note: This register is applicable for extended mode and compatibility mode. For compatibility mode, it reflects the position of the only touch point	

Register Definition 58 – REG_CTOUCH_TOUCH_XY Definition

REG_CTOUCH_TOUCH1_XY Definition	
31	16 15 0
r/o	r/o
Offset: 0x11C	Reset Value: 0x80008000
Bit 31 - 16: The value of these bits is X coordinate of the second touch point	
Bit 15 - 0: The value of these bits is Y coordinate of the second touch point.	
Note: This register is only applicable in the extended mode	

Register Definition 59 – REG_CTOUCH_TOUCH1_XY Definition

REG_CTOUCH_TOUCH2_XY Definition	
31	16 15 0
r/o	r/o
Offset: 0x18C	Reset Value: 0x80008000
Bit 31 - 16: The value of these bits is X coordinates of the third touch point	
Bit 15 - 0: The value of these bits is Y coordinates of the third touch point.	
Note: This register is only applicable in the extended mode	

Register Definition 60 – REG_CTOUCH_TOUCH2_XY Definition

REG_CTOUCH_TOUCH3_XY Definition	
31	16 15 0
r/o	r/o
Offset: 0x190	Reset Value: 0x80008000
Bit 31 - 16: The value of these bits is X coordinate of the fourth touch point	
Bit 15 - 0: The value of these bits is Y coordinate of the fourth touch point.	
Note: This register is only applicable in the extended mode	

Register Definition 61 – REG_CTOUCH_TOUCH3_XY Definition

REG_CTOUCH_TOUCH4_X Definition	
15	0
r/o	
Offset: 0x16C	Reset Value: 0x8000
Bit 15 - 0: The value of these bits is X coordinate of the fifth touch point.	
Note: This register is only applicable in the extended mode. It is a 16-bit register.	

Register Definition 62 – REG_CTOUCH_TOUCH4_X Definition

REG_CTOUCH_TOUCH4_Y Definition	
15	0
r/o	
Offset: 0x120	Reset Value: 0x8000
Bit 15 – 0: The value of these bits is Y coordinate of the fifth touch point.	
Note: This register is only applicable in the extended mode. It is a 16-bit register.	

Register Definition 63 – REG_CTOUCH_TOUCH4_Y Definition

REG_CTOUCH_RAW_XY Definition	
31	16 15
r/o	
Offset: 0x11C	Reset Value: 0xFFFFFFFF
Bit 31 – 16: The value of these bits is the X coordinate of a touch point before going through calibration process	
Bit 15 - 0: The value of these bits is the Y coordinate of a touch point before going through calibration process	
Note: This register is only applicable in the compatibility mode	

Register Definition 64 – REG_CTOUCH_RAW_XY Definition

REG_CTOUCH_TAG Definition	
31	8 7
Reserved	
r/o	
Offset: 0x12C	Reset Value: 0x0
Bit 31 – 8: Reserved Bits	
Bit 7 – 0: These bits are set as the tag value of the specific graphics object on the screen which is being touched. These bits are updated once all the lines of the current frame are scanned out to the screen. It works in both extended mode and compatibility mode. In extended mode, it is the tag of the first touch point, i.e., the tag value mapping to the coordinate in REG_CTOUCH_TAG_XY	
Note: The valid tag value range is from 1 to 255, therefore the default value of this register is zero, meaning there is no touch by default. In extended mode, it refers to the first touch point	

Register Definition 65 – REG_CTOUCH_TAG Definition

REG_CTOUCH_TAG1 Definition	
31	8 7
Reserved	
r/o	
Offset: 0x134	Reset Value: 0x0
Bit 31 – 8: Reserved Bits	
Bit 7 - 0: These bits are set as the tag value of the specific graphics object on the screen which is being touched. It is the second touch point in extended mode. These bits are updated once when all the lines of the current frame are scanned out to the screen.	
Note: The valid tag value range is from 1 to 255, therefore the default value of this register is zero, meaning there is no touch by default. This register is only applicable in the extended mode.	

Register Definition 66 – REG_CTOUCH_TAG1 Definition

REG_CTOUCH_TAG2 Definition	
31	8 7
Reserved	
r/o	
Offset: 0x13C	Reset Value: 0x0
Bit 31 – 8: Reserved Bits	
Bit 7 – 0: These bits are set as the tag value of the specific graphics object on the screen which is being touched. It is the third touch point in extended mode. These bits are updated once when all the lines of the current frame are scanned out to the screen.	
Note: The valid tag value range is from 1 to 255, therefore the default value of this register is zero, meaning there is no touch by default. This register is only applicable in the extended mode.	

Register Definition 67 – REG_CTOUCH_TAG2 Definition

REG_CTOUCH_TAG3 Definition		
31	8 7	0
Reserved		r/o
Offset: 0x144		Reset Value: 0x0
Bit 31 – 8: Reserved Bits		
Bit 7 – 0: These bits are set as the tag value of the specific graphics object on the screen which is being touched. It is the fourth touch point in extended mode. These bits are updated once when all the lines of the current frame are scanned out to the screen.		
Note: The valid tag value range is from 1 to 255, therefore the default value of this register is zero, meaning there is no touch by default. This register is only applicable in the extended mode.		

Register Definition 68 – REG_CTOUCH_TAG3 Definition

REG_CTOUCH_TAG4 Definition		
31	8 7	0
Reserved		r/o
Offset: 0x14C		Reset Value: 0x0
Bit 31 – 8: Reserved Bits		
Bit 7 – 0: These bits are set as the tag value of the specific graphics object on the screen which is being touched. It is the fifth touch point in extended mode. These bits are updated once when all the lines of the current frame are scanned out to the screen.		
Note: The valid tag value range is from 1 to 255, therefore the default value of this register is zero, meaning there is no touch by default. This register is only applicable in the extended mode.		

Register Definition 69 – REG_CTOUCH_TAG4 Definition

REG_CTOUCH_TAG_XY Definition		
31	16 15	0
r/o		r/o
Offset: 0x128		Reset Value: 0x0
Bit 31 – 16: The value of these bits is X coordinate of the touch screen, used by the touch engine to look up the tag result.		
Bit 15 - 0: The value of these bits is Y coordinate of the touch screen, used by the touch engine to look up the tag result.		
Note: The Host can read this register to check the coordinates used by the touch engine to update the tag register REG_CTOUCH_TAG.		

Register Definition 70 – REG_CTOUCH_TAG_XY Definition

REG_CTOUCH_TAG1_XY Definition		
31	16 15	0
r/o		r/o
Offset: 0x130		Reset Value: 0x0
Bit 31 – 16: The value of these bits is X coordinate of the touch screen to look up the tag result.		
Bit 15 – 0: The value of these bits is Y coordinate of the touch screen to look up the tag result.		
Note: The Host can read this register to check the coordinates used by the touch engine to update the tag register REG_CTOUCH_TAG1.		

Register Definition 71 – REG_CTOUCH_TAG1_XY Definition

REG_CTOUCH_TAG2_XY Definition		
31	16 15	0
r/o		r/o
Offset: 0x138		Reset Value: 0x0
Bit 31 – 16: The value of these bits is X coordinate of the touch screen to look up the tag result.		
Bit 15 - 0: The value of these bits is Y coordinate of the touch screen to look up the tag result.		
Note: The Host can read this register to check the coordinates used by the touch engine to update the tag register REG_CTOUCH_TAG2.		

Register Definition 72 – REG_CTOUCH_TAG2_XY Definition

REG_CTOUCH_TAG3_XY Definition			
31		16	15
			0
r/o		r/o	
Offset: 0x140		Reset Value: 0x0	
Bit 31 - 16: The value of these bits is X coordinate of the touch screen to look up the tag result.			
Bit 15 - 0: The value of these bits is Y coordinate of the touch screen to look up the tag result.			
Note: The Host can read this register to check the coordinates used by the touch engine to update the tag register REG_CTOUCH_TAG3.			

Register Definition 73 – REG_CTOUCH_TAG3_XY Definition

REG_CTOUCH_TAG4_XY Definition			
31		16	15
			0
r/o		r/o	
Offset: 0x148		Reset Value: 0x0	
Bit 31 - 16: The value of these bits is X coordinate of the touch screen to look up the tag result.			
Bit 15 - 0: The value of these bits is Y coordinate of the touch screen to look up the tag result.			
Note: The Host can read this register to check the coordinates used by the touch engine to update the tag register REG_CTOUCH_TAG4.			

Register Definition 74 – REG_CTOUCH_TAG4_XY Definition

3.4.5 Calibration

The calibration process is started using the **CMD_CALIBRATE** command and is compatible with both the **RTE** and **CTSE**; however, it is only available when the **CTSE** is compatibility mode. The results of the calibration, though, apply to both compatibility mode and extended modes. Therefore, it is recommended that users complete the calibration before switching to extended mode.

Once the calibration process is finished, the registers **REG_TOUCH_TRANSFORM_A-F** are updated accordingly.

3.5 Coprocessor Engine Registers

REG_CMD_DL Definition			
31		13	12
			0
Reserved		r/w	
Offset: 0x100		Reset Value: 0x0	
Bit 31 - 13: Reserved Bits			
Bit 12 - 0: These bits indicate the offset from RAM_DL of the display list commands generated by the coprocessor engine. The coprocessor engine depends on these bits to determine the address in the display list buffer of generated display list commands. It will update this register if the display list commands are generated into the display list buffer. By setting this register properly, the host can specify the starting address in the display list buffer for the coprocessor engine to generate display commands. The valid value range is from 0 to 8191 (sizeof(RAM_DL)-1).			

Register Definition 75 – REG_CMD_DL Definition

REG_CMD_WRITE Definition			
31		12	11
			0
Reserved		r/w	
Offset: 0xFC		Reset Value: 0x0	
Bit 31 - 12: Reserved Bits			
Bit 11 - 0: These bits are updated by the MCU to inform the coprocessor engine of the ending address of valid data feeding into its FIFO . Typically, the host will update this register after it has downloaded the coprocessor commands into its FIFO. The valid range is from 0 to 4095, i.e., within the size of the FIFO .			
Note: The FIFO size of the command buffer is 4096 bytes, and each coprocessor instruction is 4 bytes in size. The value to be written into this register must be 4 bytes aligned.			

Register Definition 76 – REG_CMD_WRITE Definition

REG_CMD_READ Definition			
31		12	11
Reserved		r/o	
Offset: 0xF8		Reset Value: 0x0	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: These bits are updated by the coprocessor engine if the coprocessor engine fetched the command from its FIFO. The host can read this register to determine the FIFO fullness of the coprocessor engine. The valid value range is from 0 to 4095. In the case of an error, the coprocessor engine writes 0xFFF to this register.			
Note: The host shall not write into this register unless in an error recovery case. The default value is zero after the coprocessor engine is reset.			

Register Definition 77 – REG_CMD_READ Definition

REG_CMDB_SPACE Definition			
31		12	11
Reserved		r/o	
Offset: 0x574		Reset Value: 0xFFC	
Bit 31 – 12: Reserved Bits			
Bit 11 – 0: These bits are updated by the coprocessor engine to indicate the free space in RAM_CMD. The host can read this register to determine how many bytes are available to be written into RAM_CMD before writing to RAM_CMD .			
Note: The host shall not write into this register unless in an error recovery case. The default value is zero after the coprocessor engine is reset.			

Register Definition 78 – REG_CMDB_SPACE Definition

REG_CMDB_WRITE Definition			
31			0
w/o			
Offset: 0x578		Reset Value: 0x0	
Bit 31 – 0: The data or command to be written into RAM_CMD . The Host can issue one write transfer with this register address to transfer data less than or equal to the amount of REG_CMDB_SPACE to make bulky data transfer possible.			
Note: This register helps programmers write to the coprocessor FIFO(RAM_CMD) . It was introduced from the FT810 series chip. Always write this register with 4 bytes aligned data.			

Register Definition 79 – REG_CMDB_WRITE Definition

3.6 Miscellaneous Registers

This chapter covers miscellaneous registers, including those for backlight control, interrupts, GPIO, and other functional purposes.

REG_CPURESET Definition				
31		3	2	1
Reserved		r/w	r/w	r/w
Offset: 0x20		Reset Value: 0x0		
Bit 31 – 3: Reserved Bits				
Bit 2: Control the reset of audio engine.				
Bit 1: Control the reset of touch engine.				
Bit 0: Control the reset of coprocessor engine.				
Note: Write 1 to reset the corresponding engine. Write 0 to go back to normal working status. Reading 1 means the engine is in reset status, and reading zero means the engine is in working status.				

Register Definition 80 – REG_CPURESET Definition

REG_MACRO_1 Definition	
31	0
r/w	
Offset: 0xDC	Reset Value: 0x0
Bit 31 – 0: Display list command macro 1. The value of this register will be copied over to RAM_DL to replace the display list command MACRO if its parameter is 1.	

Register Definition 81 – REG_MACRO_1 Definition

REG_MACRO_0 Definition	
31	0
r/w	
Offset: 0xD8	Reset Value: 0x0
Bit 31 – 0: Display list command macro 0. The value of this register will be copied over to RAM_DL to replace the display list command MACRO if its parameter is 0.	

Register Definition 82 – REG_MACRO_0 Definition

REG_PWM_DUTY Definition		
31	8 7	0
Reserved		r/w
Offset: 0xD4	Reset Value: 0x80	
Bit 31 – 8: Reserved Bits		
Bit 7 – 0: These bits define the backlight PWM output duty cycle. The valid range is from 0 to 128. 0 means backlight completely off; 128 means backlight in max brightness.		

Register Definition 83 – REG_PWM_DUTY Definition

REG_PWM_HZ Definition		
31	14 13	0
Reserved		r/w
Offset: 0xD0	Reset Value: 0xFA	
Bit 31 – 14: Reserved Bits		
Bit 13 – 0: These bits define the backlight PWM output frequency in HZ . The default is 250 Hz after reset. The valid frequency is from 250Hz to 10000Hz.		

Register Definition 84 – REG_PWM_HZ Definition

REG_INT_MASK Definition		
31	8/9 7/8	0
Reserved		r/w
Offset: 0xB0	Reset Value: 0xFF/0x1FF	
Bit 31 – 8: Reserved Bits for BT815/6. Or		
Bit 31 – 9: Reserved Bits for BT817/8.		
Bit 7 – 0: INT_MASK for BT815/6. Or		
Bit 8 – 0: INT_MASK for BT817/8.		
These bits are used to mask the corresponding interruption. 1 means to enable the corresponding interrupt source, 0 means to disable the corresponding interrupt source. After reset, all the interrupt source are eligible to trigger an interrupt by default.		
Interrupt source:		
Bit 0: INT_SWAP		
Bit 1: INT_TOUCH		
Bit 2: INT_TAG		
Bit 3: INT_SOUND		
Bit 4: INT_PLAYBACK		
Bit 5: INT_CMDEEMPTY		
Bit 6: INT_CMDFLAG		
Bit 7: INT_CONV_COMPLETE		
Bit 8: INT_UNDERRUN (BT817/8 only)		

Register Definition 85 – REG_INT_MASK Definition

REG_INT_EN Definition	
31	1 0
Reserved	r/w
Offset: 0xAC Reset Value: 0x0	
Bit 31 – 1: Reserved bits	
Bit 0: The host can set this bit to 1 to enable global interruption. To disable the global interrupt, the host can set this bit to 0.	
Note: Refer to the section "Interrupts" in BT817/8 datasheet for more details.	

Register Definition 86 – REG_INT_EN Definition

REG_INT_FLAGS Definition	
31	8/9 7/8 0
Reserved	r/w
Offset: 0xA8 Reset Value: 0x0	
Bit 31 – 8: Reserved Bits for BT815/6. Or	
Bit 31 – 9: Reserved Bits for BT817/8.	
Bit 7 – 0: INT_FLAGS for BT815/6. Or	
Bit 8 – 0: INT_FLAGS for BT817/8.	
These bits are interrupt flags. The host can read these bits to determine which interrupt takes place. These bits are cleared automatically by reading. The host shall not write to this register.	
Interrupt source:	
Bit 0: INT_SWAP	
Bit 1: INT_TOUCH	
Bit 2: INT_TAG	
Bit 3: INT_SOUND	
Bit 4: INT_PLAYBACK	
Bit 5: INT_CMDEEMPTY	
Bit 6: INT_CMDFLAG	
Bit 7: INT_CONV_COMPLETE	
Bit 8: INT_UNDERRUN (BT817/8 only)	

Register Definition 87 – REG_INT_FLAGS Definition

REG_GPIO_DIR Definition	
31	8 7 6 2 1 0
Reserved	r/w Reserved r/w r/w
Offset: 0x90 Reset Value: 0x80	
Bit 31 – 8: Reserved Bits	
Bit 7: It controls the direction of pin DISP .	
Bit 6 – 2: Reserved Bits	
Bit 1: It controls the direction of GPIO1 .	
Bit 0: It controls the direction of GPIO0 .	
Note: 1 is for output, 0 is for input direction. This register is a legacy register for backward compatibility only	

Register Definition 88 – REG_GPIO_DIR Definition

REG_GPIO Definition	
31	8 7 6 5 4 3 2 1 0
Reserved	r/w r/w r/w r/w r/w r/w
Offset: 0x94 Reset Value: 0x0	
Bit 31 – 8: Reserved Bits	
Bit 7: It controls the high or low level of pin DISP .	
Bit 6-5: Drive strength settings for pins GPIO0, GPIO1, CTP_RST_N :	
0b'00:5mA – default, 0b'01:10mA, 0b'10:15mA, 0b'11:20mA	
Bit 4: Drive strength settings for pins PCLK, DISP, VSYNC, HSYNC, DE, R, G, B, BACKLIGHT :	
0b'0: 1.2mA – default, 0b'1: 2.4mA	

Bit 3-2: Drive Strength Setting for pins MISO, MOSI, INT_N : 0b'00:5mA – default, 0b'01:10mA, 0b'10:15mA, 0b'11:20mA
Bit 1: It controls the high or low level of pin GPIO1 .
Bit 0: It controls the high or low level of pin GPIO0 .
Note: Refer to BT817/8 datasheet. This register is a legacy register for backward compatibility only.

Register Definition 89 – REG_GPIO Definition

REG_GPIOX_DIR Definition										
31		16	15	14				4	3	0
Reserved			r/w	reserved				r/w		
Offset: 0x98 Reset Value: 0x8000										
Bit 31 – 16: Reserved Bits										
Bit 15: It controls the direction of pin DISP . The default value is 1, meaning output.										
Bit 14 – 4: Reserved Bits										
Bit 3: It controls the direction of GPIO3 .										
Bit 2: It controls the direction of GPIO2 .										
Bit 1: It controls the direction of GPIO1 .										
Bit 0: It controls the direction of GPIO0 .										
Note: 1 is for output, 0 is for input direction										

Register Definition 90 – REG_GPIOX_DIR Definition

REG_GPIOX Definition														
31		16	15	14	13	12	11	10	9	8		4	3	0
Reserved			r/w	r/w	r/w	r/w	r/w	r/w	Reserved		r/w			
Offset: 0x9C Reset Value: 0x80														
Bit 31 – 16: Reserved Bits														
Bit 15: It controls the high or low level of pin DISP . 1 for high level (default) and 0 for low level.														
Bit 14-13: Drive strength settings for pins GPIO0, GPIO1, GPIO2, GPIO3, CTP_RST_N : 0b'00:5mA – default, 0b'01:10mA, 0b'10:15mA, 0b'11:20mA														
Bit 12: Drive strength settings for pins PCLK, DISP, VSYNC, HSYNC, DE, R, G, B, BACKLIGHT : 0b'0: 1.2mA – default, 0b'1: 2.4mA														
Bit 11-10: Drive Strength Setting for pins MISO, MOSI, INT_N, IO2, IO3, SPIM_SCLK, SPIM_SS_N, SPIM_MOSI, SPIM_MISO, SPIM_IO2, SPIM_IO3 : 0b'00:5mA – default, 0b'01:10mA, 0b'10:15mA, 0b'11:20mA														
Bit 9: It controls the type of pin INT_N . 0b'0: Open Drain – default, 0b'1: Push-pull														
Bit 8 – 4: Reserved Bits														
Bit 3: It controls the high or low level of pin GPIO3 .														
Bit 2: It controls the high or low level of pin GPIO2 .														
Bit 1: It controls the high or low level of pin GPIO1 .														
Bit 0: It controls the high or low level of pin GPIO0 .														
Note: Refer to BT817/8 datasheet for more details. By default, the 8-bit REG_GPIO/REG_GPIO_DIR take control. Once there's any access to REG_GPIOX/REG_GPIOX_DIR, the REG_GPIOX/REG_GPIOX_DIR will take control until a reset happens.														

Register Definition 91 – REG_GPIOX Definition

REG_FREQUENCY Definition	
31	0
r/w	
Offset: 0xC	Reset Value: 0x3938700
Bit 31 – 0: The main clock frequency is 60MHz by default. The value is in Hz. If the host selects the alternative frequency, this register must be updated accordingly.	

Register Definition 92 – REG_FREQUENCY Definition

REG_CLOCK Definition	
31	0
r/o	
Offset: 0x8	Reset Value: 0x0
Bit 31 – 0: These bits are set to zero after reset. The register counts the number of main clock cycles since reset. If the main clock’s frequency is 60Mhz, it will wrap around after about 71 seconds.	

Register Definition 93 – REG_CLOCK Definition

REG_FRAMES Definition	
31	0
r/o	
Offset: 0x4	Reset Value: 0x0
Bit 31 – 0: These bits are set to zero after reset. The register counts the number of screen frames. If the refresh rate is 60Hz, it will wrap up till about 828 days after reset.	

Register Definition 94 – REG_FRAMES Definition

REG_ID Definition	
31	0
Reserved	r/o
Offset: 0x0	Reset Value: 0x7C
Bit 31 – 8: Reserved Bits	
Bit 7 – 0: These bits are the built-in ID of the chip. The value shall always be 0x7C . The host can read this to determine if the chip belongs to the EVE series and is in working mode after booting up.	

Register Definition 95 – REG_ID Definition

REG_SPI_WIDTH Definition	
31	0
Reserved	r/w
Offset: 0x188	Reset Value: 0x0
Bit 31 – 3: Reserved Bits	
Bit 2: Extra dummy on SPI read transfer. Writing 1 to enable one extra dummy byte on SPI read transfer.	
Bit 1 – 0: SPI data bus width: 0b'00: 1 bit – default 0b'01: 2 bit (Dual-SPI) 0b'10: 4 bit (Quad-SPI) 0b'11: undefined	
Note: Refer to BT81Xdatasheet for more details.	

Register Definition 96 – REG_SPI_WIDTH Definition

REG_ADAPTIVE_FRAMERATE Definition	
7	0
Reserved	r/w
Offset: 0x57C	Reset Value: 0x1
Bit 7 – 1: Reserved bits	
Bit 0: Reduce the framerate during complex drawing. 0: Disable 1: Enable	
Note: Please check if the LCD panel datasheet supports the variable frame rate.	

Register Definition 97 – REG_ADAPTIVE_FRAMERATE Definition

REG_UNDERRUN Definition	
31	0
r/o	
Offset: 0x60C	Reset Value: 0x0
Bit 31 – 0: It counts underrun lines. When a line underruns, it is incremented. An application can sample it on each frame swap to determine if the previous frame suffered an underrun.	
Note: BT817/8 specific register.	

Register Definition 98 – REG_UNDERRUN Definition

REG_AH_HCYCLE_MAX Definition			
31	12	11	0
Reserved		r/w	
Offset: 0x610	Reset Value: 0x0		
Bit 31 – 12: Reserved bits			
Bit 11 – 0: The maximum PCLK count of horizontal line when adaptive HSYNC is enabled. Value 0 means adaptive HSYNC feature is disabled. The valid value shall be greater than REG_HCYCLE .			
Note: BT817/8 specific register.			

Register Definition 99 – REG_AH_HCYCLE_MAX Definition

REG_PCLK_FREQ Definition				
31	12	11	10	9
Reserved		r/w	r/w	
Offset: 0x614	Reset Value: 0x8A1			
Bit 31 – 12: Reserved bits				
Bit 11 – 10: Configure the range of output fractional PCLK frequency for EXTSYNC mode. Refer to BT817/8 datasheet for details.				
Bit 9 - 0: Configure the output fractional PCLK frequency for EXTSYNC mode, i.e., REG_PCLK is set to 1. Refer to BT817/8 datasheet for details.				
Note: BT817/8 specific register.				
It is recommended to refer to the table <i>RGB PCLK Frequency in EXTSYNC mode</i> in the Parallel RGB Interface section of the BT817/8 datasheet which has recommended values.				
Coprocessor command CMD_PCLKFREQ can be used to set the register too. See the section CMD_PCLKFREQ for information on this command.				

Register Definition 100 – REG_PCLK_FREQ Definition

REG_PCLK_2X Definition	
7	0
Reserved	
Offset: 0x618	Reset Value: 0x0
Bit 7 – 1: Reserved bits	
Bit 0: graphics engine outputs 1 or 2 pixels per PCLK . 0 means 1 pixel per clock, 1 means 2 pixel per clock.	
Note: BT817/8 specific register. When graphics engine outputs 2 pixels per PCLK , the values loaded in the following registers must be even:	
<ul style="list-style-type: none"> • REG_HSIZE • REG_HOFFSET • REG_HCYCLE • REG_HSYNC0 • REG_HSYNC1 	

Register Definition 101 – REG_PCLK_2X Definition

3.7 Special Registers

The registers listed here are not mapped to **RAM_REG**; instead, they reside at special addresses.

REG_TRACKER Definition			
31	16	15	0
r/o	reserved		r/o
Offset: 0x7000		Reset Value: 0x0	
Bit 31 – 16: These bits are set to indicate the tracking value for the tracked graphics objects. The coprocessor calculates the tracking value that the touching point takes within the predefined range. Please check the CMD_TRACK for more details.			
Bit 15 – 8: Reserved Bits			
Bit 7 – 0: These bits are set to indicate the tag value of a graphics object which is being touched.			

Register Definition 102 – REG_TRACKER Definition

REG_TRACKER_1 Definition			
31	16	15	0
r/o	reserved		r/o
Offset: 0x7004		Reset Value: 0x0	
Bit 31 – 16: These bits are set to indicate the tracking value for the tracked graphics objects. The coprocessor calculates the tracking value that the touching point takes within the predefined range. Please check the CMD_TRACK for more details.			
Bit 15 – 8: Reserved Bits			
Bit 7 – 0: These bits are set to indicate the tag value of a graphics object which is being touched as the second point.			
Note: It is only applicable for extended mode of CTSE .			

Register Definition 103 – REG_TRACKER_1 Definition

REG_TRACKER_2 Definition			
31	16	15	0
r/o	reserved		r/o
Offset: 0x7008		Reset Value: 0x0	
Bit 31 – 16: These bits are set to indicate the tracking value for the tracked graphics objects. The coprocessor calculates the tracking value that the touching point takes within the predefined range. Please check the CMD_TRACK for more details.			
Bit 15 – 8: Reserved Bits			
Bit 7 – 0: These bits are set to indicate the tag value of a graphics object which is being touched as the third point.			
Note: It is only applicable for extended mode of CTSE .			

Register Definition 104 – REG_TRACKER_2 Definition

REG_TRACKER_3 Definition			
31	16	15	0
r/o	reserved		r/o
Offset: 0x700C		Reset Value: 0x0	
Bit 31 – 16: These bits are set to indicate the tracking value for the tracked graphics objects. The coprocessor calculates the tracking value that the touching point takes within the predefined range. Please check the CMD_TRACK for more details.			
Bit 15 – 8: Reserved Bits			
Bit 7 – 0: These bits are set to indicate the tag value of a graphics object which is being touched as the fourth point.			
Note: It is only applicable for extended mode of CTSE .			

Register Definition 105 – REG_TRACKER_3 Definition

REG_TRACKER_4 Definition			
31	16	15	0
	r/o	reserved	r/o
Offset: 0x7010		Reset Value: 0x0	
Bit 31 – 16: These bits are set to indicate the tracking value for the tracked graphics objects. The coprocessor calculates the tracking value that the touching point takes within the predefined range. Please check the CMD_TRACK for more details.			
Bit 15 – 8: Reserved Bits			
Bit 7 – 0: These bits are set to indicate the tag value of a graphics object which is being touched as the fifth point.			
Note: It is only applicable for extended mode of CTSE .			

Register Definition 106 – REG_TRACKER_4 Definition

REG_MEDIAFIFO_READ Definition	
31	0
	r/o
Offset: 0x7014	Reset Value: 0x0
Bit 31 – 0: The value specifies the read pointer pointing to the address in RAM_G as the media FIFO .	

Register Definition 107 – REG_MEDIAFIFO_READ Definition

REG_MEDIAFIFO_WRITE Definition	
31	0
	w/o
Offset: 0x7018	Reset Value: 0x0
Bit 31 – 0: The value specifies the write pointer pointing to the address in RAM_G as the media FIFO .	

Register Definition 108 – REG_MEDIAFIFO_WRITE Definition

REG_PLAY_CONTROL Definition	
7	0
	w/o
Offset: 0x714E	Reset Value: 0x1
Bit 7 – 0: video playback control. The following values are defined: 0: pause playback 1: play normally 0xFF: exit playback	

Register Definition 109 – REG_PLAY_CONTROL Definition

REG_ANIM_ACTIVE Definition	
31	0
	r/o
Offset: 0x702C	Reset Value: 0x0
Bit 31 – 0: 32-bit mask of currently playing animations. Each bit indicates the active state of animation channel. 0 means animation ends, and 1 means animation runs.	
Note: Only applicable for the animation channel is played with ANIM_ONCE flag.	

Register Definition 110 – REG_ANIM_ACTIVE Definition

REG_COPRO_PATCH_PTR Definition	
15	0
	r/o
Offset: 0x7162	Reset Value: NA
Bit 15 – 0: The address of coprocessor patch pointer.	
Note: This register shall be only used for the coprocessor recovery purpose. Refer to Coprocesor Faults .	

Register Definition 111 – REG_COPRO_PATCH_PTR Definition

4 Display List Commands

The graphics engine retrieves instructions from the display list memory, **RAM_DL**, in the form of commands. Each command is 4 bytes long, and since **RAM_DL** is 8 KB in size, a single display list can contain up to 2048 commands. The graphics engine executes the corresponding operations based on the definitions of these commands.

4.1 Graphics State

The graphics state, which determines the effects of a drawing operation, is maintained within the graphics context. Individual elements of this state can be modified using appropriate display list commands, while the entire current state can be saved and restored with the **SAVE_CONTEXT** and **RESTORE_CONTEXT** commands.

Note that the bitmap drawing state is handled differently: While the bitmap handle itself is included in the graphics context, the parameters for each bitmap handle are not. These parameters are neither saved nor restored by the **SAVE_CONTEXT** and **RESTORE_CONTEXT** commands. Instead, they are configured using the **BITMAP_SOURCE**, **BITMAP_LAYOUT/BITMAP_LAYOUT_H** and **BITMAP_SIZE/BITMAP_SIZE_H** commands. Once set, these parameters can be used in any display list by referencing the same bitmap handle until they are updated.

The **SAVE_CONTEXT** and **RESTORE_CONTEXT** commands use a 4-level stack in addition to the current graphics context. The table below provides a detailed overview of the parameters stored within the graphics context.

Parameters	Default values	Commands
func & ref	ALWAYS, 0	ALPHA_FUNC
func & ref	ALWAYS, 0	STENCIL_FUNC
src & dst	SRC_ALPHA, ONE_MINUS_SRC_ALPHA	BLEND_FUNC
Cell value	0	CELL
Alpha value	0	COLOR_A
Red, Blue, Green colors	(255,255,255)	COLOR_RGB
Line width in 1/16 pixels	16	LINE_WIDTH
Point size in 1/16 pixels	16	POINT_SIZE
Width & height of scissor	HSIZE,2048	SCISSOR_SIZE
Starting coordinates of scissor	(x, y) = (0,0)	SCISSOR_XY
Current bitmap handle	0	BITMAP_HANDLE
Bitmap transform coefficients	+1.0,0,0,0,+1.0,0	BITMAP_TRANSFORM_A-F
Stencil clear value	0	CLEAR_STENCIL
Tag clear value	0	CLEAR_TAG
Mask value of stencil	255	STENCIL_MASK
spass and sfail	KEEP,KEEP	STENCIL_OP
Tag buffer value	255	TAG
Tag mask value	1	TAG_MASK
Alpha clear value	0	CLEAR_COLOR_A
RGB clear color	(0,0,0)	CLEAR_COLOR_RGB
Palette source address	RAM_G	PALETTE_SOURCE
Units of pixel precision	1/16 pixel	VERTEX_FORMAT, VERTEX2F

Table 10 – Graphics Context

4.2 Command Encoding

Each display list command is encoded as a 32-bit value. The most significant bits specify the command type, while any command parameters are contained in the least significant bits. Bits marked as “reserved” must always be set to zero.

4.3 Command Groups

4.3.1 Setting Graphics State

ALPHA_FUNC	set the alpha test function
BITMAP_EXT_FORMAT	specify the extended format of the bitmap
BITMAP_HANDLE	set the bitmap handle
BITMAP_LAYOUT/ BITMAP_LAYOUT_H	set the source bitmap memory format and layout for the current handle
BITMAP_SIZE/ BITMAP_SIZE_H	set the screen drawing of bitmaps for the current handle
BITMAP_SOURCE	set the source address for bitmap graphics. It can be a flash address.
BITMAP_SWIZZLE	specify the color channel swizzle for a bitmap
BITMAP_TRANSFORM_A-F	set the components of the bitmap transform matrix
BLEND_FUNC	set pixel arithmetic function
CELL	set the bitmap cell number for the VERTEX2F command
CLEAR	clear buffers to preset values
CLEAR_COLOR_A	set clear value for the alpha channel
CLEAR_COLOR_RGB	set clear values for red, green and blue channels
CLEAR_STENCIL	set clear value for the stencil buffer
CLEAR_TAG	set clear value for the tag buffer
COLOR_A	set the current color alpha
COLOR_MASK	enable or disable writing of color components
COLOR_RGB	set the current color red, green and blue
LINE_WIDTH	set the line width
POINT_SIZE	set point size
RESTORE_CONTEXT	restore the current graphics context from the context stack
SAVE_CONTEXT	push the current graphics context on the context stack
SCISSOR_SIZE	set the size of the scissor clip rectangle
SCISSOR_XY	set the top left corner of the scissor clip rectangle
STENCIL_FUNC	set function and reference value for stencil testing
STENCIL_MASK	control the writing of individual bits in the stencil planes
STENCIL_OP	set stencil test actions
TAG	set the current tag value
TAG_MASK	control the writing of the tag buffer
VERTEX_FORMAT	set the precision of VERTEX2F coordinates
VERTEX_TRANSLATE_X	specify the vertex transformation's X translation component
VERTEX_TRANSLATE_Y	specify the vertex transformation's Y translation component
PALETTE_SOURCE	Specify the base address of the palette

4.3.2 Drawing Actions

BEGIN	start drawing a graphics primitive
END	finish drawing a graphics primitive
VERTEX2F	supply a vertex with fractional coordinates
VERTEX2II	supply a vertex with unsigned coordinates

4.3.3 Execution Control

NOP	No Operation
JUMP	execute commands at another location in the display list
MACRO	execute a single command from a macro register
CALL	execute a sequence of commands at another location in the display list
RETURN	return from a previous CALL command
DISPLAY	end the display list

4.4 ALPHA_FUNC

Specify the alpha test function

Encoding

31	24	23	11	10	8	7	0
0x09	reserved			func		ref	

Parameters

func

Specifies the test function, one of NEVER, LESS, LEQUAL, GREATER, GEQUAL, EQUAL, NOTEQUAL, or ALWAYS. The initial value is ALWAYS (7)

NAME	VALUE
NEVER	0
LESS	1
LEQUAL	2
GREATER	3
GEQUAL	4
EQUAL	5
NOTEQUAL	6
ALWAYS	7

ref

Specifies the reference value for the alpha test. The initial value is 0

Graphics context

The values of func and ref are part of the graphics context, as described in section 4.1.

SEE ALSO

None

4.5 BEGIN

Start drawing a graphics primitive.

