FT800 系列程序员指南

Version 2.0

发布日期: **2016-03-23**

此文件是 FT800 系列芯片的程序员指南。这份指南详细描述芯片的功能及程序。若要查询 FT801 的特定功能及程序，请参考章节 FT801。

Future Technology Devices International Limited (FTDI)
Unit 1, 2 Seaward Place, Glasgow G41 1HH, United Kingdom
Tel.: +44 (0) 141 429 2777 Fax: + 44 (0) 141 429 2758
Web Site: http://ftdichip.com

目录

源代码片段清单

图清单

表清单

寄存器清单

1 简介

这份文件叙述了 FT800 系列芯片的图形指令、小工具(widget)指令的编程细节，以及 FT800 系列芯片的配置方法，以提供流畅而生动的的屏幕效果。

FT800 系列的芯片是图形的控制器，拥有音频播放及触控能力等附加功能。它们包含了丰富的图形物件(包括基元(primitive)以及小工具(widget))集合，可以用在一系列产品的菜单和截图显示上，例如家用电器、玩具、工厂机械、家庭自动化、电梯以及其它相关的应用。

1.1 概观

这份文件针对每个特定指令都有简单易用的范例，因此对于了解指令集很有用处。除此之外，也有包括不同的电源模式、音频和触控功能以及各自的使用方式解说。

引脚设定、硬件模型以及硬件配置信息都可以参考 FT800 的数据表（DS_FT800_Embedded_Video_Engine）或是 FT801 的数据表（DS_FT801）

1.2 文件范围

这份文件目的在供软件编程员和系统设计者在有 SPI 或 I^2C 主接口的系统处理器上，开发有图形用户界面（GUI）的应用程序。

1.3 API 参考定义

在这份文件有用到的 API，其功能性及其命名方式如下所示

wr8() –在指定的地址位置写入 8 个 bits

wr16() –在指定的地址位置写入 16 个 bits

wr32() –在指定的地址位置写入 32 个 bits

wr8s() – 在指定的地址位置写入 8 个 bits 的字符串

rd8() –从指定的地址位置读取 8 个 bits

rd16() –从指定的地址位置读取 16 个 bits

rd32() –从指定的地址位置读取 32 个 bits

rd8s() –从指定的地址位置读取 8 个 bits 的字符串

cmd() –写入 32 个 bits 的指令到协处理器引擎 FIFO RAM_CMD

cmd_*() –写入 32 个 bits 的指令到协处理器引擎 FIFO(RAM_CMD)，包含必须的参数。

dl() –写入指定的 32 个 bits 显示清单指令到 RAM_DL。更多信息可参考 2.5.4 节。

host_command() –发送主机命令到 FT800。更多信息参考 FT800 的数据表。

2 程序模型

FT800 对于主机 MCU 来说是一个存储器映射的 SPI 或是 I^2C 装置。主机以读取或是写入 8MB 的地址空间。

在这份文件里，DL 指令、协处理器引擎指令、寄存器值的读写、输入的 RGB 位图数据、以及 ADPCM(自适应差分脉冲编码调制)输入数据的字节序都是`(Little Endian)小字节'格式。

2.1 一般性软件架构

软件架构可以大致上分类成不同的层次，例如定制的应用程序、图形/GUI 管理器、视频管理器、音频管理器、以及驱动程式等等。FT800 的上级图形引擎指令及协处理器引擎的小工具指令是属于图形/GUI 管理器的一部份。视频以及音频的控制&数据路径是属于视频管理器及音频管理器。图形/GUI 管理器及硬件的沟通是透过 SPI 及 I^2C 驱动程序。

通常显示的屏幕截图是透过定制的应用程序所建构，而这些定制的应用程序是根据由图形/GUI 处理器曝露的帧。

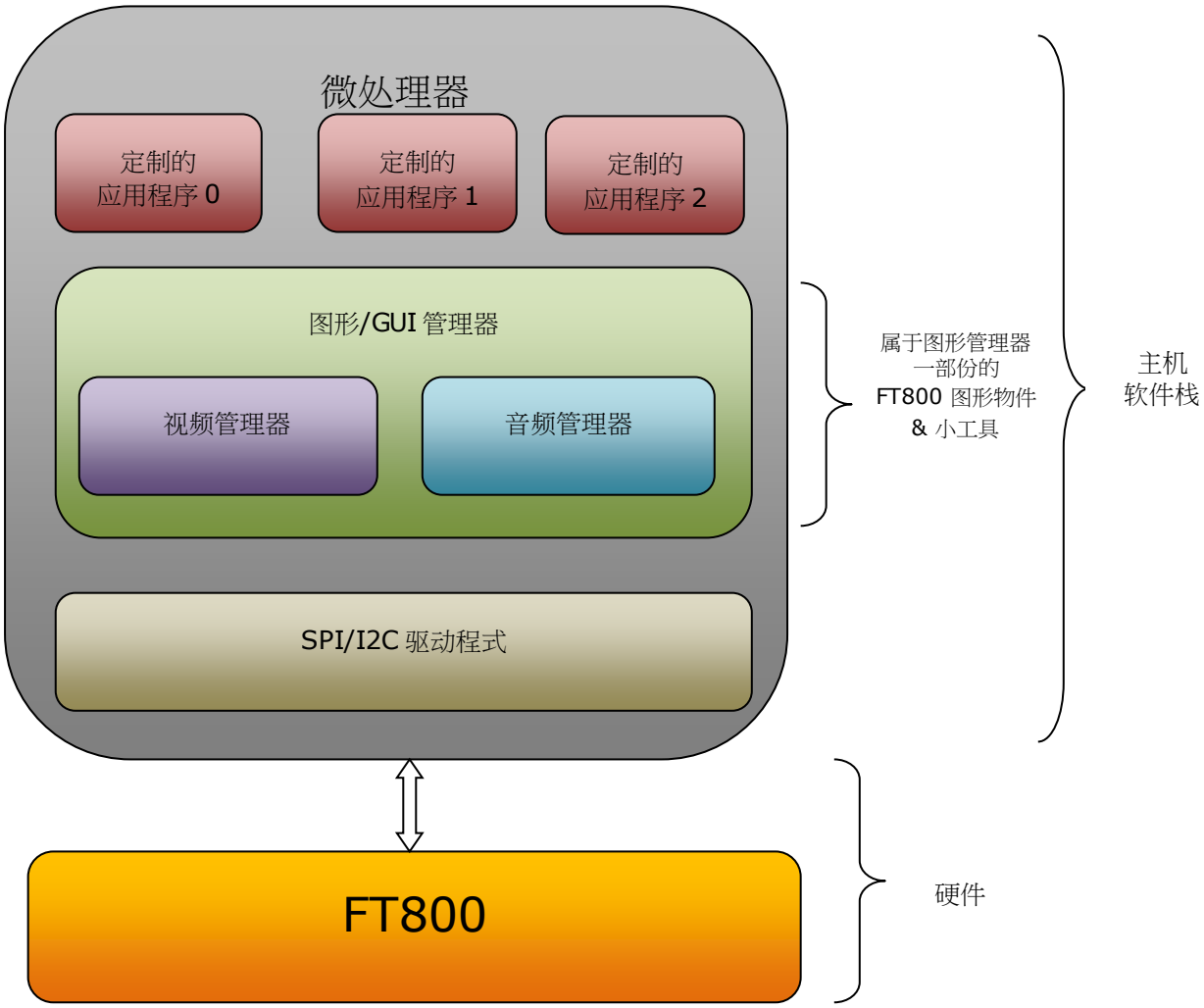


图 1: 软件架构

2.2 显示配置及初始化

显示的配置，需将特定数值载入时序控制的寄存器以达到想要的显示结果。以下这些寄存器控制水平方向的时序：

- REG_PCLK
- REG_PCLK_POL
- REG_HCYCLE
- REG_HOFFSET
- REG_HSIZE
- REG_HSYNC0
- REG_HSYNC1

以下这些寄存器控制垂直方向的时序：

- REG_VCYCLE

- REG_VOFFSET
- REG_VSIZE
- REG_VSYNC0
- REG_VSYNC1

寄存器 REG_CSPREAD 改变颜色时序以减少系统噪音。

GPIO bit 7 是用来作为 LCD 模块的显示启动引脚。将 GPIO bit 的方向设定成”输出”之后，可以写入数值”1”到 GPIO bit 7 启动显示，或是写入数值”0”到 GPIO bit 7 关闭显示。预设的 GPIO bit 7 方向是”输出”，数值为”0”。

注意:有关显示寄存器集合的信息请参考 FT800 的数据表。

2.2.1 水平时序

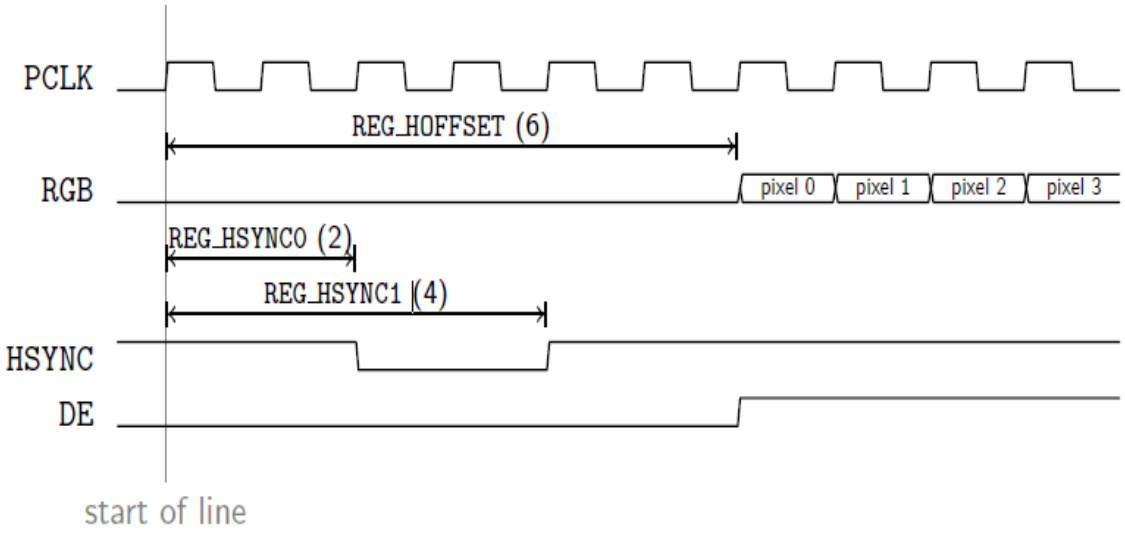


图 2: 水平时序

REG_PCLK 控制 PCLK 的频率。此寄存器会设定一个除数给主时钟 48MHz，例如除数为 4，PCLK 则输出一个 12MHz 的时钟。但如果 REG_PCLK 设为 0，则所有显示输出会暂停。REG_PCLK_POL 控制 PCLK 的极性。”0”表示数据输出于 PCLK 的上升沿，若为”1”则表示数据输出于 PCLK 的下降沿。