Encoding

31	24	23	4	3	0
0x1F	reserved			prim	

Parameters

prim

The graphics primitive to be executed. The valid values are defined as below:

Name	Value	Description
BITMAPS	1	Bitmap drawing primitive
POINTS	2	Point drawing primitive
LINES	3	Line drawing primitive
LINE_STRIP	4	Line strip drawing primitive
EDGE_STRIP_R	5	Edge strip right side drawing primitive
EDGE_STRIP_L	6	Edge strip left side drawing primitive

EDGE_STRIP_A	7	Edge strip above drawing primitive
EDGE_STRIP_B	8	Edge strip below side drawing primitive
RECTS	9	Rectangle drawing primitive

Table 11 - Graphics Primitive Definition

Description

All supported primitives are listed in the table above. A primitive is selected using the **BEGIN** command, and it remains active until a new primitive is chosen with another **BEGIN** command.

It is important to note that the actual drawing of the primitives does not occur until **VERTEX2II** or **VERTEX2F** is executed.

Examples

Drawing points, lines and bitmaps:



```

dl ( BEGIN (POINTS) );
dl ( VERTEX2II (50, 5, 0, 0) );
dl ( VERTEX2II (110, 15, 0, 0) );
dl ( BEGIN (LINES) );
dl ( VERTEX2II (50, 45, 0, 0) );
dl ( VERTEX2II (110, 55, 0, 0) );
dl ( BEGIN (BITMAPS) );
dl ( VERTEX2II (50, 65, 31, 0x45) );
dl ( VERTEX2II (110, 75, 31, 0x46) );
```

Graphics context

None

See also

[END](#)

4.6 BITMAP_EXT_FORMAT

Define the bitmap using the extended format.

Encoding

31	24	23	16	15	0
0x2E	reserved			format	

Parameters

format
 Bitmap pixel format.

Description

If **BITMAP_LAYOUT** specifies a format for **GLFORMAT**, the actual format is instead determined by **BITMAP_EXT_FORMAT**.

Valid values for the field format are:

Format Name	Value	Bits per Pixel
ARGB1555	0	16
L1	1	1
L4	2	4
L8	3	8
RGB332	4	8
ARGB2	5	8
ARGB4	6	16
RGB565	7	16
TEXT8X8	9	8
TEXTVGA	10	8
BARGRAPH	11	8
PALETTED565	14	8
PALETTED4444	15	8
PALETTED8	16	8
L2	17	2
COMPRESSED_RGBA_ASTC_4x4_KHR	37808	8.00
COMPRESSED_RGBA_ASTC_5x4_KHR	37809	6.40
COMPRESSED_RGBA_ASTC_5x5_KHR	37810	5.12
COMPRESSED_RGBA_ASTC_6x5_KHR	37811	4.27
COMPRESSED_RGBA_ASTC_6x6_KHR	37812	3.56
COMPRESSED_RGBA_ASTC_8x5_KHR	37813	3.20
COMPRESSED_RGBA_ASTC_8x6_KHR	37814	2.67
COMPRESSED_RGBA_ASTC_8x8_KHR	37815	2.00
COMPRESSED_RGBA_ASTC_10x5_KHR	37816	2.56
COMPRESSED_RGBA_ASTC_10x6_KHR	37817	2.13
COMPRESSED_RGBA_ASTC_10x8_KHR	37818	1.60
COMPRESSED_RGBA_ASTC_10x10_KHR	37819	1.28
COMPRESSED_RGBA_ASTC_12x10_KHR	37820	1.07
COMPRESSED_RGBA_ASTC_12x12_KHR	37821	0.89

Table 12 – Bitmap formats and bits per pixel

Graphics context

None

See also

[BITMAP_LAYOUT](#)

4.7 BITMAP_HANDLE

Select the bitmap handle to use.

Encoding

31	24 23	5 4	0
0x05	reserved	handle	

Parameters

handle

Bitmap handle. The initial value is 0. The valid value range is from 0 to 31.

Description

By default, bitmap handles 16 through 31 are reserved for built-in fonts, while handle 15 serves as scratch bitmap for coprocessor engine commands such as **CMD_GRADIENT**, **CMD_BUTTON**, **CMD_KEYS**, **CMD_PLAYVIDEO** and **CMD_ANIMFRAME**.

Graphics context

The bitmap handle value is included in the graphics context, as described in Section 4.1.

See also

[BITMAP_LAYOUT](#), [BITMAP_SIZE](#)

4.8 BITMAP_LAYOUT

Specify the source bitmap memory format and layout for the current handle.

Encoding

31	2423	1918	98	0
0x07	format	linstride	height	

Parameters

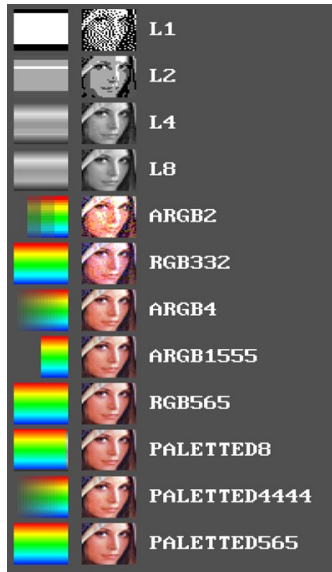
format

Bitmap pixel format. The valid range is from 0 to 17 and defined as per the table below.

Name	Value	Bits/pixel	Alpha bits	Red bits	Green bits	Blue bits
ARGB1555	0	16	1	5	5	5
L1	1	1	1	0	0	0
L4	2	4	4	0	0	0
L8	3	8	8	0	0	0
RGB332	4	8	0	3	3	2
ARGB2	5	8	2	2	2	2
ARGB4	6	16	4	4	4	4
RGB565	7	16	0	5	6	5
TEXT8X8	9	-	-	-	-	-
TEXTVGA	10	-	-	-	-	-
BARGRAPH	11	-	-	-	-	-
PALETTED565	14	8	0	5	6	5
PALETTED4444	15	8	4	4	4	4
PALETTED8	16	8	8	8	8	8
L2	17	2	2	0	0	0
GLFORMAT	31	Check BITMAP_EXT_FORMAT				

Table 13 – BITMAP_LAYOUT Format List

The examples of the various supported bitmap formats, excluding **TEXTVGA** are shown as below:



BARGRAPH – This format renders bitmap data as a bar graph. For each x-coordinate, the corresponding byte in a data array is checked; if the byte value is less than y-coordinate, an opaque pixel is drawn; otherwise, the pixel is transparent. The result is a visual bar graph of the bitmap data. Using the BARGRAPH format, a bitmap of up to screen width x 256 pixels can be displayed. The graph’s orientation, width and height can be modified through the bitmap transform matrix.

TEXT8X8 – is a display format that uses fixed 8x8 font for rendering text. In this format, the bitmap is stored as a byte array in graphics RAM, where each byte corresponds to a character indexed in an internal 8x8 CP437¹ font. Drawing routines use built-in bitmap handles 16 & 17 to render the TEXT8X8 format. Essentially, the bitmap functions as a character grid, allowing a single bitmap to be drawn over all or part of the display. Each byte in the bitmap data represents one 8x8 pixel character cell, enabling efficient text rendering on the display.

TEXTVGA – is a display format that uses fixed 8x16 font with TEXTVGA syntax for rendering. The bitmap is stored as a TEXTVGA array in graphics RAM, where each element indexes an internal 8x16 CP437 font. Built-in bitmap handles 18 and 19 are used to render the TEXTVGA format, supporting additional control information such as background color, foreground color, and cursor positioning. The bitmap effectively functions as a TEXTVGA grid, allowing a single bitmap to be drawn over all or part of the display. Each TEXTVGA data element in the bitmap corresponds to one 8x16 pixel character cell, enabling precise text and attribute rendering.

¹ https://en.wikipedia.org/wiki/Code_page_437

linestride – Bitmap line strides, in bytes. This represents the amount of memory used for each line of bitmap pixels.

For **L1, L2, and L4** format, the necessary data must be padded to make it byte aligned. Normally, it can be calculated with the following formula:

$$\text{linestride} = \text{width} * \text{byte/pixel}$$

For example, if one bitmap is 64x32 pixels in L4 format, the line stride shall be
 $(64 * 1/2 = 32)$

height - Bitmap height, in lines

Description

For more details about memory layout according to pixel format, refer to the figures and tables:

L1 Format							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Pixel 0	Pixel 1	Pixel 2	Pixel 3	Pixel 4	Pixel 5	Pixel 6	Pixel 7

L2 Format			
Bit 7	6	Bit 5	4
Pixel 0	Pixel 1	Pixel 2	Pixel 3

L4 Format	
7	0
Pixel 0	Pixel 1

L8 Format	
7	0
Pixel 0	

Table 14 – L1/L2/L4/L8 Pixel Format

ARGB2 Format			
7	6	5	4
Alpha Channel	Red Channel	Green Channel	Blue Channel

RGB332 Format		
7	5	4
Red Channel	Green Channel	Blue Channel

Table 15 – ARGB2/RGB332 Pixel Format

RGB565/PALETTE565 Format		
15	11	10
Red Channel	Green Channel	Blue Channel

Table 16 – RGB565/PALETTE565 Pixel Format

ARGB1555 Format		
15	14	10
Alpha Channel	Red Channel	Green Channel

ARGB4/PALETTE4444 Format			
15	12	11	8
Alpha Channel	Red Channel	Green Channel	Blue Channel

Table 17 – ARGB1555/ARGB4/PALETTE4444 Pixel Format

PALETTE8 Format			
31	24	23	0
Alpha Channel	Red Channel	Green Channel	Blue Channel

Table 18 – PALETTE8 Pixel Format

Note: PALETTE8 is 8 bits per pixel as each pixel is represented by an 8-bit index value in the look-up table. It has a color depth of 24-bits and 8-bit alpha.

Graphics Context

None

Note: PALETTE8 format is supported indirectly, and it is different from PALETTE format in FT80X. To render Alpha, Red, Green and Blue channels, multi-pass drawing action is required.

The following display list snippet shows:

```

//addr_pal is the starting address of palette lookup table in RAM_G
//bitmap source(palette indices) is starting from address 0

dl(BITMAP_HANDLE(0))
dl(BITMAP_LAYOUT(PALETTE8, width, height))
dl(BITMAP_SIZE(NEAREST, BORDER, BORDER, width, height))

dl(BITMAP_SOURCE(0)) //bitmap source(palette indices)

dl(BEGIN(BITMAPS))
dl(BLEND_FUNC(ONE, ZERO))

//Draw Alpha channel
dl(COLOR_MASK(0,0,0,1))
dl(PALETTE_SOURCE(addr_pal+3))
dl(VERTEX2II(0, 0, 0, 0))

//Draw Red channel
dl(BLEND_FUNC(DST_ALPHA, ONE_MINUS_DST_ALPHA))
dl(COLOR_MASK(1,0,0,0))
dl(PALETTE_SOURCE(addr_pal+2))
dl(VERTEX2II(0, 0, 0, 0))

//Draw Green channel
dl(COLOR_MASK(0,1,0,0))
dl(PALETTE_SOURCE(addr_pal + 1))
dl(VERTEX2II(0, 0, 0, 0))

//Draw Blue channel
dl(COLOR_MASK(0,0,1,0))
dl(PALETTE_SOURCE(addr_pal))
dl(VERTEX2II(0, 0, 0, 0))

```

Code Snippet 12 – PALETTE8 Drawing Example

See also

[BITMAP_HANDLE](#), [BITMAP_SIZE](#), [BITMAP_SOURCE](#), [PALETTE_SOURCE](#)

4.9 BITMAP_LAYOUT_H

Specify the 2 most significant bits of the source bitmap memory format and layout for the current handle.

Encoding

31		24	23			4	3	2	1	0	
0x28			reserved					linstride		height	

Parameters

linstride

The 2 most significant bits of the 12-bit line stride parameter value specified to **BITMAP_LAYOUT**.

height

The 2 most significant bits of the 11-bit height parameter value specified to **BITMAP_LAYOUT**.

Description

This command is the extension command of **BITMAP_LAYOUT** for bitmap larger than 511 by 511 pixels.

Examples

NA

See also

[BITMAP_LAYOUT](#)

4.10 BITMAP_SIZE

Specify the screen drawing of bitmaps for the current handle

Encoding

31		24	23	21	20	19	18	17		9	8		0
0x08			reserved	filter	wrapx	wrapy	width				height		

Parameters

filter

Bitmap filtering mode, one of NEAREST or BILINEAR.
 The value of NEAREST is 0 and the value of BILINEAR is 1.

wrapx

Bitmap x wrap mode, one of REPEAT or BORDER
 The value of BORDER is 0 and the value of REPEAT is 1.

wrapy

Bitmap y wrap mode, one of REPEAT or BORDER
 The value of BORDER is 0 and the value of REPEAT is 1.

width

Drawn bitmap width, in pixels. From 1 to 511. Zero has special meaning.

height

Drawn bitmap height, in pixels. From 1 to 511. Zero has special meaning.

Description

This command controls bitmap rendering, including the on-screen size of the bitmap, wrap behavior, and the filtering function. Note that if wrapx or wrapy is set to **REPEAT**, the corresponding memory layout dimension (BITMAP_LAYOUT line stride or height) must be power of two; otherwise the result is undefined.

For width and height, the value from 1 to 511 specify the bitmap width and height in pixels. The value zero has a special meaning: if no **BITMAP_SIZE_H** is present before, or if the high bit in **BITMAP_SIZE_H** is zero, a value of zero represents 2048 pixels rather than 0 pixels.

4.11 BITMAP_SIZE_H

Specifies the 2 most significant bits of bitmap dimensions for the current handle.

Encoding

31	24 23	4	3	2	1	0
0x29	Reserved	width	height			

Parameters

width

2 most significant bits of bitmap width. The initial value is zero.

Height

2 most significant bits of bitmap height. The initial value is zero.

Description

This command is the extension command of **BITMAP_SIZE** for bitmap larger than 511 by 511 pixels.

Graphics context

None

See also

[BITMAP_HANDLE](#), [BITMAP_LAYOUT](#), [BITMAP_SOURCE](#), [BITMAP_SIZE](#)

4.12 BITMAP_SOURCE

Specifies the source address of the bitmap data in **RAM_G** or **flash memory**.

Encoding

31		24 23	0
0x01		addr	

Parameters

addr

Bitmap address in **RAM_G** or **flash memory**, aligned with respect to the bitmap format. For example, if the bitmap format is **RGB565/ARGB4/ARGB1555**, the bitmap source shall be aligned to 2 bytes.

Description

The bitmap source address determines the location of the bitmap graphic data:

- If bit 23 is 0, bits 0-22 indicate the byte address in **RAM_G**.
- If bit 23 is 1, bits 0-22, multiplied by 32, represent the byte address in external flash memory.

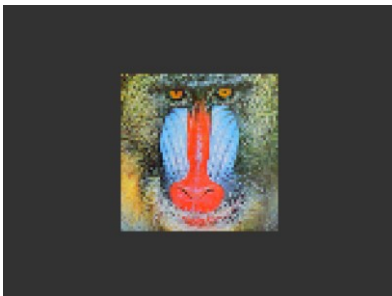
Note that in rare cases when setting the bitmap source address in **RAM_G**, the calculated address may be negative (for example, when loading a font that begins at **RAM_G+0** and uses a pointer to raw data that results in a negative value). In such cases, the value written to **BITMAP_SOURCE** should be masked so that only bits 0-22 are set, ensuring that bit 23 is not written as 1.

For example, if **addr** is (0x800000 | 422), the byte address in external flash memory refers to 13,504(422 x 32).

However, only bitmap data in ASTC-specific formats can be rendered directly from flash memory. Bitmap data in any non-ASTC format stored in flash memory must be copied into **RAM_G** using **CMD_FLASHREAD** before EVE can render it correctly.

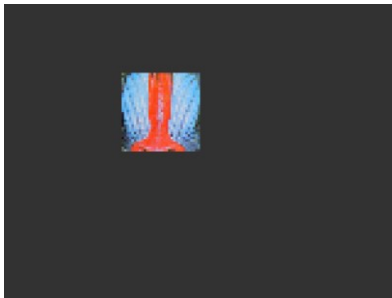
Examples

Drawing a 64 x 64 bitmap, loaded at address 0:



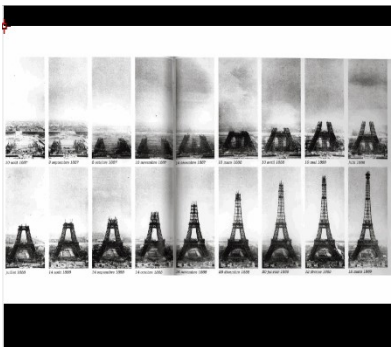
```
dl( BITMAP_SOURCE(0) );
dl( BITMAP_LAYOUT( RGB565, 128, 64 ) );
dl( BITMAP_SIZE( NEAREST, BORDER, BORDER, 64, 64 ) );
dl( BEGIN( BITMAPS ) );
dl( VERTEX2II( 48, 28, 0, 0 ) );
```

Using the same graphics data, but modifying the source and size to display only a 32 x 32 pixel detail:



```
dl ( BITMAP_SOURCE(128 * 16 + 32) );
dl ( BITMAP_LAYOUT(RGB565, 128, 64) );
dl ( BITMAP_SIZE(NEAREST, BORDER, BORDER, 32, 32) );
dl ( BEGIN(BITMAPS) );
dl ( VERTEX2II(48, 28, 0, 0) );
```

Display one 800x480 image by using extended display list commands mentioned above:



```
dl (BITMAP_HANDLE(0));
dl (BITMAP_SOURCE(0));
dl (BITMAP_SIZE_H(1, 0));
dl (BITMAP_SIZE(NEAREST, BORDER, BORDER, 288, 480));
dl (BITMAP_LAYOUT_H(1, 0));
dl (BITMAP_LAYOUT(ARGB1555, 576, 480));
dl (BEGIN(BITMAPS));
dl (VERTEX2II(76, 25, 0, 0));
dl (END());
```

Graphics context

None

See also

[BITMAP_LAYOUT](#), [BITMAP_SIZE](#)

4.13 BITMAP_SWIZZLE

Set the source for the red, green, blue and alpha channels of a bitmap.

Encoding

31		24	23			12	11	9	8	6	5	3	2	0
0x2f				Reserved				r	g	b	a			

Parameters

- r**
red component source channel

- g**
green component source channel

- b**
blue component source channel

- a**
alpha component source channel

Description

Bitmap swizzle allows the channels of a bitmap to be rearranged or copied into the final color channels. Each resulting color component can be taken from any of the bitmap's color components, or it can be set to a constant value of zero or one. The valid values for each source are:

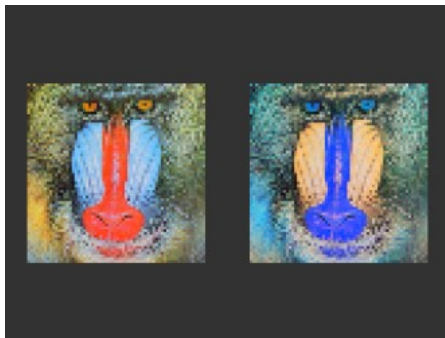
Name	Value	Description
ZERO	0	Set the source channel to zero
ONE	1	Set the source channel to 1
RED	2	Specify RED component as source channel
GREEN	3	Specify GREEN component as source channel
BLUE	4	Specify BLUE component as source channel
ALPHA	5	Specify ALPHA component as source channel

Bitmap swizzle is applied only when the format parameter of **BITMAP_LAYOUT** is set to **GLFORMAT**. Otherwise, the four components remain in their default order, with the default swizzle being (RED, GREEN, BLUE, ALPHA)

Note: Please refer to OpenGL API specification for more details

Examples

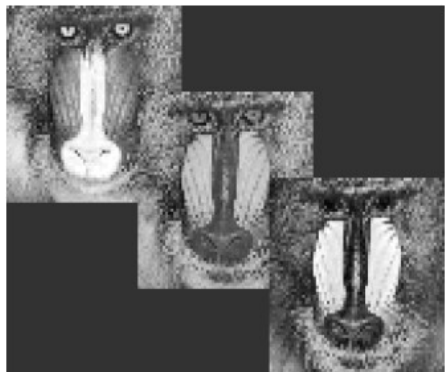
Bitmap drawn with default swizzle, and with red/blue exchanged:



```

dl(BITMAP_SOURCE(0));
dl(BITMAP_LAYOUT(GLFORMAT, 128, 64));
dl(BITMAP_EXT_FORMAT(RGB565));
dl(BITMAP_SIZE(NEAREST, BORDER, BORDER, 64, 64));
dl(BEGIN(BITMAPS));
dl(BITMAP_SWIZZLE(RED, GREEN, BLUE, ALPHA));
dl(VERTEX2II(8, 28, 0, 0));
dl(BITMAP_SWIZZLE(BLUE, GREEN, RED, ALPHA));
dl(VERTEX2II(88, 28, 0, 0));
  
```

Red, green, and blue channels extracted to create three grayscale images:



```

dl(BITMAP_LAYOUT(GLFORMAT, 128, 64));
dl(BITMAP_EXT_FORMAT(RGB565));
dl(BEGIN(BITMAPS));
dl(BITMAP_SWIZZLE(RED, RED, RED, ALPHA));
dl(VERTEX2II(0, 0, 0, 0));
dl(BITMAP_SWIZZLE(GREEN, GREEN, GREEN, ALPHA));
dl(VERTEX2II(48, 28, 0, 0));
dl(BITMAP_SWIZZLE(BLUE, BLUE, BLUE, ALPHA));
dl(VERTEX2II(96, 56, 0, 0));
  
```

Graphics Context

None

See also

None

4.14 BITMAP_TRANSFORM_A

Specifies the A coefficient of the bitmap transformation matrix.

Encoding

31		24	23		18	17	16		0
0x15			reserved			p	v		

Parameters

p
 Precision control: 0 is 8.8, 1 is 1.15. The initial value is 0.

v
 The A component of the bitmap transformation matrix, represented in signed 8.8 or 1.15 fixed-point format. Its initial value is 256.

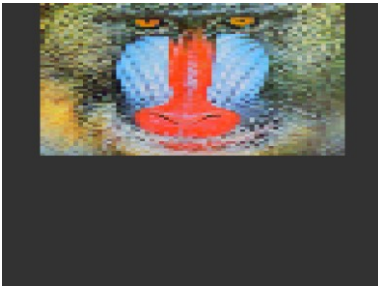
Note: The parameters of this command are changed in BT81X.

Description

The **BITMAP_TRANSFORM_A-F** coefficients define the bitmap transformation matrix used to perform operations such as scaling, rotation, and translation, similar to the transformation functionality in OpenGL.

Examples

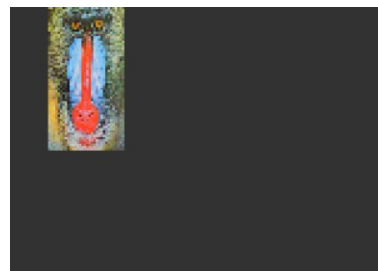
A value of 0.5 (128) causes the bitmap to appear double width:



```

dl ( BITMAP_SOURCE ( 0 ) );
dl ( BITMAP_LAYOUT ( RGB565, 128, 64 ) );
dl ( BITMAP_TRANSFORM_A ( 128 ) );
dl ( BITMAP_SIZE ( NEAREST, BORDER, BORDER, 128, 128 ) );
dl ( BEGIN ( BITMAPS ) );
dl ( VERTEX2II ( 16, 0, 0, 0 ) );
  
```

A value of 2.0 (512) gives a half-width bitmap:



```

dl ( BITMAP_SOURCE ( 0 ) );
dl ( BITMAP_LAYOUT ( RGB565, 128, 64 ) );
dl ( BITMAP_TRANSFORM_A ( 512 ) );
dl ( BITMAP_SIZE ( NEAREST, BORDER, BORDER, 128, 128 ) );
dl ( BEGIN ( BITMAPS ) );
dl ( VERTEX2II ( 16, 0, 0, 0 ) );
  
```

Graphics Context

The value of p, v is part of the graphics context, as described in section [4.1](#)

See also

None

4.15 BITMAP_TRANSFORM_B

Specifies the B coefficient of the bitmap transformation matrix

Encoding

31	24	23	18	17	16	0
0x16	reserved	P	v			

Parameters

p

Precision control: 0 is 8.8, 1 is 1.15. The initial value is 0.

v

The B component of the bitmap transformation matrix, represented in signed 8.8 or 1.15 fixed-point format. Its initial value is 0.

Description

The **BITMAP_TRANSFORM_A-F** coefficients define the bitmap transformation matrix, which is used to perform operations such as scaling, rotation and translation functionality similar to OpenGL transform functionality.

Note: The parameters of this command are changed in BT81X.

Graphics context

The value of p, v is part of the graphics context, as described in Section 4.1.

See also

None

4.16 BITMAP_TRANSFORM_C

Specifies the C coefficient of the bitmap transformation matrix.

Encoding

31	24	23	0
0x17	C		

Parameters

c

The c component of the bitmap transformation matrix, represented in signed 15.8 fixed-point format. Its initial value is 0.

Description

The **BITMAP_TRANSFORM_A-F** coefficients form the bitmap transformation matrix, enabling operations such as scaling, rotation, and translation, similar to the transformation functions in OpenGL.

Graphics context

The value of c is part of the graphics context, as described in section 4.1.

See also

None

4.17 BITMAP_TRANSFORM_D

Specifies the d coefficient of the bitmap transformation matrix.

Encoding

31		24	23		18	17	16		0
0x18			reserved			p	v		

Parameters

p
 Precision control: 0 is 8.8, 1 is 1.15. The initial value is 0.

v
 The d component of the bitmap transformation matrix, stored in signed 8.8 or 1.15 fixed-point format. Its initial value is 0.

Note: The parameters of this command are changed in BT81X.

Description

BITMAP_TRANSFORM_A-F coefficients are used to perform bitmap transform functionalities such as scaling, rotation and translation. These are like OpenGL transform functionality.

Graphics context

The value of p,v of the graphics context, as described in Ssection 4.1.

See also

None

4.18 BITMAP_TRANSFORM_E

Specify the E coefficient of the bitmap transform matrix.

Encoding

31		24	23		18	17	16		0
0x19			reserved			p	v		

Parameters

p
 Precision control: 0 is 8.8, 1 is 1.15. The initial value is 0.

v
 The e component of the bitmap transformation matrix, in signed 8.8 or 1.15 fixed point form. Its initial value is 256.

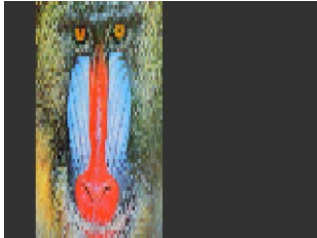
Description

BITMAP_TRANSFORM_A-F coefficients are used to perform bitmap transform functions such as scaling, rotation, and translation. These operate in a manner like OpenGL transform functionality.

Note: The parameters of this command are changed in BT81X.

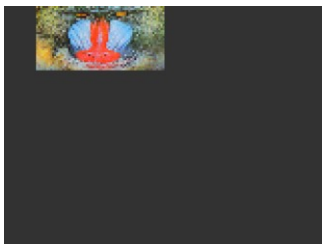
Examples

A value of 0.5 (128) causes the bitmap to appear double height:



```
dl( BITMAP_SOURCE(0) );
dl( BITMAP_LAYOUT( RGB565, 128, 64 ) );
dl( BITMAP_TRANSFORM_E(128) );
dl( BITMAP_SIZE( NEAREST, BORDER, BORDER, 128, 128 ) );
dl( BEGIN( BITMAPS ) );
dl( VERTEX2II( 16, 0, 0, 0 ) );
```

A value of 2.0 (512) gives a half-height bitmap:



```
dl( BITMAP_SOURCE(0) );
dl( BITMAP_LAYOUT( RGB565, 128, 64 ) );
dl( BITMAP_TRANSFORM_E(512) );
dl( BITMAP_SIZE( NEAREST, BORDER, BORDER, 128, 128 ) );
dl( BEGIN( BITMAPS ) );
dl( VERTEX2II( 16, 0, 0, 0 ) );
```

Graphics context

The value of p and v of the graphics context, as described in section [4.1](#)

See also

None

4.19 BITMAP_TRANSFORM_F

Specify the f coefficient of the bitmap transform matrix

Encoding

31	24	23	0
0x1A	f		

Parameters

f
 The f component of the bitmap transformation matrix, stored in signed 15.8-bit fixed-point format. Its initial value is 0.

Description

BITMAP_TRANSFORM_A-F coefficients are used to perform bitmap transform functionalities such as scaling, rotation and translation. These are like **OpenGL** transformation functionality.

Graphics context

The value of f is part of the graphics context, as described in Section [4.1](#).

See also

None

4.20 BLEND_FUNC

Specify pixel arithmetic

Encoding

31	24	23	6	5	3	2	0
0x0B	reserved				src	dst	

Parameters

src

Specifies how the source blending factor is computed. One of ZERO, ONE, SRC_ALPHA, DST_ALPHA, ONE_MINUS_SRC_ALPHA or ONE_MINUS_DST_ALPHA. The initial value is SRC_ALPHA (2).

dst

Specifies how the destination blending factor is computed, one of the same constants as src. The initial value is ONE_MINUS_SRC_ALPHA(4)

Name	Value	Description
ZERO	0	Check OpenGL definition
ONE	1	Check OpenGL definition
SRC_ALPHA	2	Check OpenGL definition
DST_ALPHA	3	Check OpenGL definition
ONE_MINUS_SRC_ALPHA	4	Check OpenGL definition
ONE_MINUS_DST_ALPHA	5	Check OpenGL definition

Table 19 – BLEND_FUNC Constant Value Definition

Description

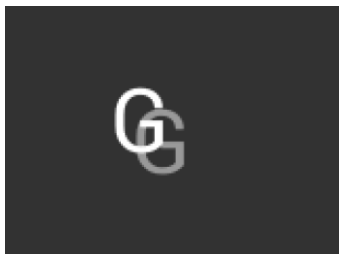
The blend function controls how new color values are combined with the values already in the color buffer. Given a pixel value source and a previous value in the color buffer destination, the computed color is:

$$source \times src + destination \times dst$$

For each color channel: red, green, blue and alpha.

Examples

The default blend function of (SRC_ALPHA, ONE_MINUS_SRC_ALPHA) causes drawing to overlay the destination using the alpha value:



```

d1 ( BEGIN (BITMAPS) );
d1 ( VERTEX2II (50, 30, 31, 0x47) );
d1 ( COLOR_A ( 128 ) );
d1 ( VERTEX2II (60, 40, 31, 0x47) );
```

A destination factor of zero means that destination pixels are not used:



```
dl ( BEGIN (BITMAPS) );
dl ( BLEND_FUNC (SRC_ALPHA, ZERO) );
dl ( VERTEX2II (50, 30, 31, 0x47) );
dl ( COLOR_A ( 128 ) );
dl ( VERTEX2II (60, 40, 31, 0x47) );
```

Using the source alpha to control how much of the destination to keep:



```
dl ( BEGIN (BITMAPS) );
dl ( BLEND_FUNC (ZERO, SRC_ALPHA) );
dl ( VERTEX2II (50, 30, 31, 0x47) );
```

Graphics context

The values of src and dst are part of the graphics context, as described in Section [4.1](#).

See also

[COLOR_A](#)

4.21 CALL

Execute a sequence of commands at another location in the display list

Encoding

31		24	23		16	15		0
0x1D			reserved			dest		

Parameters

dest

The display list number which the display command is to be switched. **EVE** has the stack to store the return address. To come back to the next command of source address, the **RETURN** command can help.

The valid range is from 0 to 2047(sizeof(**RAM_DL**)/4-1).

Description

CALL and RETURN have a 4-level stack in addition to the current pointer. Any additional CALL/RETURN done will lead to unexpected behavior.

Graphics context

None

See also

[JUMP](#), [RETURN](#)

4.22 CELL

Specify the bitmap cell number for the **VERTEX2F** command.

Encoding

31	24	23	7	6	0
0x06		reserved			cell

Parameters

cell

bitmap cell number. The initial value is 0

Graphics context

The value of cell is part of the graphics context, as described in section [4.1](#).

See also

None

4.23 CLEAR

Clear buffers to preset values.

Encoding

31	24	23	3	2	1	0	
0x26		reserved			c	s	t

Parameters

c

Clear color buffer. Setting this bit to 1 will clear the color buffer to the pre-set value. Setting this bit to 0 will maintain the color buffer with an unchanged value. The preset value is defined in command CLEAR_COLOR_RGB for RGB channel and CLEAR_COLOR_A for alpha channel.

s

Clear stencil buffer. Setting this bit to 1 will clear the stencil buffer to the pre-set value. Setting this bit to 0 will maintain the stencil buffer with an unchanged value. The preset value is defined in command CLEAR_STENCIL.

t

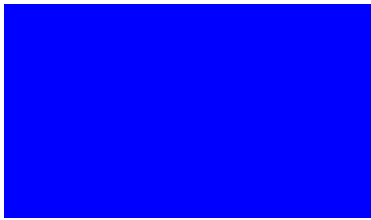
Clear tag buffer. Setting this bit to 1 will clear the tag buffer to the pre-set value. Setting this bit to 0 will maintain the tag buffer with an unchanged value. The pre-set value is defined in command CLEAR_TAG.

Description

The scissor test and the buffer write masks affect the clear operation. The scissor test limits the cleared rectangle, and the buffer write masks limit which buffers are affected. The alpha function, blend function, and stencil state do not affect the clear.

Examples

To clear the screen to bright blue:



```
dl( CLEAR_COLOR_RGB(0, 0, 255) );
dl( CLEAR(1, 0, 0) );
```

To clear part of the screen to gray, part to blue using scissor rectangles:



```
dl( CLEAR_COLOR_RGB(100, 100, 100) );
dl( CLEAR(1, 1, 1) );
dl( CLEAR_COLOR_RGB(0, 0, 255) );
dl( SCISSOR_SIZE(30, 120) );
dl( CLEAR(1, 1, 1) );
```

Graphics context

None

See also

[CLEAR_COLOR_A](#), [CLEAR_STENCIL](#), [CLEAR_TAG](#), [CLEAR_COLOR_RGB](#)

4.24 CLEAR_COLOR_A

Specifies clear value for the alpha channel.

Encoding

31	24	23	8	7	0
0x0F		reserved		alpha	

Parameters

alpha

Alpha value is used when the color buffer is cleared. The initial value is 0.

Graphics context

The value of alpha is part of the graphics context, as described in section 4.1.

See also

[CLEAR_COLOR_RGB](#), [CLEAR](#)

4.25 CLEAR_COLOR_RGB

Specify clear values for red, green and blue channels.

Encoding

31	24	23	16	15	8	7	0
0x02		red		blue		green	

Parameters

red

Red value is used when the color buffer is cleared. The initial value is 0.

green

Green value is used when the color buffer is cleared. The initial value is 0.

blue

Blue value is used when the color buffer is cleared. The initial value is 0.

Description

Sets the color values used by the following **CLEAR**.

Examples

To clear the screen to bright blue:



```
dl ( CLEAR_COLOR_RGB(0, 0, 255) );
dl ( CLEAR(1, 1, 1) );
```

To clear part of the screen to gray, part to blue using scissor rectangles:



```
dl ( CLEAR_COLOR_RGB(100, 100, 100) );
dl ( CLEAR(1, 1, 1) );
dl ( CLEAR_COLOR_RGB(0, 0, 255) );
dl ( SCISSOR_SIZE(30, 120) );
dl ( CLEAR(1, 1, 1) );
```

Graphics context

The values of red, green and blue are part of the graphics context, as described in Section 4.1.

See also

[CLEAR_COLOR_A](#), [CLEAR](#)

4.26 CLEAR_STENCIL

Specify clear value for the stencil buffer.

Encoding

31	24	23	reserved	8	7	0
0x11						s

Parameters

s
 Value used when the stencil buffer is cleared. The initial value is 0

Graphics context

The value of s is part of the graphics context, as described in section 4.1.

See also

[CLEAR](#)

4.27 CLEAR_TAG

Specify clear value for the tag buffer

Encoding

31	24	23	8	7	0
0x12		reserved		t	

Parameters

t
 Value used when the tag buffer is cleared. The initial value is 0.

Graphics context

The value of s is part of the graphics context, as described in Section 4.1.

See also

[TAG](#), [TAG_MASK](#), [CLEAR](#)

4.28 COLOR_A

Set the current color alpha.

Encoding

31	24	23	8	7	0
0x10		reserved		alpha	

Parameters

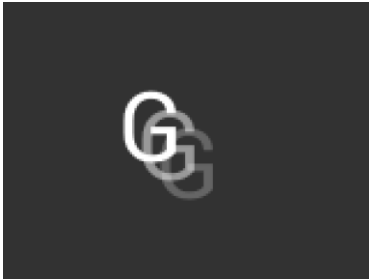
alpha
 Alpha value for the current color. The initial value is 255.

Description

Sets the alpha value applied to drawn elements – points, lines, and bitmaps. How the alpha value affects image pixels depends on BLEND_FUNC; the default behavior is transparent blend.

Examples

Drawing three characters with transparency 255, 128, and 64:



```

dl ( BEGIN (BITMAPS) );
dl ( VERTEX2II (50, 30, 31, 0x47) );
dl ( COLOR_A ( 128 ) );
dl ( VERTEX2II (58, 38, 31, 0x47) );
dl ( COLOR_A ( 64 ) );
dl ( VERTEX2II (66, 46, 31, 0x47) );
  
```

Graphics context

The value of alpha is part of the graphics context, as described in Section [4.1](#).

See also

[COLOR_RGB](#), [BLEND_FUNC](#)

4.29 COLOR_MASK

Enable or disable writing of color components.

Encoding

31	24	23	4	3	2	1	0		
0x20	reserved					r	g	b	a

Parameters

- r**
Enables or disables the red channel update of the color buffer. The initial value is 1, which means enabled.

- g**
Enables or disables the green channel update of the color buffer. The initial value is 1, which means enabled.

- b**
Enables or disables the blue channel update of the color buffer. The initial value is 1, which means enabled.

- a**
Enables or disables the alpha channel update of the color buffer. The initial value is 1, which means enabled.

Description

The color mask controls whether a pixel’s color values are updated. It can be used to selectively update only the red, green, blue or alpha channels of an image. More often, it is used to completely disable color updates while updating the tag and stencil buffers.

Examples

Draw an ‘8’ digit in the middle of the screen. Then paint an invisible 40-pixel circular touch area into the tag buffer:



```
dl( BEGIN(BITMAPS) );
dl( VERTEX2II(68, 40, 31, 0x38) );
dl( POINT_SIZE(40 * 16) );
dl( COLOR_MASK(0, 0, 0, 0) );
dl( BEGIN(POINTS) );
dl( TAG( 0x38 ) );
dl( VERTEX2II(80, 60, 0, 0) );
```

Graphics context

The values of r, g, b and a are part of the graphics context, as described in section 4.1.

See also

[TAG_MASK](#)

4.30 COLOR_RGB

Set the current color red, green and blue.

Encoding

31		24	23		16	15		8	7		0
0x04			red			blue			green		

Parameters

red

Red value for the current color. The initial value is 255.

green

Green value for the current color. The initial value is 255.

blue

Blue value for the current color. The initial value is 255.

Description

Sets the red, green and blue values of the color buffer which will be applied to the following drawing operation.

Examples

Drawing three characters with different colors:



```
dl( BEGIN(BITMAPS) );
dl( VERTEX2II(50, 38, 31, 0x47) );
dl( COLOR_RGB( 255, 100, 50 ) );
dl( VERTEX2II(80, 38, 31, 0x47) );
dl( COLOR_RGB( 50, 100, 255 ) );
dl( VERTEX2II(110, 38, 31, 0x47) );
```

Graphics context

The values of red, green and blue are part of the graphics context, as described in Section 4.1.

See also

[COLOR_A](#)

4.31 DISPLAY

End the display list. All the commands following this command will be ignored.

Encoding

31	24	23	0
0x0	reserved		

Parameters

None

Graphics context

None

See also

None

4.32 END

End drawing a graphics primitive.

Encoding

31	24	23	0
0x21	reserved		

Parameters

None

Description

It is recommended to have an **END** for each **BEGIN**. However, advanced users may avoid the usage of **END** to save space for extra graphics instructions in RAM_DL.

Graphics context

None

See also

[BEGIN](#)

4.33 JUMP

Execute commands at another location in the display list.

Encoding

31	24	23	16	15	0
0x1E	reserved			dest	

Parameters

dest

Display list number to be jumped. The valid range is from 0 to 2047(sizeof(**RAM_DL**)/4-1).

Graphics context

None

See also

[CALL](#)

4.34 LINE_WIDTH

Specify the width of lines to be drawn with primitive LINES in 1/16-pixel precision.

Encoding

31	24	23	12	11	0
0x0E	reserved			width	

Parameters

width

Line width in 1/16-pixel precision. The initial value is 16.

Description

Sets the width of drawn lines. The width is the distance from the center of the line to the outermost drawn pixel, in units of 1/16 pixel. The valid range is from 1 to 4095. i.e., from 1 to 255 pixels.

Please note the LINE_WIDTH command will affect the LINES, LINE_STRIP, RECTS, EDGE_STRIP_A/B/R/L primitives.

Note: The lines are drawn with the requested width, but below around 6 the pixels become very dark and are hard to see. Half pixel lines (width 8) are totally usable.

Examples

The second line is drawn with a width of 80, for a 5-pixel radius:



```
dl ( BEGIN(LINES) );
dl ( VERTEX2F(16 * 10, 16 * 30) );
dl ( VERTEX2F(16 * 150, 16 * 40) );
dl ( LINE_WIDTH(80) );
dl ( VERTEX2F(16 * 10, 16 * 80) );
dl ( VERTEX2F(16 * 150, 16 * 90) );
```

Graphics context

The value of width is part of the graphics context, as described in Section 4.1.

See also

None

4.35 MACRO

Execute a single command from a macro register.

Encoding

31	24	23	10
0x25	reserved		m

Parameters

m

Macro registers to read. Value 0 means the content in **REG_MACRO_0** is to be fetched and inserted in place. Value 1 means **REG_MACRO_1** is to be fetched and inserted in place. The content of **REG_MACRO_0** or **REG_MACRO_1** shall be a valid display list command, otherwise the behavior is undefined.

Graphics context

None

See also

None

4.36 NOP

No operation.

Encoding

31	24	23	0
0x2D	reserved		

Parameters

None

Description

Does nothing. May be used as a spacer in display lists, if required.

Graphics context

None

See also

None

4.37 PALETTE_SOURCE

Specify the base address of the palette.

Encoding

31	24	23	22	21	0
0x2A	reserved	addr			

Parameters

addr

Address of palette in **RAM_G**, 2-byte alignment is required if pixel format is **PALETTED4444** or **PALETTED565**. The initial value is **RAM_G**.

Description

Specify the base address in **RAM_G** for palette

Graphics context

The value of addr is part of the graphics context, as described in Section 4.1.

See also

None

4.38 POINT_SIZE

Specify the radius of points

Encoding

31	24 23	13 12	0
0x0D	reserved	size	

Parameters

size

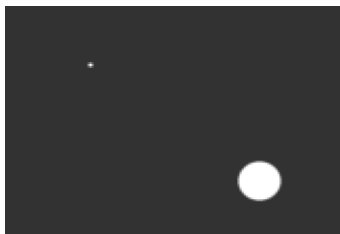
Point radius in 1/16-pixel precision. The initial value is 16. The valid range is from zero to 8191, i.e., from 0 to 511 pixels.

Description

Sets the size of drawn points. The width is the distance from the center of the point to the outermost drawn pixel, in units' of 1/16 pixels.

Examples

The second point is drawn with a width of 160, for a 10-pixel radius:



```
dl( BEGIN(POINTS) );
dl( VERTEX2II(40, 30, 0, 0) );
dl( POINT_SIZE(160) );
dl( VERTEX2II(120, 90, 0, 0) );
```

Graphics context

The value of size is part of the graphics context, as described in Section 4.1.

See also

None

4.39 RESTORE_CONTEXT

Restore the current graphics context from the context stack.

Encoding

31		24		0
0x23		reserved		

Parameters

None

Description

Restores the current graphics context, as described in Section 4.1. Four levels of **SAVE** and **RESTORE** stacks are available. Any extra **RESTORE_CONTEXT** will load the default values into the present context.

Examples

Saving and restoring context means that the second 'G' is drawn in red, instead of blue:



```

dl( BEGIN(BITMAPS) );
dl( COLOR_RGB( 255, 0, 0 ) );
dl( SAVE_CONTEXT() );
dl( COLOR_RGB( 50, 100, 255 ) );
dl( VERTEX2II(80, 38, 31, 0x47) );
dl( RESTORE_CONTEXT() );
dl( VERTEX2II(110, 38, 31, 0x47) );
  
```

Graphics context

None

See also

[SAVE_CONTEXT](#)

4.40 RETURN

Return from a previous **CALL** command.

Encoding

31		24		0
0x24		reserved		

Parameters

None

Description

CALL and **RETURN** have 4 levels of stack in addition to the current pointer. Any additional **CALL/RETURN** done will lead to unexpected behavior.

Graphics context

None

See also

[CALL](#)

4.41 SAVE_CONTEXT

Push the current graphics context on the context stack.

Encoding

31	24	23	0
0x22	reserved		

Parameters

None

Description

Saves the current graphics context, as described in section 4.1. Any extra **SAVE_CONTEXT** will throw away the earliest saved context.

Examples

Saving and restoring context means that the second 'G' is drawn in red, instead of blue:



```

d1 ( BEGIN (BITMAPS) );
d1 ( COLOR_RGB ( 255, 0, 0 ) );
d1 ( SAVE_CONTEXT ( ) );
d1 ( COLOR_RGB ( 50, 100, 255 ) );
d1 ( VERTEX2II (80, 38, 31, 0x47) );
d1 ( RESTORE_CONTEXT ( ) );
d1 ( VERTEX2II (110, 38, 31, 0x47) );
```

Graphics context

None

See also

[RESTORE_CONTEXT](#)

4.42 SCISSOR_SIZE

Specify the size of the scissor clip rectangle.

Encoding

31	24	23	12	11	0
0x1C	width		height		

Parameters

width

The width of the scissor clip rectangle, in pixels. The initial value is 2048.
 The value of zero will cause zero output on screen.
 The valid range is from zero to 2048.

height

The height of the scissor clip rectangle, in pixels. The initial value is 2048.
 The value of zero will cause zero output on screen.
 The valid range is from zero to 2048.

Description

Sets the width and height of the scissor clip rectangle, which limits the drawing area.

Examples

Setting a 40 x 30 scissor rectangle clips the clear and bitmap drawing:



```
dl( SCISSOR_XY(40, 30) );
dl( SCISSOR_SIZE(80, 60) );
dl( CLEAR_COLOR_RGB(0, 0, 255) );
dl( CLEAR(1, 1, 1) );
dl( BEGIN(BITMAPS) );
dl( VERTEX2II(35, 20, 31, 0x47) );
```

Graphics context

The values of width and height are part of the graphics, as described in Section [4.1](#).

See also

None

4.43 SCISSOR_XY

Specify the top left corner of the scissor clip rectangle.

Encoding

31	24	23	22	21	11	10	0
0x1B		reserved		x		y	

Parameters

x

The unsigned x coordinate of the scissor clip rectangle, in pixels. The initial value is 0. The valid range is from zero to 2047.

y

The unsigned y coordinates of the scissor clip rectangle, in pixels. The initial value is 0. The valid range is from zero to 2047.

Description

Sets the top-left position of the scissor clip rectangle, which limits the drawing area.

Examples

Setting a 40 x 30 scissor rectangle clips the clear and bitmap drawing:



```
dl ( SCISSOR_XY (40, 30) );
dl ( SCISSOR_SIZE (80, 60) );
dl ( CLEAR_COLOR_RGB (0, 0, 255) );
dl ( CLEAR (1, 1, 1) );
dl ( BEGIN (BITMAPS) );
dl ( VERTEX2II (35, 20, 31, 0x47) );
```

Graphics context

The values of x and y are part of the graphics, as described in Section [4.1](#).

See also

None

4.44 STENCIL_FUNC

Set function and reference value for stencil testing.

Encoding

31	24	23	20	19	16	15	8	7	0
0x0A		reserved	func	ref			mask		

Parameters

func

Specifies the test function, one of **NEVER**, **LESS**, **LEQUAL**, **GREATER**, **GEQUAL**, **EQUAL**, **NOTEQUAL**, or **ALWAYS**. The initial value is **ALWAYS**.

About the value of these constants, refer to **ALPHA_FUNC**.

ref

Specifies the reference value for the stencil test. The initial value is 0.

mask

Specifies a mask that is ANDed with the reference value and the stored stencil value. The initial value is 255

Description

Stencil test rejects or accepts pixels depending on the result of the test function defined in func parameter, which operates on the current value in the stencil buffer against the reference value.

Examples

Refer to [STENCIL_OP](#).

Graphics context

The values of func, ref and mask are part of the graphics context, as described in Section [4.1](#).

See also

[STENCIL_OP](#), [STENCIL_MASK](#)

4.45 STENCIL_MASK

Control the writing of individual bits in the stencil planes

Encoding

31	24	23	8	7	0
0x13		reserved		mask	

Parameters

mask

The mask is used to enable writing stencil bits. The initial value is 255

Graphics context

The value of mask is part of the graphics context, as described in Section 4.1.

See also

[STENCIL_FUNC](#), [STENCIL_OP](#), [TAG_MASK](#)

4.46 STENCIL_OP

Set stencil test actions.

Encoding

31	24	23	6	5	3	2	0
0x0C		reserved		sfail		spass	

Parameters

sfail

Specifies the action to take when the stencil test fails, one of KEEP, ZERO, REPLACE, INCR, DECR, and INVERT. The initial value is KEEP (1)

spass

Specifies the action to take when the stencil test passes, one of the same constants as sfail. The initial value is KEEP (1)

Name	Value	Description
ZERO	0	check OpenGL definition
KEEP	1	check OpenGL definition
REPLACE	2	check OpenGL definition
INCR	3	check OpenGL definition
DECR	4	check OpenGL definition
INVERT	5	check OpenGL definition

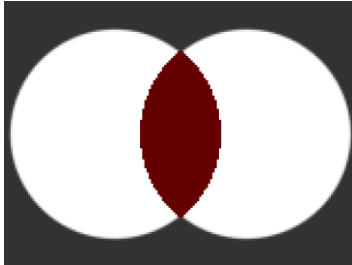
Table 20 – STENCIL_OP Constants Definition

Description

The stencil operation specifies how the stencil buffer is updated. The operation selected depends on whether the stencil test passes or not.

Examples

Draw two points, incrementing stencil at each pixel, then draw the pixels with value 2 in red:



```

dl( STENCIL_OP(INCR, INCR) );
dl( POINT_SIZE(760) );
dl( BEGIN(POINTS) );
dl( VERTEX2II(50, 60, 0, 0) );
dl( VERTEX2II(110, 60, 0, 0) );
dl( STENCIL_FUNC(EQUAL, 2, 255) );
dl( COLOR_RGB(100, 0, 0) );
dl( VERTEX2II(80, 60, 0, 0) );
  
```

Graphics context

The values of `sfail` and `spass` are part of the graphics context, as described in Section [4.1](#).

See also

[STENCIL_FUNC](#), [STENCIL_MASK](#)

4.47 TAG

Attach the tag value for the following graphics objects drawn on the screen. The initial buffer tag is 255.

Encoding

31	24	23	8	7	0
0x03	reserved			s	

Parameters

s
 Tag value. Valid value range is from 1 to 255.

Description

The initial value of the tag buffer is specified by command **CLEAR_TAG** and takes effect by issuing command **CLEAR**. The **TAG** command can specify the value of the tag buffer that applies to the graphics objects when they are drawn on the screen. This **TAG** value will be assigned to all the following objects, unless the **TAG_MASK** command is used to disable it. Once the following graphics objects are drawn, they are attached with the tag value successfully. When the graphics objects attached with the tag value are touched, the register **REG_TOUCH_TAG** will be updated with the tag value of the graphics object being touched.

If there are no TAG commands in one display list, all the graphics objects rendered by the display list will report the tag value from the tag buffer in **REG_TOUCH_TAG** when they are touched.

Graphics context

The value of `s` is part of the graphics context, as described in Section [4.1](#).

See also

[CLEAR_TAG](#), [TAG_MASK](#)

4.48 TAG_MASK

Control the writing of the tag buffer

Encoding

31		24		23		1	0
0x14				reserved			mask

Parameters

mask

Allow updates to the tag buffer. The initial value is one and it means the tag buffer is updated with the value given by the TAG command. Therefore, the following graphics objects will be attached to the tag value given by the TAG command.

Value zero means the tag buffer is set as the default value, rather than the value given by **TAG** command in the display list.

Description

Every graphics object drawn on screen is attached with the tag value which is defined in the tag buffer. The tag buffer can be updated by the **TAG** command.

The default value of the tag buffer is determined by **CLEAR_TAG** and **CLEAR** commands. If there is no **CLEAR_TAG** command present in the display list, the default value in tag buffer shall be 0.

TAG_MASK command decides whether the tag buffer takes the value from the default value of the tag buffer or the TAG command of the display list.

Graphics context

The value of mask is part of the graphics context, as described in Section [4.1](#).

See also

[TAG](#), [CLEAR_TAG](#), [STENCIL_MASK](#), [COLOR_MASK](#)

4.49 VERTEX2F

Start the operation of graphics primitives at the specified screen coordinate, in the pixel precision defined by **VERTEX_FORMAT**.

Encoding

31		30		29		15	14	0
0x1		x				y		

Parameters

x

Signed x-coordinate in units of pixel precision defined in command **VERTEX_FORMAT**, which by default is 1/16-pixel precision.

y

Signed y-coordinate in units of pixel precision defined in command **VERTEX_FORMAT**, which by default is 1/16-pixel precision.

Description

The pixel precision depends on the value of **VERTEX_FORMAT**. The maximum range of coordinates depends on pixel precision and is described in the VERTEX_FORMAT instruction.

Graphics context

None

See also

[VERTEX_FORMAT](#)

4.50 VERTEX2II

Start the operation of graphics primitive at the specified coordinates in pixel precision.

Encoding

31	30	29		21	20		12	11	7	6	0	
0x2		x				y				handle		cell

Parameters

x
 X-coordinate in pixels, unsigned integer ranging from 0 to 511.

y
 Y-coordinate in pixels, unsigned integer ranging from 0 to 511.

handle
 Bitmap handle. The valid range is from 0 to 31.

cell
 Cell number. Cell number is the index of the bitmap with same bitmap layout and format. For example, for handle 31, cell 65 means the character "A" in built in font 31.

Note: The handle and cell parameters are ignored unless the graphics primitive is specified as bitmap by command **BEGIN(BITMAPS)**, prior to this command.

Description

To draw the graphics primitives beyond the coordinate range [(0,0), (511, 511)], use VERTEX2F instead.

Graphics context

None

See also

[BITMAP_HANDLE](#), [CELL](#), [VERTEX2F](#)

4.51 VERTEX_FORMAT

Set the precision of **VERTEX2F** coordinates.

Encoding

31	24 23	32	0
0x27	reserved	frac	

Parameters

frac

Number of fractional bits in X, Y coordinates. Valid range is from 0 to 4. The initial value is 4.

Description

VERTEX2F uses 15-bit signed numbers for its (X,Y) coordinate. This command controls the interpretation of these numbers by specifying the number of fractional bits.

By varying the format, an application can trade range against precision.

Frac value	Unit of pixel precision	VERTEX2F range
0	1 <i>pixel</i>	-16384 to 16383
1	1/2 <i>pixel</i>	-8192 to 8191
2	1/4 <i>pixel</i>	-4096 to 4095
3	1/8 <i>pixel</i>	-2048 to 2047
4	1/16 <i>pixel</i>	-1024 to 1023

Table 21 – VERTEX_FORMAT and Pixel Precision

Graphics context

The value of **frac** is part of the graphics context, as described in Section [4.1](#).

See also

[VERTEX2F](#), [VERTEX_TRANSLATE_X](#), [VERTEX_TRANSLATE_Y](#)

4.52 VERTEX_TRANSLATE_X

Specify the vertex transformations X translation component.

Encoding

31	24	23	17	16	0
0x2B		reserved		x	

Parameters

x

Signed x-coordinate in 1/16 pixel. The initial value is 0.

Description

Specifies the offset added to vertex X coordinates. This command allows drawing to be shifted to the screen. It applies to both **VERTEX2F** and **VERTEX2II** commands.

Graphics context

The value of **x** is part of the graphics context

See also

NONE

4.53 VERTEX_TRANSLATE_Y

Specify the vertex transformation's Y translation component.

Encoding

31	24	23	17	16	0
0x2C	reserved	reserved	reserved	reserved	y

Parameters

y
 Signed y-coordinate in 1/16 pixel. The initial value is 0

Description

Specifies the offset added to vertex Y coordinates. This command allows drawing to be shifted to the screen. It applies to both **VERTEX2F** and **VERTEX2II** commands.

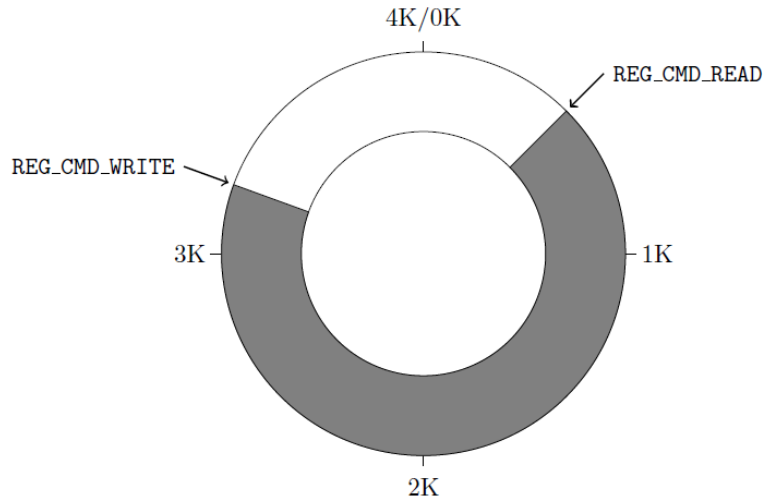
Graphics context

The value of y is part of the graphics context

5 Coprocessor Engine

5.1 Command FIFO

The coprocessor engine is fed via a 4K byte **FIFO** called **RAM_CMD**. The MCU writes coprocessor commands or display list commands into the **FIFO**, and the coprocessor engine reads and executes the commands. The MCU updates the register **REG_CMD_WRITE** to indicate that there are new commands in the **FIFO**, and the coprocessor engine updates **REG_CMD_READ** after the commands have been executed. Therefore, when **REG_CMD_WRITE** is equal to **REG_CMD_READ**, it indicates the **FIFO** is empty and all the commands are executed without error.



To compute the free space, the MCU can apply the following formula:

$$\begin{aligned} \text{fullness} &= (\text{REG_CMD_WRITE} - \text{REG_CMD_READ}) \bmod 4096 \\ \text{free space} &= (4096 - 4) - \text{fullness}; \end{aligned}$$

This calculation does not report 4096 bytes of free space, to prevent completely wrapping the circular buffer and making it appear empty.

If enough space is available in FIFO, the MCU writes the commands at the appropriate location in the **FIFO** and then updates **REG_CMD_WRITE**. To simplify the MCU code, **EVE** automatically wraps continuous writes from the top address (**RAM_CMD** + 4095) back to the bottom address (**RAM_CMD** + 0) if the starting address of a write transfer is within **RAM_CMD**.

FIFO entries are always 4 bytes wide – it is an error for either **REG_CMD_READ** or **REG_CMD_WRITE** to have a value that is not a multiple of 4 bytes. Each command issued to the coprocessor engine may take 1 or more words: the length depends on the command itself, and any appended data. Some commands are followed by variable-length data, so the command size may not be a multiple of 4 bytes. In this case the coprocessor engine ignores the extra 1, 2 or 3 bytes and continues reading the next command at the following 4-byte boundary.

To offload work from the MCU for checking the free space in the circular buffer, **EVE** offers a pair of registers **REG_CMDB_SPACE** and **REG_CMDB_WRITE**. It enables the MCU to write commands and data to the coprocessor in a bulk transfer, without computing the free space in the circular buffer and increasing the address. If the amount of data transferred is less than the value in the register **REG_CMDB_SPACE**, the MCU can safely write all the data to **REG_CMDB_WRITE** in one write transfer. All write to **REG_CMDB_WRITE** are appended to the command **FIFO** and may be of any length that is a multiple of 4 bytes. To determine the free space of **FIFO**, reading **REG_CMDB_SPACE** and checking if it is equal to 4092 is easier and faster than comparing **REG_CMD_WRITE** and **REG_CMD_READ**.

5.2 Widgets

The Coprocessor engine provides pre-defined widgets for users to construct screen designs easily. The picture below illustrates the commands to render widgets and effects.

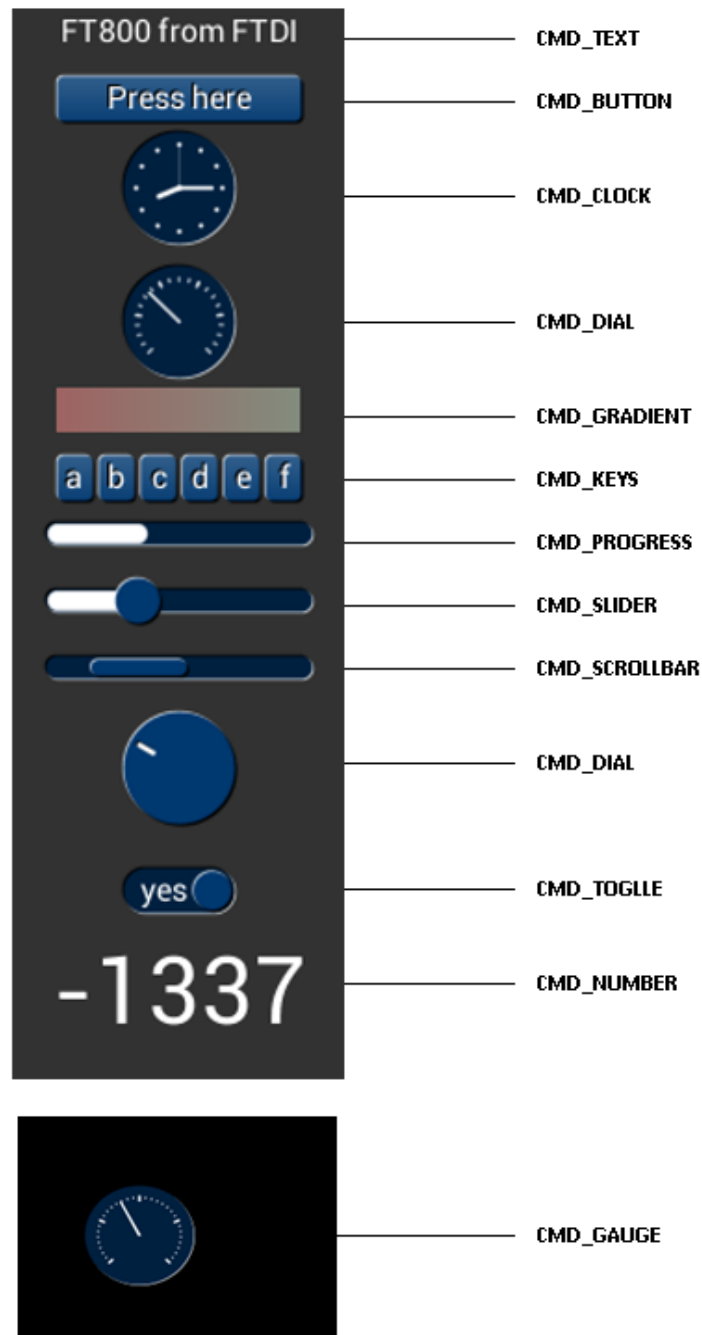


Figure 3 – Widget List

5.2.1 Common Physical Dimensions

This section contains the common physical dimensions of the widgets, unless it is specified in the widget introduction.

- All rounded corners have a radius that is computed from the font used for the widget (curvature of lowercase 'o' character).

$$\text{Radius} = \text{font height} * 3 / 16$$

- All 3D shadows are drawn with:
 - Highlight offsets 0.5 pixels above and left of the object
 - Shadow offsets 1.0 pixel below and right of the object.
- For widgets such as progress bar, scrollbar and slider, the output will be a vertical widget in the case where width and height parameters are of same value.

5.2.2 Color Settings

Coprocessor engine widgets are drawn with the color designated by the precedent commands: **CMD_FGCOLOR**, **CMD_BGCOLOR** and **COLOR_RGB**. The coprocessor engine will determine to render the different areas of the widgets in different colors according to these commands.

Usually, **CMD_FGCOLOR** affects the interaction area of coprocessor engine widgets if they are designed for interactive UI elements, for example, **CMD_BUTTON**, **CMD_DIAL**. **CMD_BGCOLOR** applies the background color of widgets with the color specified. Please see the table below for more details.

Widget	CMD_FGCOLOR	CMD_BGCOLOR	COLOR_RGB
CMD_TEXT	NO	NO	YES
CMD_BUTTON	YES	NO	YES(label)
CMD_GAUGE	NO	YES	YES(needle and mark)
CMD_KEYS	YES	NO	YES(text)
CMD_PROGRESS	NO	YES	YES
CMD_SCROLLBAR	YES(Inner bar)	YES(Outer bar)	NO
CMD_SLIDER	YES(Knob)	YES(Right bar of knob)	YES(Left bar of knob)
CMD_DIAL	YES(Knob)	NO	YES(Marker)
CMD_TOGGLE	YES(Knob)	YES(Bar)	YES(Text)
CMD_NUMBER	NO	NO	YES
CMD_CALIBRATE	YES(Animating dot)	YES(Outer dot)	NO
CMD_SPINNER	NO	NO	YES

Table 22 – Widgets Color Setup Table

5.2.3 Caveat

The behavior of widgets is not defined if the parameter values are out of the valid range.

5.3 Interaction with RAM_DL

If the coprocessor command is to generate respective display list commands, the coprocessor engine will write them to **RAM_DL**. The current writing location in **RAM_DL** is held in the register **REG_CMD_DL**. Whenever the coprocessor engine writes a word to the display list, it increments the register **REG_CMD_DL**. The special command **CMD_DLSTART** sets **REG_CMD_DL** to zero, for the start of a new display list.

All display list commands can also be written to command **FIFO**. The coprocessor engine has the intelligence to differentiate and copy them into the current display list location specified by **REG_CMD_DL**. For example, the following code snippet writes a small display list:

```
cmd(CMD_DLSTART); // start a new display list
cmd(CLEAR_COLOR_RGB(255, 100, 100)); // set clear color
cmd(CLEAR(1, 1, 1)); // clear screen
//.....
cmd(DISPLAY()); // display
```

Of course, this display list could have been written directly to RAM_DL. The advantage of this technique is that you can mix low-level operations and high-level coprocessor engine commands in a single stream:

```
cmd(CMD_DLSTART); // start a new display list
cmd(CLEAR_COLOR_RGB(255, 100, 100)); // set clear color
cmd(CLEAR(1, 1, 1)); // clear screen
cmd_button(20, 20, // x, y
           60, 60, // width, height in pixels
           30, // font 30
           0, // default options
           "OK!"); // Label of button
cmd(DISPLAY()); // Mark the end of display list
```

5.3.1 Synchronization between MCU & Coprocessor Engine

At some points, it is necessary to wait until the coprocessor engine has processed all outstanding commands. When the coprocessor engine completes the last outstanding command in the command buffer, it raises the **INT_CMDEEMPTY** interrupt. Another approach to detecting synchronization is that the MCU can poll **REG_CMD_READ** until it is equal to **REG_CMD_WRITE**.

One situation that requires synchronization is to read the value of **REG_CMD_DL**, when the MCU needs to do direct writes into the display list. In this situation the MCU should wait until the coprocessor engine is idle before reading **REG_CMD_DL**.

5.4 ROM and RAM Fonts

Fonts in **EVE** are treated as a set of bitmap-graphics with metrics block indexed by handles from 0 to 31. The following commands are using fonts:

- **CMD_BUTTON**
- **CMD_KEYS**
- **CMD_TOGGLE**
- **CMD_TEXT**
- **CMD_NUMBER**

For any EVE series ICs **prior to** BT81X Series, only **ASCII** characters are possible to be displayed by the commands above. There is one font metrics block associated with each font, which is called "legacy font metrics block" below. With it, up to 128 characters for each font are ready to be used. In BT81X Series, extended font metrics block is introduced to support a full range of **Unicode** characters with **UTF-8** coding points (note: the CMD_KEYS command does not support Unicode characters).

5.4.1 Legacy Font Metrics Block

For each font, there is one 148-bytes font metrics block associated with it.

The format of the 148-bytes font metrics block is as below:

Address	Size	Value	Description
p + 0	128	Width	width of each font character, in pixels
p + 128	4	Format	bitmap format as defined in BITMAP_EXT_FORMAT , except TEXTVGA , TEXT8X8 , BARGRAPH and PALETTED formats.
P + 132	4	line stride	font bitmap line stride, in bytes

p + 136	4	pixel width	font screen width, in pixels
p + 140	4	pixel height	font screen height, in pixels
p + 144	4	gptr	pointer to glyph data in memory

Table 23 – Legacy Font Metrics Block

For ROM fonts, these blocks are in built-in **ROM**, in an array of length 19. The address of this array is held in ROM location **ROM_FONTROOT**.

For custom fonts, these blocks shall be in **RAM_G**.

5.4.2 Example to find the width of character

To find the width of character 'g' (ASCII 0x67) in ROM font 34:

read 32-bit pointer p from **ROM_FONTROOT**

$widths = p + (148 * (34 - 16))$ (table starts at font 16)

read byte from memory at $widths[0x67]$

5.4.3 Extended Font Metrics Block

The extended font metrics block is a new feature introduced in **BT81X** series, which can handle fonts with a full range of **Unicode** code points. It shall reside at **RAM_G**.

The font block is variable-sized, depending on the number of characters.