在一条水平线上 PCLK 时钟的总数为 REG_HCYCLE。在这水平线上是扫描输出的像素，像素总数以 REG_HSIZE 表示。像素的扫描输出是在 REG_HOFFSET 个时钟周期后开始。当像素正被扫描输出的时候，信号 DE 的值为高电平。

在信号 HSYNC 上的水平同步时序是被 REG_HSYNC0 及 REG_HSYNC1 所控制。两者分别代表 HSYNC 上升及下降的时间。

2.2.2 垂直时序

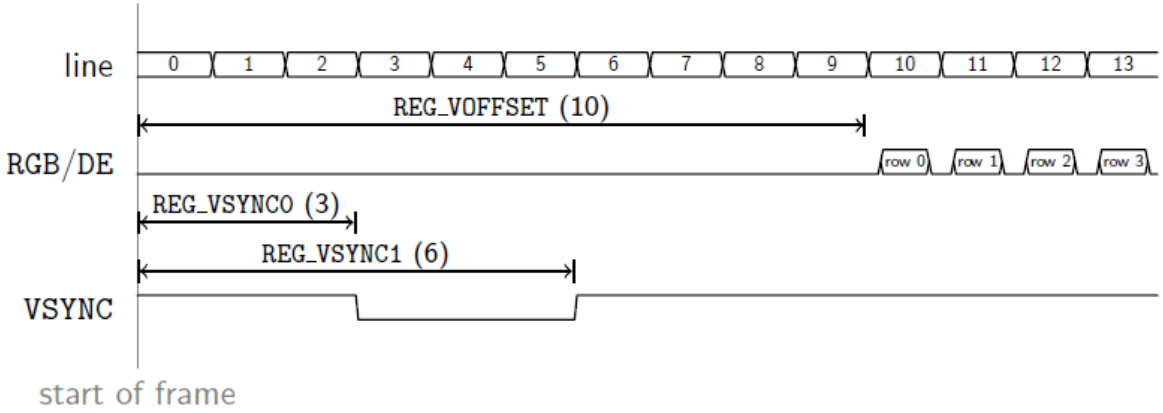


图 3: 垂直时序

垂直时序是被线的数量所指定。在一个帧里所有线的总数为 REG_VCYCLE。而像素行数总共有 REG_VSIZE 个。这些线是在经过 REG_VOFFSET 个时钟周期后开始。

在信号 VSYNC 上的垂直同步时序是由 REG_VSYNC0 及 REG_VSYNC1 所控制。两者分别代表 VSYNC 上升及下降的时间。

2.2.3 信号更新时序控制

当 REG_CSPREAD 为关闭状态，所有颜色信号在同一个时间更新：

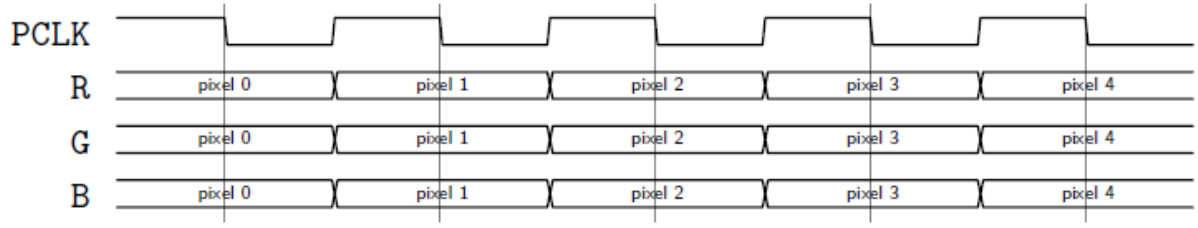


图 4: 没有 CSPREAD 的像素时钟

若 REG_CSPREAD 为启动状态，颜色信号的时序会被些微调整，所以只有较少的信号会同时改变：

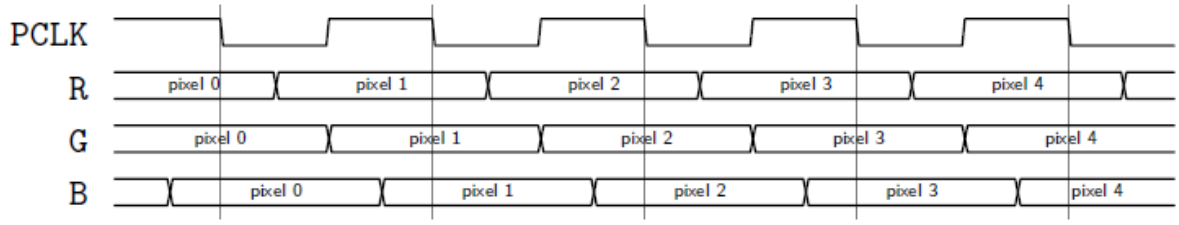


图 5: 有 CSPREAD 的像素时钟

2.2.4 时序范例: 480x272 分辨率，更新频率 60Hz

显示的更新频率若为 60Hz，每一帧有 48000000/60=800000 个快速时钟。将 PCLK 的除数 REG_PCLK 设为 5，则 PCLK 的频率变为 9.6 MHz，而

每一帧有 800000/5= 160000 个 PCLK 周期。

对于一个 480 x 272 分辨率的显示屏来说，典型的水平周期是 525 个时钟周期的时间，垂直周期是 286 行。经过一些运算得到 548 x 292 配置下一帧有 160016 个周期，很接近目标。故当 REG_HCYCLE=548 而 REG_VCYCLE=292，显示频率几乎为 60Hz。其他寄存器设定可以直接透过查找显示屏幕的数据表来设定。

2.2.5 初始程序

这一部份描述在不同情况下的初始化程序。

- 开机过程的初始化程序:
  1. 使用的 MCU SPI 时钟不要超过 11MHz。
  2. 传送主机命令"CLKEXT" 到 FT800。
  3. 传送主机命令"ACTIVE"，以启动送到 FT800 的时钟。
  4. 配置视频时序寄存器(REG_PCLK 除外)。
  5. 写入第一个显示清单。
  6. 写入 REG_DLSWAP, FT800 即刻交换(swap)显示清单。
  7. 启动背光控制以供显示。
  8. 写入 REG_PCLK, 从第一个显示清单开始视频输出。
  9. 使用的 MCU SPI 时钟不要超过 30MHz。

```
MCU_SPI_CLK_Freq(<11MHz);// 使用小于11MHz的MCU SPI时钟

host_command(CLKEXT);// 传送主机命令"CLKEXT" 到FT800
host_command(ACTIVE);// 传送主机命令"ACTIVE" 到FT800


/* 配置视频时序寄存器- 以展示WQVGA的解析度*/
wr16(REG_HCYCLE,548);
wr16(REG_HOFFSET,43);
wr16(REG_HSYNC0,0);
wr16(REG_HSYNC1,41);
wr16(REG_VCYCLE,292);
wr16(REG_VOFFSET,12);
wr16(REG_VSYNC0,0);
wr16(REG_VSYNC1,10);
wr8(REG_SWIZZLE,0);
wr8(REG_PCLK_POL,1);
wr8(REG_CSPREAD,1);
wr16(REG_HSIZE,480);
wr16(REG_VSIZE,272);

/* 写入第一个显示清单 */
wr32(RAM_DL+0,CLEAR_COLOR_RGB(0,0,0));
wr32(RAM_DL+4,CLEAR(1,1,1));
wr32(RAM_DL+8,DISPLAY());

wr8(REG_DLSWAP,DLSWAP_FRAME);//交换显示清单
```

```
wr8(REG_GPIO_DIR,0x80| Ft_Gpu_Hal_Rd8(phost,REG_GPIO_DIR));
wr8(REG_GPIO,0x080|  Ft_Gpu_Hal_Rd8(phost,REG_GPIO));//启动显示的bit

wr8(REG_PCLK,5);//此后LCD上可以看到图像

MCU_SPI_CLK_Freq(<30Mhz);//  使用的MCU  SPI时钟不要超过30MHz
```

源代码片段 1 初始化过程

- 由掉电引脚 PD_N 进行初始化程序：
    1. 将掉电引脚 PD_N 驱动至高电平
    2. 等待至少 20 毫秒
    3. 执行 ”开机过程的初始化程序”的步骤 1 至步骤 9
- 从睡眠模式初始化的程序:
    1. 传送“ACTIVE”指令给主机以启动送到 FT800 的时钟
    2. 等待至少 20 毫秒
    4. 执行 ”开机过程的初始化程序”的步骤 5 至步骤 8
- 从待机模式初始化的程序:

    执行”从睡眠模式初始化的程序”的步骤(除了步骤 2 的等待至少 20 毫秒)

注意:电源模式的信息可参考 FT800 的数据表。在掉电及复位操作时的音频管理请跟随 **Error! Reference source not found.** 节的指示。

2.3 声音合成器

播放在木琴上的 C8 音符示例代码：

```
wr8(REG_VOL_SOUND,0xFF);//  将音量设成最大
wr16(REG_SOUND,(0x6C<<8)|0x41);//  木琴的C8 MIDI音符
wr8(REG_PLAY,1);//  拨放声音
```

源代码片段 2 声音合成器拨放木琴上的 C8 音符

确认声音播放状态的示例代码:

```
Sound_status  = rd8(REG_PLAY);//  1-播放正在进行, 0-播放已结束
```

源代码片段 3 声音合成器确认声音播放状态

停止声音播放的示例代码:

```
wr16(REG_SOUND,0x0);//  将播放的声音设置为静音
wr8(REG_PLAY,1);//  播放声音
Sound_status  = rd8(REG_PLAY);//  1-播放正在进行, 0-播放已经结束
```

源代码片段 4 声音合成器停止声音播放

为了避免音频在复位或是电源状态改变时候出现爆音，触发一个”无声”的声音，并等待完成(声音播放完成是指 REG_PLAY 包含一个”0”值)。這樣將使輸出值設定為”0”。在重新開機時，音頻引擎會播放”关闭无声”的聲音，將輸出值驅動至中間電平。

注意: 有關聲音合成器及音頻播放器的更多資訊可以參考 FT800 的數據表。

2.4 音频播放

FT800 支援三种类型的音频格式：4 Bit IMA ADPCM，8 Bit 有符号的 PCM，8 Bit u-Law。对于 IMA ADPCM 格式，请注意字节的顺序：在一个字节内，第一个取样(4 bits)会放置在 bit 0 到 bit 3 之间，而第二个取样会放置在 bit 4 到 bit 7 之间。