Address	Size	Value	Description
p + 0	4	signature	Must be 0x0100AAFF
p + 4	4	size	Total size of the font block, in bytes
p + 8	4	format	Bitmap format, as defined in BITMAP_EXT_FORMAT , except TextVGA, Text8x8, BarGraph and Paletted formats.
P + 12	4	swizzle	Bitmap swizzle value, see BITMAP_SWIZZLE
p + 16	4	layout width	Font bitmap line stride, in bytes
p + 20	4	layout height	Font bitmap height, in pixels
p + 24	4	pixel_width	Font screen width, in pixels
p + 28	4	pixel_height	Font screen height, in pixels
p + 32	4	start_of_graphic_data	Pointer to font graphic data in memory, including flash.
P + 36	4	number_of_characters, N	Total number of characters in font, must be multiple of 128
p + 40	4 x $[N/128]$	gptr	Offsets to glyph data
p + 40 + 4 x $[N/128]$	4 x $[N/128]$	wptr	Offsets to width data
p + 40 + 8 x $[N/128]$	N	width_data	Width data, one byte per character

Table 24 – Extended Font Metrics Block

The table gptr contains offsets to graphic data. There is one offset for every 128 code points. The offsets are all relative to the start_of_graphic_data. The start_of_graphic_data may be an address in **RAM_G** or flash, specified in the same way as **BITMAP_SOURCE**. Similarly, the table wptr contains offsets to width data, but the offsets are relative to p , the start of the font block itself. So, to find the bitmap address and width of a code point cp , please refer to the pseudo-code below:

```
struct xfont {
    uint32_t signature;
    uint32_t size;
    uint32_t format;
    uint32_t swizzle;
    uint32_t layout_width;
    uint32_t layout_height;
    uint32_t pixel_width;
    uint32_t pixel_height;
    uint32_t start_of_graphic_data;
    uint32_t number_of_characters;
    uint32_t gptr[N/128];
    uint32_t wptr[N/128];
    uint8_t width_data[N];
};

uint32_t cp_address(xfont *xf, uint32_t cp)
{
    uint32_t bytes_per_glyph;
    bytes_per_glyph = xf->layout_width * xf->layout_height;

    if (xf->start_of_graphic_data >= 0x800000)
        //if the graphic data is in flash
        return (xf->start_of_graphic_data +
            (xf->gptr[cp / 128] + bytes_per_glyph * (cp % 128)) / 32);
    else
        //if the graphic data is in RAM_G
        return (xf->start_of_graphic_data +
            (xf->gptr[cp / 128] + bytes_per_glyph * (cp % 128)));
}

uint8_t cp_width(xfont *xf, uint32_t cp)
{
    return *(
        (uint8_t*)xf +
        xf->wptr[cp / 128] + (cp % 128));
}
```

Note that the structure above is shown to illustrate the fields of the xfont block clearly. A code implementation of the above structure could use the following definitions. The definitions help to ensure that the structure can be compiled without errors due to the variable sizes of the last three entries in the structure.

```

#define XF_GPTR(xf) ( (unsigned int*)&(((int*)xf)[10]) )
#define XF_WPTR(xf) ( (unsigned int*)&(((char*)xf)[40 + 4 * \
(xf->number_of_characters / 128)]) )
#define XF_WIDTH(xf) ( (unsigned char*)&(((char*)xf)[40 + 8 * \
(xf->number_of_characters / 128)]) )

typedef struct
{
    uint32_t signature; // Must be 0x0100AAFF
    uint32_t size;      // Total size of the font block, in bytes
    uint32_t format;    // Bitmap format
    uint32_t swizzle;   // Bitmap swizzle value
    uint32_t layout_width; // Font bitmap line stride, in bytes
    uint32_t layout_height; // Font bitmap height, in pixels
    uint32_t pixel_width; // Font screen width, in pixels
    uint32_t pixel_height; // Font screen height, in pixels
    uint32_t start_of_graphic_data; // Pointer to font graphic data
    uint32_t number_of_characters; // Total number of characters in font: N (multiple of 128)
} XFONT_EXTENDED;

uint32_t cp_address(const XFONT_EXTENDED * xf, uint32_t cp)
{
    uint32_t bytes_per_glyph;
    bytes_per_glyph = xf->layout_width * xf->layout_height;

    if (xf->start_of_graphic_data >= 0x800000)
        //if the graphic data is in flash
        return (xf->start_of_graphic_data +
            (XF_GPTR(xf)[cp / 128] + bytes_per_glyph * (cp % 128)) / 32);
    else
        //if the graphic data is in RAM_G
        return (xf->start_of_graphic_data +
            (XF_GPTR(xf)[cp / 128] + bytes_per_glyph * (cp % 128)));
}

uint8_t cp_width(const XFONT_EXTENDED * xf, uint32_t cp)
{
    uint32_t offset = XF_WPTR(xf)[cp / 128] + (cp % 128);
    return XF_WIDTH(xf)[offset];
}

```

5.4.4 ROM Fonts (Built-in Fonts)

In total, there are 19 **ROM** fonts numbered from 16 to 34.

By default, **ROM** fonts 16 to 31 are attached to bitmap handles 16 to 31 and users may use these fonts by specifying bitmap handle from 16 to 31.

To use ROM font 32 to 34, the user needs to call [CMD_ROMFONT](#) to assign the bitmap handle with the ROM font number. Refer to [CMD_ROMFONT](#) for more details. To reset ROM fonts to default bitmap handle, use [CMD_RESETFONTS](#).

For **ROM** fonts 16 to 34 (except 17 and 19), each font includes 95 printable **ASCII** characters from 0x20 to 0x7E inclusive. All these characters are indexed by their corresponding **ASCII** value. For ROM fonts 17 and 19, each font includes 127 printable **ASCII** characters from 0x80 to 0xFF, inclusive. All these characters are indexed using value from 0x0 to 0x7F, i.e., code 0 maps to **ASCII** character 0x80 and code 0x7F maps to **ASCII** character 0xFF. Users are required to handle this mapping manually.

The picture below shows the **ROM** font effects:

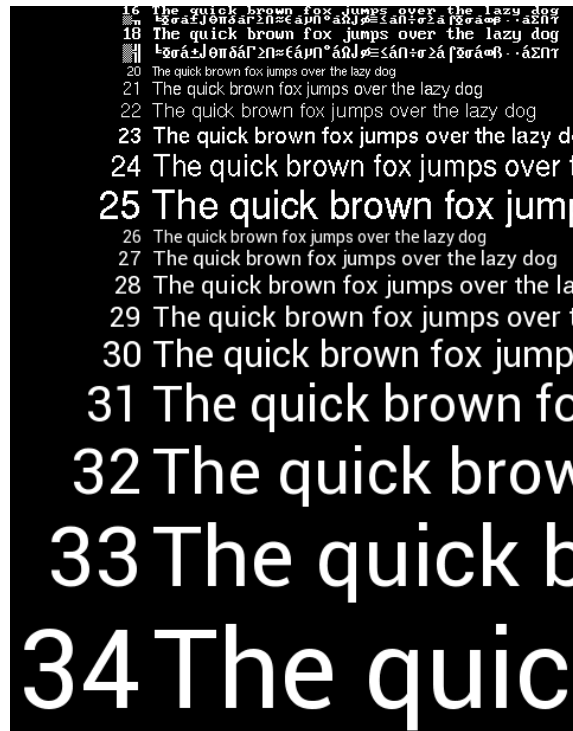


Figure 4 – ROM Font List

5.4.5 Using Custom Font

Users can define custom fonts by following the steps below:

- Select a bitmap handle 0-31
- Load the font bitmap(glyph) into **RAM_G** or **flash memory**
- Create or load a font metrics block in **RAM_G**

Then either:

1. Set up bitmap parameters by using display list command:
 - **BITMAP_SOURCE,**
 - **BITMAP_LAYOUT/BITMAP_LAYOUT_H,**
 - **BITMAP_SIZE/BITMAP_SIZE_H**
 - **BITMAP_EXT_FORMAT** if font is based on ASTC format bitmaps

or:

- using the coprocessor command **CMD_SETBITMAP.**

2. Use command **CMD_SETFONT** to register the new font with the handle 0-31

or:

Use command **CMD_SETFONT2** to register the new font with the handle 0-31.
(Recommended method)

After this setup, the font's handle 0-31 can be used as a font argument of coprocessor commands.

5.5 Animation support

Based on **ASTC** format of bitmap data, BT81X can play back the animation efficiently with minimum **MCU** effort and memory usage. To achieve that, the animation data and object are defined. The utility has been provided to generate these animation assets.

The animation data consists of a sequence of display list fragments. Each fragment must be 64-byte aligned and have a length that is a multiple of 4. The animation object is also 64-byte aligned, and contains:

- a signature
- a frame count
- an array of references to the display list fragments.

```
// A fragment is: a pointer to display list data, and a size
struct fragment {
    uint32_t nbytes; // must be 4-byte aligned
    uint32_t ptr;    // must be 64-byte aligned
};

struct animation_header {
    uint32_t signature; // always ANIM_SIGNATURE (0xAAAA0100)
    uint32_t num_frames;
    struct fragment table[num_frames];
};
```

Note that a fragment can appear multiple times in a table, for example for animation that is slower than the frame rate. Fragments contain regular display list commands. The fragment code is appended to the display list as follows in order that the fragment can:

1. change graphics state,
2. load and use any bitmaps using the current bitmap handle.

Typically, the bitmap data for a fragment also resides in flash and a typical display list to show the fragment is as below:

```
SAVE_CONTEXT
BITMAP_HANDLE(scratch_handle)
<fragment>
RESTORE_CONTEXT
```

Animations can be run on channels. A channel keeps track of the animation's state. There are 32 animation channels. Each channel can handle one animation. The animation commands are:

- **CMD_ANIMFRAME** - render one frame of an animation
- **CMD_ANIMSTART** - start an animation
- **CMD_ANIMSTOP** - stop animation
- **CMD_ANIMXY** - set the (x; y) coordinates of an animation
- **CMD_ANIMDRAW** - draw active animations

All animation functions accept channel number 0-31. Register **REG_ANIM_ACTIVE** to indicate the state of animation channels.

In BT815/6, animation objects and data are only limited to being in flash and require flash in the fast/full mode when it is running. In BT817/8, animation objects and data are also allowed to be in **RAM_G**. Therefore, there are the following commands introduced:

- **CMD_ANIMFRAMERAM** - render one frame of an animation in **RAM_G**
- **CMD_ANIMSTARTRAM** - start an animation in **RAM_G**

In addition, another command **CMD_RUNANIM** is also introduced in **BT817/8** to simplify the playing back animation.

When using **CMD_ANIMSTART** or **CMD_ANIMSTARTRAM**, animations can specify their play mode.

Name	Value	Description
ANIM_ONCE	0	plays the animation once, then cancel it
ANIM_LOOP	1	plays the animation in a loop
ANIM_HOLD	2	plays the animation once, then displays the final frame

Examples 1:

```

/**
play back an animation once in flash
***/

//set up an channel 1
cmd_animstart(1,4096, ANIM_ONCE);
cmd_animxy(400, 240); //The center of animation

//draw each frame in the animation object in a while loop.
while (0 == rd32 (REG_DLSWAP)) {
    cmd_dlstart();
    cmd_animdraw();

    cmd_swap();

    if (0 == rd32(REG_ANIM_ACTIVE))
        break;
}
cmd_animstop(1);

```

Examples 2:

```

/**
play back the animation from frame to frame using cmd_animframe.
FRAME_COUNT is the number of frames to be rendered.
***/
for (int i = 0; i < FRAME_COUNT; i++)
{
    cmd_dlstart();
    cmd(CLEAR(1, 1, 1));
    cmd_animframe(400, 240, 4096, i); //draw the ith frame.
    cmd(DISPLAY());
    cmd_swap();
}

```

5.6 String Formatting

Some coprocessor commands, such as **CMD_TEXT**, **CMD_BUTTON**, **CMD_TOGGLE**, accept a zero-terminated string argument. This string may contain UTF-8 characters, if the selected font contains the appropriate code points.

If the **OPT_FORMAT** option is given in the command, then the string is interpreted as a printf-style format string. The supported formatting is a subset of standard C99. The output string may be up to 256 bytes in length. Arguments to the format string follow the string and its padding. They are always 32-bit and aligned to 32-bit boundaries. So, for example the command:

```

char ss[10];
snprintf(ss, sizeof(ss), "%d", 237);
cmd_text(0, 0, 26, OPT_FORMAT, ss);

```

Should be serialized as:

Offset	Size (In Bytes)	Value	Remarks
0	4	0xFFFFFFFF0C	CMD_TEXT
4	2	0	X coordinate
6	2	0	Y coordinate
8	2	26	Font handle
10	2	OPT_FORMAT	Options
12	1	'%'	Format specifier
13	1	'd'	Conversion specifier
14	1	0	Padding bytes for 32 bits alignment
15	1	0	
16	4	237	

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments from the input stream. Each conversion specification is introduced by the character specifier. In between there may be (in this order) zero or more flags, an optional minimum field width and an optional precision.

5.6.1 The Flag Characters

The character % is followed by zero or more of the following flags:

Flag	Description
0	The value should be zero padded. For d, I, u, o, x, and X conversions, the converted value is padded on the left with zeros rather than blanks. If the 0 and-- flags both appear, the 0 flag is ignored. For other conversions, the behavior is undefined.
-	The converted value is to be left adjusted on the field boundary. (The default is right justification.) The converted value is padded on the right with blanks, rather than on the left with blanks or zeros
' ' (a space)	A blank should be left before a positive number (or empty string) produced by a signed conversion
+	A sign (+ or -) should always be placed before a number produced by a signed conversion. By default, a sign is used only for negative numbers.

Table 25 - Flag Characters

5.6.2 The Field Width

An optional decimal digit string (with nonzero first digit) specifies the minimum field width. If the converted value contains fewer characters than this width, it is padded with spaces on the left (or on the right, if the left-adjustment flag is present). Instead of a decimal digit string, * may be used to indicate that the field width is supplied in the next argument. A negative field width is interpreted as a '-' flag followed by the corresponding positive field width. A missing or smaller field width never causes truncation; if the converted result exceeds the specified width, the field is expanded to accommodate the entire result.

5.6.3 The Precision

An optional precision, specified by a period ('.') followed by an optional decimal digit string. In place of a decimal digit string, '*' may be used to indicate that the precision is supplied by the next argument. If the precision specified as just '.', the precision is taken to be zero. The precision determines the minimum number of digits to appear for d, i, u, o, x, and X conversions; the number of digits to appear after the radix character for a, A, e, E, f, and F conversions; the maximum number of significant digits for g and G conversions; or the maximum number of characters to be printed from a string for s and S conversions.

5.6.4 The Conversion Specifier

A character that specifies the type of conversion to be applied. The conversion specifiers and their meanings are:

Specifiers	Meaning
d,i	The integer argument is converted to signed decimal notation. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros
u, o, x, X	The unsigned integer argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x and X) notation. The letters abcdef are used for x conversions; the letters ABCDEF are used for X conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.
c (lower case)	The integer argument is treated as a Unicode code point, and encoded as UTF-8
s (lower case)	The argument is expected to be an address of RAM_G storing an array of characters. Characters from the array are written up to (but not including) a terminating null byte; if a precision is specified, no more than the number specified are written. If precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.
%	A '%' is written. No argument is converted. The complete conversion specification is '%%'.

Table 26 – String Format Specifier

Examples:

Format string	Output	Assumption
"%3d%% complete", c	51% complete	int c = 51
"base address %06x", a	base address 12a000	int a = 0x12a000
"%+5.3umV", mv	+1947 mV	unsigned int mv = 1947
"Temp %d%.1d degree", t / 10, t % 10	Temp 68.0 degrees	int c = 680
"%s %d times", RAM_G + 4, nTimes	Hello 5 times	" RAM_G +4" is the starting address of the string int nTimes = 5

5.7 Coprocessor Faults

Some commands can cause coprocessor faults. These faults arise because the coprocessor cannot continue. For example:

- An invalid JPEG is supplied to **CMD_LOADIMAGE**
- An invalid data stream is supplied to **CMD_INFLATE/CMD_INFLATE2**
- An attempt is made to write more than 2048 instructions into a display list

In the fault condition, the coprocessor:

1. writes a 128-byte diagnostic string to memory starting at **RAM_ERR_REPORT**.
2. sets **REG_CMD_READ** to 0xff (an illegal value because all command buffer data is 32-bit aligned),
3. raises the **INT_CMDEEMPTY** interrupt
4. stops accepting new commands

The diagnostic string gives details of the problem, and the command that triggered it. The string is up to 128 bytes long, including the terminating 0x00. It always starts with the text "ERROR" For example, after a fault the memory buffer might contain:

```

45 52 52 4f 52 3a 20 69 6c 6c 65 67 61 6c 20 6f |ERROR: illegal o|
70 74 69 6f 6e 20 69 6e 20 63 6d 64 5f 69 6e 66 |ption in cmd_inf|
6c 61 74 65 32 28 29 00 00 00 00 00 00 00 00 |late2().....|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|

```

The possible errors are:

Error string	Remarks
corrupted AVI	The AVI data is corrupted
corrupted DEFLATE data	The input supplied to CMD_INFLATE data is invalid
corrupted JPEG	The JPEG image data is corrupted
corrupted PNG	The PNG image data is corrupted
display list must be empty	CMD_CLEARCACHE was called when the display list was not empty
display list overflow	More than 2048 drawing operations in the display list
fontcache overflow	The font cache was not large enough to hold all required characters
format buffer overflow	The format output buffer used more than 256 bytes
illegal alignment	Flash commands only support certain alignments
illegal flash address	A 32-bit address was used when the flash is in 24-bit mode
illegal font or bitmap handle	Valid handles are 0-31
illegal option	A command's option was not recognized
image type not recognized	The image is not a PNG or JPG
invalid animation	The supplied input is not an animation
invalid animation channel	The supplied animation channel is invalid
invalid base	A number base was given outside the range 2-36
invalid format character	An invalid character appeared in a format
invalid format string	The format conversion specifier was not found
invalid size	A radius, width, or height was negative or zero
out of channels	There are no available animation channels
region too small	The region allocated for the font cache is too small
uninitialized font	Font should be set up with CMD_ROMFONT or CMD_SETFONT
unknown bitmap format	CMD_SETBITMAP was called with an unknown bitmap format
unsupported JPEG	The JPEG image is not supported (e.g. progressive)
unsupported PNG	The PNG image is not supported
unsupported command	The issued command is not supported on this version of EVE

Table 27 – Coprocessor Faults Strings

When the host MCU encounters the fault condition, it can recover as follows:

1. Read **REG_COPRO_PATCH_PTR** into a local variable "*patch_address*".
2. Set **REG_CPURESET** to 1, to hold the coprocessor engine in the reset condition
3. Set **REG_CMD_READ**, **REG_CMD_WRITE**, **REG_CMD_DL** to zero
4. Set **REG_CPURESET** to 0, to restart the coprocessor engine
5. Write the variable "*patch_address*" of step 1 to **REG_COPRO_PATCH_PTR**.
6. To enable coprocessor access flash content, send commands "**CMD_FLASHATTACH**" following "**CMD_FLASHFAST**". It will make sure flash enters full-speed mode.
7. Restore **REG_PCLK** to the original value if the error string is '*display list must be empty*' because **REG_PCLK** is set to zero when that specific error takes place.

5.8 Coprocessor Graphics State

The coprocessor engine maintains a small set of internal state variables used for graphics drawing. These states are initialized to their default values on coprocessor engine reset and by the **CMD_COLDSTART** command. The state values are not modified by **CMD_DLSTART** or **CMD_SWAP**, so an application only needs to configure them once during startup.

State	Default	Commands
background color	<i>dark blue (0x002040)</i>	CMD_BGCOLOR
foreground color	<i>light blue (0x003870)</i>	CMD_FGCOLOR
gradient color	<i>white (0xFFFFFFFF)</i>	CMD_GRADCOLOR
Spinner	<i>None</i>	CMD_SPINNER
object trackers	<i>all disabled</i>	CMD_TRACK
interrupt timer	<i>None</i>	CMD_INTERRUPT
bitmap transform matrix: $\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix}$	$\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix}$	CMD_LOADIDENTITY, CMD_TRANSLATE, CMD_SCALE, CMD_ROTATE, CMD_ROTATEAROUND
Scratch bitmap handle	<i>15</i>	CMD_SETSCRATCH
Font pointers 0-15	<i>Undefined</i>	CMD_SETFONT, CMD_SETFONT2
Font pointer 16-31	<i>ROM fonts 16-31</i>	CMD_SETFONT, CMD_SETFONT2, CMD_ROMFONT
base of number	<i>10</i>	CMD_SETBASE
Media FIFO	<i>Address is zero and length is zero</i>	CMD_MIDEAFIFO,

Table 28 – Coprocessor Engine Graphics State

5.9 Parameter OPTION

The following table defines the parameter “OPTION” mentioned in this chapter.

Name	Value	Description	Commands
OPT_3D	0	3D effect	CMD_BUTTON, CMD_CLOCK, CMD_KEYS, CMD_GAUGE, CMD_SLIDER, CMD_DIAL, CMD_TOGGLE, CMD_PROGRESS, CMD_SCROLLBAR
OPT_RGB565	0	Decode the source image to RGB565 format	CMD_LOADIMAGE
OPT_MONO	1	Decode the source JPEG image to L8 format, i.e., monochrome	CMD_LOADIMAGE
OPT_NODL	2	No display list commands generated	CMD_LOADIMAGE
OPT_FLAT	256	No 3D effect	CMD_BUTTON, CMD_CLOCK, CMD_KEYS, CMD_GAUGE, CMD_SLIDER, CMD_DIAL, CMD_TOGGLE, CMD_PROGRESS, CMD_SCROLLBAR
OPT_SIGNED	256	The number is treated as a 32-bit signed integer	CMD_NUMBER

Name	Value	Description	Commands
OPT_CENTERX	512	Horizontally centered style	CMD_KEYS, CMD_TEXT, CMD_NUMBER
OPT_CENTERY	1024	Vertically centered style	CMD_KEYS, CMD_TEXT, CMD_NUMBER
OPT_CENTER	1536	horizontally and vertically centered style	CMD_KEYS, CMD_TEXT, CMD_NUMBER
OPT_RIGHTX	2048	Right justified style	CMD_KEYS, CMD_TEXT, CMD_NUMBER
OPT_NOBACK	4096	No background drawn	CMD_CLOCK, CMD_GAUGE
OPT_FILL	8192	Breaks the text at spaces into multiple lines, with maximum width set by CMD_FILLWIDTH.	CMD_BUTTON, CMD_TEXT
OPT_FLASH	64	Fetch the data from flash memory	CMD_INFLATE2, CMD_LOADIMAGE, CMD_PLAYVIDEO,
OPT_FORMAT	4096	Flag of string formatting	CMD_TEXT, CMD_BUTTON, CMD_TOGGLE
OPT_NOTICKS	8192	No Ticks	CMD_CLOCK, CMD_GAUGE
OPT_NOHM	16384	No hour and minute hands	CMD_CLOCK
OPT_NOPOINTER	16384	No pointer	CMD_GAUGE
OPT_NOSECS	32768	No second hands	CMD_CLOCK
OPT_NOHANDS	49152	No hands	CMD_CLOCK
OPT_NOTEAR	4	Synchronize video updates to the display blanking interval, avoiding horizontal "tearing" artefacts.	CMD_PLAYVIDEO
OPT_FULLSCREEN	8	Zoom the video so that it fills as much of the screen as possible.	CMD_PLAYVIDEO
OPT_MEDIAFIFO	16	source video/image/compressed(zlib) data from the defined media FIFO	CMD_PLAYVIDEO CMD_VIDEOFRAME CMD_LOADIMAGE CMD_INFLATE2
OPT_OVERLAY	128	Append the video bitmap to an existing display list	CMD_PLAYVIDEO
OPT_SOUND	32	Decode the audio data	CMD_PLAYVIDEO
OPT_DITHER	256	Enable dithering feature in decoding PNG process	CMD_LOADIMAGE

Table 29 – Parameter OPTION Definition

5.10 Resources Utilization

The coprocessor engine does not change the state of the graphics engine. That is, graphics states such as color and line width are not to be changed by the coprocessor engine.

However, the widgets do reserve some hardware resources, which the user must consider:

- Bitmap handle 15 is used by the 3D-effect buttons, keys and gradient, unless it is set to another bitmap handle using **CMD_SETSCRATCH**.
- One graphics context is used by objects, and the effective stack depth for **SAVE_CONTEXT** and **RESTORE_CONTEXT** commands are 3 levels.

5.11 Command list

In BT817/8, coprocessor adds a new feature “command list”, which enables user to construct a series of coprocessor command or display list at **RAM_G**. There are the following new commands to facilitate:

- **CMD_NEWLIST**
- **CMD_CALLLIST**
- **CMD_RETURN**
- **CMD_ENDLIST**

The examples can be found in the sections of the commands above.

5.12 Command Groups

These commands begin and finish the display list:

- **CMD_DLSTART**-- start a new display list
- **CMD_SWAP**-- swap the current display list

Commands to draw graphics objects:

- **CMD_TEXT**-- draw a **UTF-8** text string
- **CMD_BUTTON**-- draw a button with a **UTF-8** label.
- **CMD_CLOCK**-- draw an analog clock
- **CMD_BGCOLOR**-- set the background color
- **CMD_FGCOLOR**-- set the foreground color
- **CMD_GRADCOLOR** – set up the highlight color used in 3D effects for **CMD_BUTTON** and **CMD_KEYS**
- **CMD_GAUGE**-- draw a gauge
- **CMD_GRADIENT**-- draw a smooth color gradient
- **CMD_KEYS**-- draw a row of keys
- **CMD_PROGRESS**-- draw a progress bar
- **CMD_SCROLLBAR**-- draw a scroll bar
- **CMD_SLIDER**-- draw a slider
- **CMD_DIAL**-- draw a rotary dial control
- **CMD_TOGGLE**-- draw a toggle switch with **UTF-8** labels
- **CMD_NUMBER**-- draw a decimal number
- **CMD_SETBASE**-- set the base for number output
- **CMD_FILLWIDTH**-- set the text fill width

Commands to operate on **RAM_G**:

- **CMD_MEMCRC**-- compute a **CRC-32** for **RAM_G**
- **CMD_MEMZERO**-- write zero to **RAM_G**
- **CMD_MEMSET**-- fill **RAM_G** with a byte value
- **CMD_MEMWRITE**-- write bytes into **RAM_G**
- **CMD_MEMCPY**-- copy a block of **RAM_G**
- **CMD_APPEND**-- append more commands to display list

Commands for loading data into **RAM_G**:

- **CMD_INFLATE** - decompress data into **RAM_G**
- **CMD_INFLATE2** - decompress data into **RAM_G** with more options
- **CMD_LOADIMAGE**-- load a JPEG/PNG image into **RAM_G**
- **CMD_MEDIAFIFO**-- set up a streaming media FIFO in **RAM_G**
- **CMD_VIDEOFRAME** – load video frame from **RAM_G** or flash memory.

Commands for setting the bitmap transform matrix:

- **CMD_LOADIDENTITY**-- set the current matrix to identity
- **CMD_TRANSLATE**-- apply a translation to the current matrix
- **CMD_SCALE**-- apply a scale to the current matrix
- **CMD_ROTATE**-- apply a rotation to the current matrix
- **CMD_ROTATEAROUND**-- apply a rotation and scale around the specified pixel
- **CMD_SETMATRIX**-- write the current matrix as a bitmap transform
- **CMD_GETMATRIX**-- retrieves the current matrix coefficients

Commands for flash operation:

- **CMD_FLASHERASE** – erase all of flash
- **CMD_FLASHWRITE** – write data from **RAM_CMD** to blank flash
- **CMD_FLASHUPDATE** – write data to flash, erasing if necessary
- **CMD_FLASHPROGRAM** – write data from **RAM_G** to blank flash
- **CMD_FLASHDETACH** – detach from flash
- **CMD_FLASHATTACH** – attach to flash
- **CMD_FLASHFAST** – enter full-speed mode
- **CMD_FLASHPIDESEL** –SPI bus: deselect device
- **CMD_FLASHSPITX** – SPI bus: write bytes
- **CMD_FLASHSPIRX** – SPI bus: read bytes
- **CMD_CLEARCACHE** – clear the flash cache
- **CMD_FLASHSOURCE** – specify the flash source address for the following coprocessor commands
- **CMD_VIDEOSTARTF** – initialize video frame decoder
- **CMD_APPENDF** – Read data from flash to **RAM_DL**

Commands for video playback:

- **CMD_VIDEOSTART** – Initialize the video frame decoder
- **CMD_VIDEOSTARTF** –Initialize the video frame decoder for video data in flash
- **CMD_VIDEOFRAME** – Load video frame data
- **CMD_PLAYVIDEO**-- play back motion-**JPEG** encoded AVI video

Commands for animation:

- **CMD_ANIMFRAME** – render one frame of an animation
- **CMD_ANIMFRAMERAM** – render one frame in **RAM_G** of an animation
- **CMD_ANIMSTART** – start an animation
- **CMD_ANIMSTOP** – stop animation
- **CMD_ANIMXY** – set the (x,y) coordinates of an animation
- **CMD_ANIMDRAW** – draw active animation

Other commands:

- **CMD_COLDSTART**-- set coprocessor engine state to default values
- **CMD_INTERRUPT**-- trigger interrupt INT_CMDFLAG
- **CMD_REGREAD**-- read a register value
- **CMD_CALIBRATE**-- execute the touch screen calibration routine
- **CMD_ROMFONT** – load a ROM font into bitmap handle
- **CMD_SETROTATE**-- Rotate the screen and set up transform matrix accordingly
- **CMD_SETBITMAP** – Set up display list commands for specified bitmap
- **CMD_SPINNER**-- start an animated spinner
- **CMD_STOP**-- stop any spinner, screensaver or sketch
- **CMD_SCREENSAVER**-- start an animated screensaver
- **CMD_SKETCH**-- start a continuous sketch update
- **CMD_SNAPSHOT**-- take a snapshot of the current screen
- **CMD_SNAPSHOT2**-- take a snapshot of part of the current screen with more format option
- **CMD_LOGO**-- play device logo animation

5.13 CMD_APILEVEL

This command sets the API level used by the coprocessor.

C prototype

```
void cmd_apilevel( uint32_t level );
```

Parameter

level

API level to use. Level 1 is BT815/6 compatible and is the default. Level 2 is BT817/8.

Command layout

+0	CMD_APILEVEL (0xFFFF FF63)
+4	level

Description

To use the BT817/8 specific commands or other improvement, level 2 must be sent.

Example

```
//At startup, the API level is 1. To set it to 2:  
cmd_apilevel(2);
```

Note: BT817/8 specific command

5.14 CMD_DLSTART

This command starts a new display list. When the coprocessor engine executes the command, it waits until the current display list is ready for writing and then sets **REG_CMD_DL** to zero.

C prototype

```
void cmd_dlstart( );
```

Command layout

+0	CMD_DLSTART (0xFFFF FF00)
----	----------------------------------

5.15 CMD_INTERRUPT

This command is used to trigger the CMDFLAG Interrupt. When the coprocessor engine executes this command, it generates an interrupt that sets the **CMDFLAG** bit in **REG_INT_FLAGS**, unless the corresponding bit in **REG_INT_MASK** is zero.

C prototype

```
void cmd_interrupt( uint32_t ms );
```

Parameters

ms

The delay before the interrupt triggers, in milliseconds. The interrupt is guaranteed not to fire before this delay. If ms is zero, the interrupt fires immediately.

Notes: The maximum allowable value for **ms** across all different system clock frequencies is 10000, which corresponds to 10 seconds.

Command layout

+0	CMD_INTERRUPT(0xFFFF FF02)
+4	ms

Examples

```
//To trigger an interrupt after a JPEG has finished loading:
cmd_loadimage();
//...
cmd_interrupt(0); // previous load image complete, trigger interrupt

//To trigger an interrupt in 0.5 seconds:
cmd_interrupt(500);
//...
```

5.16 CMD_COLDSTART

This command sets the coprocessor engine to default reset states.

C prototype

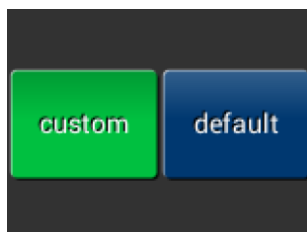
```
void cmd_coldstart( );
```

Command layout

+0	CMD_COLDSTART(0xFFFF FF32)
----	-----------------------------------

Examples

Change to a custom color scheme, and then restore the default colors:



```
cmd_fgcolor(0x00c040);
cmd_gradcolor(0x000000);
cmd_button( 2, 32, 76, 56, 26,0, "custom" );
cmd_coldstart();
cmd_button( 82, 32, 76, 56, 26,0, "default" );
```

5.17 CMD_SWAP

This command is used to swap the current display list. When executed by the coprocessor engine, it requests a display list swap immediately after the current display list has been scanned out. Internally, the coprocessor engine implements this command by writing the value **0x02** to **REG_DLSWAP**.

This coprocessor engine command does not generate any display list commands in the display list memory RAM_DL. It is intended to be used in conjunction with **CMD_DLSTART**.

C prototype

```
void cmd_swap( );
```

Command layout

+0	CMD_SWAP(0xFFFF FF01)
----	------------------------------

5.18 CMD_APPEND

This command appends additional commands residing in **RAM_G** to the current display list at the memory address specified by the offset in **REG_CMD_DL**.

C prototype

```
void cmd_append( uint32_t ptr,  
                uint32_t num );
```

Parameters

ptr
Starting address of source commands in RAM_G

num
Number of bytes to copy. This must be a multiple of 4.

Command layout

+0	CMD_APPEND(0xFFFF FF1E)
+4	ptr
+8	num

Description

After the appending operation is complete, the coprocessor engine will increment **REG_CMD_DL** by num to make sure the display list remains in sequence.

Examples

```
cmd_dlstart();  
cmd_append(0, 40); // copy 10 commands from main memory address 0  
cmd(DISPLAY); // finish the display list  
cmd_swap();
```

5.19 CMD_REGREAD

This command is used to read a register value.

C prototype

```
void cmd_regread( uint32_t ptr,  
                 uint32_t result );
```

Parameters

ptr
Address of the register to be read

result

The register value read from the ptr address. **OUTPUT** parameter.

Write a dummy 32-bit value 0x00000000 for this parameter; the coprocessor will replace this value with the result after the command has been executed.

After execution, the host should read the address of this parameter in **RAM_CMD** to retrieve the result value.

Command layout

+0	CMD_REGREAD(0xFFFF FF19)
+4	ptr
+8	result

Examples

```

//To capture the exact time when a command completes:
uint16_t x = rd16(REG_CMD_WRITE);
cmd_regread(REG_CLOCK, 0);
//...
printf("%08x\n", rd32(RAM_CMD + (x + 8) % 4096));
```

5.20 CMD_MEMWRITE

This command writes the specified bytes into the memory. This command can be used to set register values or to update memory contents at specific times.

C prototype

```
void cmd_memwrite( uint32_t ptr,
                  uint32_t num );
```

Parameters

ptr
The memory address to be written

num
Number of bytes to be written.

Command layout

+0	CMD_MEMWRITE(0xFFFF FF1A)	
+4	ptr	
+8	num	
+12	byte ₁	
	...	
	byte _n	

Description

The data bytes should immediately follow in the command buffer. If the number of bytes is not a multiple of 4, then 1, 2 or 3 padding bytes should be appended to ensure 4-byte alignment for the next command; these padding bytes can have any value. Completion of this command can be detected when the value of **REG_CMD_READ** equals **REG_CMD_WRITE**.

Note: Improper use of this command may result in memory corruption.

Examples

```
//To change the backlight brightness to 0x64 (half intensity) for a particular screen shot:
//...
cmd_swap(); // finish the display list
cmd_dlstart(); // wait until after the swap
cmd_memwrite(REG_PWM_DUTY, 4); // write to the PWM_DUTY register
cmd(100);
```

5.21 CMD_INFLATE

This command decompresses the specified compressed data into **RAM_G**. The data must be compressed using the **DEFLATE** algorithm, such as with the **ZLIB** library. This command is especially useful for loading graphics data.

C prototype

```
void cmd_inflate( uint32_t ptr );
```

Parameters

ptr
 Destination address in **RAM_G**. The data byte should immediately follow in the command buffer.

Command layout

+0	CMD_INFLATE(0xFFFF FF22)
+4	ptr
+8	byte ₁
	...
	byte _n

Description

If the number of bytes is not a multiple of 4, then 1, 2 or 3 padding bytes should be appended to ensure 4-byte alignment for the next command. These padding bytes can contain any value.

Examples

To load graphics data to main memory address 0x8000:

```
cmd_inflate(0x8000);
// zlib-compressed data follows
```

5.22 CMD_INFLATE2

This command decompresses the specified compressed data into **RAM_G**. The data may be supplied via the command buffer, the media **FIFO**, or from flash memory. It must be compressed using the **DEFLATE** algorithm, such as with the **ZLIB** library. This command is especially useful for loading graphics data.

C prototype

```
void cmd_inflate2( uint32_t ptr,
                  uint32_t options );
```

Parameters

ptr
 destination address to put the decompressed data.

options
 If the option **OPT_MEDIAFIFO** is specified, the compressed data is sourced from the media **FIFO**. If the option **OPT_FLASH** is specified, the data is read from flash memory. When using flash as the source, **CMD_FLASHSOURCE** must be called before this command to specify the source address. See [CMD_FLASHSOURCE](#). Otherwise, a **zero** value should be given, and the compressed data must immediately follow.

Command layout

+0	CMD_INFLATE2(0xFFFF FF50)	
+4	ptr	
+8	options	
+12	byte ₁	
	...	
	byte _n	

Description

If the number of bytes is not a multiple of 4, then 1, 2 or 3 padding bytes should be appended to ensure 4-byte alignment for the next command. These padding bytes can contain any value.

5.23 CMD_LOADIMAGE

This command loads a JPEG or PNG image, decompressing the provided **JPEG** or **PNG** image data into a specific **EVE** bitmap format stored in **RAM_G**. The image data must conform to the following formats:

- Regular baseline **JPEG (JFIF)** with chroma subsampling patterns 4:2:0, 4:2:2, 4:4:4
- or
- **PNG** with bit-depth 8 only and no interlace

C prototype

```
void cmd_loadimage( uint32_t ptr,
                   uint32_t options );
```

Parameters

ptr
 Destination address, within **RAM_G**

options

For **JPEG** images, the bitmap is loaded in either **RGB565** or **L8** format bitmap. If **OPT_MONO** is specified, **L8** format is used; otherwise, **RGB565** is used. Note that **OPT_RGB565** and **OPT_MONO** options apply only to **JPEG** images.