为了播放在 FT800 RAM 上的音频数据， FT800 必须要求在 REG_PLAYBACK_START 的开始地址 要对齐到 64 bit (8 Bytes) 的整数倍。除此之外， 寄存器 REG_PLAYBACK_LENGTH 上指示的音频数据长度也必须对齐到 64 bit (8 Bytes)的整数倍。

为了学习播放音频数据，请参照以下的范例代码:

```
wr8(REG_VOL_PB,0xFF);//配置音频播放的音量
wr32(REG_PLAYBACK_START,0);//配置音频缓冲器的启始地址
wr32(REG_PLAYBACK_LENGTH,100*1024);//配置音频缓冲器的长度
wr16(REG_PLAYBACK_FREQ,44100);//设定音频取样频率
wr8(REG_PLAYBACK_FORMAT,ULAW_SAMPLES);//配置音频格式
wr8(REG_PLAYBACK_LOOP,0);//配置成一次或是连续播放
wr8(REG_PLAYBACK_PLAY,1);//开始音频播放
```

源代码片段 5 音频播放

```
AudioPlay_Status = rd8(REG_PLAYBACK_PLAY);//1-音频播放正在进行，0-音频播放已经结束
```

源代码片段 6 确认音频播放的状态

```
wr32(REG_PLAYBACK_LENGTH,0);//将播放长度配置为0
wr8(REG_PLAYBACK_PLAY,1);//  开始音频播放
```

源代码片段 7 停止音频播放

2.5 图表的常规性工作

这个部份描述了图表功能并提供了一些范例。

2.5.1 开始

这个简短的例子产生一个屏幕显示了"FTDI"这个文字以及一个红点。

图 6:产生一个启始影像的范例。

绘制这样屏幕的源代码如下：

```
wr32(RAM_DL +0, CLEAR(1,1,1));//  清除屏幕
wr32(RAM_DL +4, BEGIN(BITMAPS));//开始绘制位图
wr32(RAM_DL +8, VERTEX2II(220,110,31,'F'));//  字体31的ascii F
wr32(RAM_DL +12, VERTEX2II(244,110,31,'T'));// ascii T
wr32(RAM_DL +16, VERTEX2II(270,110,31,'D'));// ascii D
wr32(RAM_DL +20, VERTEX2II(299,110,31,'I'));// ascii I
wr32(RAM_DL +24, END());
wr32(RAM_DL +28, COLOR_RGB(160,22,22));//  改变颜色成红色
wr32(RAM_DL +32, POINT_SIZE(320));//将点的半径大小设定成20像素（20*16）
wr32(RAM_DL +36, BEGIN(POINTS));//开始绘制点
wr32(RAM_DL +40, VERTEX2II(192,133,0,0));//红点
wr32(RAM_DL +44, END());
wr32(RAM_DL +48, DISPLAY());//显示影像
```

源代码片段 8 开始

在载入上述的绘图指令到显示清单的 RAM 之后，为了使新的显示清单在下一个帧刷新时有效，寄存器 REG_DLSWAP 必须被设定为 0x02。

注意:

- 显示清单都是以 RAM_DL 为开始地址
- 因为每个指令的宽度都是 32 bit，地址固定的增加量是 4 bytes。

- 在做任何新图操作时，建议先执行 CLEAR 指令，以确保 FT800 图形引擎在已知的状态。
- 显示清单的结束固定都以指令 DISPLAY 标记。

2.5.2 坐标平面

下图说明了图形的坐标平面及可见区域。

若精度为 1 个像素，有效的 X 及 Y 坐标范围是从-1024 到 1023 个像素单位，若精度达 1/16 个像素，则坐标范围为-16384 到 16383 个单位。

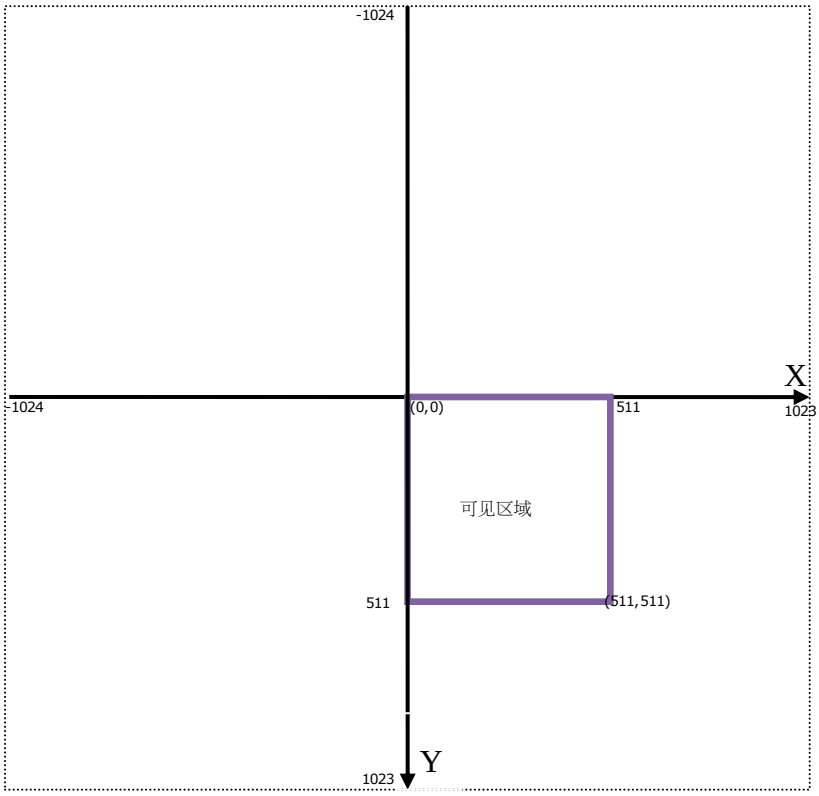


图 7: 以像素为精度表示 FT800 图形坐标平面

2.5.3 绘制图案

一般绘制图案的方法如下：

- 以 BEGIN 指令开始时，会同时选一种图形基元类型当作开始
- 输入一个或多个顶点，以指定图形基元在屏幕上的位置。
- 以 END 指令标示图形基元的结束。

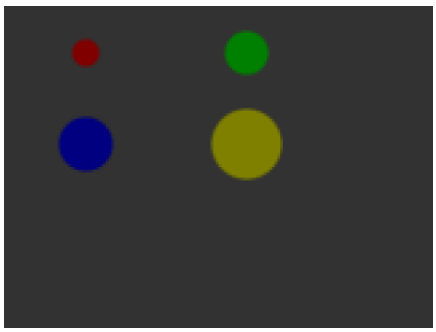
  (注意: 之后许多例子不会明确列出 END 指令)

图形引擎支持的图形基元如下：

- BITMAPS 位图 – 矩形的像素阵列，有不同的颜色格式。
- POINTS 点 - 抗混叠的点，点的半径可设成 1 到 256 个像素
- LINES 线 - 抗混叠的的线，宽度可设成 0 到 4095 个 1/16 像素单位。(宽度定义是指从线的中心到边界的距离)
- LINE_STRIP 线带 – 抗混叠的线条，头尾相接。
- RECTS 矩形 -圆角的矩形，圆角的弧度可用 LINE_WIDTH 指令调整。
- EDGE_STRIP_A/B/L/R - 边条

例子

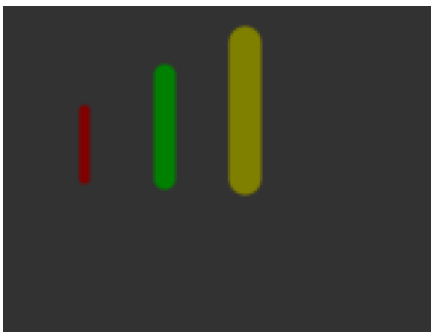
以不同的颜色，绘制半径范围从 5 个像素变化到 13 个像素的点：



```
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) );
```

利用 VERTEX2F 指令可以得到圆心的位置。

以不同的颜色画出尺寸范围从 2 个像素变化至 6 个像素的线(线宽度定义是从线的中心到边界)：



```
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) );
```

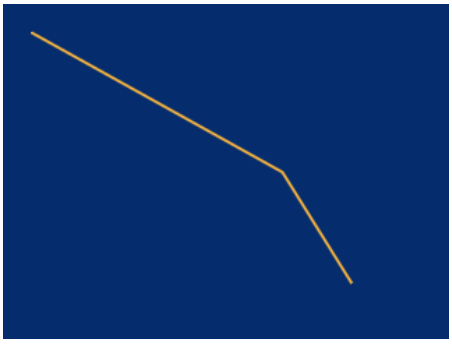
VERTEX2F 指令需以成对方式使用以定义线的开始及结束。

绘制三个尺寸分别为 5x25、10x38、15x50 的矩形(line_width 是用来定义拐角的弧度，除了矩形尺寸以外，LINE_WIDTH 像素也会加在两个方向。)

```
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) );
```

VERTEX2F 指令需以成对方式使用，以定义矩形的左上及右下拐角位置。

以多组坐标点绘制线条：



```
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) );
```

往上绘制边条：

```
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) );
```

往下绘制边条：

```
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) );
```

往右绘制边条：

```
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) );
```

往左绘制边条：

```
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) );
```

2.5.4 写入显示清单

以 wr32()写入显示清单条目相当耗时而且容易出错，因此可以采用函数的方式：

```
static size_t dli;
staticvoid dl(unsignedlong cmd)
{
wr32(RAM_DL + dli, cmd);
dli +=4;
}
...
dli =0;//开始写入显示清单
dl(CLEAR(1,1,1));//清除屏幕
dl(BEGIN(BITMAPS));//开始绘制位图
dl(VERTEX2II(220,110,31,'F'));//  字体31的ascii F
dl(VERTEX2II(244,110,31,'T'));// ascii T
dl(VERTEX2II(270,110,31,'D'));// ascii D
dl(VERTEX2II(299,110,31,'I'));// ascii I
dl(END());
dl(COLOR_RGB(160,22,22));//将颜色改为红色
dl(POINT_SIZE(320));//设定点的尺寸
dl(BEGIN(POINTS));//开始画点
dl(VERTEX2II(192,133,0,0));//红点
dl(END());
dl(DISPLAY());//显示影像
```