For **PNG** images, the **PNG** standard defines multiple color formats. Each format is loaded into a bitmap in the following manner:

Color type	Format	Loaded by CMD_LOADIMAGE
0	Grayscale	L8
2	Truecolor	RGB565
3	Indexed	PALETTED565 or PALETTED4444
4	Grayscale and alpha	not supported
6	Truecolor and alpha	ARGB4

OPT_FULLSCREEN: Scales the bitmap to fill as much of the screen as possible.

OPT_MEDIAFIFO: Uses the media FIFO as the source for the image data.

OPT_FLASH: Uses flash memory as the source for the image data.

If neither **OPT_MEDIAFIFO** nor **OPT_FLASH** is specified, the image data must immediately follow in the command **FIFO**. When using flash as the source, **CMD_FLASHSOURCE** must be called before this command to specify the source address. See [CMD_FLASHSOURCE](#).

To simplify rendering the loaded image, a set of display list commands is generated and appended to the current display list, unless **OPT_NODL** is specified.

Command layout

+0	CMD_LOADIMAGE(0xFFFF FF24)	
+4	ptr	
+8	options	
+12	byte ₁	
	...	
	byte _n	

Description

If **OPT_MEDIAFIFO** or **OPT_FLASH** is **NOT** set, the data bytes must immediately follow in the command **FIFO**. If the number of bytes is not a multiple of 4, then 1, 2 or 3 padding bytes should be appended to ensure 4-byte alignment for the next command; these padding bytes can contain any value. The host application must parse the **JPEG/PNG** header to determine the properties of the **JPEG/PNG** image. Behavior is unpredictable in cases of non-baseline JPEG images or if the decompressed output exceeds the size of **RAM_G**.

Note: When loading a **PNG** image, the top 42 KB of **RAM_G** will be used as a temporary buffer for the decoding process and will be overwritten.

Examples

To load a JPEG image at address 0 then draw the bitmap at (10, 20) and (100, 20):

```
cmd_loadimage(0, 0);
... // JPEG file data follows
cmd(BEGIN(BITMAPS));
cmd(VERTEX2II(10, 20, 0, 0)); // draw bitmap at (10,20)
cmd(VERTEX2II(100, 20, 0, 0)); // draw bitmap at (100,20)
```

5.24 CMD_MEDIAFIFO

This command sets up a streaming media **FIFO**. It allocates the specified area of **RAM_G** and configures it as a streaming media **FIFO**, which is used by:

- **MJPEG** video playback: **CMD_PLAYVIDEO/CMD_VIDEOFRAME**
- **JPEG/PNG** image decoding: **CMD_LOADIMAGE**
- Compressed data by zlib: **CMD_INFLATE2**

if the option **OPT_MEDIAFIFO** is selected.

C prototype

```
void cmd_mediafifo( uint32_t ptr,
                   uint32_t size );
```

Parameters

ptr
starting address of media **FIFO**

size
number of bytes of media **FIFO**

Command layout

+0	CMD_MEDIAFIFO (0xFFFF FF39)
+4	ptr
+8	size

Examples

To set up a 64-Kbyte FIFO at the top of **RAM_G** for JPEG streaming and to report the initial values of the read and write pointers:

```
cmd_mediafifo(0x100000 - 65536, 65536); //0x100000 is the top of RAM_G
printf("R=0x%08X W=0x%08X\n", rd32(REG_MEDIAFIFO_READ), rd32(REG_MEDIAFIFO_WRITE));
```

It prints:

R=0x000F0000 W=0x000F0000

Which is the value of 1M(RAM_G size) minus 64K(media FIFO size).

5.25 CMD_PLAYVIDEO

This command plays back **MJPEG**-encoded AVI video.

Playback starts immediately, and the command completes when playback ends. Playback can be paused or stopped by writing to **REG_PLAY_CONTROL**. The value written to this register controls playback as follows:

- -1(0xFF) exit playback
- 0 pause playback
- 1 play normally

During command execution, an **RGB565** bitmap is created at the starting address of **RAM_G**, with a size of $2 \times W \times H$ bytes, where **W** and **H** are the width and height of the video. If **OPT_SOUND** is specified, then a 32 Kbyte audio buffer follows the bitmap. This means that the corresponding area of **RAM_G** will be overwritten by **CMD_PLAYVIDEO**.

C prototype

```
void cmd_playvideo( uint32_t opts );
```

Parameters

opts: Options for playing video

OPT_FULLSCREEN: Zooms the video to fill as much of the screen as possible.

OPT_MEDIAFIFO: Sources AVI video data from the media FIFO in **RAM_G** instead of the command buffer.

OPT_FLASH: Sources video data from flash. When using flash **CMD_FLASHSOURCE** must be called before this command to specify the source address. See **CMD_FLASHSOURCE**.

OPT_NOTEAR: Synchronizes video updates to the display blanking interval to avoid horizontal tearing artifacts.

OPT_SOUND: Decodes the audio data, if present, following the video data.

OPT_OVERLAY: Appends the video bitmap to an existing display list rather than starting a new display list.

OPT_NODL: Does not modify the current display list. A display list rendering the video bitmap must already exist.

data

The video data must immediately follow in the command buffer unless **opts** includes **OPT_MEDIAFIFO** or **OPT_FLASH**.

Command layout

+0	CMD_PLAYVIDEO (0xFFFF FF3A)	
+4	opts	
+8	byte ₁	
	...	
	byte _n	

Data following parameter "opts" shall be padded to 4 bytes aligned with zero.

Note:

- For the audio data encoded into AVI video, three formats are supported:

4 Bit IMA ADPCM, 8 Bit signed PCM, 8 Bit u-Law

In addition, 16 Bit PCM audio is partially supported by discarding the least significant 8 bits of each audio sample.

- When video data is supplied from the command **FIFO**, the command completes only after the video playback finishes. However, if the video data comes from the media **FIFO** or flash, the command completes immediately, but the coprocessor remains busy. During this time, the coprocessor will not process any further commands until the video playback is finished. To check if the video playback has ended, a **CMD_NOP** command can be sent to the coprocessor, and its completion can be awaited.

Examples

To play back an AVI video, full screen:

```
cmd_playvideo(OPT_FULLSCREEN | OPT_NOTEAR);  
//... append AVI data ...
```

5.26 CMD_VIDEOSTART

This command is used to initialize video frame decoder. The video data is supplied using the media FIFO. This command processes the video header information from the media FIFO and completes when the header has been fully consumed.

C prototype

```
void cmd_videostart( );
```

Command layout

+0	CMD_VIDEOSTART (0xFFFF FF40)
----	------------------------------

Examples

To load frames of video at address 4:

```
cmd_videostart();  
cmd_videoframe(4, 0);
```

5.27 CMD_VIDEOFRAME

This command is used to load the next frame of a video. The video data is supplied using the media **FIFO** or flash memory. This command extracts the next video frame and completes when it has been fully consumed.

C prototype

```
void cmd_videoframe( uint32_t dst,  
                    uint32_t ptr );
```

Parameters**dst**

Memory location to load the frame data, this will be in **RAM_G**.

ptr

Completion pointer. The command writes a 32-bit word to this location. It is set to 1 if at least one more frame is available in the video. A value of 0 indicates that this is the last frame. The value of ptr shall be within **RAM_G**.

Command layout

+0	CMD_VIDEOFRAME (0xFFFF FF41)
+4	dst
+8	ptr

Examples

To load frames of video at address 4:

```
cmd_videostart();
do {
  cmd_videoframe(4, 0);
  //... display frame ...
} while (rd32(0) != 0);
```

5.28 CMD_MEMCRC

This command computes a CRC-32 for a block of **RAM_G** memory.

C prototype

```
void cmd_memcrc( uint32_t ptr,
                 uint32_t num,
                 uint32_t result );
```

Parameters

ptr

Starting address of the memory block

num

Number of bytes in the source memory block

result

Output parameter; written with the CRC-32 after command execution.

Command layout

+0	CMD_MEMCRC(0xFFFF FF18)
+4	ptr
+8	num
+12	result

Examples

To compute the CRC-32 of the first 1 KB of memory, first record the value of **REG_CMD_WRITE**, execute the command, wait for completion, and then read the 32-bit value at result:

```
uint16_t x = rd16(REG_CMD_WRITE);
cmd_memcrc(0, 1024, 0);

//wait till the command is complete
printf("CRC result is %08x\n", rd32(RAM_CMD + (x + 12) % 4096));
```

5.29 CMD_MEMZERO

This command is used to write zero to a block of memory.

C prototype

```
void cmd_memzero( uint32_t ptr,
                  uint32_t num );
```

Parameters

ptr
Starting address of the memory block

num
Number of bytes in memory block

Command layout

+0	CMD_MEMZERO(0xFFFF FF1C)
+4	ptr
+8	num

Examples

```
//To erase the first 1K of main memory:
cmd_memzero(0, 1024);
```

5.30 CMD_MEMSET

This command is used to fill a block of memory with a byte value.

C prototype

```
void cmd_memset( uint32_t ptr,
                 uint32_t value,
                 uint32_t num );
```

Parameters

ptr
Starting address of the memory block

value
Value to be written to memory

num
Number of bytes in memory block

Command layout

+0	CMD_MEMSET(0xFFFF FF1B)
+4	ptr
+8	value
+12	num

Examples

```
//To write 0xff the first 1K of main memory:
cmd_memset(0, 0xff, 1024);
```

5.31 CMD_MEMCPY

This command is used to copy a block of memory.

C prototype

```
void cmd_memcpy( uint32_t dest,
                uint32_t src,
                uint32_t num );
```

Parameters

dest
address of the destination memory block

src
address of the source memory block

num
number of bytes to copy

Command layout

+0	CMD_MEMCPY(0xFFFF FF1D)
+4	dst
+8	src
+12	num

Examples

```
//To copy 1K byte of memory from 0 to 0x8000:
cmd_memcpy(0x8000, 0, 1024);
```

5.32 CMD_BUTTON

This command is used to draw a button with a UTF-8 label.

C prototype

```
void cmd_button( int16_t x,
                int16_t y,
                uint16_t w,
                uint16_t h,
                uint16_t font,
                uint16_t options,
                const char* s );
```

Parameters

x
X-coordinate of button top-left, in pixels

y
Y-coordinate of button top-left, in pixels

w
width of button, in pixels

h
height of button, in pixels

font
bitmap handle to specify the font used in the button label. [See ROM and RAM Fonts.](#)

options
By default, the button is drawn with a 3D effect and a value of zero. **OPT_FLAT** removes the 3D effect. The value of **OPT_FLAT** is 256.

s
Button label. It must be a single string terminated with null character ('\0' in C language) and UTF-8 encoded. If **OPT_FILL** is not specified, the string may include newline characters ('\n', ASCII 0x0A) in C to indicate line breaks. **See 5.6 String Formatting.**

Command layout

+0	CMD_BUTTON(0xFFFF FF0D)
+4	x
+6	y
+8	w
+10	h
+12	font
+14	options
+16	s
	...
	0 (null character to terminate string)

Description

Refer to [Coprocessor engine widgets physical dimensions](#) for more information.

Examples

A 140x100 pixel button with large text:



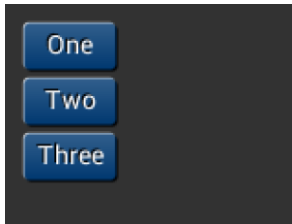
```
cmd_button(10, 10, 140, 100, 31, 0, "Press!");
```

Without the 3D look:



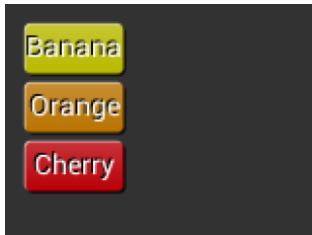
```
cmd_button(10, 10, 140, 100, 31, OPT_FLAT, "Press!");
```

Several smaller buttons:



```
cmd_button(10, 10, 50, 25, 26, 0, "One");
cmd_button(10, 40, 50, 25, 26, 0, "Two");
cmd_button(10, 70, 50, 25, 26, 0, "Three");
```

Changing button color



```
cmd_fgcolor(0xb9b900),
cmd_button(10, 10, 50, 25, 26, 0, "Banana");
cmd_fgcolor(0xb97300),
cmd_button(10, 40, 50, 25, 26, 0, "Orange");
cmd_fgcolor(0xb90007),
cmd_button(10, 70, 50, 25, 26, 0, "Cherry");
```

5.33 CMD_CLOCK

This command is used to draw an analog clock.

C prototype

```
void cmd_clock( int16_t x,
               int16_t y,
               uint16_t r,
               uint16_t options,
               uint16_t h,
               uint16_t m,
               uint16_t s,
               uint16_t ms );
```

Parameters

x
x-coordinate of clock center, in pixels

y
y-coordinate of clock center, in pixels

r
the radius of clock, in pixels

options

By default, the clock dial is drawn with a 3D effect (**OPT_3D**). Setting **OPT_FLAT** removes the 3D effect.

OPT_NOBACK disables drawing the background.

OPT_NOTICKS disables drawing the twelve-hour ticks.

OPT_NOSECS disables drawing the second hand.

OPT_NOHANDS disables drawing all hands.

OPT_NOHM disables drawing the hour and minutes hands.

h
hours

m
minutes

s
seconds

ms
milliseconds

Command layout

+0	CMD_CLOCK(0xFFFF FF14)
+4	x
+6	y
+8	r
+10	options
+12	h
+14	m
+16	s
+18	ms

Description

The details of the physical dimensions are:

- The 12 tick marks are placed on a circle of radius $r*(200/256)$.
- Each tick is a point of radius $r*(10/256)$
- The seconds hand has length $r*(200/256)$ and width $r*(3/256)$
- The minutes hand has length $r*(150/256)$ and width $r*(9/256)$
- The hours hand has length $r*(100/256)$ and width $r*(12/256)$

Refer to [Coprocessor engine widgets physical dimensions](#) for more information.

Examples

A clock with radius 50 pixels, showing a time of 8.15:



```
cmd_clock(80, 60, 50, 0, 8, 15, 0, 0);
```

Setting the background color



```
cmd_bgcolor(0x401010);
```

Without the 3D look:



```
cmd_clock(80, 60, 50, OPT_FLAT, 8, 15, 0, 0);
```

The time fields can have large values. In this example, the hours are 7 x 3600s, the minutes are 38 x 60s, and the seconds are 59. This creates a clock face showing the time as 7:38.59.



```
cmd_clock(80, 60, 50, 0,0, 0, (7 * 3600) + (38 * 60) + 59, 0);
```

No seconds hand:



```
cmd_clock(80, 60, 50, OPT_NOSECS, 8, 15, 0, 0);
```

No Background:



```
cmd_clock(80, 60, 50, OPT_NOBACK, 8, 15, 0, 0);
```

No ticks:



```
cmd_clock(80, 60, 50, OPT_NOTICKS, 8, 15, 0, 0);
```

No hands:



```
cmd_clock(80, 60, 50, OPT_NOHANDS, 8, 15, 0, 0);
```

5.34 CMD_FGCOLOR

This command is used to set the foreground color.

C prototype

```
void cmd_fgcolor( uint32_t c );
```

Parameters

c

New foreground color specified as a 24-bit RGB value.

Red occupies the most significant 8 bits, and blue the least significant 8 bits. For example, 0xff0000 represents bright red.

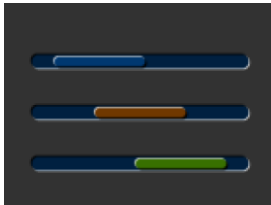
The foreground color applies to user-interactive elements such as handles and buttons.

Command layout

+0	CMD_FGCOLOR(0xFFFF FF0A)
+4	c

Examples

The top scrollbar uses the default foreground color, while the others use a modified color:



```
cmd scrollbar(20, 30, 120, 8, 0, 10, 40, 100);
cmd_fgcolor(0x703800);
cmd_scrollbar(20, 60, 120, 8, 0, 30, 40, 100);
cmd_fgcolor(0x387000);
cmd_scrollbar(20, 90, 120, 8, 0, 50, 40, 100);
```

5.35 CMD_BGCOLOR

This command is used to set the background color.

C prototype

```
void cmd_bgcolor( uint32_t c );
```

Parameters

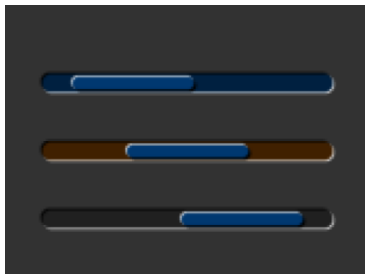
c
 New background color, specified as a 24-bit RGB value. Red occupies the most significant 8 bits, and blue the least significant 8 bits. For example, 0xff0000 represents bright red. Background color applies to the elements that the user cannot move such as the area behind gauges, sliders and similar components.

Command layout

+0	CMD_BGCOLOR(0xFFFF FF09)
+4	c

Examples

The top scrollbar uses the default background color, while the others use a modified color:



```
cmd scrollbar(20, 30, 120, 8, 0, 10, 40, 100);
cmd_bgcolor(0x402000);
cmd_scrollbar(20, 60, 120, 8, 0, 30, 40, 100);
cmd_bgcolor(0x202020);
cmd_scrollbar(20, 90, 120, 8, 0, 50, 40, 100);
```

5.36 CMD_GRADCOLOR

This command is used to set the 3D button highlight color.

C prototype

```
void cmd_gradcolor( uint32_t c );
```

Parameters

c

New highlight gradient color, specified as a 24-bit RGB value.
 The default is white 0xFFFFFF.

Red occupies the most significant 8 bits, and blue the least significant 8 bits. For example, 0xff0000 represents bright red.

Gradient highlights are supported only for Button and Keys widgets.

Command layout

+0	CMD_GRADCOLOR(0xFFFF FF34)
+4	c

Examples

Changing the gradient color: white, red, green and blue:



```
cmd_fgcolor(0x101010);
cmd_button( 2, 2, 76, 56, 31, 0, "W");
cmd_gradcolor(0xff0000);
cmd_button( 82, 2, 76, 56, 31, 0, "R");
cmd_gradcolor(0x00ff00);
cmd_button( 2, 62, 76, 56, 31, 0, "G");
cmd_gradcolor(0x0000ff);
cmd_button( 82, 62, 76, 56, 31, 0, "B");
```

The gradient color is also used for keys:



```
cmd_fgcolor(0x101010);
cmd_keys(10, 10, 140, 30, 26, 0, "abcde");
cmd_gradcolor(0xff0000);
cmd_keys(10, 50, 140, 30, 26, 0, "fghij");
```

5.37 CMD_GAUGE

This command is used to draw a Gauge.

C prototype

```
void cmd_gauge( int16_t x,
                int16_t y,
                uint16_t r,
                uint16_t options,
                uint16_t major,
                uint16_t minor,
                uint16_t val,
                uint16_t range );
```

Parameters

x

X-coordinate of gauge center, in pixels

y

Y-coordinate of gauge center, in pixels

r
 Radius of the gauge, in pixels

options

By default, the gauge dial is drawn with a 3D effect, and the value of options is zero. Setting OPT_FLAT removes the 3D effect.
 OPT_NOBACK disables drawing the background
 OPT_NOTICKS disables drawing the tick marks
 OPT_NOPOINTER disables drawing the pointer.

major

Number of major subdivisions on the dial, 1-10

minor

Number of minor subdivisions on the dial, 1-10

val

Gauge indicated value, between 0 and range, inclusive

range

Maximum value

Command layout

+0	CMD_GAUGE(0xFFFF FF13)
+4	x
+6	y
+8	r
+10	options
+12	major
+14	minor
+16	value
+18	range

Description

The details of physical dimension are:

- The tick marks are placed on a 270-degree arc, clockwise, starting at the south-west position.
- Minor ticks are lines of width $r*(2/256)$, major ticks are $r*(6/256)$.
- Ticks are drawn at a distance of $r*(190/256)$ to $r*(200/256)$ from the center.
- The pointer is drawn with lines of width $r*(4/256)$ extending to a point $r*(190/256)$ from the center.
- The other ends of the lines are positioned 90 degrees perpendicular to the pointer direction, at a distance of $r*(3/256)$ from the center.

Refer to [Coprocesor engine widgets physical dimensions](#) for more information.

Examples

A gauge with a radius of 50 pixels, five divisions of four ticks each, showing a value of 30%:



```
cmd_gauge(80, 60, 50, 0, 5, 4, 30, 100);
```

Without the 3D look:



```
cmd_gauge(80, 60, 50, OPT_FLAT, 5, 4, 30, 100);
```

Ten major divisions with two minor divisions each:



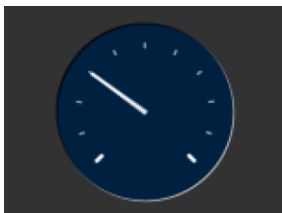
```
cmd_gauge(80, 60, 50, 0, 10, 2, 30, 100);
```

Setting the number of minor divisions to 1 cause them to disappear:



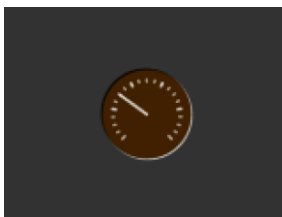
```
cmd_gauge(80, 60, 50, 0, 10, 1, 30, 100);
```

Setting the number of major divisions to 1 result in only the minor divisions being drawn:



```
cmd_gauge(80, 60, 50, 0, 1, 10, 30, 100);
```

A smaller gauge with a brown background:



```
cmd_bgcolor(0x402000);  
cmd_gauge(80, 60, 25, 0, 5, 4, 30, 100);
```

Scale 0-1000, indicating 1000:



```
cmd_gauge(80, 60, 50, 0, 5, 2, 1000, 1000);
```

Scaled 0-65535, indicating 49152:



```
cmd_gauge(80, 60, 50, 0, 4, 4, 49152, 65535);
```

No background:



```
cmd_gauge(80, 60, 50, OPT_NOBACK, 4, 4, 49152, 65535);
```

No tick marks:



```
cmd_gauge(80, 60, 50, OPT_NOTICKS, 4, 4, 49152, 65535);
```

No pointer:



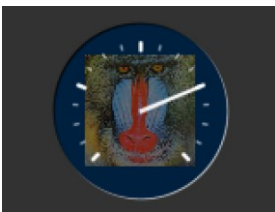
```
cmd_gauge(80, 60, 50, OPT_NOPOINTER, 4, 4, 49152, 65535);
```

Drawing the gauge in two passes, with bright red for the pointer:



```
GAUGE_0 = OPT_NOPOINTER;  
GAUGE_1 = OPT_NOBACK | OPT_NOTICKS;  
cmd_gauge(80, 60, 50, GAUGE_0, 4, 4, 49152, 65535);  
cmd(COLOR_RGB(255, 0, 0));  
cmd_gauge(80, 60, 50, GAUGE_1, 4, 4, 49152, 65535);
```

Add a custom graphic to the gauge by drawing its background, then a bitmap, and finally its foreground:



```
GAUGE_0 = OPT_NOPOINTER | OPT_NOTICKS;  
GAUGE_1 = OPT_NOBACK;  
cmd_gauge(80, 60, 50, GAUGE_0, 4, 4, 49152, 65535);  
cmd(COLOR_RGB(130, 130, 130));  
cmd(BEGIN(BITMAPS));  
cmd(VERTEX2II(80 - 32, 60 - 32, 0, 0));  
cmd(COLOR_RGB(255, 255, 255));  
cmd_gauge(80, 60, 50, GAUGE_1, 4, 4, 49152, 65535);
```

5.38 CMD_GRADIENT

This command is used to create a smooth transition between color gradients.

C prototype

```
void cmd_gradient( int16_t x0,
                  int16_t y0,
                  uint32_t rgb0,
                  int16_t x1,
                  int16_t y1,
                  uint32_t rgb1 );
```

Parameters

x0

x-coordinate of point 0, in pixels

y0

y-coordinate of point 0, in pixels

rgb0

Color of point 0, represented as a 24-bit RGB value. Red occupies the most significant 8 bits, and blue the least significant 8 bits. For example, 0xff0000 represents bright red.

x1

x-coordinate of point 1, in pixels

y1

y-coordinate of point 1, in pixels

rgb1

Color of point 1, same definition as **rgb0**.

Command layout

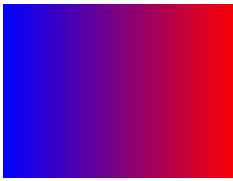
+0	CMD_GRAGIENT(0xFFFF FF0B)	
+4	x0	
+6	y0	
+8	rgb0	
+12	x1	
+14	y1	
+16	rgb1	

Description

All color step values are determined from a smooth curve interpolated between the RGB0 and RGB1 parameters. The smooth curve is calculated separately for each color channel using the formula $R0 + t * (R1 - R0)$, where t varies from 0 to 1. The gradient should be applied together with the Scissor function to produce the intended gradient effect.

Examples

A horizontal gradient transitioning from blue to red



```
cmd_gradient(0, 0, 0x0000ff, 160, 0, 0xff0000);
```

A vertical gradient



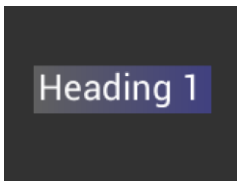
```
cmd_gradient(0, 0, 0x808080, 0, 120, 0x80ff40);
```

The same colors in a diagonal gradient



```
cmd_gradient(0, 0, 0x808080, 160, 120, 0x80ff40);
```

Using a scissor rectangle to render a gradient stripe as a background for a title:



```
cmd(SCISSOR_XY(20, 40));  
cmd(SCISSOR_SIZE(120, 32));  
cmd_gradient(20, 0, 0x606060, 140, 0, 0x404080);  
cmd_text(23, 40, 29, 0, "Heading 1");
```

5.39 CMD_GRADIENTA

This command is used to draw smooth color gradient with transparency. Each of the two points has RGB color values and an alpha value specifying opacity from 0x00 to 0xff.

C prototype

```
void cmd_gradienta( int16_t x0,  
                   int16_t y0,  
                   uint32_t argb0,  
                   int16_t x1,  
                   int16_t y1,  
                   uint32_t argb1 );
```

Parameters

x0
x-coordinate of point 0, in pixels

y0
y-coordinate of point 0, in pixels

argb0
color of point 0, represented as a 32-bit ARGB value. The most significant of 8 bits (A) define opacity, and the least significant 8bits (B) define blue. For example, 0x80ff0000 is 50% transparent bright red, and 0xff0000ff is fully opaque blue.

x1
x-coordinate of point 1, in pixels

y1
 y-coordinate of point 1, in pixels

argb1
 color of point 1

Command layout

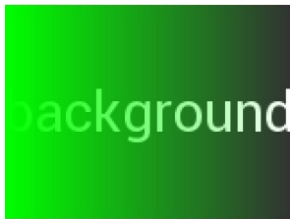
+0	CMD_GRADIENTA(0xFFFF FF57)	
+4	x0	
+6	y0	
+8	argb0	
+12	x1	
+14	y1	
+16	argb1	

Description

All color step values are computed from a smooth curve interpolated between RGB0 and RGB1 parameters. The smooth curve is calculated independently for each color channel using the formula $R0 + t * (R1 - R0)$, where "t" varies from 0 to 1. The gradient should be used together with the scissor function to achieve the intended gradient display.

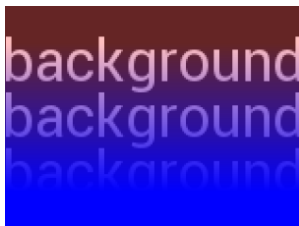
Examples

A solid green gradient, transparent on the right:



```
cmd_text(80, 60, 30, OPT_CENTER, "background");
cmd_gradienta(0,0,0xff00ff00,160,0,0x0000ff00);
```

A vertical gradient from transparent red to solid blue:



```
cmd_text(80, 30, 30, OPT_CENTER, "background");
cmd_text(80, 60, 30, OPT_CENTER, "background");
cmd_text(80, 90, 30, OPT_CENTER, "background");
cmd_gradienta(0,20,0x40ff0000,0,100,0xff0000ff);
```

5.40 CMD_KEYS

This command is used to render a row of keys.

C prototype

```
void cmd_keys( int16_t x,
               int16_t y,
               uint16_t w,
               uint16_t h,
               uint16_t font,
               uint16_t options,
               const char* s );
```

Parameters

x

x-coordinate of keys top-left, in pixels

y

y-coordinate of keys top-left, in pixels

font

Bitmap handle to specify the font used in key label. The valid range is from 0 to 31

options

By default, keys are drawn with a 3D effect, and the option value is zero. Specifying OPT_FLAT removes the 3D effect. If OPT_CENTER is set, the keys are drawn at their minimum size, centered within the w x h rectangle. Otherwise, the keys are expanded to fully fill the available space. If an ASCII code is provided, that key is drawn in a "pressed" state-- using the background color with any 3D effect removed.

w

The width of the keys

h

The height of the keys

s

key labels, with one character assigned per key. The TAG value for each key is set to its ASCII code, allowing key presses to be detected through the REG_TOUCH_TAG register.

Command layout

+0	CMD_KEYS(0xFFFF FF0E)	
+4	x	
+6	y	
+8	w	
+10	h	
+12	font	
+14	options	
+16	s	
	...	
	0	(null character to terminate string)

Description

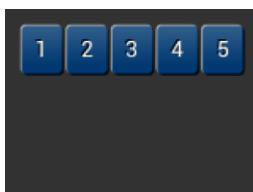
The physical dimension are as follows:

- The gap between keys is 3 pixels.
- In the **OPT_CENTERX** case, each key is (font width + 1.5) pixels wide; otherwise keys are sized to fully fill the available width.

Refer to [Coprocessor engine widgets physical dimensions](#) for more information.

Examples

A row of keys:



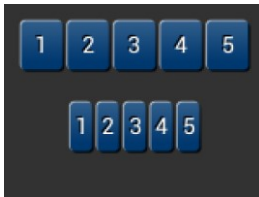
```
cmd_keys(10, 10, 140, 30, 26, 0, "12345");
```

Without the 3D look:



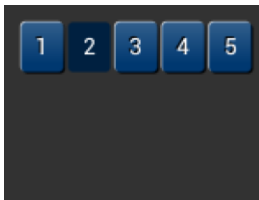
```
cmd_keys(10, 10, 140, 30, 26, OPT_FLAT, "12345");
```

Default vs. centered:



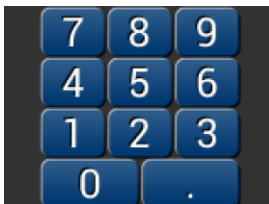
```
cmd_keys(10, 10, 140, 30, 26, 0, "12345");  
cmd_keys(10, 60, 140, 30, 26, OPT_CENTER, "12345");
```

Setting the options to display the "" key pressed "" is ASCII code 0x32):



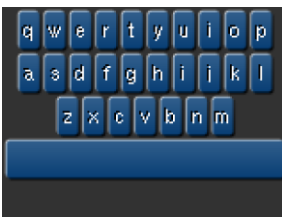
```
cmd_keys(10, 10, 140, 30, 26, 0x32, "12345");
```

A calculator-style keyboard using font 29:



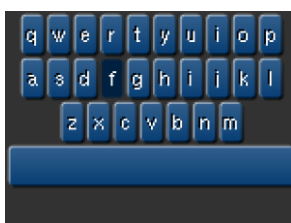
```
cmd_keys(22, 1, 116, 28, 29, 0, "789");  
cmd_keys(22, 31, 116, 28, 29, 0, "456");  
cmd_keys(22, 61, 116, 28, 29, 0, "123");  
cmd_keys(22, 91, 116, 28, 29, 0, "0.");
```

A compact keyboard drawn in font 20:



```
cmd_keys(2, 2, 156, 21, 20, OPT_CENTER, "qwertyuiop");  
cmd_keys(2, 26, 156, 21, 20, OPT_CENTER, "asdfghijkl");  
cmd_keys(2, 50, 156, 21, 20, OPT_CENTER, "zxcvbnm");  
cmd_button(2, 74, 156, 21, 20, 0, "");
```

Showing the f (ASCII 0x66) key pressed:



```
k = 0x66;  
cmd_keys(2, 2, 156, 21, 20, k | OPT_CENTER, "qwertyuiop");  
cmd_keys(2, 26, 156, 21, 20, k | OPT_CENTER, "asdfghijkl");  
cmd_keys(2, 50, 156, 21, 20, k | OPT_CENTER, "zxcvbnm");  
Cmd_button(2, 74, 156, 21, 20, 0, "");
```

5.41 CMD_PROGRESS

This command is used to draw a progress bar.

C prototype

```
void cmd_progress( int16_t x,
                  int16_t y,
                  uint16_t w,
                  uint16_t h,
                  uint16_t options,
                  uint16_t val,
                  uint16_t range );
```

Parameters

x
x-coordinate of progress bar top-left, in pixels

y
y-coordinate of progress bar top-left, in pixels

w
width of progress bar, in pixels

h
height of progress bar, in pixels

options
By default, the progress bar is drawn with a 3D effect and options value is set to zero. The OPT_FLAT option removes the 3D effect and has a value of 256.

val
Displayed value of progress bar, between 0 and range inclusive.

range
Maximum value

Command layout

+0	CMD_PROGRESS(0xFFFF FF0F)
+4	x
+6	y
+8	w
+10	h
+12	options
+14	val
+16	range

Description

The physical dimensions are defined as follows:

- x,y,w,h specifies the outer dimensions of the progress bar. The bar radius (r) is calculated as $\min(w,h)/2$
- The radius of the inner progress line is $r*(7/8)$

Refer to [Coprocessor engine widgets physical dimensions](#) for more information.

Examples

A progress bar indicating 50% completion:



```
cmd_progress(20, 50, 120, 12, 0, 50, 100);
```

Without the 3D look:



```
cmd_progress(20, 50, 120, 12, OPT_FLAT, 50, 100);
```

A 4-pixel-tall bar, spanning 0-65535, on a brown background:



```
cmd_bgcolor(0x402000);  
cmd_progress(20, 50, 120, 4, 0, 9000, 65535);
```

5.42 CMD_SCROLLBAR

This command is used to draw a scroll bar.

C prototype

```
void cmd_scrollbar( int16_t x,  
                  int16_t y,  
                  uint16_t w,  
                  uint16_t h,  
                  uint16_t options,  
                  uint16_t val,  
                  uint16_t size,  
                  uint16_t range );
```

Parameters

x
x-coordinate of scroll bar top-left, in pixels

y
y-coordinate of the scroll bar's top-left corner, in pixels.

w
Width of the scroll bar, in pixels. If the width is greater than the height, the scroll bar is drawn horizontally.

h
Height of the scroll bar, in pixels. If the height is greater than the width, the scroll bar is drawn vertically.

options

By default, the scroll bar is drawn with a 3D effect, and the value of options is zero. The OPT_FLAT Option removes the 3D effect and has a value of 256.

val

Displayed value of the scroll bar, between 0 and the range inclusive.

size

Displayed size of the scroll bar, between 0 and the range inclusive.

range

Maximum value

Command layout

+0	CMD_SCROLLBAR(0xFFFF FF11)
+4	x
+6	y
+8	w
+10	h
+12	options
+14	val
+16	size
+18	range

Description

Refer to **CMD_PROGRESS** for more information on physical dimension.

Examples

A scroll bar indicates 10-50%:



```
cmd_scrollbar(20, 50, 120, 8, 0, 10, 40, 100);
```

Without the 3D look:



```
cmd_scrollbar(20, 50, 120, 8, OPT_FLAT, 10, 40, 100);
```

A brown-themed vertical scroll bar:



```
cmd_bgcolor(0x402000);
cmd_fgcolor(0x703800);
cmd_scrollbar(140, 10, 8, 100, 0, 10, 40, 100);
```

5.43 CMD_SLIDER

This command is to draw a slider.

C prototype

```
void cmd_slider( int16_t x,
                int16_t y,
                uint16_t w,
                uint16_t h,
                uint16_t options,
                uint16_t val,
                uint16_t range );
```

Parameters

x

x-coordinate of slider's top-left corner, in pixels.

y

y-coordinate of slider's top-left corner, in pixels.

w

width of the slider, in pixels. If the width is greater than the height, the scroll bar is drawn horizontally.

h

height of the slider, in pixels. If the height is greater than the width, the scroll bar is drawn vertically.

options

By default, the slider is drawn with a 3D effect. The OPT_FLAT option removes the 3D effect.

val

Displayed value of slider, between 0 and the range inclusive.

range

Maximum value

Command layout

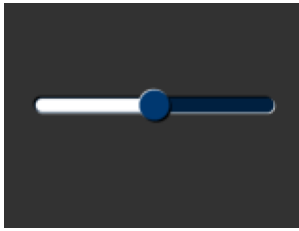
+0	CMD_SLIDER(0xFFFF FF10)
+4	x
+6	y
+8	w
+10	h
+12	options
+14	val
+16	range

Description

Refer to **CMD_PROGRESS** for more information on physical dimension.

Examples

A slider set to 50%:



```
cmd_slider(20, 50, 120, 8, 0, 50, 100);
```

Without the 3D look:



```
cmd_slider(20, 50, 120, 8, OPT_FLAT, 50, 100);
```

A brown-themed vertical slider, with range 0-65535:



```
cmd bgcolor(0x402000);  
cmd fgcolor(0x703800);  
cmd_slider(76, 10, 8, 100, 0, 20000, 65535);
```

5.44 CMD_DIAL

This command is used to draw rotary dial control.

C prototype

```
void cmd_dial( int16_t x,  
              int16_t y,  
              uint16_t r,  
              uint16_t options,  
              uint16_t val );
```

Parameters

x
x-coordinate of the dial's center, in pixels.

y
y-coordinate of the dial's center, in pixels.

r
Radius of the dial, in pixels.

options

By default, the dial is drawn with a 3D effect, and the value of options is zero. The OPT_FLAT option removes the 3D effect and has a value of 256.

val

Specify the position of the dial pointer by setting a value between 0 and 65535 inclusive. A value of 0 points the dial straight down, 0x4000 points left, 0x8000 points up, and 0xc000 points right.

Command layout

+0	CMD_DIAL(0xFFFF FF2D)
+4	x
+6	y
+8	r
+10	options
+12	val

Description

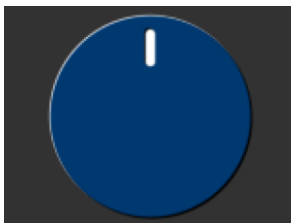
The details of the physical dimension are:

- The marker is a line with a width of $r*(12/256)$, drawn at a distance from the center ranging from $r*(140/256)$ to $r*(210/256)$.

Refer to [Coprocessor engine widgets physical dimensions](#) for more information.

Examples

A dial set to 50%:



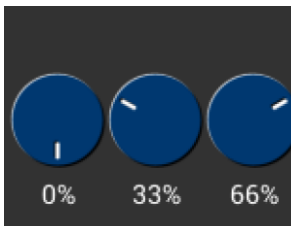
```
cmd_dial(80, 60, 55, 0, 0x8000);
```

Without the 3D look:



```
cmd_dial(80, 60, 55, OPT_FLAT, 0x8000);
```

Dials set to 0%, 33% and 66%:



```
cmd_dial(28, 60, 24, 0, 0x0000);
cmd_text(28, 100, 26, OPT_CENTER, "0%");
cmd_dial(80, 60, 24, 0, 0x5555);
cmd_text(80, 100, 26, OPT_CENTER, "33%");
cmd_dial(132, 60, 24, 0, 0xaaaa);
cmd_text(132, 100, 26, OPT_CENTER, "66%");
```

5.45 CMD_TOGGLE

This command is used to draw a toggle switch with UTF-8 labels.

C prototype

```
void cmd_toggle( int16_t x,
                int16_t y,
                uint16_t w,
                uint16_t font,
                uint16_t options,
                uint16_t state,
                const char* s );
```

Parameters

x

x-coordinate of the toggle's top-left corner, in pixels.

y

y-coordinate of the toggle's top-left corner, in pixels.

w

width of the toggle, in pixels

font

Font to use for text, 0-31. See [ROM and RAM Fonts](#).

options

By default, the toggle is drawn with a 3D effect, and the value of options is zero. The **OPT_FLAT** option removes the 3D effect and has a value of 256.

state

State of the toggle: 0 is off, 65535 is on.

s

String labels for toggle, in UTF-8 encoding. A character value of 255 (in C, written as '\xff') separates the label strings. See [5.6_String Formatting](#).

Command layout

+0	CMD_TOGGLE(0xFFFF FF12)	
+4	x	
+6	y	
+8	w	
+10	font	
+12	options	
+14	state	
+16	s	
	...	
	0	(null character to terminate string)

Description

The details of physical dimension are:

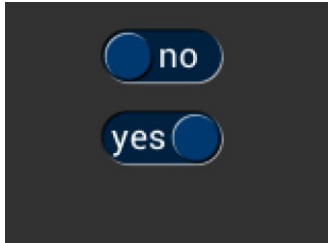
- Widget height (h) is font height * (20/16) pixels.
- Outer bar radius (r) is font height * (10/16) pixels.
- Knob radius is (r-1.5) pixels, where r is the outer bar radius above.

- The center of the outer bar's left rounded end is at $(x, y + r/2)$.

Refer to [Coprocesor engine widgets physical dimensions](#) for more information.

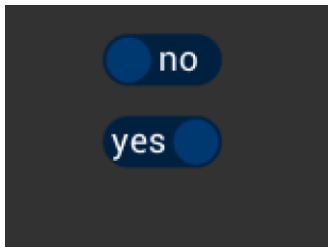
Examples

Using a medium font, in the two states:



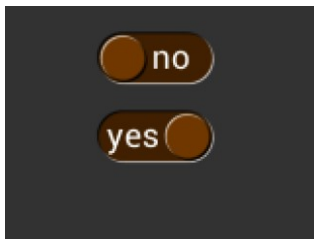
```
cmd_toggle(60, 20, 33, 27, 0, 0, "no" "\xff" "yes");
cmd_toggle(60, 60, 33, 27, 0, 65535, "no" "\xff" "yes");
```

Without the 3D look:



```
cmd_toggle(60, 20, 33, 27, OPT_FLAT, 0, "no" "\xff" "yes");
cmd_toggle(60, 60, 33, 27, OPT_FLAT, 65535, "no" "\xff" "yes");
```

With different background and foreground colors:



```
cmd_bgcolor(0x402000);
cmd_fgcolor(0x703800);
cmd_toggle(60, 20, 33, 27, 0, 0, "no" "\xff" "yes");
cmd_toggle(60, 60, 33, 27, 0, 65535, "no" "\xff" "yes");
```

5.46 CMD_FILLWIDTH

This command sets the pixel fill width for **CMD_TEXT**, **CMD_BUTTON**, and **CMD_BUTTON** with the **OPT_FILL** option.

C prototype

```
void cmd_fillwidth( uint32_t s );
```

Parameters

s
line fill width, in pixels

Command layout

+0	CMD_FILLWIDTH(0xFFFF FF58)
+4	s

Examples

Long text split into lines of no more than 160 pixels:

This text
doesn't fit
on one line

```
cmd_fillwidth(160);
cmd_text(0, 0, 30, OPT_FILL, "This text doesn't fit on one
line");
```

5.47 CMD_TEXT

This command is used to draw a UTF-8 text string.

C prototype

```
void cmd_text( int16_t x,
               int16_t y,
               uint16_t font,
               uint16_t options,
               const char* s );
```

Parameters

x
x-coordinate of the text baseline, in pixels.

y
y-coordinate of the text baseline, in pixels.

font
Font to use for text, 0-31. See [ROM and RAM Fonts](#)

options
By default (x,y) is the top-left pixel of the text, and the value of options is zero.
OPT_CENTERX centers the text horizontally.
OPT_CENTERY centers the text vertically.
OPT_CENTER centers the text both horizontally and vertically.
OPT_RIGHTX right-justifies the text, so that the x is the rightmost pixel.
OPT_FORMAT processes the text as a format string, see String formatting.
OPT_FILL breaks the text at spaces into multiple lines, with a maximum width set by **CMD_FILLWIDTH**.

s
Text string in UTF-8 encoding. If **OPT_FILL** is not specified, the string may contain newline (\n) characters to indicate line breaks. See [5.6 String Formatting](#).

Command layout

+0	CMD_TEXT(0xFFFF FFOC)
+4	x
+6	y
+8	font
+10	options
+12	s
	...
	0 (null character to terminate string)

Examples

Plain text at (0,0) in the largest font:



```
cmd_text(0, 0, 31, 0, "Text!");
```

Using a smaller font:



```
cmd_text(0, 0, 26, 0, "Text!");
```

Centered horizontally:



```
cmd_text(80, 60, 31, OPT_CENTERX, "Text!");
```

Right justified:



```
cmd_text(80, 60, 31, OPT_RIGHTX, "Text!");
```

Centered vertically:



```
cmd_text(80, 60, 31, OPT_CENTERY, "Text!");
```

Centered both horizontally and vertically:



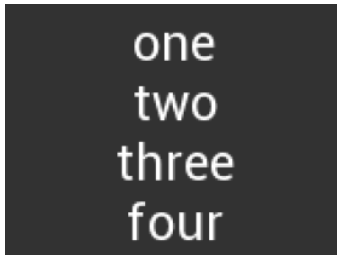
```
cmd_text(80, 60, 31, OPT_CENTER, "Text!");
```

Text with explicit newline:



```
cmd_text(80, 60, 29, OPT_CENTER, "one two\nthree four");
```

Text split into lines and centered:



```
cmd_fillwidth(80);  
cmd_text(80, 60, 29, OPT_FILL | OPT_CENTER, "one two three  
four");
```

5.48 CMD_SETBASE

This command is used to set the base for number output.

C prototype

```
void cmd_setbase( uint32_t b );
```

Parameters

b
 Numeric base, valid values are from 2 to 36:
 2 for binary,
 8 for octal,
 10 for decimal,
 16 for hexadecimal

Command layout

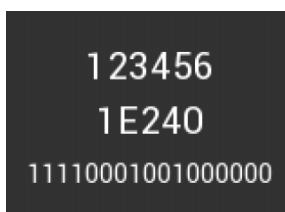
+0	CMD_SETBASE(0xFFFF FF38)
+4	b

Description

Set up numeric base for **CMD_NUMBER**

Examples

The number 123456 displayed in decimal, hexadecimal and binary:



```
cmd_number(80, 30, 28, OPT_CENTER, 123456);  
cmd_setbase(16);  
cmd_number(80, 60, 28, OPT_CENTER, 123456);  
cmd_setbase(2);  
cmd_number(80, 90, 26, OPT_CENTER, 123456);
```

5.49 CMD_NUMBER

This command is used to draw a number.

C prototype

```
void cmd_number( int16_t x,
                int16_t y,
                uint16_t font,
                uint16_t options,
                int32_t n );
```

Parameters

x

x-coordinate of the text's baseline, in pixels.

y

y-coordinate of the text's baseline, in pixels.

font

font to use for the text numbered 0-31. See ROM and RAM Fonts.

options

By default (x,y) is the top-left pixel of the text.

OPT_CENTERX centers the text horizontally.

OPT_CENTERY centers the text vertically.

OPT_CENTER centers the text both horizontally and vertically.

OPT_RIGHTX right-justifies the text, so that the x is the rightmost pixel.

By default, the number is displayed with no leading zeroes, but if a width 1-9 is specified in options, then the number is padded with leading zeros, if necessary, to reach the specified width. If OPT_SIGNED is set, the number is treated as signed and prefixed with a minus sign if negative.

n

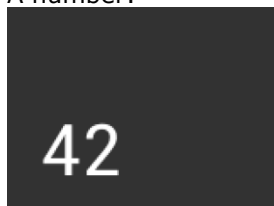
The number to display is either an unsigned or signed 32-bit value, in the base specified by the preceding [CMD_SETBASE](#). If no CMD_SETBASE precedes **CMD_NUMBER**, the number is displayed in decimal.

Command layout

+0	CMD_NUMBER(0xFFFF FF2E)	
+4	x	
+6	y	
+8	font	
+10	options	
+12	n	

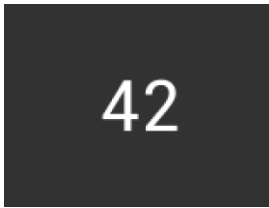
Examples

A number:



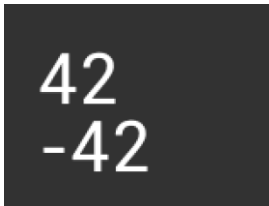
```
cmd number(20, 60, 31, 0, 42);
```

Centered:



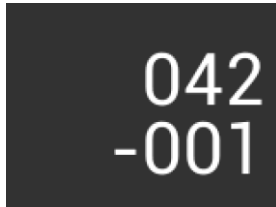
```
cmd_number(80, 60, 31, OPT_CENTER, 42);
```

Signed output of positive and negative numbers:



```
cmd_number(20, 20, 31, OPT_SIGNED, 42);  
cmd_number(20, 60, 31, OPT_SIGNED, -42);
```

Forcing width to 3 digits, right justified



```
cmd_number(150, 20, 31, OPT_RIGHTX | 3, 42);  
cmd_number(150, 60, 31, OPT_SIGNED | OPT_RIGHTX | 3, -1);
```

5.50 CMD_NOP

This command is a placeholder and performs no action.

C prototype

```
void cmd_nop( );
```

Command layout

+0	CMD_NOP(0xFFFF FF5B)
----	-----------------------------

5.51 CMD_LOADIDENTITY

This command instructs the coprocessor engine to set the current matrix to the identity matrix, allowing it to form a new matrix as specified by **CMD_SCALE**, **CMD_ROTATE**, or **CMD_TRANSLATE** command. For more information on the identity matrix, refer to the [Bitmap Transformation Matrix](#) section.

C prototype

```
void cmd_loadidentity( );
```

Command layout

+0	CMD_LOADIDENTITY(0xFFFF FF26)
----	--------------------------------------

5.52 CMD_SETMATRIX

The coprocessor engine assigns the value of the current matrix to the graphic engine's bitmap transform matrix by generating display list commands, i.e., BITMAP_TRANSFORM_A-F. After this command, subsequent bitmap rendering operations will be affected by the new transform matrix.

C prototype

```
void cmd_setmatrix( );
```

Command layout

+0	CMD_SETMATRIX(0xFFFF FF2A)
----	----------------------------

5.53 CMD_GETMATRIX

This command retrieves the current matrix from the coprocessor engine. Note that this matrix will not affect bitmap transformation until it is applied to the graphics engine using **CMD_SETMATRIX**.

C prototype

```
void cmd_getmatrix( int32_t a,  
                   int32_t b,  
                   int32_t c,  
                   int32_t d,  
                   int32_t e,  
                   int32_t f );
```

Parameters

a

output parameter; receives the matrix coefficient a. See the parameter description of the BITMAP_TRANSFORM_A command for formatting.

b

output parameter; receives the matrix coefficient b. See the parameter b of the BITMAP_TRANSFORM_B command for formatting.

c

output parameter; receives the matrix coefficient c. See the parameter c of the BITMAP_TRANSFORM_C command for formatting.

d

output parameter; receives the matrix coefficient d. See the parameter d of the BITMAP_TRANSFORM_D command for formatting.

e

output parameter; receives the matrix coefficient e. See the parameter e of the BITMAP_TRANSFORM_E command for formatting.

f

output parameter; receives the matrix coefficient f. See the parameter f of the BITMAP_TRANSFORM_F command for formatting.

Command layout

+0	CMD_GETMATRIX(0xFFFF FF33)
+4	a
+8	b
+12	c
+16	d
+20	e
+24	f

5.54 CMD_GETPTR

This command returns to the first unallocated memory location.

At API level 1, the allocation pointer is advanced by the following commands:

- cmd_inflate
- cmd_inflate2

At API level 2, the allocation pointer is additionally advanced by:

- cmd_loadimage
- cmd_playvideo
- cmd_videoframe
- cmd_endlist

C prototype

```
void cmd_getptr( uint32_t result );
```

Parameters

result

The first unallocated memory location.

Command layout

+0	CMD_GETPTR (0xFFFF FF23)
+4	result

Examples

```
cmd_inflate(1000); //Decompress the data into RAM_G + 1000
.....           //Following the zlib compressed data
While(rd16(REG_CMD_WRITE) != rd16(REG_CMD_READ)); //Wait till the compression was done

uint16_t x = rd16(REG_CMD_WRITE);
uint32_t ending_address = 0;
cmd_getptr(0);
ending_address = rd32(RAM_CMD + (x + 4) % 4096);
```

5.55 CMD_GETPROPS

This command returns the source address and size of the bitmap loaded by the previous **CMD_LOADIMAGE**.

C prototype

```
void cmd_getprops( uint32_t ptr,
                  uint32_t width,
                  uint32_t height );
```

Parameters

ptr

Source address of bitmap.

Note:

At API Level 2 this parameter returns the source address of the decoded image data in RAM_G

At API level 1, this parameter has different meaning depending on the input image format of CMD_LOADIMAGE:

For JPEG, it is the source address of the decoded image data in RAM_G.
 For PNG, it is the first unused address in RAM_G after decoding process.
 This is an output parameter.

width

The width of the image decoded by the last CMD_LOADIMAGE executed before this command. It is an output parameter.

height

The height of the image decoded by the last CMD_LOADIMAGE executed before this command. It is an output parameter.

Command layout

+0	CMD_GETPROPS (0xFFFF FF25)
+4	ptr
+8	width
+12	height

Description

This command retrieves the properties of the image decoded by **CMD_LOADIMAGE**. The coprocessor updates the respective image properties after this command executes successfully.

Examples

Please refer to the [CMD_GETPTR](#).

5.56 CMD_SCALE

This command is used to apply a scaling transformation to the current matrix.

C prototype

```
void cmd_scale( int32_t sx,
               int32_t sy );
```

Parameters

sx
x scale factor, in signed 16.16-bit fixed-point format.

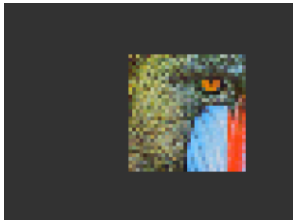
sy
y scale factor, in signed 16.16-bit fixed-point format.

Command layout

+0	CMD_SCALE(0xFFFF FF28)
+4	sx
+8	sy

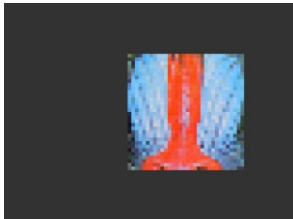
Examples

To zoom a bitmap 2X:



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_scale(2 * 65536, 2 * 65536);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

To zoom a bitmap 2X around its center:



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_translate(65536 * 32, 65536 * 32);
cmd_scale(2 * 65536, 2 * 65536);
cmd_translate(65536 * -32, 65536 * -32);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

5.57 CMD_ROTATE

This command is used to apply a rotation to the current matrix.

C prototype

```
void cmd_rotate( uint32_t a );
```

Parameters

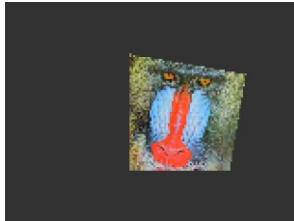
a
Clockwise rotation angle, in units of 1/65536 of a full circle.

Command layout

+0	CMD_ROTATE(0xFFFF FF29)
+4	a

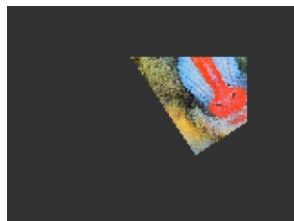
Examples

To rotate the bitmap clockwise by 10 degrees relative to the top-left corner of the bitmap:



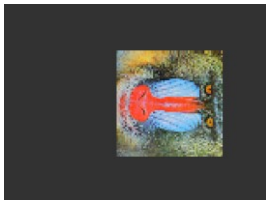
```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_rotate(10 * 65536 / 360);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

To rotate the bitmap counterclockwise by 33 degrees around the top-left corner of the bitmap:



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_rotate(-33 * 65536 / 360);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

Rotating a 64 x 64 bitmap around its center:



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_translate(65536 * 32, 65536 * 32);
cmd_rotate(90 * 65536 / 360);
cmd_translate(65536 * -32, 65536 * -32);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

5.58 CMD_ROTATEAROUND

This command is used to apply rotation and scaling around a specified coordinate.

C prototype

```
void cmd_rotatearound( int32_t x,
                      int32_t y,
                      uint32_t a,
                      int32_t s );
```

Parameters

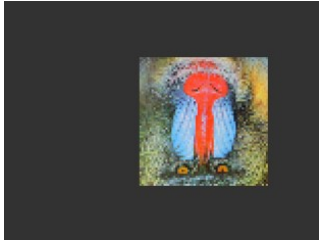
- x**
x-coordinate of the rotation/scaling center.
- y**
y-coordinate of the rotation/scaling center.
- a**
clockwise rotation angle, in units of 1/65536 of a full circle.
- s**
scale factor, in signed 16.16-bit fixed-point format.

Command layout

+0	CMD_ROTATEAROUND(0xFFFF FF51)
+4	x
+8	y
+12	a
+16	s

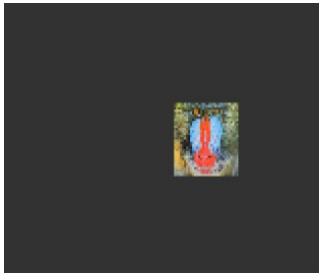
Examples

Rotating a 64 x 64 bitmap around its center:



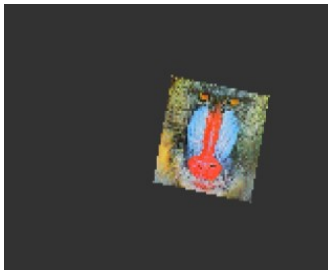
```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_rotatearound(32,32, 180 * 65536 /360, 65536 *1);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

To halve the bitmap size around its center:



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_rotatearound(32, 32, 0, 0.5 * 65536);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

A combined 11-degree rotation and shrink by 0.75



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_rotatearound(32, 32, 11*65536/360, 0.75 * 65536);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

5.59 CMD_TRANSLATE

This command is used to apply a translation to the current matrix.

C prototype

```
void cmd_translate( int32_t tx,
                  int32_t ty );
```

Parameters

tx
 x translation factor, in signed 16.16-bit fixed-point format.

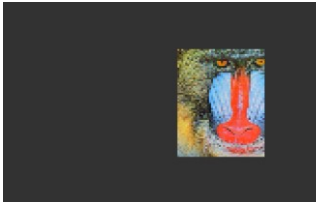
ty
 y translation factor, in signed 16.16-bit fixed-point format.

Command layout

+0	CMD_TRANSLATE(0xFFFF FF27)
+4	tx
+8	ty

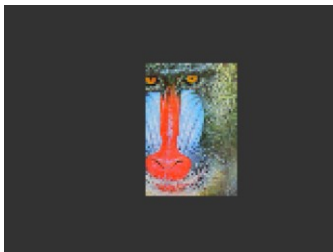
Examples

To translate the bitmap 20 pixels to the right:



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_translate(20 * 65536, 0);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

To translate the bitmap 20 pixels to the left:



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_translate(-20 * 65536, 0);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

5.60 CMD_CALIBRATE

This command is used to execute the touch screen calibration routine. During calibration, the system collects three touch points from the screen, then calculates and loads the corresponding transformation matrix into **REG_TOUCH_TRANSFORM_A-F**. To use the feature, create a display list and include **CMD_CALIBRATE**. The coprocessor engine overlays the calibration targets on the current display list, captures the touch input, and updates **REG_TOUCH_TRANSFORM_A-F**. There is no need to add a **DISPLAY** command or manually swap the frame, as the coprocessor engine handles this automatically when the command is executed.

Please note that this command applies only to the RTE and compatibility mode of CTSE.

C prototype

```
void cmd_calibrate( uint32_t result );
```

Parameters

result
 output parameter; written with 0 on failure of calibration.

Command layout

+0	CMD_CALIBRATE(0xFFFF FF15)
+4	Result

Description

The completion of this function is detected when the value of **REG_CMD_READ** equals **REG_CMD_WRITE**.

Examples

```
cmd_dlstart();
cmd(CLEAR(1,1,1));
cmd_text(80, 30, 27, OPT_CENTER, "Please tap on the dot");
cmd_calibrate(0);
```

5.61 CMD_CALIBRATESUB

This command executes the touch screen calibration routine for a sub-window. It is like **CMD_CALIBRATE**, but instead of using the full screen area, it operates on a smaller sub-window specified by the command. This is intended for panels that do not use the entire defined surface.

Please note that this command applies only to the RTE and compatibility mode of CTSE.

C prototype

```
void cmd_calibratesub( uint16_t x,
                      uint16_t y,
                      uint16_t w,
                      uint16_t h,
                      uint32_t result );
```

Parameters

x
x-coordinate of the top-left corner of the sub-window, in pixels.

y
y-coordinate of the top-left corner of the sub-window, in pixels.

w
width of the sub-window, in pixels.

h
height of the sub-window, in pixels.

result
output parameter; written with 0 on failure.

Command layout

		CMD_CALIBRATESUB(0xFFFF FF60)
+0		
+4	X	
+6	Y	
+8	W	
+10	H	
+12	result	

Note: BT817/8 specific command

Examples

```
cmd_dlstart();
cmd(CLEAR(1,1,1));
cmd_text(600, 140, 31, OPT_CENTER, "Please tap on the dot");
//Calibrate a touch screen for 1200x280 screen
cmd_calibratesub(0,0, 1200,280,0);
```

5.62 CMD_SETROTATE

This command is used to rotate the screen.

C prototype

```
void cmd_setrotate( uint32_t r );
```

Parameters

r
 The value ranges from 0 to 7 and follows the same definition as the value in **REG_ROTATE**. See section [Rotation](#) for more details.

Command layout

+0	CMD_SETROTATE (0xFFFF FF36)
+4	r

Description

CMD_SETROTATE sets **REG_ROTATE** to the specified value *r*, causing the screen to rotate. It also appropriately adjusts the touch transform matrix accordingly so that touch point coordinates are mapped correctly to the rotated coordinate system.

Examples

```
cmd_setrotate(2); //Put the display in portrait mode
```

5.63 CMD_SPINNER

This command is used to start an animated spinner. The spinner is an overlay animation that indicates to the user that a task is in progress. To trigger the spinner, create a display list and include **CMD_SPINNER**. The coprocessor engine overlays the spinner on the current display list, swaps the display list to make it visible, then continuously animates until **CMD_STOP** is received. **REG_MACRO_0** and **REG_MACRO_1** register are used to generate the animation effect. The movement frequency of the points is determined by the configured display frame rate.

For a 480×272 display panel, the typical frame rate is ~60 fps. At 60 fps:

- **Style 0:** the point completes its sequence in 2 seconds
- **Style 1:** the point completes its sequence in 1.25 seconds
- **Style 2:** the clock hand completes its sequence in 2 seconds
- **Style 3:** the moving dots complete their sequence in 1 second

Note that only one **CMD_SKETCH**, **CMD_SCREENSAVER**, or **CMD_SPINNER** can be active at a time.

C prototype

```
void cmd_spinner( int16_t x,
                 int16_t y,
                 uint16_t style,
                 uint16_t scale );
```

Parameters

x
 The X-coordinate of the top-left corner of the spinner.

y
 The Y-coordinate of the top-left corner of the spinner.

style
 The style of the spinner. Valid values range from 0 to 3.

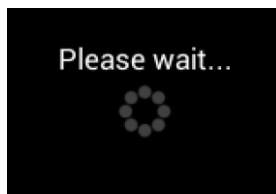
scale
 The scaling coefficient of the spinner. A value of 0 means no scaling.

Command layout

+0	CMD_SPINNER(0xFFFF FF16)
+4	x
+6	Y
+8	Style
+10	Scale

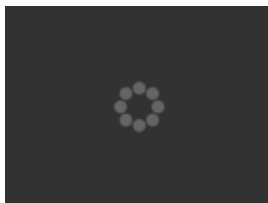
Examples

Create a display list, then start the spinner:

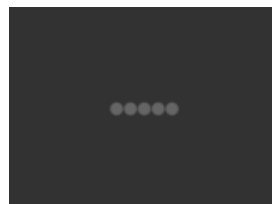


```
cmd_dlstart();
cmd(CLEAR(1,1,1));
cmd_text(80, 30, 27, OPT_CENTER, "Please wait...");
cmd_spinner(80, 60, 0, 0);
```

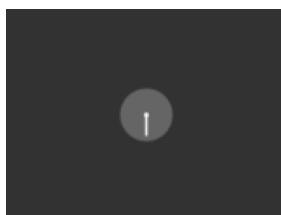
style 0, a circle of dots:



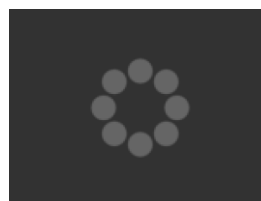
Style 1, a line of dots:



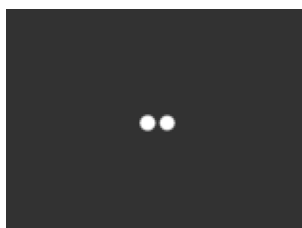
Style 2, a rotating clock hand:



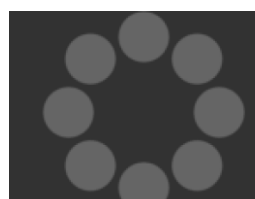
Half screen, scale:



Style 3, two orbiting dots:



Full screen, scale 2:



5.64 CMD_SCREENSAVER

This command is used to start an animated screensaver. Once the screensaver command is issued, the coprocessor engine continuously updates **REG_MACRO_0** using **VERTEX2F** with varying (x,y) coordinates. When used with a suitable display list, this causes a bitmap to move around the screen without any MCU intervention. If the current pixel precision is not set to 1/16th, the **VERTEX_FORMAT(4)** command must be sent before issuing this command.

The command **CMD_STOP** stops the update process. Note that only one **CMD_SKETCH**, **CMD_SCREENSAVER**, or **CMD_SPINNER** can be active at a time.

C prototype

```
void cmd_screensaver( );
```

Command layout

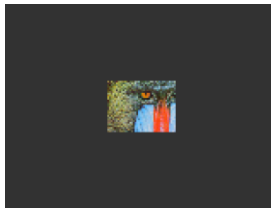
+0	CMD_SCREENSAVER(0xFFFF FF2F)
----	-------------------------------------

Description

REG_MACRO_0 is updated according to the frame rate, based on the display register configuration. For a 480x272 display, the typical frame rate is approximately 60 frames per second.

Examples

To start the screensaver, create a display list using a MACRO instruction. The coprocessor engine will then update it continuously:



```
cmd_screensaver( );
cmd(BITMAP_SOURCE(0));
cmd(BITMAP_LAYOUT(RGB565, 128, 64));
cmd(BITMAP_SIZE(NEAREST,BORDER,BORDER, 40, 30));
cmd(BEGIN(BITMAPS));
cmd(MACRO(0));
cmd(DISPLAY());
```

5.65 CMD_SKETCH

This command is used to start a continuous sketch update. The coprocessor engine continuously samples touch inputs and paints pixels into a bitmap based on the provided touch (x, y) coordinates. This allows user touch input to be drawn directly into the bitmap without any MCU intervention. **CMD_STOP** must be sent to stop the sketch process.

Note that only one **CMD_SKETCH**, **CMD_SCREENSAVER**, or **CMD_SPINNER** can be active at a time.

C prototype

```
void cmd_sketch( int16_t x,
                int16_t y,
                uint16_t w,
                uint16_t h,
                uint32_t ptr,
                uint16_t format );
```

Parameters

x
The X-coordinate of the top-left corner of the sketch area, in pixels

y
The Y-coordinate of the top-left corner of the sketch area, in pixels

w
The width of the sketch area, in pixels

h
The height of the sketch area, in pixels

ptr
The base address of the sketch bitmap

format
The format of the sketch bitmap, either L1 or L8

Note: The update frequency of the bitmap data at ptr depends on the sampling rate of the built-in ADC circuit, which can be up to 1000 Hz.

Command layout

+0	CMD_SKETCH(0xFFFF FF30)
+4	x
+6	y
+8	w
+10	h
+12	ptr
+16	format

Examples

To start sketching into a 480x272 L1 bitmap:

```
cmd_memzero(0, 480 * 272 / 8);
cmd_sketch(0, 0, 480, 272, 0, L1);

//Then to display the bitmap
cmd(BITMAP_SOURCE(0));
cmd(BITMAP_LAYOUT(L1, 60, 272));
cmd(BITMAP_SIZE(NEAREST, BORDER, BORDER, 480, 272));
cmd(BEGIN(BITMAPS));
cmd(VERTEX2II(0, 0, 0, 0));

//Finally, to stop sketch updates
cmd_stop();
```

5.66 CMD_STOP

This command instructs the coprocessor engine to stop the periodic operation initiated by **CMD_SKETCH**, **CMD_SPINNER** or **CMD_SCREENSAVER**.

C prototype

```
void cmd_stop( );
```

Command layout

+0	CMD_STOP(0xFFFF FF17)
----	------------------------------

Description

For **CMD_SPINNER** and **CMD_SCREENSAVER**, updates to **REG_MACRO_0** and **REG_MACRO_1** will be stopped.

For **CMD_SKETCH**, updates to the bitmap data in **RAM_G** will be stopped.

Examples

See [CMD_SKETCH](#), [CMD_SPINNER](#), [CMD_SCREENSAVER](#).

5.67 CMD_SETFONT

CMD_SETFONT is used to register a custom-defined bitmap font with the coprocessor engine. Once registered, the coprocessor engine can use the bitmap font with the appropriate commands.

Note that **CMD_SETFONT** does not configure the font's bitmap parameters. The MCU must set these parameters before using the font. For further details on setting up a custom font, refer to the [ROM and RAM Font](#).

C prototype

```
void cmd_setfont( uint32_t font,  
                 uint32_t ptr );
```

Parameters

font

The bitmap handle, with valid values from 0 to 31.

ptr

The address of the metrics block in **RAM_G**. Must be 4-byte aligned.

Command layout

+0	CMD_SETFONT(0xFFFF FF2B)
+4	Font
+8	Ptr

Examples

With a suitable font metrics block loaded in **RAM_G** at address 1000, it can be set it up for use as font7 for objects:

```
cmd(BITMAP_LAYOUT(L8,16, 10));
cmd(BITMAP_SIZE(NEAREST,BORDER,BORDER, 16, 10));
cmd(BITMAP_SOURCE(1000));
cmd_setfont(7, 1000);
cmd_button( 20, 20, // x,y
            120, 40, // width,height in pixels
            7, // font 7, just loaded
            0, // default options,3D style
            "custom font!" );
```

5.68 CMD_SETFONT2

This command is used to set up a custom font. To use a custom font with the coprocessor objects, create the font definition data in **RAM_G** and issue **CMD_SETFONT2**, as described in ROM and RAM Fonts.

Note that **CMD_SETFONT2** sets up the font's bitmap parameters by appending **BITMAP_SOURCE**, **BITMAP_LAYOUT** and **BITMAP_SIZE** commands to the current display list.

For details on setting up a custom font, refer to [ROM and RAM Fonts](#).

C prototype

```
void cmd_setfont2( uint32_t font,
                  uint32_t ptr,
                  uint32_t firstchar );
```

Parameters

font

The bitmap handle, with valid values from 0 to 31.

ptr

The 32-bit aligned memory address in **RAM_G** of the font metrics block.

firstchar

The **ASCII** value of the first character in the font. **For an extended font block, this value should be zero.**

Command layout

+0	CMD_SETFONT2(0xFFFF FF3B)
+4	font
+8	ptr
+12	firstchar

Examples

With a suitable font metrics block loaded in RAM_G at address 0x100000 and the first character's ASCII value set to 32, it can be configured for use as font20:



```
cmd_setfont2(20, 100000, 32);
cmd_button(15, 30, 130, 20, 18, 0, "This is font 18");
cmd_button(15, 60, 130, 20, 20, 0, "This is font 20");
```

5.69 CMD_SETSCRATCH

This command is used to set the scratch bitmap for widget use. Graphical objects use a bitmap handle for rendering, which defaults to bitmap handle 15. This command allows the handle to be changed to any other bitmap, enabling safe use of bitmap handle 15 for other purposes.

C prototype

```
void cmd_setscratch( uint32_t handle );
```

Parameters

handle

The bitmap handle number, with valid values from 0 to 31.