源代码片段 9 DL 函数的定义

2.5.5 位图变换矩阵

为了做到位图变换，FT800 里有如下所示的位图变换阵列，并以 m 来表示。

$$m = \begin{bmatrix} BITMAP\_TRANSFORM\_A & BITMAP\_TRANSFORM\_B & BITMAP\_TRANSFORM\_C \\ BITMAP\_TRANSFORM\_D & BITMAP\_TRANSFORM\_E & BITMAP\_TRANSFORM\_F \end{bmatrix}$$

预设 $m = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix}$, 称作单位矩阵。

位图变换后的坐标x', y'是用以下的式子计算：

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

也就是:

$$x' = x * A + y * B + C$$
$$y' = x * D + y * E + F$$

A,B,C,E,D,E,F 代表被指令 BITMAP_TRANSFORM_A-F 所分配的值

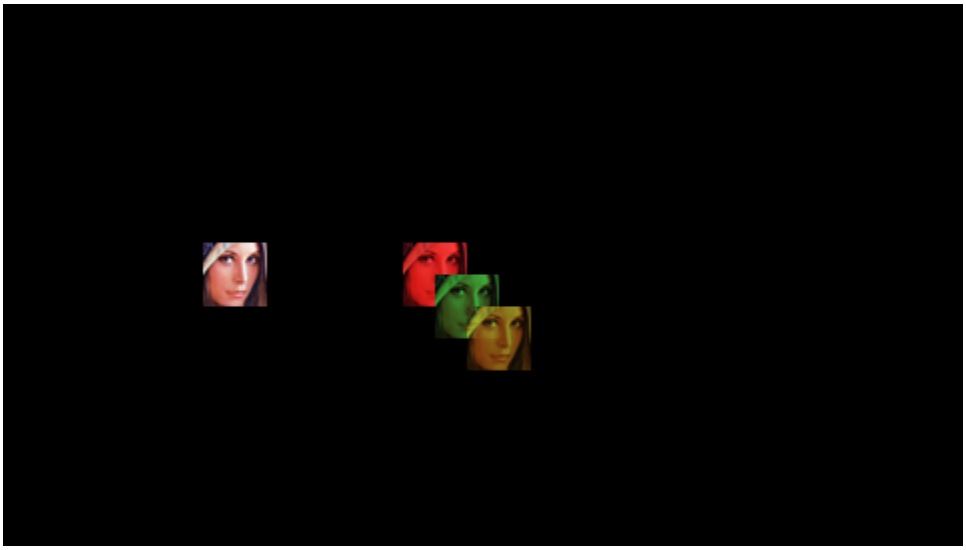
2.5.6 颜色及透明度

在屏幕上，可以在不止一处的位置画出同样的位图形，可分别设定颜色及透明度。

```
dl(COLOR_RGB(255,64,64));//  画在(200, 120)，颜色为红色
dl(VERTEX2II(200,120,0,0));
dl(COLOR_RGB(64,180,64));//  画在(216, 136)，颜色为绿色
dl(VERTEX2II(216,136,0,0));
dl(COLOR_RGB(255,255,64));//  画在(232, 152)，颜色为透明黄色
dl(COLOR_A(150));
dl(VERTEX2II(232,152,0,0));
```

源代码片段 10 颜色及透明度

COLOR_RGB 指令可以改变目前用来绘图的颜色，也就是用来绘制位图的颜色。COLOR_A 指令改变目前绘图的 alpha 值，也就是改变绘图的透明度：alpha 等于 0 表示完全透明而 alpha 等于 255 表示完全不透明。 以这个例子来说，alpha 等于 150 表现出部份透明的效果。



2.5.7 VERTEX2II 及 VERTEX2F

先前用过的 VERTEX2II 指令仅允许正值的屏幕坐标。假如位图是部份超出屏幕范围，例如在屏幕滚动上，则有必要指定到负值的屏幕坐标。VERTEX2F 指令可以允许负值坐标。它也允许分数型的坐标，因为他可指定以 1/16 像素为单位的屏幕坐标(x,y)

例如，以 VERTEX2F 指令绘制相同的位图于(-10,-10)的位置：

```
dl(BEGIN(BITMAPS));
dl(VERTEX2F(-160,-160));
dl(END());
```

源代码片段 11 负值屏幕坐标的例子

2.5.8 屏幕截图

以下的源代码展示如何利用寄存器及 RAM_SCREENSHOT 指令，以全像素数值截取目前的屏幕。每一个像素是以 32 bits 及 BGRA 的格式表示。然而，这个过程可能造成闪烁及撕裂的效果。

```
#define SCREEN_WIDTH    480
#define SCREEN_HEIGHT   272

uint32 screenshot[SCREEN_WIDTH*SCREEN_HEIGHT];

wr8(REG_SCREENSHOT_EN, 1);
for (int ly = 0; ly < SCREEN_HEIGHT; ly++) {
  wr16(REG_SCREENSHOT_Y, ly);
  wr8(REG_SCREENSHOT_START, 1);

  //读取64 bit寄存器，检查寄存器是否忙录
  while (rd32(REG_SCREENSHOT_BUSY) | rd32(REG_SCREENSHOT_BUSY + 4));

  wr8(REG_SCREENSHOT_READ , 1);
  for (int lx = 0; lx < SCREEN_WIDTH; lx ++) {
      //从RAM_SCREENSHOT读取32 bit像素值
      //像素格式为BGRA:蓝色是最低的地址，Alpha是最高的地址
      screenshot[ly*SCREEN_HEIGHT + lx] = rd32(RAM_SCREENSHOT + lx*4);
  }
  wr8(REG_SCREENSHOT_READ, 0);
}
wr8(REG_SCREENSHOT_EN, 0);
```

源代码片段 12 全像素数值的屏幕截图

2.5.9 性能

图形引擎没有帧缓存区：它使用动态的影像合成的方式，在扫描输出过程中建立每一条显示线。因此，绘制每一条线的所需的时间是一个有限的时间。这个时间依扫描输出的参数所决定(REG_PCLK 及 REG_HCYCLE)，但是绝不会少于 2048 个内部时钟的周期。

一些性能上的局限性:

- 显示清单的长度必须小于 2048 个指令，因为图形引擎在每个时钟抓取显示清单指令一次。
- 图形引擎渲染的像素数量是每时钟 4 个像素。对于有 2048 个显示指令的线条，全部绘制出来的像素表现，必须小于 8192 个像素。
- 对于一些位图格式，绘制的速率为每时钟周期一个像素。这些格式为 TEXT8X8、TEXTVGA、及 PALETTED。
- 对于双线性过滤的像素，绘制的速率会减少到每时钟周期 1/4 个像素。多数的位图的格式是每时钟周期绘制一个像素，而上述提到的格式(TEXT8X8、TEXTVGA 、及 PALETTED)是每 4 个时钟周期画一个像素。

以上总结:

表 1 位图呈现的性能

过滤模式	格式	速率
最近(Nearest)	TEXT8X8、TEXTVGA、及 PALETTED	每个时钟周期 1 个像素
最近(Nearest)	所有其他格式	每个时钟周期 4 个像素
双线性(BILINEAR)	TEXT8X8、TEXTVGA、及 PALETTED	每个时钟周期 1/4 个像素
双线性(BILINEAR)	所有其他格式	每个时钟周期 1 个像素

3 寄存器描述

在这一章里，所有 FT800 寄存器被分类成 5 个群组：图形引擎寄存器、音频引擎寄存器、触摸引擎寄存器、协处理器引擎寄存器，以及其他寄存器。这一章会为每一种寄存器下详细的定义。如要检视 FT800 寄存器的摘要，请参考数据表。

另外，请注意所有保留的 bits 都是只读的，而且值为零。所有 16 进位的数值都有 0x 作为前缀。 为了能对操作有更好的理解，强烈鼓励读者可以交叉参考本文件其他章节的部份。

3.1 图形引擎寄存器

寄存器定义 1 REG_PCLK 定义

REG_PCLK 定义	
保留	R/W

31 8 7 0

地址: 0x10246C **复位值:** 0x0

Bit 0 - 7 : 这些bits是被设定来当作主时钟的除数，以供输出为PCLK。如果典型的主时钟频率为48MHz而且这些bits的值为5，则PCLK的频率为9.6MHz。假如这些bits的值为0，则没有PCLK输出。

注意事项: 无

寄存器定义 2 REG_PCLK_POL 定义

REG_PCLK_POL Definition	
保留	R/W

31 1 0

地址: 0x102468 Reset Value: 0x0

Bit 0： 这个bit控制PCLK的极性。如果设定为0，则PCLK的极性在上升边缘。如果设定为1，则PCLK的极性是在下降边缘。

注意事项: 无

寄存器定义3 REG_CSPREAD 定义

更多细节请参考 2.2.3 节

REG_CSPREAD 定义	
保留	读/写

31 1 0

地址：0x102464 复位值： 0x1

Bit 0:这个 bit 以 PCLK 的时钟的有效边缘控制 RGB 信号的变换， 当 REG_CSPREAD=0，R[7:2]、G[7:2]、及 B[7:2]在 PCLK 的有效边缘之后做转换。当 REG_CSPREAD=1，R[7:2]在一个 PCLK 时钟周期之前做转换，而 B[7:2]在一个 PCLK 时钟周期之后做转换，这样做可以帮忙减少系统噪音。

Bit 1-31: 保留

注意事项： 无

寄存器定义4 REG_SWIZZLE 定义

REG_SWIZZLE 定义	
保留	读/写

31 4 3 0

地址：0x102460 复位值： 0x0

Bit 0 – 3:这些 bits 是用来控制输出的 RGB 引脚的排部，可能有助于支持不同的 LCD 屏幕。细节请参考以下表格。

注意事项： 无

表 2 REG_SWIZZLE 及 RGB 引脚映射表

REG_SWIZZLE				引脚			
b3	b2	b1	b0	R7, R6, R5, R4, R3, R2	G7, G6, G5, G4, G3, G2	B7, B6, B5, B4, B3, B2	
0	X	0	0	R[7:2]	G[7:2]	B[7:2]	上电后的预设值
0	X	0	1	R[2:7]	G[2:7]	B[2:7]	
0	X	1	0	B[7:2]	G[7:2]	R[7:2]	
0	X	1	1	B[2:7]	G[2:7]	R[2:7]	
1	0	0	0	G[7:2]	B[7:2]	R[7:2]	
1	0	0	1	G[2:7]	B[2:7]	R[2:7]	
1	0	1	0	G[7:2]	R[7:2]	B[7:2]	
1	0	1	1	G[2:7]	R[2:7]	B[2:7]	
1	1	0	0	B[7:2]	R[7:2]	G[7:2]	
1	1	0	1	B[2:7]	R[2:7]	G[2:7]	
1	1	1	0	R[7:2]	B[7:2]	G[7:2]	
1	1	1	1	R[2:7]	B[2:7]	G[2:7]	