Command layout

+0	CMD_SETSCRATCH (0xFFFF FF3C)
+4	Handle

Examples

Using the SETSCRATCH command, set the scratch bitmap to handle 31. This makes handle 15 available for application use, for example as a font.



```
cmd_setscratch(31);
cmd_setfont2(15, 100000, 32);
cmd_button(15, 30, 130, 20, 15, 0, "This is font 15");

//Restore bitmap handle 31 to ROM Font number 31.
cmd_romfont(31, 31);
```

5.70 CMD_ROMFONT

This command is to load a ROM font into a specified bitmap handle. By default, ROM fonts 16-31 are mapped to bitmap handles 16-31. This command allows any ROM font 16-34 to be loaded into any bitmap handle.

C prototype

```
void cmd_romfont( uint32_t font,
                  uint32_t romslot );
```

Parameters

font

The bitmap handle number, with valid values from 0 to 31.

romslot

The ROM font number, with valid values from 16 to 34.

Command layout

+0	CMD_ROMFONT (0xFFFF FF3F)
+4	Font
+8	Romslot

Examples

To load hardware fonts 31-34 into bitmap handle 1:



```
cmd_romfont(1, 31);
cmd_text(0, 0, 1, 0, "31");
cmd_romfont(1, 32);
cmd_text(0, 60, 1, 0, "32");
cmd_romfont(1, 33);
cmd_text(80, -14, 1, 0, "33");
cmd_romfont(1, 34);
cmd_text(60, 32, 1, 0, "34");
```

5.71 CMD_RESETFONTS

This command loads bitmap handles 16-31 with their corresponding default ROM fonts.

C prototype

```
void cmd_resetfonts( );
```

Command layout

+0	CMD_RESETFONTS (0xFFFF FF52)
----	-------------------------------------

5.72 CMD_TRACK

This command is used to track touches for a graphics object. **EVE** can assist the **MCU** by handling touch tracking on graphical objects. For example, when a dial object is tracked, EVE can report the touch position as an angle, eliminating the need for the MCU to calculate it. To use this feature, MCU draws the object with a selected tag value and defines a tracking area associated with that tag. After this setup, any touch on the object is reported in **REG_TRACKER**, with additional simultaneous touches (if supported by the touch panel) reported in **REG_TRACKER_1**, **REG_TRACKER_2**, **REG_TRACKER_3**, and **REG_TRACKER_4**.

The MCU detects touches by reading the 32-bit values in these five registers. The lower 8 bits contain the current tag value, or zero if no touch is detected. The upper sixteen bits contain the tracked value. For a rotary tracker, used with clocks, gauges and dials, the tracked value represents the touch angle relative to the object center, expressed in units of 1/65536 of a full circle. An angle of 0 corresponds to straight down, 0x4000 to left, 0x8000 to up, and 0xc000 to right.

For a linear tracker, used with sliders and scrollbars, the tracked value represents the position along the tracked object, ranging from 0 to 65535.

Note: Multiple touch points are supported only in BT81X series devices when a capacitive touch display is connected.

C prototype

```
void cmd_track( int16_t x,
               int16_t y,
               uint16_t w,
               uint16_t h,
               uint16_t tag );
```

Parameters

x
 For linear tracker functionality, this specifies the x-coordinate of the top-left corner of the track area, in pixels.

For rotary tracker functionality, this specifies the x-coordinate of the center of the track area, in pixels.

y

For linear tracker functionality, this specifies the y-coordinate of the top-left corner of the track area, in pixels.

For rotary tracker functionality, this specifies the y-coordinate of the center of the track area, in pixels.

w

Width of the track area, in pixels.

h

Height of the track area, in pixels.

Note: A width (w) and height (h) of (1,1) indicate a rotary tracker, which reports an angle value in **REG_TRACKER**. A width and height of (0,0) disable the coprocessor's tracking functionality. Any other width and height values select a linear tracker, which reports position values along its length from 0 to 65535 in **REG_TRACKER**.

tag

Tag value of the graphics object to be tracked, in the range 1-255.

Command layout

+0	CMD_TRACK(0xFFFF FF2C)
+4	x
+6	y
+8	w
+10	h
+12	tag

Description

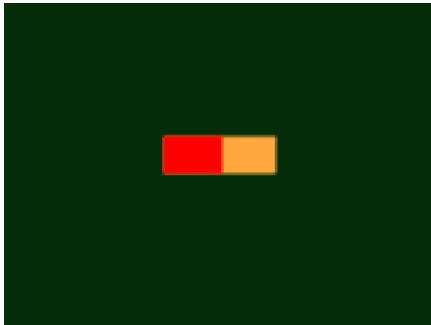
The Coprocessor engine supports tracking of a graphic object in both rotary tracker and linear tracker modes:

- Rotary tracker mode – Tracks the angle between the touch point and the center of the graphics object identified by the specified tag value. The value is in units of 1/65536 of a circle. 0 means that the angle is straight down, 0x4000 left, 0x8000 up, and 0xC000 right from the center.
- Linear tracker mode – If parameter w is greater than h, the coprocessor tracks the relative distance of the touch point along the width of the graphics object identified by the specified tag value. If parameter w is not greater than h, the relative distance is tracked along the height of the graphics object. The reported value is in units of 1/65536 of the object's width or height. The distance is measured from the top-left pixel of the graphics object to the touch point coordinate.

Note: The behavior of **CMD_TRACK** is undefined if the center of the tracked object (for rotary tracking) or the top-left corner of the tracked object (for linear tracking) lies outside the visible region of the display panel.

Examples

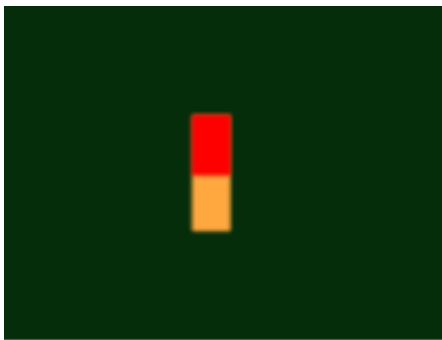
A 40 x 12 pixel rectangle is tracked horizontally, and the current touch position is at the middle (50%) of the track:



```

dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1, 1, 1) );
dl( BEGIN(RECTS) );
dl( VERTEX2F(60 * 16, 50 * 16) );
dl( VERTEX2F(100 * 16, 62 * 16) );
dl( COLOR_RGB(255, 0, 0) );
dl( VERTEX2F(60 * 16, 50 * 16) );
dl( VERTEX2F(80 * 16, 62 * 16) );
dl( COLOR_MASK(0, 0, 0, 0) );
dl( TAG(1) );
dl( VERTEX2F(60 * 16, 50 * 16) );
dl( VERTEX2F(100 * 16, 62 * 16) );
cmd_track(60, 50, 40, 12, 1);
  
```

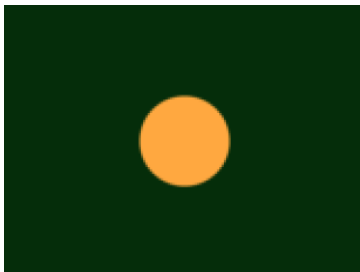
A 12 x 40 pixel rectangle is tracked vertically, and the current touch position is at the middle (50%) of the track:



```

dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1, 1, 1) );
dl( BEGIN(RECTS) );
dl( VERTEX2F(70 * 16, 40 * 16) );
dl( VERTEX2F(82 * 16, 80 * 16) );
dl( COLOR_RGB(255, 0, 0) );
dl( VERTEX2F(70 * 16, 40 * 16) );
dl( VERTEX2F(82 * 16, 60 * 16) );
dl( COLOR_MASK(0, 0, 0, 0) );
dl( TAG(1) );
dl( VERTEX2F(70 * 16, 40 * 16) );
dl( VERTEX2F(82 * 16, 80 * 16) );
cmd_track(70, 40, 12, 40, 1);
  
```

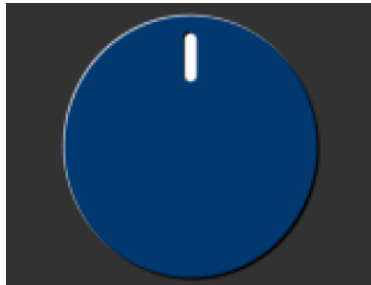
A circular track is centered at the display coordinates (80,60):



```

dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1, 1, 1) );
dl( TAG(1) );
dl( BEGIN(POINTS) );
dl( POINT_SIZE(20 * 16) );
dl( VERTEX2F(80 * 16, 60 * 16) );
cmd_track(80, 60, 1, 1, 1);
  
```

To draw a dial with tag 33 centered at (80, 60) that can be adjusted by touch:

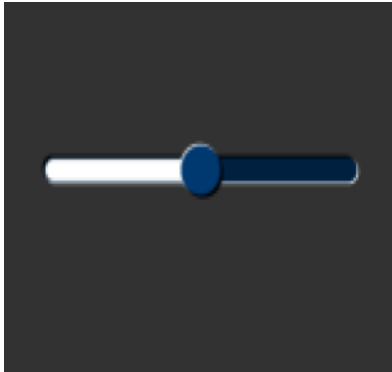


```

uint16_t angle = 0x8000;
cmd_track(80, 60, 1, 1, 33);
while (1) {
  cmd(TAG(33));
  cmd_dial(80, 60, 55, 0, angle);

  uint32_t tracker = rd32(REG_TRACKER);
  if ((tracker & 0xff) == 33)
    angle = tracker >> 16;
}
  
```

To create an adjustable slider with tag 34:



```
uint16_t val = 0x8000;
cmd_track(20, 50, 120, 8, 34);
while (1) {
  cmd(TAG(34));
  cmd_slider(20, 50, 120, 8, val, 65535);

  uint32_t tracker = rd32(REG_TRACKER);
  if ((tracker & 0xff) == 34)
    val = tracker >> 16;
}
```

5.73 CMD_SNAPSHOT

This command instructs the coprocessor engine to capture a snapshot of the current screen and write the result into **RAM_G** as an **ARGB4** bitmap. The bitmap size matches screen dimensions, as defined by the **REG_HSIZE** and **REG_VSIZE** registers.

To avoid display glitches, the display should be disabled during the snapshot by setting **REG_PCLK** to 0. The destination address in **RAM_G** must not be used or referenced by the graphics engine while the coprocessor writes the bitmap.

C prototype

```
void cmd_snapshot( uint32_t ptr );
```

Parameters

ptr

Destination address in **RAM_G** where the snapshot will be stored.

Command layout

+0	CMD_SNAPSHOT(0xFFFF FF1F)
+4	Ptr

Examples

To capture a snapshot of the current 160 x 120 screen and use it as a bitmap in the new display list:

```
wr(REG_PCLK,0);//Turn off the PCLK
wr16(REG_HSIZE,120);
wr16(REG_WSIZE,160);

cmd_snapshot(0);//Taking snapshot.

wr(REG_PCLK,5);//Turn on the PCLK
wr16(REG_HSIZE,272);
wr16(REG_WSIZE,480);

cmd_dlstart();
cmd(CLEAR(1,1,1));
cmd(BITMAP_SOURCE(0));
cmd(BITMAP_LAYOUT(ARGB4, 2 * 160, 120));
cmd(BITMAP_SIZE(NEAREST, BORDER, BORDER, 160, 120));
cmd(BEGIN(BITMAPS));
cmd(VERTEX2II(10, 10, 0, 0));
```

5.74 CMD_SNAPSHOT2

The snapshot command makes the coprocessor capture a portion of the current screen and store it in graphics memory as a bitmap. The size, position and format of the bitmap may be specified. During the snapshot process, the display output is suspended. On LCD screens, timing variations are tolerated, so the snapshot typically causes no noticeable flicker.

C prototype

```
void cmd_snapshot2( uint32_t fmt,
                   uint32_t ptr,
                   int16_t x,
                   int16_t y,
                   uint16_t w,
                   uint16_t h);
```

Parameters

fmt

Format of the output bitmap, which can be **RGB565**, **ARGB4** or **0x20**. Using **0x20** generates the snapshot in **ARGB8** format.

Refer to [BITMAP_LAYOUT](#) for format List.

ptr

Destination address in **RAM_G** where the snapshot bitmap will be stored.

x

x-coordinate of the top-left corner of the snapshot area, in pixels

y

y-coordinate of the top-left corner of the snapshot area , in pixels

w

width of the snapshot area, in pixels. **Note:** if the *format (fmt)* is **0x20**, i.e., in **ARGB8** format, the width value must be doubled.

h

height of the snapshot area, in pixels.

Command layout

+0	CMD_SNAPSHOT2(0xFFFF FF37)	
+4	Fmt	
+8	Ptr	
+12	X	
+14	Y	
+16	W	
+18	H	

Examples

To take a 32 x 32 snapshot of the top-left corner of the screen and use it as a bitmap in the new display list:

```
cmd_snapshot2( RGB565, 0, 0, 0, 32, 32 );
cmd_dlstart();
cmd_setbitmap(0, RGB565, 32, 32);
cmd(CLEAR(1,1,1));
cmd(BEGIN(BITMAPS));
cmd(VERTEX2II(10, 10, 0, 0));
```

Note: For ARGB8 format, pixel memory layout is as below:

ARGB8 Pixel Format			
31	24	23	16
15	8	7	0
Alpha Channel	Red Channel	Green Channel	Blue Channel

5.75 CMD_SETBITMAP

This command generates the corresponding display list commands for a given bitmap, saving the effort of writing the display list manually. The display list commands that can be generated include:

- **BITMAP_SOURCE**
- **BITMAP_LAYOUT/ BITMAP_LAYOUT_H**
- **BITMAP_SIZE/ BITMAP_SIZE_H**
 In the generated display list commands, the filter/wrapx/wrapy parameters in **BITMAP_SIZE** are always set to **NEAREST/BORDER/BORDER**.
- **BITMAP_EXT_FORMAT**

C prototype

```
void cmd_setbitmap( uint32_t source,
                  uint16_t fmt,
                  uint16_t width,
                  uint16_t height );
```

Parameters

source

Source address of the bitmap, located in **RAM_G** or flash memory, specified as a **BITMAP_SOURCE** parameter. When stored in flash memory, the address must be given in block units (each block is 32 bytes).

fmt

Bitmap format. See the definition in **BITMAP_EXT_FORMAT**.

width

bitmap width, in pixels. 2-byte value.

height

bitmap height, in pixels. 2-byte value.

Command layout

+0	CMD_SETBITMAP(0xFFFF FF43)
+4	Source
+8	Fmt
+10	Width
+12	Height

Examples

Display an ASTC image with a width of 35 pixels and a height of 35 pixels, stored in flash memory at address 6016 (188 x 32):

```
cmd_dlstart();
cmd_setbitmap(0x800000 | 188, COMPRESSED_RGBA_ASTC_5x5_KHR, 35, 35);
cmd(CLEAR(1,1,1));
cmd(BEGIN(BITMAPS));
cmd(VERTEX2II(10, 10, 0, 0));
cmd_swap();
```

Note:

- Two bytes are required to be appended after the last parameter to ensure 4-byte alignment.
- When the format is **PALETTED4444**, **PALETTED8**, or **PALETTED565**, no **PALETTE_SOURCE** display list command is generated. In these cases, the user must manually issue the **PALETTE_SOURCE** command.

```
cmd_dlstart();
cmd(BITMAPS(2)); //Use bitmap handle 2
cmd_setbitmap(RAM_G, PALETTED565, 35, 35); //Set up parameters for bitmap handle 2
cmd(PALETTE_SOURCE(RAM_G + 35*35)); //Specify the address of palette(color table),
//assuming it comes right after the index
cmd(CLEAR(1,1,1));
cmd(BEGIN(BITMAPS));
cmd(VERTEX2II(10, 10, 2, 0)); //draw image with bitmap handle 2
cmd_swap();
```

5.76 CMD_LOGO



The LOGO command instructs the coprocessor engine to play back a short Bridgetek logo animation. During logo playback, the MCU must not write or render any display list. After approximately 2.5 seconds, the coprocessor engine writes zero to **REG_CMD_READ** and **REG_CMD_WRITE**, then enters the command wait state. Once the command completes, the MCU should write the next command to the starting address of **RAM_CMD**.

C prototype

```
void cmd_logo( );
```

Command layout

+0	CMD_LOGO(0xFFFF FF31)
----	------------------------------

Examples

To play back the logo animation:

```
cmd_logo();
delay(3000); // Optional to wait
//Wait till both read & write pointer register are equal.
While(rd16(REG_CMD_WRITE) != rd16(REG_CMD_READ));
```

5.77 CMD_FLASHERASE

This command erases the attached flash storage.

C prototype

```
void cmd_flasherace( );
```

Command layout

+0	CMD_FLASHERASE(0xFFFF FF44)
----	------------------------------------

5.78 CMD_FLASHWRITE

This command writes the following inline data to the flash storage. The flash must have been previously erased using **CMD_FLASHERASE**.

C prototype

```
void cmd_flashwrite( uint32_t ptr,
                    uint32_t num );
```

Parameters

ptr

Destination address in flash memory. The address must be 256-byte aligned, with the first block starting at address **zero**.

num

Number of bytes to write. The value must be a multiple of 256.

Command layout

+0	CMD_FLASHEWRTITE (0xFFFF FF45)	
+4	Ptr	
+8	Num	
+12	bytes ₁	
	...	
	byte _n	

5.79 CMD_FLASHPROGRAM

This command writes data to blank flash. It assumes the flash is previously programmed to all-ones, the default state from the manufacturer. Before using this command, the flash should be set to Full Speed mode via CMD_FLASHFAST.

C prototype

```
void cmd_flashprogram( uint32_t dest,
                      uint32_t src,
                      uint32_t num );
```

Parameters

dst

destination address in flash memory. The address must be 4096-byte aligned, with the first block starting at address **zero**.

src

source data in main memory. The address must be 4-byte aligned.

num

number of bytes to write. The value must be a multiple of 4096.

Command layout

+0	CMD_FLASHEPROGRAM (0xFFFF FF70)
+4	Dst
+8	Src
+12	Num

5.80 CMD_FLASHREAD

This command reads data from flash memory into main memory.

C prototype

```
void cmd_flashread ( uint32_t dest,
                    uint32_t src,
                    uint32_t num );
```

Parameters

dest

Destination address in **RAM_G**. The address must be 4-byte aligned, with the first block starting at address 0.

src

source address in flash memory. The address must be 64-byte aligned.

num

number of bytes to read. The value must be a multiple of 4.

Command layout

+0	CMD_FLASHEREAD(0xFFFF FF46)
+4	Dest
+8	Src
+12	Num

Examples

```
// Read all main RAM (1M bytes) from flash:
cmd_flashread(0, 4096, 1048576);
```

5.81 CMD_APPENDF

This command appends data from flash memory to the current display list at the address specified by **REG_CMD_DL**.

C prototype

```
void cmd_appendf( uint32_t ptr,
                 uint32_t num );
```

Parameters

ptr

Start address of the source commands in flash memory. The address must be 64-byte aligned, with the first block starting at address **zero**.

num
 number of bytes to copy. The value must be a multiple of 4.

Command layout

+0	CMD_APPENDF (0xFFFF FF59)
+4	Ptr
+8	Num

Description

After appending, the coprocessor engine increments **REG_CMD_DL** by the specified num to keep the display list in order. Note that any appended data still counts toward the 8 KB maximum of **RAM_DL**.

5.82 CMD_FLASHUPDATE

This command writes the specified data to flash memory. If the data already matches the existing flash contents, no action is taken. Otherwise, the flash is erased in 4K units and the new data is written.

C prototype

```
void cmd_flashupdate( uint32_t dest,
                     uint32_t src,
                     uint32_t num );
```

Parameters

dest
 Destination address in flash memory. The address must be 4096-byte aligned, with the first block starting at address **zero**.

src
 source address in **RAM_G** (main memory). The address must be 4-byte aligned.

num
 number of bytes to write. The values must be a multiple of 4096.

Command layout

+0	CMD_FLASHUPDATE (0xFFFF FF47)
+4	Dest
+8	Src
+12	Num

Example

```
// The pseudo code below shows how to program the blob file to first block of flash
// Assume the flash is in detach mode and now attach it
cmd_flashattach();

// Now check if the flash is in basic mode after attaching
while (FLASH_STATUS_BASIC != rd8(REG_FLASH_STATUS));

//Write the BLOB file into the first block of flash
//Assume the BLOB file is in RAM_G
cmd_flashupdate(0, RAM_G, 4096);

// To check if the blob is valid, try to switch to full mode
cmd_flashfast(0);

while (FLASH STATUS FULL != rd8(REG FLASH STATUS));
```

5.83 CMD_FLASHDETACH

This command puts the SPI device lines into a high-impedance (hi-z) state. When detached, the only valid flash operations are the low-level SPI access commands listed below:

- **CMD_FLASHSPIDSEL**
- **CMD_FLASHSPITX**
- **CMD_FLASHSPIRX**
- **CMD_FLASHATTACH**

Refer to the section - Flash interface in BT817/8 datasheet.

C prototype

```
void cmd_flashdetach( );
```

Command layout

+0	CMD_FLASHDETACH (0xFFFF FF48)
----	--------------------------------------

5.84 CMD_FLASHATTACH

This command reconnects **EVE** to the attached SPI flash storage. After the execution of the command, **REG_FLASH_STATUS** should read **FLASH_STATUS_BASIC**. Refer to the section - Flash interface in BT817/8 datasheet.

C prototype

```
void cmd_flashattach( );
```

Command layout

+0	CMD_FLASHATTACH (0xFFFF FF49)
----	--------------------------------------

5.85 CMD_FLASHFAST

This command sets the BT81X chip to drive the attached flash in full-speed mode, if supported. Refer to the section - Flash interface in BT817/8 datasheet.

C prototype

```
void cmd_flashfast( uint32_t result );
```

Parameters

result

The result code is written after execution. A value of indicates that the command succeeded. Otherwise, an error code is set as follows:

Error Code	Meaning
0xE001	flash is not supported
0xE002	no header detected in sector 0 – is flash blank?
0xE003	sector 0 data failed integrity check
0xE004	device/blob mismatch – was correct blob loaded?
0xE005	failed full-speed test – check board wiring

Command layout

+0	CMD_FLASHFAST (0xFFFF FF4A)
+4	result

Note: To access any data in flash by EVE, the host must send this command at least once to put EVE in full-speed flash mode. In addition, the flash chip must have the correct blob file programmed in its first block (4096 bytes); Otherwise, the command will fail.

5.86 CMD_FLASHSPIDESEL

This command de-asserts the SPI CS signal. It is valid only when the flash has been detached using **CMD_FLASHDETACH**.

C prototype

```
void cmd_flashspidesel();
```

Command layout

+0	CMD_FLASHSPIDESEL (0xFFFFFFF4B)
----	--

5.87 CMD_FLASHSPITX

This command transmits the specified bytes over the flash SPI interface. It is only valid when the flash has been detached using **CMD_FLASHDETACH**.

C prototype

```
void cmd_flashspitx( uint32_t num );
```

Parameters

num
number of bytes to transmit

Command layout

+0	CMD_FLASHSPITX (0xFFFF FF4C)	
+4	num	
+8	byte ₁	
	...	
	byte _n	

Example

Transmit single-byte 06:

```
cmd_flashdetach();
cmd_flashspidesel();
cmd_flashspitx(1);
cmd(0x00000006);
```

5.88 CMD_FLASHSPIRX

This command reads bytes from the flash SPI interface and writes them to main memory. It is valid only when the flash has been detached using **CMD_FLASHDETACH**.

C prototype

```
void cmd_flashspirx( uint32_t ptr,
                    uint32_t num );
```

Parameters

ptr
destination address in **RAM_G**

num
number of bytes to receive

Command layout

+0	CMD_FLASHSPIRX (0xFFFF FF4D)
+4	Ptr
+4	Num

Example

Read 3 bytes from the SPI flash to main memory locations 100,101,102:

```
cmd_flashdetach();
cmd_flashspidesel();
cmd_flashspirx(100, 3);
```

5.89 CMD_CLEARCACHE

This command clears the graphics engine's internal flash cache. It should be used after modifying flash graphics data with **CMD_FLASHUPDATE** or **CMD_FLASHWRITE**; otherwise, bitmaps from flash may display "stale" data. The command must be executed when the display list is empty, immediately after a **CMD_DLSTART** command. If executed at the wrong time, it triggers a coprocessor fault ("display list must be empty") and sets **REG_PCLK** to zero.

C prototype

```
void cmd_clearcache( );
```

Command layout

+0	CMD_CLEARCACHE (0xFFFF FF4F)
-----------	-------------------------------------

Example

```
//Flash is in Full mode and has the right content working with EVE
//Update the 4th block of flash chip with new bitmap data located at RAM_G+1024
cmd_flashupdate(4*4096, RAM_G+1024, 4*4096);

//To continue rendering the bitmap data in flash, need to call cmd_clearcache
cmd_dlstart();
cmd_clearcache();
cmd_swap();
```

5.90 CMD_FLASHSOURCE

This command sets the source address in flash for data loaded by **CMD_LOADIMAGE**, **CMD_PLAYVIDEO**, **CMD_VIDEOSTARTF**, and **CMD_INFLATE2** commands when using the **OPT_FLASH** option.

C prototype

```
void cmd_flashsource( uint32_t ptr );
```

Parameters

ptr

flash address. The address must be 64-byte aligned, with the first block starting at address zero.

Command layout

+0	CMD_FLASHSOURCE (0xFFFF FF4E)
+4	ptr

5.91 CMD_VIDEOSTARTF

This command is used to initialize the video frame decoder. The video data must reside in flash memory, with its address previously set using **CMD_FLASHSOURCE**. The command processes the video header information and completes once the header has been fully consumed.

C prototype

```
void cmd_videostartf( );
```

Command layout

+0	CMD_VIDEOSTARTF (0xFFFF FF5F)
----	--------------------------------------

Example

```
cmd_flashsource( LOGO_VIDEO_FLASH_ADDRESS );
cmd_videostartf( );
cmd_videoframe( 4, 0 );
```

5.92 CMD_ANIMSTART

This command starts an animation. If the channel is already in use, the previous animation is replaced.

C prototype

```
void cmd_animstart( int32_t ch,
                   uint32_t aoptr,
                   uint32_t loop );
```

Parameters

ch

Animation channel number 0-31. If no channel is available, an "out of channels" exception is raised.

aoptr

Address of the animation object is in flash memory.

loop

Loop flags. See [Animation support](#).

Command layout

+0	CMD_ANIMSTART (0xFFFF FF53)
+4	Ch
+8	Aoptr
+12	Loop

Example

See [CMD_ANIMFRAME](#).

5.93 CMD_ANIMSTARTRAM

This command starts an animation stored in **RAM_G**. If the channel is already in use, the previous animation is replaced.

C prototype

```
void cmd_animstartram( int32_t ch,
                      uint32_t aoptr,
                      uint32_t loop );
```

Parameters

ch

Animation channel number 0-31. If no channel is available, an "out of channels" exception is raised.

aoptr

Pointer to the animation object in RAM_G. The address must be 64-byte aligned.

loop

Loop flags. See [Animation support](#).

Command layout

+0	CMD_ANIMSTARTRAM(0xFFFF FF6E)
+4	Ch
+8	Aoptr
+12	Loop

Example

See [CMD_ANIMFRAMERAM](#).

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.94 CMD_RUNANIM

This command is used to play or run animations until completion. Playback stops when either the specified animation finishes, or the host **MCU** writes to a control byte. Note that only animations started with **ANIM_ONCE** will complete. Pseudocode for **CMD_RUNANIM** is:

```
do {
  if ((play != -1) && (*play != 0))
    break;
  CMD_DLSTART();
  Clear(1,1,1);
  CMD_ANIMDRAW(-1);
  CMD_SWAP();
} while ((waitmask & REG_ANIM_ACTIVE) == 0);
```

C prototype

```
void cmd_runanim( uint32_t waitmask,
                 uint32_t play );
```

Parameters

waitmask

32-bit mask specifying which animation channels to wait for. Animation completes when the logical **AND** of this mask and **REG_ANIM_ACTIVE** becomes zero.

play

Address of the play control byte. Animation stops when the byte at this address is non-zero. If this feature is not required, use the special value -1 (0xFFFF FFFF) to indicate that no control byte used.

Command layout

+0	CMD_RUNANIM(0xFFFF FF6F)
+4	Waitmask
+8	Play

Example

```
/** play back several animations simultaneously assume the animation is in flash**/
/* set up a channel for first animation */
cmd_animstart(1,4096, ANIM_ONCE);
cmd_animxy(100, 240); //The center of animation

/* set up another channel for second animation */
cmd_animstart(2,4096 + 10*1024, ANIM_ONCE);
cmd_animxy(200, 240); //The center of animation
/* set up another channel for third animation */
cmd_animstart(2,4096 + 10*1024, ANIM_ONCE);
cmd_animxy(300, 240); //The center of animation

/* play back both animations and set up the control byte at 0xF0000 of RAM_G */
wr32(0xF0000, 1);
cmd_runanim(-1, 0xF0000); //The animation will be shown on display.

//.....

/* To stop the animation before it ends, write the control byte to zero */
wr32(0xF0000, 0);
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.95 CMD_ANIMSTOP

This command stops one or more active animations.

C prototype

```
void cmd_animstop( int32_t ch );
```

Parameters

ch

Animation channel number, 0-31. If ch is =-1, all active animations are stopped.

Command layout

+0	CMD_ANIMSTOP (0xFFFF FF54)
+4	ch

5.96 CMD_ANIMXY

This command sets the coordinates of an animation.

C prototype

```
void cmd_animxy( int32_t ch,  
                int16_t x,  
                int16_t y );
```

Parameters

ch

Animation channel number, 0-31.

x

x-coordinate of the animation center on the screen, in pixels.

y

y-coordinate of the animation center on the screen, in pixels.

Command layout

+0	CMD_ANIMXY (0xFFFF FF55)
+4	Ch
+8	x
+10	y

NOTE: If the pixel precision is not set to 1/16 in the current graphics context, a **VERTEX_FORMAT(4)** command must be issued before this command.

5.97 CMD_ANIMDRAW

This command draws one or more active animations.

C prototype

```
void cmd_animdraw( int32_t ch );
```

Parameters

ch

Animation channel number, 0-31. If ch = -1, all undrawn animations are drawn in ascending order.

Command layout

+0	CMD_ANIMDRAW(0xFFFF FF56)
+4	Ch

5.98 CMD_ANIMFRAME

This command draws a specified frame of an animation.

C prototype

```
void cmd_animframe( int16_t x,
                   int16_t y,
                   uint32_t aoptr,
                   uint32_t frame );
```

Parameters

x

x-coordinate of the animation center on the screen, in pixels.

y

y-coordinate of the animation center on the screen, in pixels.

aoptr

Address of the animation object is in flash memory.

frame

Frame number to draw, starting from zero.

Command layout

+0	CMD_ANIMFRAME (0xFFFF FF5A)	
+4	X	
+6	Y	
+8	Aoptr	
+12	Frame	

NOTE: If the pixel precision in the current graphics context is not set to 1/16, a **VERTEX_FORMAT(4)** command must be issued before this command.

Example

```
//Draw a frame located at the first available address of flash onto (0,400).
cmd_animframe(0, 400, 4096, 65);
```

5.99 CMD_ANIMFRAMERAM

This command draws a specified frame of an animation stored in **RAM_G**.

C prototype

```
void cmd_animframeram( int16_t x,
                      int16_t y,
                      uint32_t aoptr,
                      uint32_t frame );
```

Parameters

x
x-coordinate of the animation center on the screen, in pixels.

y
y-coordinate of the animation center on the screen, in pixels.

aoptr
The address of the animation object in **RAM_G**. Must be 64-byte aligned.

frame
Frame number to draw, starting from zero.

Command layout

+0	CMD_ANIMFRAMERAM (0xFFFF FF6D)	
+4	x	
+6	y	
+8	Aoptr	
+12	Frame	

NOTE: If the pixel precision in the current graphics context is not set to 1/16, a **VERTEX_FORMAT(4)** command must be issued before this command.

Example

```
//Draw the 65th frame of the animation onto (400,240). The animation object is in RAM_G+4096
cmd_animframeram(400, 240, 4096, 65);
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.100 CMD_SYNC

This command waits for the end of the video scan-out period and then returns immediately. It can be used to synchronize screen updates that are not part of a display list and to accurately measure frame rendering time.

C prototype

```
void cmd_sync( );
```

Command layout

+0	CMD_SYNC(0xFFFF FF42)
-----------	------------------------------

Examples

```
//To synchronize with a frame:
cmd_sync();

//To update REG_MACRO_0 at the end of scan out, to avoid tearing:
cmd_sync();
cmd_memwrite(REG_MACRO_0, 4);
cmd(value);

//To measure frame duration
cmd_sync();
cmd_memcpy(0, REG_CLOCK, 4);
cmd_sync();
cmd_memcpy(4, REG_CLOCK, 4);
//the difference between locations 4 and 0 give the frame interval in clocks.
```

5.101 CMD_BITMAP_TRANSFORM

This command computes a bitmap transform and appends commands BITMAP_TRANSFORM_A through BITMAP_TRANSFORM_F to the display list. It calculates the transformation using three corresponding points in screen space and bitmap space. From these points, the command computes a matrix that maps bitmap coordinates to screen coordinates and appends the necessary display list commands to set the bitmap matrix.

C prototype

```
void cmd_bitmap_transform( int32_t x0,
                          int32_t y0,
                          int32_t x1,
                          int32_t y1,
                          int32_t x2,
                          int32_t y2,
                          int32_t tx0,
                          int32_t ty0,
                          int32_t tx1,
                          int32_t ty1,
                          int32_t tx2,
                          int32_t ty2,
                          uint16_t result );
```

Parameters

X₀,Y₀

Screen coordinates of Point 0, in pixels

X₁,Y₁

Screen coordinates of Point 1, in pixels

X₂,Y₂

Screen coordinates of Point 2, in pixels

tx₀,ty₀

Bitmap coordinates of Point 0, in pixels

tx₁,ty₁

Bitmap coordinates of Point 1, in pixels

tx₂,ty₂

Bitmap coordinates of Point 2, in pixels

result

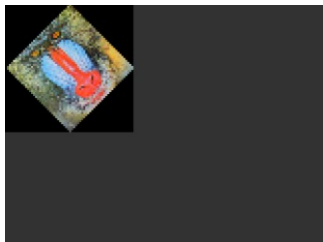
Result return value: -1 indicates success, while 0 means it was not possible to compute a solution matrix.

Command layout

+0	CMD_BITMAP_TRANSFORM(0xFFFF FF21)
+4	x0
+8	y0
+10	x1
+16	y1
+20	x2
+24	y2
+28	tx0
+32	ty0
+36	tx1
+40	ty1
+44	tx2
+48	ty2
+52	result

Examples

Transform a 64x64 bitmap:

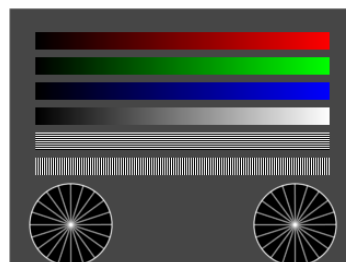


```
cmd(BLEND_FUNC(ONE, ZERO));
cmd_bitmap_transform(32,0, 64,32, 32,64,0,0, 0,64,
64,64, 0);
cmd(BEGIN(BITMAPS));
cmd(VERTEX2II(0, 0, 0, 0));
```

5.102 CMD_TESTCARD

The **TESTCARD** command loads a display list containing a testcard graphic and executes **CMD_SWAP** to display it. The graphic is automatically scaled for the current display size, based on **REG_HSIZE**, **REG_VSIZE**, and **REG_ROTATE**. Features of the testcard are:

- White border at the edges to verify screen boundaries and clock stability
- Red, green, blue and white gradients to check color bit depth
- Horizontal and vertical checker patterns to verify signal integrity
- Circle graphics to confirm aspect ratio
- Radial line pattern to test antialiasing performance



C prototype

```
void cmd_testcard( );
```

Command layout

+0	CMD_TESTCARD(0xFFFF FF61)
----	----------------------------------

Examples

```
//To display a test card, call the command:
cmd_testcard();
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.103 CMD_WAIT

This command waits for a specified number of microseconds. Delays longer than 1 second (1000000 μ s) are not supported.

C prototype

```
void cmd_wait( uint32_t us );
```

Parameters

us
 Delay time, in microseconds (μ s)

Command layout

+0	CMD_WAIT(0xFFFF FF65)
+4	Us

Examples

```
//To delay for 16.7 ms:
cmd_wait(16700);
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.104 CMD_NEWLIST

This command starts compiling a command list into **RAM_G**. Instead of being executed immediately, the following commands are appended to the list until a **CMD_ENDLIST** is encountered. The compiled list can later be executed using **CMD_CALLIST**. The following commands are not supported in command lists and their behavior is undefined:

- **CMD_FLASHSPITX**
- **CMD_FLASHWRITE**
- **CMD_INFLATE**
- **CMD_NEWLIST**

The following commands are supported only when using the **OPT_MEDIAFIFO** option:

- **CMD_INFLATE2**
- **CMD_LOADIMAGE**
- **CMD_PLAYVIDEO**

C prototype

```
void cmd_newlist( uint32_t a );
```

Parameters

a
memory address of the start of the command list.

Command layout

+0	CMD_NEWLIST(0xFFFF FF68)
+4	A

Examples

```
/**
 * Create a command list at RAM_G address 0xF0000 by
 * sending the following commands to command buffer
 */
cmd_newlist(RAM_G + 0xF0000);
cmd(COLOR_RGB(255, 100, 0));
cmd_button(20, 20, 60, 60, 30, 0, "OK!");
cmd_endlist();

//.....

/**
 * Invoke the command list
 */
cmd_dlstart();
cmd(COLOR_RGB(255, 255, 255));
cmd(CLEAR(1,1,1));
cmd_calllist(RAM_G + 0xF0000);
cmd(DISPLAY());
cmd_swap();
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.105 CMD_ENDLIST

This command terminates the compilation of a command list into **RAM_G**. Afterward, **CMD_GETPTR** can be used to find the first unused memory address following the command list.

C prototype

```
void cmd_endlist( );
```

Command layout

+0	CMD_ENDLIST(0xFFFF FF69)
----	---------------------------------

Examples

See [CMD_NEWLIST](#).

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.106 CMD_CALLLIST

This command calls a command list. When executed, all commands compiled between **CMD_NEWLIST** and **CMD_ENDLIST** are run as if they were executed at the point of **CMD_CALLLIST**. A command list may also contain nested **CMD_CALLLIST** commands, up to a depth of 4 levels.

C prototype

```
void cmd_calllist( uint32_t a );
```

Parameters

a
memory address of the command list, in **RAM_G**

Command layout

+0	CMD_CALLLIST(0xFFFF FF67)
+4	A

Examples

See [CMD_NEWLIST](#).

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.107 CMD_RETURN

This command ends a command list. Normally it is not required, because **CMD_ENDLIST** appends **CMD_RETURN** to the list. However, it can be used when constructing command lists offline.

C prototype

```
void cmd_return( );
```

Command layout

+0	CMD_RETURN(0xFFFF FF66)
-----------	--------------------------------

Examples

```
/**
Construct a command list in RAM_G to show a button
***/
wr32(RAM_G + 0 * 4, SAVE_CONTEXT());
wr32(RAM_G + 1 * 4, COLOR_RGB(125, 125, 128));
wr32(RAM_G + 2 * 4, CMD_BUTTON);

wr16(RAM_G + 3 * 4, 160); //x coordinate of button
wr16(RAM_G + 3 * 4 + 2, 160); //y coordinate of button
wr16(RAM_G + 4 * 4, 123); //w
wr16(RAM_G + 4 * 4 + 2, 234); //h
wr16(RAM_G + 5 * 4, 31); //Font handle
wr16(RAM_G + 5 * 4 + 2, 0); //option parameter of cmd_button
wr8(RAM_G + 6 * 4, 'T');
wr8(RAM_G + 6 * 4 + 1, 'E');
wr8(RAM_G + 6 * 4 + 2, 'S');
wr8(RAM_G + 6 * 4 + 3, 'T');

wr8(RAM_G + 7 * 4, '\0'); //the null terminators for string "TEST"

//Append extra 3 bytes for alignment purpose
wr8(RAM_G + 7 * 4 + 1, '\0');
wr8(RAM_G + 7 * 4 + 2, '\0');
wr8(RAM_G + 7 * 4 + 3, '\0');

wr32(phost, RAM_G + 8 * 4, RESTORE_CONTEXT());
wr32(phost, RAM_G + 9 * 4, CMD_RETURN); //Indicate the end of command list

//.....
```

```
/**
Invoke the command list in RAM_G to render the button
***/
cmd_dlistart();
cmd(COLOR_RGB(255, 255, 255));
cmd(CLEAR(1,1,1));
cmd_calllist(RAM_G);
cmd(DISPLAY());
cmd_swap();
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.108 CMD_FONTCACHE

This command enables the font cache, which loads all the bitmaps (glyph) from a flash-based font into a **RAM_G** area. This eliminates the need to render bitmaps from flash, at the cost of using some **RAM_G**. The cache area must be large enough to hold all the bitmaps used in **two** consecutive frames. This feature applies only to **ASTC**-based custom fonts.

C prototype

```
void cmd_fontcache( uint32_t font,
                   uint32_t ptr,
                   uint32_t num );
```

Parameters

font

font handle to be cached. The font must be in extended format.

ptr

Start address of the cache area in **RAM_G**. The address must be 64-byte aligned.

num

Size of the cache area in bytes. The value must be 4 byte-aligned and at least 16 Kbytes.

Command layout

+0	CMD_FONTCACHE(0xFFFF FF6B)
+4	Font
+8	Ptr
+12	Num

Examples

To cache font 13 using a 64 Kbyte font cache located at the top of memory:

```
cmd_fontache(13, 0xf0000, 0x10000);
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.109 CMD_FONTCACHEQUERY

This command queries the capacity and utilization of the font cache.

C prototype

```
void cmd_fontcachequery( uint32_t total,
                        uint32_t used );
```

Parameters

total

Output parameter; Total number of bytes available in the font cache, in bytes.

used

Output parameter; Number of bitmaps used in the font cache, in bytes

Command layout

+0	CMD_FONTCACHEQUERY(0xFFFF FF6C)
+4	Total
+8	Used

Examples

```
uint32_t total, used;
uint16_t ram_fifo_offset = rd16(REG_CMD_WRITE);
cmd_fontcachequery(total, used);

total = rd32(RAM_CMD + (ram_fifo_offset + 4) % 4096);
used = rd32(RAM_CMD + (ram_fifo_offset + 8) % 4096);

printf("Font cache usage: %d / %d", used, total);
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.110 CMD_GETIMAGE

This command returns all attributes of the bitmap created by the previous CMD_LOADIMAGE, CMD_PLAYVIDEO, CMD_VIDEOSTART or CMD_VIDEOSTARTF command.

C prototype

```
void cmd_getimage( uint32_t source,
                  uint32_t fmt,
                  uint32_t w,
                  uint32_t h,
                  uint32_t palette );
```

Parameters

source

Output parameter; source address of bitmap.

fmt

Output parameter; format of the bitmap

w

Width of the bitmap, in pixels

h

Height of the bitmap, in pixels

palette

Palette data of the bitmap if format is PALETTED565 or PALETTED4444. Otherwise, zero.

Command layout

+0	CMD_GETIMAGE (0xFFFF FF64)
+4	Source
+8	Fmt
+12	W
+16	h
+20	palette

Examples

```

//To find the base address and dimensions of the previously loaded image
uint32_t source, fmt, w, h, palette;
uint16_t cmd_fifo_offset = rd16(REG_CMD_WRITE);

cmd_getimage(src, fmt, w, h, palette);

src = rd32(RAM_CMD + (cmd_fifo_offset) + 4 % 4096);
fmt = rd32(RAM_CMD + (cmd_fifo_offset) + 8 % 4096);
w  = rd32(RAM_CMD + (cmd_fifo_offset) + 12 % 4096);
h  = rd32(RAM_CMD + (cmd_fifo_offset) + 16 % 4096);
palette = rd32(RAM_CMD + (cmd_fifo_offset) + 20 % 4096);

cmd_setbitmap(src, fmt, w, h);
if (palette != 0) PaletteSource(palette);

```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.111 CMD_HSF

This command sets the horizontal scale factor for the graphics engine.

C prototype

```
void cmd_hsf( uint32_t w );
```

Parameters

w

Output pixel width. The value must be less than REG_HSIZE. Setting it to 0 disables HSF.

Command layout

+0	CMD_HSF (0xFFFF FF62)
+4	W

Examples

A common panel format is 800×480. This gives a logical aspect ratio of

$$800/480 = 1.6667$$

However, the physical panel size is 153.84 × 85.63mm, giving a physical aspect ratio of 1.796. This difference means the panel has non-square pixels. To maintain square pixels in rendering, we can compute the logical width of the panel while keeping the height constant:

$$480 \times (153.84/85.63) = 862.3$$

By rendering graphics at 862×480 and then resizing to 800×480, all drawing can assume square pixels. To configure this panel, set REG_HSIZE to 862, then issue this command:

```
cmd_hsf(800);
```

To disable the HSF, set the output width to 0 by issuing:

```
cmd_hsf(0);
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

5.112 CMD_PCLKFREQ

This command sets **REG_PCLK_FREQ** to generate the closest possible pixel clock frequency to the requested value. If no suitable frequency can be found, the result field is zero and **REG_PCLK_FREQ** remains unchanged.

C prototype

```
void cmd_pclkfreq( uint32_t ftarget,
                  int32_t rounding,
                  uint32_t factual );
```

Parameters

ftarget

Target pixel clock frequency, in Hertz (Hz).

rounding

Approximation mode. Valid values are:

- 0 is nearest,
- 1 is highest frequency less than or equal to target,
- 1 is lowest frequency greater than or equal to target.

factual

Output parameter; Actual frequency achieved. If no suitable frequency was found, the value is zero.

Command layout

+0	CMD_PCLKFREQ (0xFFFF FF6A)
+4	ftarget
+8	rounding
+12	factual

Important Note:

When using this command, the flash **BLOB** is required to ensure that the calculated PLL2 settings remain within the 228MHz specification. Therefore, before using this command, ensure that the following steps have been taken:

- An external flash chip is connected to the BT817/8
- The external flash chip has the **BLOB** installed in the first 4096 bytes starting at address 0
- The external flash chip is set to full-speed mode

Alternatively, REG_PCLK_FREQ can be set directly by referring to the *RGB PCLK Frequency table in EXTSYNC mode* (**Parallel RGB Interface** section of the BT817/8 datasheet which has recommended values). This method can be used instead of CMD_PCLKFREQ and is recommended when no flash is installed.

Examples

```
//Assume the first 4096 bytes of flash chip is installed with Blob
//Switch the state of flash to full-speed mode
cmd_flashfast(0);

//.....

//To set the output PCLK as close to 9 MHz as possible:
cmd_pclkfreq(9000000, 0, 0);
```

Note: This is a BT817/8 specific command, available only on the BT817 and BT818 series chips.

6 ASTC

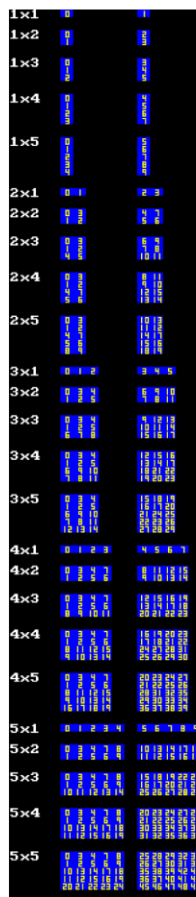
ASTC (stands for **A**daptive **S**calable **T**exture **C**ompression) is an open standard developed by **ARM**® for mobile GPUs. **ASTC** is a block-based lossy compression format, where the compressed image is divided into blocks of uniform size, allowing rapid identification of which block contains a given texel.

Each block has a fixed 128-bit memory footprint, but can represent varying numbers of texels (the block footprint). Block footprint are not restricted to powers-of-two and need not be square. For 2D formats, block dimensions range from 4x4 to 12x12 texels.

Using ASTC for large ROM fonts provide significant memory savings. Encoding the four largest fonts in ASTC format results in no noticeable quality loss while reducing ROM usage from 1M Byte to approximately 640K Kbytes.

The **BT81X** series enables advanced animation features and **Unicode** support using the **ASTC** format. With ASTC format, BT81X Series can display images directly from flash memory without consuming valuable **RAM_G** space. By storing enough ASTC images in flash memory or **RAM_G**, users can create image-rich **GUI** applications efficiently.

6.1 ASTC RAM Layout



ASTC blocks represent 4x4 to 12x12 pixels, with each block occupying 16 bytes in memory. For details, refer to Table 13 – BITMAP_LAYOUT Format List. In a nutshell, 4x4 blocks offer the lowest compression rate but the best image quality; while 12x12 blocks provide the highest compression rate but the lowest quality. Users should evaluate the image quality of different **ASTC block sizes** on hardware to achieve the trade-off.

ASTC bitmaps in main memory must be 16-byte aligned. However, for a multi-cell bitmap used with the **CELL** command, the source address of each bitmap cell must start on a multiple of 4 blocks, i.e., 64-byte aligned.

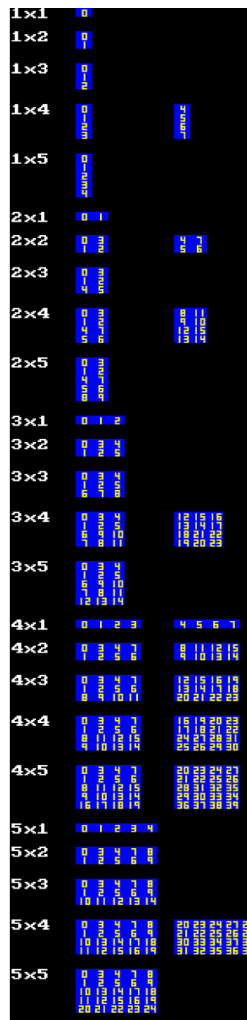
The mapping from bitmap coordinates to memory locations is not always linear. Instead, blocks are grouped into 2x2 tiles. Within each tile, the block order is:

0	3
1	2

When a line contains an odd number of blocks, the final two blocks are packed into a 1x2 configuration. When there is an odd number of rows, the last row of blocks is stored linearly.

The diagram above illustrates the same memory region loaded with ASTC blocks in ascending memory addresses. The first column corresponds to cell 0, and the second column corresponds to cell 1.

6.2 ASTC Flash Layout



ASTC bitmaps in flash must be 64-byte aligned. This means that multi-celled bitmaps must have a size that is a multiple of 4 blocks. Similarly, fonts stored in flash must use a multiple of four blocks per character. Note that only bitmaps with a size that is multiple of four blocks will have cell 1 displayed.

7 DXT1

S3 Texture Compression (S3TC), also known as DXTn, DXTC, or BCn, is a family of lossy texture compression algorithms originally developed by S3 Graphics, Ltd. DXT1 (also known as Block Compression 1 or BC1) is the most compact variant of S3TC. It stores 16 input pixels (in a 4x4 block) using 64 bits, consisting of two 16-bit RGB565 or PALETTED565 color values and a 4x4 two-bit lookup table. Although not natively supported, EVE series chips can render **DXT1** by leveraging the native bitmap format and multi-pass drawing techniques. In many user scenarios, DXT1 offers a good trade-off option between memory usage (4 bit per pixel) and image quality.

This section discusses the techniques for rendering DXT1 format bitmaps.

7.1 Principle

Each DXT1 block represents a 4x4 pixel area and consists of two 16-bit color values (c0 and c1) along with a 4x4 two-bit lookup table. The color values (c0 and c1) are stored in **PALETTED565** or **RGB565** bitmaps and rendered into the **RGB** channels, forming what is called the color layer. The lookup table, on the other hand, can be stored in a **L1** or **L2** bitmap and rendered into the alpha channel, forming the gradient layer.

7.2 Set up the bitmap handle of color layer

The color layer bitmap is stored in **RGB565** or **PALETTED565** format and occupies $\frac{1}{4}$ of the original image size, since DXT1 compresses the image 4x4 blocks. To make sure the displayed image appears at the correct size, **BITMAP_SIZE** must be adjusted to match the full image dimensions:

```
d1(BITMAP_HANDLE(colorHandle)); // Color layer
if (RGB565)
    cmd_setbitmap (colorAddr, RGB565, width / 4, height / 4);
else
    cmd_setbitmap (colorAddr, PALETTED565, width / 4, height / 4);

d1(BITMAP_SIZE(NEAREST, BORDER, BORDER, width, height));

if (PALETTED565)
    d1(PALETTE_SOURCE(paletteAddr));
```

7.3 Set up the bitmap handle of gradient layer

The gradient layer bitmap is stored in **L1** or **L2** format and occupies the full size of the image.

```
d1(BITMAP_HANDLE(gradientHandle)); // Gradient layer
if (L1)
    cmd_setbitmap( gradientAddr, L1, width, height);
else
    cmd_setbitmap( gradientAddr, L2, width, height);
```

7.4 Render the DXT1 bitmap

The gradient layer is rendered into the alpha channel, while the color layer is rendered into the RGB channels. The following code illustrates the procedure for rendering a **DXT1** bitmap.

```
d1(SAVE_CONTEXT());
d1(BEGIN(BITMAPS));
if (L1)
    d1(BLEND_FUNC(ONE, ZERO));
    d1(COLOR_A(0x55));
    d1(VERTEX2II(x, y, gradientHandle, 0));
    d1(BLEND_FUNC(ONE, ONE));
    d1(COLOR_A(0xAA));
    d1(VERTEX2II(x, y, gradientHandle, 1));
else
    d1(BLEND_FUNC(ONE, ZERO));
    d1(COLOR_A(0xFF));
    d1(VERTEX2II(x, y, gradientHandle, 0));
d1(COLOR_MASK(1, 1, 1, 0));
cmd_loadidentity();
cmd_scale(4UL * 65536UL, 4UL * 65536UL); // Color passes, scaled up 4x, nearest
cmd_setmatrix();
d1(BLEND_FUNC(DST_ALPHA, ZERO));
d1(VERTEX2II(x, y, colorHandle, 1)); // Color layer 1 (RGB*L2)
d1(BLEND_FUNC(ONE_MINUS_DST_ALPHA, ONE));
d1(VERTEX2II(x, y, colorHandle, 0)); // Color layer 0 (RGB*(1-L2) + DST)
d1(END);
d1(RESTORE_CONTEXT);
```

Note:

A key limitation of rendering DXT1 bitmaps is that the final output must only use **NEAREST** scaling and can only be scaled by integer multiples; otherwise, visual artifacts may appear. Additionally, the bitmap should not be positioned at non-integer screen coordinates or rotated.

8 Contact Information

Refer to <https://brtchip.com/contact-us/> for contact information.

Distributor and Sales Representatives

Please visit the Sales Network page of the [Bridgetek Web site](#) for the contact details of our distributor(s) and sales representative(s) in your country.

System and equipment manufacturers and designers are responsible to ensure that their systems, and any Bridgetek Pte Ltd (BRTChip) devices incorporated in their systems, meet all applicable safety, regulatory, and system-level performance requirements. All application-related information in this document (including application descriptions, suggested Bridgetek devices, and other materials) is provided for reference only. While Bridgetek has taken care to assure it is accurate, this information is subject to customer confirmation, and Bridgetek disclaims all liability for system designs and any application assistance provided by Bridgetek. Use of Bridgetek devices in life support and/or safety applications is entirely at the user's risk, and the user agrees to defend, indemnify, and hold harmless Bridgetek from any damages, claims, suits, or expenses resulting from such use. This document is subject to change without notice. No freedom to use patents or other intellectual property rights is implied by the publication of this document. Neither the whole nor any part of the information contained in, or the product described in this document, may be adapted, or reproduced in any material or electronic form without the prior written consent of the copyright holder. Bridgetek Pte Ltd, 1 Tai Seng Avenue, Tower A, #03-05, Singapore 536464. Singapore Registered Company Number: 201542387H.

Appendix A – References

Document References

[BT815/6 Datasheet](#)
[BT817/8 Datasheet](#)
[OpenGL 4.5 Reference Pages](#)

Acronyms and Abbreviations

Terms	Description
ADC	Analog-to-digital
API	Application Programming Interface
AVI	Audio Video Interactive
ASTC	Adaptive Scalable Texture Compression
ASCII	American Standard Code for Information Interchange
CTPM	Capacitive Touch Panel Module
CTSE	Capacitive Touch Screen Engine
EVE	Embedded Video Engine
FIFO	First In First Out buffer
I ² C	Inter-Integrated Circuit
JPEG	Joint Photographic Experts Group
LCD	Liquid Crystal Display
MCU	Micro controller unit
MPU	Microprocessor Unit
PCB	Printed Circuit Board
PCM	Pulse-Code Modulation
PNG	Portable Network Graphics
PWM	Pulse Width Modulation
RAM	Random Access Memory
RTE	Resistive Touch Engine
ROM	Read Only Memory
SPI	Serial Peripheral Interface

Appendix B – List of Tables/ Figures/ Registers/ Code Snippets

List of Tables

Table 1 - API Reference Definitions	11
Table 2 - Updated Commands in BT81X	12
Table 3 – Memory Map	14
Table 4 – Read Chip Identification Code	16
Table 5 – Flash Interface states and commands.....	20
Table 6 – Bitmap Rendering Performance.....	28
Table 7 – Common Registers Summary	36
Table 8 – RTE Registers Summary	38
Table 9 – CTSE Registers Summary.....	41
Table 10 – Graphics Context.....	54
Table 11 - Graphics Primitive Definition	57
Table 12 – Bitmap formats and bits per pixel.....	58
Table 13 – BITMAP_LAYOUT Format List	59
Table 14 – L1/L2/L4/L8 Pixel Format	61
Table 15 – ARGB2/RGB332 Pixel Format.....	61
Table 16 – RGB565/PALETTED565 Pixel Format	61
Table 17 – ARGB1555/ARGB4/PALETTED4444 Pixel Format.....	61
Table 18 – PALETTED8 Pixel Format	62
Table 19 – BLEND_FUNC Constant Value Definition	72
Table 20 – STENCIL_OP Constants Definition.....	88
Table 21 – VERTEX_FORMAT and Pixel Precision	92
Table 22 – Widgets Color Setup Table.....	96
Table 23 – Legacy Font Metrics Block	98
Table 24 – Extended Font Metrics Block	98
Table 25 - Flag Characters	104
Table 26 – String Format Specifier	105
Table 27 – Coprocessor Faults Strings	106
Table 28 – Coprocessor Engine Graphics State.....	107
Table 29 – Parameter OPTION Definition	108

List of Figures

Figure 1 – BT81X data flow	15
Figure 2 – Getting Started Example.....	22
Figure 3 – Widget List.....	95
Figure 4 – ROM Font List.....	101

List of Registers

Register Definition 1 – REG_TAG Definition	29
Register Definition 2 – REG_TAG_Y Definition	29
Register Definition 3 – REG_TAG_X Definition.....	29
Register Definition 4 – REG_PCLK Definition	30
Register Definition 5 – REG_PCLK_POL Definition	30
Register Definition 6 – REG_CSPREAD Definition	30
Register Definition 7 – REG_SWIZZLE Definition	30
Register Definition 8 – REG_DITHER Definition	30
Register Definition 9 – REG_OUTBITS Definition.....	31
Register Definition 10 – REG_ROTATE Definition	31
Register Definition 11 – REG_DLSWAP Definition	31
Register Definition 12 – REG_VSYNC1 Definition	31
Register Definition 13 – REG_VSYNC0 Definition	31
Register Definition 14 – REG_VSIZE Definition.....	32
Register Definition 15 – REG_VOFFSET Definition.....	32
Register Definition 16 – REG_VCYCLE Definition.....	32
Register Definition 17 – REG_HSYNC1 Definition	32
Register Definition 18 – REG_HSYNC0 Definition	32
Register Definition 19 – REG_HSIZE Definition.....	32
Register Definition 20 – REG_HOFFSET Definition.....	32
Register Definition 21 – REG_HCYCLE Definition	33
Register Definition 22 – REG_PLAY Definition.....	33
Register Definition 23 – REG_SOUND Definition	33
Register Definition 24 – REG_VOL_SOUND Definition.....	33
Register Definition 25 – REG_VOL_PB Definition	33
Register Definition 26 – REG_PLAYBACK_PLAY Definition	34
Register Definition 27 – REG_PLAYBACK_LOOP Definition	34
Register Definition 28 – REG_PLAYBACK_FORMAT Definition	34
Register Definition 29 – REG_PLAYBACK_FREQ Definition.....	34
Register Definition 30 – REG_PLAYBACK_READPTR Definition	34
Register Definition 31 – REG_PLAYBACK_LENGTH Definition.....	35
Register Definition 32 – REG_PLAYBACK_START Definition	35
Register Definition 33 – REG_PLAYBACK_PAUSE Definition.....	35
Register Definition 34 – REG_FLASH_STATUS Definition	35
Register Definition 35 – REG_FLASH_SIZE Definition	35
Register Definition 36 – REG_TOUCH_CONFIG Definition	36
Register Definition 37 – REG_TOUCH_TRANSFORM_F Definition.....	36
Register Definition 38 – REG_TOUCH_TRANSFORM_E Definition.....	37

Register Definition 39 – REG_TOUCH_TRANSFORM_D Definition	37
Register Definition 40 – REG_TOUCH_TRANSFORM_C Definition	37
Register Definition 41 – REG_TOUCH_TRANSFORM_B Definition	37
Register Definition 42 – REG_TOUCH_TRANSFORM_A Definition	37
Register Definition 43 – REG_TOUCH_TAG Definition	38
Register Definition 44 – REG_TOUCH_TAG_XY Definition	38
Register Definition 45 – REG_TOUCH_SCREEN_XY Definition	38
Register Definition 46 – REG_TOUCH_DIRECT_Z1Z2 Definition	39
Register Definition 47 – REG_TOUCH_DIRECT_XY	39
Register Definition 48 – REG_TOUCH_RZ Definition	39
Register Definition 49 – REG_TOUCH_RAW_XY Definition	39
Register Definition 50 – REG_TOUCH_RZTHRESH Definition	39
Register Definition 51 – REG_TOUCH_OVERSAMPLE Definition	40
Register Definition 52 – REG_TOUCH_SETTLE Definition	40
Register Definition 53 – REG_TOUCH_CHARGE Definition	40
Register Definition 54 – REG_TOUCH_ADC_MODE Definition	40
Register Definition 55 – REG_TOUCH_MODE Definition	40
Register Definition 56 – REG_CTOUCH_MODE Definition	41
Register Definition 57 – REG_CTOUCH_EXTENDED Definition	42
Register Definition 58 – REG_CTOUCH_TOUCH_XY Definition	42
Register Definition 59 – REG_CTOUCH_TOUCH1_XY Definition	42
Register Definition 60 – REG_CTOUCH_TOUCH2_XY Definition	42
Register Definition 61 – REG_CTOUCH_TOUCH3_XY Definition	42
Register Definition 62 – REG_CTOUCH_TOUCH4_X Definition	42
Register Definition 63 – REG_CTOUCH_TOUCH4_Y Definition	43
Register Definition 64 – REG_CTOUCH_RAW_XY Definition	43
Register Definition 65 – REG_CTOUCH_TAG Definition	43
Register Definition 66 – REG_CTOUCH_TAG1 Definition	43
Register Definition 67 – REG_CTOUCH_TAG2 Definition	43
Register Definition 68 – REG_CTOUCH_TAG3 Definition	44
Register Definition 69 – REG_CTOUCH_TAG4 Definition	44
Register Definition 70 – REG_CTOUCH_TAG_XY Definition	44
Register Definition 71 – REG_CTOUCH_TAG1_XY Definition	44
Register Definition 72 – REG_CTOUCH_TAG2_XY Definition	44
Register Definition 73 – REG_CTOUCH_TAG3_XY Definition	45
Register Definition 74 – REG_CTOUCH_TAG4_XY Definition	45
Register Definition 75 – REG_CMD_DL Definition	45
Register Definition 76 – REG_CMD_WRITE Definition	45
Register Definition 77 – REG_CMD_READ Definition	46
Register Definition 78 – REG_CMDB_SPACE Definition	46

Register Definition 79 – REG_CMDB_WRITE Definition	46
Register Definition 80 – REG_CPURESET Definition.....	46
Register Definition 81 – REG_MACRO_1 Definition.....	47
Register Definition 82 – REG_MACRO_0 Definition.....	47
Register Definition 83 – REG_PWM_DUTY Definition	47
Register Definition 84 – REG_PWM_HZ Definition	47
Register Definition 85 – REG_INT_MASK Definition.....	47
Register Definition 86 – REG_INT_EN Definition	48
Register Definition 87 – REG_INT_FLAGS Definition.....	48
Register Definition 88 – REG_GPIO_DIR Definition	48
Register Definition 89 – REG_GPIO Definition	49
Register Definition 90 – REG_GPIOX_DIR Definition	49
Register Definition 91 – REG_GPIOX Definition	49
Register Definition 92 – REG_FREQUENCY Definition	50
Register Definition 93 – REG_CLOCK Definition	50
Register Definition 94 – REG_FRAMES Definition	50
Register Definition 95 – REG_ID Definition	50
Register Definition 96 – REG_SPI_WIDTH Definition	50
Register Definition 97 – REG_ADAPTIVE_FRAMERATE Definition.....	50
Register Definition 98 – REG_UNDERRUN Definition	51
Register Definition 99 – REG_AH_HCYCLE_MAX Definition.....	51
Register Definition 100 – REG_PCLK_FREQ Definition	51
Register Definition 101 – REG_PCLK_2X Definition	51
Register Definition 102 – REG_TRACKER Definition	52
Register Definition 103 – REG_TRACKER_1 Definition	52
Register Definition 104 – REG_TRACKER_2 Definition	52
Register Definition 105 – REG_TRACKER_3 Definition	52
Register Definition 106 – REG_TRACKER_4 Definition	53
Register Definition 107 – REG_MEDIAFIFO_READ Definition	53
Register Definition 108 – REG_MEDIAFIFO_WRITE Definition.....	53
Register Definition 109 – REG_PLAY_CONTROL Definition	53
Register Definition 110 – REG_ANIM_ACTIVE Definition	53
Register Definition 111 – REG_COPRO_PATCH_PTR Definition	53

List of Code Snippets

Code Snippet 1 – Initialization Sequence	17
Code Snippet 2 - Setting before SLEEP	17
Code Snippet 3 - Resume registers after SLEEP	18
Code Snippet 4 – Play C8 on the Xylophone	21
Code snippet 5 – Stop Playing Sound	21

Code snippet 6 – Avoid Pop Sound	21
Code Snippet 7 – Audio Playback	21
Code Snippet 8 – Check the status of Audio Playback	21
Code Snippet 9 – Stop the Audio Playback	22
Code Snippet 10 – Getting Started	22
Code Snippet 11 – Color and Transparency	27
Code Snippet 12 – PALETTED8 Drawing Example	62

Appendix C – Revision History

Document Title: BRT_AN_033 BT81X Series Programming Guide
 Document Reference No.: BRT_000225
 Clearance No.: BRT#129
 Product Page: <http://brtchip.com/product/>
 Document Feedback: [Send Feedback](#)

Revision	Changes	Date (DD-MM-YYYY)
Version 1.0	Initial release	14-08-2018
Version 1.1	Corrected the reset value of Bitmap_transform_A/B/D/E; Added the limitation of cmd_loadimage for PNG image: top 42K bytes of RAM_G is overwritten; Fixed the typo of cmd_flashfast; Added the flash driver information; Fixed the typo in cmd_track example; Added more explanation for cmd_interrupt Added the missing definition of OPT_FILL Removed GL_FORMAT in extended font format, Added the exception of bitmap format in font metrics block. Fixed the broken reference. Updated the ASTC RAM layout image.	03-07-2019
Version 1.2	Updated statement in CMD_VIDEOFRAME	30-03-2020
Version 2.0	Added description for BT817/8 Enhanced the register tables and command example code format (CMD_GETPTR & CMD_CLEARCACHE) Example added for CMD_CLEARCACHE The -1 definition of channel number for animation playback removed Updated Table 28 (Added Parameter option - OPT_DITHER)	07-07-2020
Version 2.1	Updated the Table of Flash Interface; Updated the Sample code to cover the RAM_CMD wrap-up use case; Updated the Section 5.4 to reflect CMD_KEYS do not support UTF8 characters; Deleted the obsolete CMD_SKETCH; Fixed the mute sound value in the code snippet "Avoid Pop Sound"	07-06-2021
Version 2.2	Fixed multiple minor format and typo issues; Used lower case for commands parameters; Updated the conversion specifier 'c' and 's' in string format section from upper case to lower case; Corrected the bit per pixel value for ASTC 8x8,10x5,10x6 in table 12: Bitmap formats and bits per pixel; Added the missing member in xfont structure	24-09-2021
Version 2.3	Updated the following - <ul style="list-style-type: none"> • Section "Register Definition 100 – REG_PCLK_FREQ Definition" to correct the bits and section "5.111 CMD_PCLKFREQ" to add notes about using the command. • Removed the 300ms delay from Code Snippet 1 – Initialization Sequence • Change the fast mode of flash to full-speed mode of flash • Example code of cmd_plkfreq • Fix the REG_SPI_WIDTH offset to 0x188 • Remove the disabled feature description of cmd_fontcache • Add RST_PULSE in the sample code of bootup sequence • Typo correction: REG_GPIO_X ->REG_GPIOX 	16-12-2022

	<ul style="list-style-type: none"> Remove the undefined DLSWAP_FRAME Add the requirement of ASTC multi-cell bitmap: 4 blocks alignment Modify the maximum bitmap size of Bargraph from 256x256 to screen width to 256. Add the missing parameter of cmd_memcpy Improve xfont structure description 	
Version 2.4	<p>Updated the following –</p> <ul style="list-style-type: none"> Change int16_t to uint16_t where it is applicable The dest parameter of JUMP/CALL command is changed to number of display list Removed the irrelevant chip ID of FT81x Fixed the typo of PALETTED4444/565 in PALETTE_SOURCE command Added example code for CMD_SETBITMAP Added cmd_nop command Moved the RST_PULSE host command from `after` to `before` ACTIVE host command in boot up sequence. Rephrased the note of the SPI clock frequency in boot up sequence. Fixed the typo cmd_crc to cmd_memcrc Updated the office address of Singapore 	17-11-2023
Version 2.5	<p>Updated the following –</p> <ul style="list-style-type: none"> Add a section of DXT1 Add delay recommendation after the host command "ACTIVE" in initial sequence Add a power management section Fix pseudo code error in CMD_FLASHUPDATE Add error string for "illegal flash address" in coprocessor fault section 	14-08-2024
Version 2.6	<p>Fixed the following errors –</p> <ul style="list-style-type: none"> Typo error for "Channel" Screen rotation picture error Sample code error for font extended format Typo error for CMD_FLASHSPITX and CMD_FLASHSPIRX Typo in print message of cmd_mediafifo <p>Added the following –</p> <ul style="list-style-type: none"> Animation play mode in animation support section Chroma subsampling patterns in cmd_loadimage <p>Updated the following –</p> <ul style="list-style-type: none"> Sample code for DXT1 CMD_NOP command Hex value CMD_FLASHWRITE and CMD_FLASHPROGRAM 	20-11-2024
Version 2.7	<p>Added note for parameter limitation for CMD_INTERRUPT;</p> <p>Corrected the Typo error for Co-processor Fault Strings;</p> <p>Added interrupt source for REG_INT_FLAG and REG_INT_MASK. Differentiate the bit settings for BT816/7 and BT817/8;</p> <p>Updated SPI clock frequency in Note in initialization sequence;</p> <p>Added note for host_command in 2.4;</p> <p>Changed 255 to the value from the tag buffer in `TAG` session;</p> <p>Aligned the font in `VERTEX_FORMAT` session;</p> <p>Updated REG_OUTBITS;</p> <p>Updated default value for REG_GPIO_DIR and REG_GPIOX;</p> <p>Updated description for REG_GPIOX;</p> <p>Corrected string format error in section 5.6;</p>	15-07-2025

	Section 3.2 update REG_VOL_PB; Updated coprocessor faults strings table in section 5.7; Description changes for CMD_APPENDF in section 5.81; Reformatted all the cmd in section 5	
2.8	Added the missing description for "size" in Section 5.42 Updated the description in Section 4.7 Added links in Section 4.5, 4.15 and 4.51	20-02-2026