寄存器定义5 REG_DITHER 定义

REG_DITHER 定义	
保留	读/写

31 1 0

地址：0x10245C 复位值： 0x1B6

Bit 0:这个 bit 设为 1 启动输出 RGB 信号的抖色功能，设为 0 关闭抖色功能。从这个 bit 读出 1 表示抖色功能有启动，从这个 bit 读出 0 表示抖色功能关闭。

注意事项： 细节请参考 REG_SWIZZLE 及 RGB 引脚的映射表格

寄存器定义 6 REG_OUTBITS 定义

REG_OUTBITS 定义	
保留	读/写

31 9 8 0

地址：0x102458 复位值： 0x1B6

Bit 0 - 8: 这 9 个 bits 会被分成三个群组，分别表示红、缘、蓝三种颜色的输出信号
Bit 0 - 2: 蓝色信号线，预设值为 6
Bit 3 - 5: 绿色信号线，预设值为 6
Bit 6 - 8: 红色信号线，预设值为 6

注意事项： 无

寄存器定义 7 REG_ROTATE 定义

REG_ROTATE 定义	
保留	读/写

31 1 0

地址：0x102454 复位值： 0x00

Bit 0: 180 度屏幕旋转开关。这个 bit 写入 0 值会关闭旋转功能。这个 bit 写入 1 会开启旋转功能 180 度的旋转效果会在下一帧渲染时出现。读取这个 bit 可以呈现目前旋转功能的状态。

注意事项： 在旋转功能打开之后，请再做一次屏幕校正

寄存器定义 8 REG_VSYNC1 定义

REG_VSYNC1 定义	
	读/写

31 10 9 0

地址：0x10244C 复位值： 0x00A

Bit 0 - 9:这些 bits 的值用来决定在新帧的开始时，信号 VSYNC 会取多少条线。

注意事项： 无

寄存器定义 9 REG_VSYNC0 定义

REG_VSYNC0 定义	
	读/写
31 10 9	0
地址：0x102448 复位值： 0x000	
Bit 0 - 9:这些 bits 的值决定在新帧的开始时，信号 VSYNC 的高电平状态会取多少条线。	
注意事项： 无	

寄存器定义 10 REG_VSIZE 定义

REG_VSIZE 定义	
保留	读/写
31 10 9	0
地址：0x102444 复位值： 0x110	
Bit 0 - 9: 这些 bits 的值决定一个时间帧里，有多少像素线条。	
注意事项：	

寄存器定义 11 REG_VOFFSET 定义

REG_VOFFSET 定义	
保留	读/写
31 10 9	0
地址：0x102440 复位值： 0x00C	
Bit 0 - 9:这些 bits 的值决定在一个新帧开始之后取多少条线。	
注意事项：	

寄存器定义 12 REG_VCYCLE 定义

REG_VCYCLE 定义	
保留	读/写
31 10 9	0
地址：0x10243C 复位值： 0x124	
Bit 0 - 9:这些 bits 的值决定一个帧里有多少条线。	
注意事项：	

寄存器定义 13 REG_HSYNC1 定义

REG_HSYNC1 定义	
保留	读/写

31 10 9 0

地址：0x102438 复位值： 0x029

Bit 0 - 9:这些 bits 的值决定在线的开始时，有多少 PCLK 时钟周期时间供 HSYNC 使用。

注意事项： 无

寄存器定义 14 REG_HSYNC0 定义

REG_HSYNC0 定义	
保留	读/写

31 10 9 0

地址：0x102434 复位值： 0x0

Bit 0 - 9:这些 bits 的值决定在线的开始时，有多少 PCLK 时钟周期时间，HSYNC 处于高电平状态。

注意事项： 无

寄存器定义 15 REG_HSIZE 定义

请参考 **Error! Reference source not found.**

REG_HSIZE 定义	
保留	读/写
31 10 9	0
地址：0x102430 复位值： 0x1E0	
Bit 0 - 9:这些 bits 的值决定每一条水平线有多少 PCLK 时钟周期时间。	
注意事项： 无	

寄存器定义 16 REG_HOFFSET 定义

请参考 **Error! Reference source not found.**

REG_HOFFSET 定义	
保留	读/写
31 10 9	0
地址：0x10242C 复位值： 0x2B	
Bit0 -9:这些 bits 的值用来决定在像素被扫描输出之前，有多少个 PCLK 时钟周期。	
注意事项： 无	

寄存器定义 17 REG_HCYCLE

请参考 **Error! Reference source not found.**

REG_HCYCLE 定义	
保留	读/写
31 10 9	0

地址：0x102428 复位值： 0x224

Bit0 -9:这些 bits 的值用来决定每一条水平线扫描，总共花多少个 PCLK 时钟周期的时间。预设值为 548，推测可支持 480x272 的屏幕解析度显示。更多细节请参考显示屏幕的规格。

注意事项： 无

寄存器定义 18 REG_TAP_MASK

REG_TAP_MASK 定义	
读/写	
31	0

地址：0x102424 复位值： 0xFFFFFFFF

Bit0 -9:这些 bits 的值可以掩盖 RGB 输出信号的值。这个结果会被用来计算 CRC 值，而 CRC 值会被更新至 REG_TAP_CRC。

注意事项： 无

寄存器定义 19 REG_TAP_CRC 定义

REG_TAP_CRC 定义	
只读	
31	0
地址：0x102420	复位值： 0x00000000
Bit0 - 31:这些 bits 的值是由 FT800 设定，当作 RGB 信号输出的 CRC 值。每次渲染显示清单时，会更新一次。	
注意事项： 无	

寄存器定义 20 REG_DLSWAP 定义

REG_DLSWAP 定义	
保留	读/写

31 2 1 0

地址：0x102450 复位值： 0x00

Bit 0 -1:这些 bits 的值被主机设定，以使 FT800 的显示清单缓冲区生效。FT800 的图形引擎会依据这些 bits 的值，决定什么时候渲染屏幕：

01:当目前的线被扫描输出后，图形引擎会立即渲染屏幕。这样可能会造成撕裂效果。

10:当目前的帧被扫描输出后，图形引擎会立即渲染屏幕。这是在多数情况下建议的作法。

00:这个寄存器不要存入这个数值。

11:这个寄存器不要存入这个数值。

这些 bits 也可以被主机读取，以确认 FT800 显示清单缓冲器的可用性。如果数值读出为零，FT800 显示清单缓冲器是安全的且可写入。否则，主机需要等到值变为零才可写入。

注意事项： 无

寄存器定义 21 REG_TAG 定义

REG_TAG 定义	
保留	只读

31 8 7 0

地址: 0x102478 复位值: 0x0

Bit 0 -7: 这些 bits 会被 FT800 的图形引擎以标记值做更新。这里的标记值是对应到 REG_TAG_X 及 REG_TAG_Y 所给的触摸点坐标值。主机可以读出这个寄存器以确认哪一个图形物件被触摸。

注意：请注意 REG_TAG 与 REG_TOUCH_TAG 的差异。REG_TAG 是依据 REG_TAG_X 及 REG_TAG_Y 所提供的 X、Y 做更新。而 REG_TOUCH_TAG 则是依据由 FT800 触摸引擎提供的目前触摸点做更新。

寄存器定义 22 REG_TAG_Y 定义

REG_TOUCH_Y 定义	
保留	读写

31 9 8 0

地址: 0x102474 复位值: 0x0

Bit 0 – 8: 这些 bits 数的值会被主机设定，当作触摸点的 Y 坐标，这样可以启动主机，使其查询标记值。这个寄存器会跟 REG_TAG_X 及 REG_TAG 一起被使用。正常来说，若是主机已经截取触摸点坐标的情况下，这个寄存器可以被更新以查询对应的触摸点坐标。

注意事项：无

寄存器定义 23 REG_TAG_X 定义

REG_TOUCH_X 定义	
保留	读写

31 9 8 0

地址: 0x102470 复位值: 0x0

Bit 0 – 8: 这些 bits 数的值会被主机设定，当作触摸点的 X 坐标，这样可以启动主机，使其查询标记值。这个寄存器会跟 REG_TAG_Y 及 REG_TAG 一起被使用。正常来说，若是主机已经截取触摸点坐标的情况下，这个寄存器可以被更新以查询对应的触摸点坐标。

注意事项：无

3.2 触屏引擎寄存器 (只用于 FT800)

寄存器定义 24 REG_TOUCH_DIRECT_Z1Z2 定义

REG_TOUCH_DIRECT_Z1Z2 定义			
保留	只读	保留	只读
31 26	25 16	15 10	9 0

地址: 0x102578 复位值: NA

Bit 0-9: 10 bit 模数转换器的值，作为触屏电阻值 Z2。

Bit 16-25: 10 bit 模数转换器的值，作为触屏电阻值 Z1。

注意事项：要知道有否触摸，请检查 REG_TOUCH_DIRECT_XY 第 31 个 bit 的值。FT800 触摸引擎会针对 Z1 及 Z2 的值做后处理，并更新至 REG_TOUCH_RZ。

寄存器定义 25 REG_TOUCH_DIRECT_XY

REG_TOUCH_DIRECT_XY 定义				
只读	**保留**	只读	**保留**	**只读**
31 26	25 16	15 10	9 0	

地址: 0x102574 复位值: 0x0

Bit 0-9: 10 bit 模数转换器的值，作为 Y 坐标的值。

Bit 16-25: 10 bit 模数转换器的值，作为 X 坐标的值。

Bit 31: 如果这个 bit 为 0，表示有检测到一个触摸，而上述两者就是检测到的 Y 坐标及 X 坐标。如果这个 bit 为 1，表示没有检测到触摸，而上述两者可被忽略。

注意事项：

寄存器定义 26 REG_TOUCH_TRANSFORM_F 定义

REG_TOUCH_TRANSFORM_F 定义		
读写		
31 30	16 15	0
地址: 0x102530	复位值: 0x0	

Bit 0 – 15: 这些 bits 数的值表示定点数字的小数部份。

Bit 16 – 30: 这些 bits 数的值表示定点数字的整数部份。

Bit 31:定点数字的符号位

注意事项：这个寄存器代表定点数字，复位后的预设值为+0.0。

寄存器定义 27 REG_TOUCH_TRANSFORM_E 定义

REG_TOUCH_TRANSFORM_E 定义		
读写		
31 30	16 15	0
地址: 0x10252C	复位值: 0x0	

Bit 0 – 15: 这些 bits 数的值表示定点数字的小数部份。

Bit 16 – 30: 这些 bits 数的值表示定点数字的整数部份。

Bit 31 :定点数字的符号位

注意事项：这个寄存器代表定点数字，复位后的预设值为+1.0。

寄存器定义 28 REG_TOUCH_TRANSFORM_D 定义

REG_TOUCH_TRANSFORM_D 定义		
读写		
31 30	16 15	0

地址: 0x102528 复位值: 0x0

Bit 0 – 15: 这些 bits 数的值表示定点数字的小数部份。

Bit 16 – 30: 这些 bits 数的值表示定点数字的整数部份。

Bit 31 :定点数字的符号位

注意事项：这个寄存器代表定点数字，复位后的预设值为+0.0。

寄存器定义 29 REG_TOUCH_TRANSFORM_C 定义

REG_TOUCH_TRANSFORM_C 定义		
读写		
31 30	16 15	0

地址: 0x102524 复位值: 0x0

Bit 0 – 15: 这些 bits 数的值表示定点数字的小数部份。

Bit 16 – 30: 这些 bits 数的值表示定点数字的整数部份。

Bit 31 :定点数字的符号位

注意事项：这个寄存器代表定点数字，复位后的预设值为+0.0。

寄存器定义 30 REG_TOUCH_TRANSFORM_B 定义

REG_TOUCH_TRANSFORM_B 定义		
读写		
31 30	16 15	0

地址: 0x102520 复位值: 0x0

Bit 0 – 15: 这些 bits 数的值表示定点数字的小数部份。

Bit 16 – 30: 这些 bits 数的值表示定点数字的整数部份。

Bit 31 :定点数字的符号位

注意事项：这个寄存器代表定点数字，复位后的预设值为+0.0。

寄存器定义 31 REG_TOUCH_TRANSFORM_A 定义

REG_TOUCH_TRANSFORM_A 定义		
读写		
31 30	16 15	0

地址: 0x10251C 复位值: 0x0

Bit 0 – 15: 这些 bits 数的值表示定点数字的小数部份。

Bit 16 – 30: 这些 bits 数的值表示定点数字的整数部份。

Bit 31 :定点数字的符号位

注意事项：这个寄存器代表定点数字，复位后的预设值为+1.0。

寄存器定义 32 REG_TOUCH_TAG 定义

REG_TOUCH_TAG 定义	
保留	只读
31 8	7 0

地址: 0x102518 复位值: 0

Bit 0 – 7: 这些 bits 是设定来当作特定图形物件的标记值，这些物件是当下在屏幕上正被触摸的物件。当目前帧上所有的线都被扫描输出到屏幕上时，这些 bits 会被更新一次。
Bit 8 – 31: 这些 bits 被保留

注意事项：有效的标记范围是从 1 到 255，因此这个寄存器的预设值是 0，表示预设情况是没有被触摸。

寄存器定义 33 REG_TOUCH_TAG_XY 定义

REG_TOUCH_TAG_XY 定义	
只读	只读
31 16	15 0

地址: 0x102514 复位值: 0

Bit 0 – 15: 这些 bits 数的值是指触屏的 Y 坐标，触屏引擎会用它来查找标记结果。

Bit 16 – 31: 这些 bits 数的值是指触屏的 X 坐标，触屏引擎会用它来查找标记结果。

注意事项：主机可以读取这个寄存器，检查触屏引擎用来更新标记寄存器 REG_TOUCH_TAG 的坐标。

寄存器定义 34 REG_TOUCH_SCREEN_XY 定义

REG_TOUCH_SCREEN_XY 定义	
只读	只读
31 16	15 0

地址: 0x102510 复位值: 0x80008000

Bit 0 - 15: 这些 bits 的值是触屏引擎的 Y 坐标。在做完校准后，会落在屏幕尺寸的高度范围以内。假如触屏没有被触摸，其值是 0x8000。

Bit 16 - 31: 这些 bits 的值是触屏引擎的 X 坐标。在做完校准后，会落在屏幕尺寸的高度范围以内。假如触屏没有被触摸，其值是 0x8000。

注意事项：这个寄存器是 FT800 的触摸引擎最后的计算输出结果。它已经被映射到屏幕的尺寸大小。

寄存器定义 35 REG_TOUCH_RZ 定义

REG_TOUCH_RZ 定义	
保留	只读
31 16	15 0

地址: 0x10250C 复位值: 0x7FFF

Bit 0 - 15: 这些 bits 是触摸触屏的阻值。有效值范围是从 0 到 0x7FFF。最高值 0x7FFF 代表没有触摸，而最低值 0 代表最大的压力。

Bit 16 - 31: 保留

寄存器定义 36 REG_TOUCH_RAW_XY 定义

REG_TOUCH_RAW_XY 定义	
保留	只读

31 16 15 0

地址: 0x102508 复位值: 0xFFFFFFFF

it 0 - 15: 这些 bits 的值表示要进行转换矩阵的原始 Y 坐标，有效范围是从 0 到 1023。若屏幕上没有触摸，这个值为 0xFFFF。

Bit 16 - 31: 这些 bits 的值表示要进行转换矩阵的原始 X 坐标，有效范围是从 0 到 1023。若屏幕上没有触摸，这个值为 0xFFFF。

注意事项：在这个寄存器里的坐标还没被映射到屏幕的坐标。若要取得屏幕坐标，请参考 REG_TOUCH_SCREEN_XY。

寄存器定义 37 REG_TOUCH_RZTHRESH 定义

REG_TOUCH_RZTHRESH 定义	
保留	读/写

31 16 15 0

地址: 0x102504 复位值: 0xFFFF

Bit 0 - 15: 这些 bits 的值控制触屏阻值的阈值。主机可以藉著设定这个寄存器调整触屏的触摸灵敏度。复位后的预设值是 0xFFFF，这是 FT800 触摸引擎可接受的最轻触摸的程度。主机可以透过一些实验方式设定这个寄存器。典型值是 1200。

寄存器定义 38 REG_TOUCH_OVERSAMPLE 定义

REG_TOUCH_OVERSAMPLE 定义	
保留	读/写

31 4 3 0

地址: 0x102500 复位值: 0x7

Bit 0 - 3: 这些 bits 控制触屏的过采样因子。这个寄存器里的值愈高，表示会以较高的功耗得到较高的精确度，但这可能不是必要的。有效范围是从 1 到 15

寄存器定义 39 REG_TOUCH_SETTLE 定义

REG_TOUCH_SETTLE 定义	
保留	读/写

31 4 3 0

地址: 0x1024FC 复位值: 0x3

Bit 0 - 3: 这些 bits 控制触屏的建立时间。单位是 6 个时钟周期。预设值是 3，表示建立时间是 18(3x6)个系统时钟周期。

注意事项：

寄存器定义 40 REG_TOUCH_CHARGE 定义

REG_TOUCH_CHARGE 定义	
保留	读/写

31 16 15 0

地址: 0x1024F8 复位值: 0x1770

Bit 0 -15：这些 bits 控制触屏的充电时间，单位是 6 个系统时钟周期。复位后的预设值是 6000，也就是指充电时间会是 6000x6 个系统时间周期。

注意事项：

寄存器定义 41 REG_TOUCH_ADC_MODE 定义

REG_TOUCH_ADC_MODE 定义	
保留	读写

31 2 1 0

地址: 0x1024F4 复位值: 0x1

Bit 0 -1:主机可以设定这个 bit 去控制 FT800 模数转换器的采样模式，依如下设定：

0：单端模式：这个模式有较低的功能，但是精准度较差。

1：差动模式：这个模式有较高的功能，但是精准度较高。这是复位后的预设模式。

寄存器定义 42 REG_TOUCH_MODE 定义

REG_TOUCH_MODE 定义	
保留	读写

31 2 1 0

地址: 0x1024F0 复位值: 0x3

Bit 0 -1:主机可以设定这个 bit 去控制 FT800 触屏引擎的触屏采样模式，依如下设定：

00：关闭模式：没有采样发生。

01：单次模式：可引起一次的采样。

10：帧模式：可在每个帧的开始引起一次采样。

11：连续模式：每秒达 1000 次的采样，这是复位后的预设值。

3.3 音频引擎寄存器

寄存器定义 43 REG_PLAY 定义

REG_PLAY 定义	
保留	读写

31 2 1 0

地址: 0x1024888 复位值: 0x0

Bit 0: 写入这个 bit 会触发合成音效的播放，该合成音效是由 REG_SOUND 指定。
若从这个 bit 读出 1 值，表示此音效正在播放。若要停止该音效，主机需藉由设定 REG_SOUND，来选择静音音效，并设定此寄存器播放。

注意事项：这个寄存器的细节请参考数据表中”声音合成器”的部份

寄存器定义 44 REG_SOUND 定义

REG_SOUND 定义	
保留	读写
31 16	15 0

地址: 0x102484 复位值: 0x0000

Bit 0 - 15: 这些 bits 用来选择合成的音效，又分作两组，分别是 Bit 0 - 7 及 Bit 8 - 15。

Bit 0 - 7:这些 bits 定义音效。有些音效可以调整音高，而音高是定义在 Bit 8 -15。而有些音效不能调整音高，若不能调整音高，Bit 8 - 15 的值会被忽略。

Bit 8 - 15：音效的 MIDI 音符，指的是定义在 Bit 0 - 7 里的音效。

注意事项：这个寄存器的细节请参考数据表中”声音合成器”的部份。

寄存器定义 45 REG_VOL_SOUND 定义

REG_VOL_SOUND 定义	
保留	读写
31 8	7 0

地址: 0x102480 复位值: 0xFF

Bit 0 - 7:这些 bits 控制合成的声音音量。预设值是 0xFF，指最高音量。若值为 0，表示静音。

注意事项：

寄存器定义 46 REG_VOL_PB 定义

REG_VOL_PB 定义	
保留	读写

31 8 7 0

地址: 0x10247C 复位值: 0xFF

Bit 0 - 7：这些 bits 控制播放音频档案的音量。预设值是 0xFF，表示最高音量。若值为 0，表示静音。

注意事项：

寄存器定义 47 REG_PLAYBACK_PLAY 定义

REG_PLAYBACK_PLAY 定义	
保留	读写

31 1 0

地址: 0x1024BC 复位值: 0x0

Bit 0：写入这个 bit 会触发音频的播放，无论写入'0'或是'1'。当播放正在进行，可读出'1'。当播放结束，可读出'0'。

注意事项： **这个寄存器的细节请参考数据表中"音频播放"的部份**

寄存器定义 48 REG_PLAYBACK_LOOP 定义

REG_PLAYBACK_LOOP 定义	
保留	读写

31 1 0

地址: 0x1024B8 复位值: 0x0

Bit 0：一旦播放完所有数据，这个 bit 可控制音频引擎，使其再从起始地址播放 RAM_G 里的音频数据。值为 1 表示 LOOP 功能是开启的，若值为 0 表示 LOOP 功能关闭。

注意事项： **这个寄存器的细节请参考数据表中"音频播放"的部份**

寄存器定义 49 REG_PLAYBACK_FORMAT 定义

REG_PLAYBACK_FORMAT 定义	
保留	读/写

31 2 1 0

地址: 0x1024B4 复位值: 0x0

Bit 0 - 1：这些 bits 定义在 RAM_G 里音频数据的格式。FT800 支持：

00：Linear 线性采样格式

01：uLaw 采样格式

10：4 bit IMA ADPCM 采样格式

11：未定义

注意事项： **这个寄存器的细节请参考数据表中"音频播放"的部份**

寄存器定义 50 REG_PLAYBACK_FREQ 定义

REG_PLAYBACK_FREQ 定义	
保留	只读
31 16	15 0

地址: 0x1024B0 复位值: 0x1F40

Bit 0 - 15: 这些 bits 指定音频播放数据的采样频率。单位是赫兹。

注意事项：更多细节请参考数据表中”音频播放”的部份。

寄存器定义 51 REG_PLAYBACK_READPTR 定义

REG_PLAYBACK_READPTR 定义	
保留	只读
31 20	19 0

地址: 0x1024AC 复位值: 0x00000

Bit 0 -19:当正从 RAM_G 播放音频数据时，这些 bits 会被 FT800 音频引擎更新。这是正在播发时，当下的音频数据地址。主机可以读取这个寄存器检查是否音频引擎已经消耗完所有的音频数据。

注意事项：更多细节请参考数据表中”音频播放”的部份。

寄存器定义 52 REG_PLAYBACK_LENGTH 定义

REG_PLAYBACK_LENGTH 定义	
保留	读/写

31 20 19 0

地址: 0x1024A8 复位值: 0x00000

Bit 0 -19:这些 bits 指定在 RAM_G 里的音频数据要播放的长度，开始是从 REG_PLAYBACK_START 寄存器所指定的地址开始。

注意事项：更多细节请参考数据表中”音频播放”的部份。

寄存器定义 53 REG_PLAYBACK_START 定义

REG_PLAYBACK_START 定义	
保留	读/写

31 20 19 0

地址: 0x1024A4 复位值: 0x00000

Bit 0 -19:i 这些 bits 指定在 RAM_G 里，所要播放的音频数据的开始地址。

注意事项：更多细节请参考数据表中”音频播放”的部份。

3.4 协处理器引擎寄存器

寄存器定义 54 REG_CMD_DL 定义

REG_CMD_DL 定义	
保留	读/写

31 14 13 0

地址: 0x1024EC 复位值: 0x0000

Bit 0 – 13:这些 bits 的值指的是一个显示清单指令的 RAM_DL 偏移量(offset)，显示清单指令是产生于协处理器引擎。协处理器引擎会依照这些 bits 判定显示清单缓冲器里的地址，显示清单缓冲器是属于产生的显示清单指令的。只要有显示清单指令被产生在显示清单缓冲器，协处理器引擎会更新这个寄存器。藉由适当地设定这个寄存器，主机可以指定显示清单缓冲器里的开始地址，以供协处理器引擎产生显示指令。有效范围是 0 到 8195。

注意事项：

寄存器定义 55 REG_CMD_WRITE 定义

REG_CMD_WRITE 定义	
保留	读/写

31 12 11 0

地址: 0x1024E8 复位值: 0x0

Bit 0 – 11:这些 bits 被 MCU 主机更新，以通知协处理器引擎将有效数据的结束地址喂入其 FIFO。通常来说，主机会在下载完协处理器指令到其 FIFO 之后，更新这个寄存器。有效范围是 0 到 4095，也就是在 FIFO 的大小范围内。

注意事项： 指令缓冲器的 FIFO 大小是 4096 个字节，而每个协处理器的指令大小是 4 个字节。所有写入这个寄存器的值需要对齐 4 个字节的整数倍。

寄存器定义 **56 REG_CMD_READ** 定义

REG_CMD_READ 定义	
保留	读/写
31 12	11 0

地址: 0x1024E4 复位值: 0x000

Bit 0 – 11:只要协处理器引擎从其 FIFO 取得指令，这些 bits 就被协处理器引擎更新。主机可读取此寄存器，以判定协处理器引擎的 FIFO 是否满载。有效范围从 0 到 4095。在有错的情况下，协处理器会写入 0xFFF 到这个寄存器。

注意事项：除非在错误回复的情况，主机不应写入这个寄存器。在协处理器引擎复位后，其预设值为 0。

寄存器定义 **57 REG_TRACKER** 定义

REG_TRACK 定义	
只读	
追踪值	标记值
31 16	15 0

地址: 0x109000 复位值: 0x0

Bit 0 – 15: 这些bits用来表示正被触摸的图形物件的标记值。

Bit 16 – 31: 这些bits是设定来标示被追踪图形物件的追踪值。协处理器可在预定义的范围里，计算目前的触摸点有多少。更多细节请参考CMD_TRACK。

注意事项： 无

3.5 其它寄存器

在这个章节里，其它寄存器包含背光控制、中断、GPIO、及其他功能的寄存器。

寄存器定义 58 REG_PWM_DUTY 定义

REG_PWM_DUTY 定义	
保留	读/写

31 8 7 0

地址: 0x1024C4 复位值: 0x80

Bit 0 - 7：这些bits定义背光脉宽调变(PWM)输出的占空比。有效范围是从0到128。0代表背光完全关闭，128代表背光是在最大的亮度。

注意事项：

寄存器定义 59 REG_PWM_HZ 定义

REG_PWM_HZ 定义	
保留	读/写

31 14 13 0

地址: 0x1024C0 复位值: 0xFA

Bit 0 - 13：这些bits定义背光脉宽调变输出的频率，单位为赫兹。复位后预设值为250赫兹。有效频率范围是从250赫兹到10000赫兹

注意事项：

寄存器定义60 REG_INT_MASK 定义

REG_INT_MASK 定义	
保留	读/写

31 8 7 0

地址: 0x1024A0 复位值: 0xFF

Bit 0 - 7 : 这些bits是用来遮盖对应的中断言号。1表示启动对应的中断源，0表示关闭对应的中断源。在复位后，预设上所有中断源都可合格地用来触发中断。

注意事项：更多细节请参考数据表的"中断"部份

寄存器定义61 REG_INT_EN 定义

REG_INT_EN 定义	
保留	读/写

31 1 0

地址: 0x10249C 复位值: 0x0

Bit 0 :主机可以设定这个bit为1，以启动FT800全域中断。若要关闭FT800全域中断，主机可将此bit设为0

注意事项：此寄存器的细节请参考数据表的"中断"部份

寄存器定义 62 REG_INT_FLAGS 定义

REG_INT_FLAGS 定义	
保留	读/写

31 8 7 0

地址: 0x102498 复位值: 0x00

Bit 0 - 7: 这些bits是FT800设定的中断标记。主机可以读取这些bits，以判定哪一个中断发生。这些bits在读取后会自动清除。主机不应写入此寄存器。在复位后，预设上没有中断发生，因此预设值为0x00。

注意事项：更多细节请参考数据表的"中断"部份

寄存器定义 63 REG_GPIO 定义

REG_GPIO 定义	
保留	读/写

31 8 7 0

地址: 0x102490 复位值: 0x00

Bit 0 -7: 这些bits是多用途的。Bit 0, 1, 7用来控制GPIO引脚的值。
Bit 2 -6: 这些bits是用来配置引脚的驱动能力大小。

注意事项：更多细节请参考数据表的"通用IO引脚"部份

寄存器定义 64 REG_GPIO_DIR 定义

REG_GPIO_DIR 定义	
保留	读/写

31 8 7 0

地址: 0x10248C 复位值: 0x80

Bit 0 – 7: 这些 bits 配置 FT800 的 GPIO 引脚方向。Bit 0 控制 GPIO0 的方向，Bit 7 控制 GPIO7 的方向。bit 值为 1 表示 GPIO 引脚设定成输出，否则则为输入。复位后，预设上只有 GPIO7 是设为输出。

寄存器定义 65 REG_CPURESET 定义

REG_CPURESET 定义	
保留	读/写

31 1 0

地址: 0x10241C 复位值: 0x00

Bit 0: 若将这个 bit 写入 1 将会使 FT800 的协处理器引擎成复位状态。若写入 0，会从复位状态继续到正常操作模式。若从这个 bit 读出 1，表示 FT800 的协处理器引擎在复位状态，否则，FT800 的协处理器引擎则在正常状态。

Bit 1 – 31: 保留

寄存器定义 66 REG_SCREENSHOT_READ 定义

REG_SCREENSHOT_READ 定义	
保留	读/写

31 1 0

地址: 0x102554 复位值: 0x0

Bit 0: 将这个 bit 设为 1，以启动选定 Y 线截屏的读出

Bit 1 – 31: 保留

注意事项：当 REG_SCREENSHOT_BUSY 寄存器清除之后，在读出选定 Y 线截屏之前，这个 bit 需要先设定好。截屏会存在 RAM_SCREENSHOT，而每个像素的格式是 32 bit BGRA 格式：蓝通道是最低地址，Alpha 透明是最高地址。

寄存器定义 67 REG_SCREENSHOT_BUSY 定义

REG_SCREENSHOT_BUSY 定义
只读

63 0

地址: 0x1024D8 复位值: 0x0

Bit 0 - 63: 屏幕截图忙录中的标记。这 64 bits，若为任何的非零值代表屏幕截图的忙录状态。这 64 bits 若为零值，表示屏幕截图已经完成。

注意事项：注意事项：屏幕截图开始之后，主机应读取此寄存器，以判定屏幕截图何时会结束

寄存器定义 68 REG_SCREENSHOT_START 定义

REG_SCREENSHOT_START 定义	
保留	读/写

31 1 0

地址: 0x102418 复位值: 0x0

Bit 0: 若屏幕截图功能已启动(enabled)，设此 bit 为 1 开始(start)屏幕截图。当屏幕截图功能关闭(disabled)时，屏幕截图会自动停止(stop)。

Bit 1 – 31: 保留

注意事项：无

寄存器定义 69 REG_SCREENSHOT_Y 定义

REG_SCREENSHOT_Y 定义	
保留	读/写

31 9 8 0

地址: 0x102414 复位值: 0x000

Bit 0 - 8: 这9 bits指定当屏幕截图功能启动(enabled)时，在水平方向要截取多少线
Bit 9 - 31: 保留

注意事项：无

寄存器定义 70 REG_SCREENSHOT_EN 定义

REG_SCREENSHOT_EN 义	
保留	读/写

31 1 0

地址: 0x102410 复位值: 0x000

Bit 0： 将此bit设为1可启动(enable)目前帧的屏幕截屏功能。清除这个bit则关闭屏幕截屏功能。
Bit 1 - 31: 保留

注意事项：无

寄存器定义 71 REG_FREQUENCY 定义

REG_FREQUENCY 定义
读/写

31 0

地址: 0x10240C 复位值: 0x2DC6C00

Bit 0 - 31: 在复位后，这些bits会设定为0x2DC6C00，也就是指，预设的主时钟频率是48 MHz。值的单位是赫兹。如果主机利用主机命令CLK36M选择其他替代的频率，这个寄存器也需要一起被更新

寄存器定义 72 REG_CLOCK 定义

REG_CLOCK 定义
只读
31 0 地址: 0x102408 复位值: 0x00000000 Bit 0 - 31: 在复位后这些bits会被设为零。此寄存器会计算复位之后，FT800主时钟周期的数量。若FT800的主时钟频率为48 MHz，此寄存器约在89秒后，数完一个循环。

寄存器定义 73 REG_FRAMES 定义

REG_FRAMES 定义
只读
31 0
地址: 0x102404 复位值: 0x00000000
Bit 0 - 31: 这些bits在复位后会设定为0。此寄存器会计算屏幕帧的数量。假如刷新速率是60赫兹，复位后数到约828天，会数完一个循环。

寄存器定义 74 REG_ID 定义

REG_ID 定义	
保留	读/写
31 8	7 0
地址: 0x102400 复位值: 0x7C	
Bit 0 - 7: 这些bits是内置的寄存器ID。主机可以读取这个寄存器以判定此芯片是否为FT800。其值应该要一直是0x7C。	

寄存器定义 75 REG_TRIM 定义

REG_TRIM 定义	
保留	读/写

31 5 4 0

地址: 0x10256C 复位值: 0x0

Bit 0 - 4： 这些bits是设定来修剪内部时钟
Bit 5 - 31: 保留

注意事项：更多细节请参考操作说明书 AN_299_FT800_FT801_Internal_Clock_Trimming

4 显示清单指令

FT800 的图形引擎从显示清单的存储器 RAM_DL 里，以指令的形式取得操作指示。每个指令长度是 4 字节，因为 RAM_DL 的尺寸是 8K 字节，所以一个显示清单可以被填满到 2048 个指令。FT800 的图形引擎会依据指令的定义，执行对应的操作。

4.1 图形状态

控制绘图的图形状态存在图形上下文(graphics context)里。个别状态可以由适当的显示清单指令改变(如：COLOR_RGB)，而整体的状态可利用 SAVE_CONTEXT 及 RESTORE_CONTEXT 指令做储存及回复。

要注意位图的绘图状态是特别的：虽然位图句柄是图形上下文的一部份，但每一个位图句柄的参数并不是图形上下文的一部份。它们无法由 SAVE_CONTEXT 储存也无法由 RESTORE_CONTEXT 回复。这些参数是经由 BITMAP_SOURCE、BITMAP_LAYOUT、及 BITMAP_SIZE 这些指令改变的。一旦这些参数被设立，它们就可被显示清单所使用，直到参数值被改变。

SAVE_CONTEXT 及 RESTORE_CONTEXT 由一个 4 层的堆栈所组成，以及目前的图形上下文。下表列出图形上下文里详细的参数。

表 3 图形上下文

参数	预设值	指令
Func & ref	ALWAYS, 0	ALPHA_FUNC
func & ref	ALWAYS, 0	STENCIL_FUNC
Src & dst	SRC_ALPHA, ONE_MINUS_SRC_ALPHA	BLEND_FUNC
单元格(cell)值	0	CELL
Alpha 值	0	COLOR_A
红、蓝、绿颜色	(255,255,255)	COLOR_RGB
线的宽度，以 1/16 像素为单	16	LINE_WIDTH
点的大小，以 1/16 像素为单位	16	POINT_SIZE
剪刀的宽度及高度	512,512	SCISSOR_SIZE
剪刀的字符串坐标	(x,y) = (0,0)	SCISSOR_XY
目前位图的句柄	0	BITMAP_HANDLE
位图转变系数	+1.0,0,0,0,+1.0,0	BITMAP_TRANSFORM_A-F
模板清除值	0	CLEAR_STENCIL
标记清除值	0	CLEAR_TAG
模板的遮盖值	255	STENCIL_MASK
spass & sfail	KEEP,KEEP	STENCIL_OP

参数	预设值	指令
标记缓冲器值	255	TAG
标记遮盖值	1	TAG_MASK
Alpha 透明清除值	0	CLEAR_COLOR_A
RGB 清除颜色	(0,0,0)	CLEAR_COLOR_RGB

这个部份的每一个显示清单指令有列出每一个它会设定的图形上下文

4.2 指令编码

每个显示清单指令有一个 32-bit 编码。编码的最高几位判定指令。指令参数(如果有的话)在最低几位。任何遮盖保留的 bits 必须为零。

FT800 支持的图形基元及其对应的值在下表。
表 4 FT800 图形基元清单

原始图形	原始值
BITMAPS	1
POINTS	2
LINES	3
LINE_STRIP	4
EDGE_STRIP_R	5
EDGE_STRIP_L	6
EDGE_STRIP_A	7
EDGE_STRIP_B	8
RECTS	9

各种 FT800 支持的位图格式及相对应的值会在下面提到。

表 5 图形位图格式表

位图格式	位图格式值
ARGB1555	0
L1	1
L4	2
L8	3
RGB332	4
ARGB2	5
ARGB4	6
RGB565	7
PALETTED	8
TEXT8X8	9
TEXTVGA	10
BARGRAPH	11

4.3 指令组

4.3.1 设定图形状态

ALPHA_FUNC 设定 alpha 测试功能

BITMAP_HANDLE 设定位图句柄

BITMAP_LAYOUT 为目前的句柄设定源头位图存储器格式及布局

BITMAP_SIZE 为目前句柄设定位图的屏幕绘制尺寸

BITMAP_SOURCE 为位图图形设定源头地址

BITMAP_TRANSFORM_A-F 设定位图转换矩阵的元素

BLEND_FUNC 设定像素算法

CELL 为 VERTEX2F 指令设定位图单元格数字

(编者按：一个 handle 可以有多个 cell)

CLEAR 清除缓冲器成预定(preset)值。

CLEAR_COLOR_A 为 alpha 通道设定清除值(clear value)

CLEAR_COLOR_RGB 为红绿蓝通道设定清除值(clear value)

CLEAR_STENCIL 为模板缓冲器设定清除值(clear value)

CLEAR_TAG 为标记缓冲器设定清除值(clear value)

COLOR_A 设定目前颜色的 alpha 通道值

COLOR_MASK	启动或关闭写入颜色元件
COLOR_RGB	设定目前颜色红绿蓝
LINE_WIDTH	设定线的宽度
POINT_SIZE	设定点的大小
RESTORE_CONTEXT	从上下文的栈恢复目前的图形上下文
SAVE_CONTEXT	将目前的图形上下文推入上下文堆栈
SCISSOR_SIZE	设定剪刀修剪矩形的大小
SCISSOR_XY	设定剪刀修剪矩形的左上角
STENCIL_FUNC	为模板测试设定功能及参考值
STENCIL_MASK	控制位于模板平面上各别 bits 的写入
STENCIL_OP	设定模板测试动作
TAG	设定目前标记值
TAG_MASK	控制标记缓冲器的写入

4.3.2 绘图动作

BEGIN	开始绘制一个图形基元
END	结束绘制一个图形基元
VERTEX2F	以小数坐标提供一个顶点
VERTEX2II	以正整数坐标提供一个顶点

4.3.3 赴行控制

JUMP	在显示清单另一个位置执行多个指令序列
MACRO	从一个宏寄存器(macro register)执行一个单一指令
CALL	在显示清单另一个位置执行一个指令的序列
RETURN	从先前的 CALL 指令返回
DISPLAY	结束显示清单

4.4 ALPHA_FUNC

指定 alpha 测试功能

编码

31 24	23 11	10 8	7	6	5	4	3	2	1	0
0x09	保留	**func**	**ref**							

参数

func

指定测试功能，指定以下其中一项：NEVER、LESS、LEQUAL、GREATER、GEQUAL、EQUAL、NOTEQUAL、或 ALWAYS。 初始值是 ALWAYS(7)

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

图 8: ALPHA_FUNC 的常数

ref

指定 alpha 测试的参考值。初始值为 0。

图形上下文

func 及 ref 的值是图形上下文的一部份，如 **Error! Reference source not found.**节所说。

请参阅

无

4.5 BEGIN

开始绘制一个图形基元

编码

31 24	23 4	3	2	1	0
0x1F	保留	prim			

参数

prim

图形基元。有效值定义如下：

表 6 FT800 图形基元操作定义

名称	值	描述
BITMAPS	1	位图绘图基元
POINTS	2	点绘图基元
LINES	3	线绘图基元
LINE_STRIP	4	线条带绘图基元
EDGE_STRIP_R	5	沿边衬条右绘图基元
EDGE_STRIP_L	6	沿边衬条左边绘图基元
EDGE_STRIP_A	7	沿边衬条上绘图基元
EDGE_STRIP_B	8	沿边衬条下绘图基元
RECTS	9	矩形绘图基元

描述

全部 FT800 支持的基元都定义在上表。透过 BEGIN 指令，选择要绘制的基元。一旦基元被选定，它会一直线持有效，直到一个新的基元再被 BEGIN 指令选择。

请注意，基元绘制的操作还不会被执行，要一直到执行 VERTEX2II 或 VERTEX2F 指令才会绘制。

范例

绘制点、线、及位图：


```
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) );
```

图形上下文

无

请参阅

END

4.6 BITMAP_HANDLE

指定位图的句柄(handle)

编码

31 24	23 5	4	3	2	1	0
0x05	保留	句柄				

参数

handle

位图句柄初始值是 0。 有效范围是从 0 到 31。

描述

句柄 16 到 31 是由 FT800 定义，供内置字体所使用，而句柄 15 是定义在协处理器引擎的指令 CMD_GRADIENT、CMD_BUTTON、及 CMD_KEYS。使用者可以从句柄 0 到 14 定义新的位图。若目前的显示清单没有协处理器引擎指令 CMD_GRADIENT、CMD_BUTTON、及 CMD_KEYS，使用者甚至可以利用句柄 15 定义一个位图。

图形上下文

句柄的值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

BITMAP_LAYOUT, BITMAP_SIZE

4.7 BITMAP_LAYOUT

为了目前的句柄，指定源头位图存储器格式及布局。

编码

31 24	23	22	21	20	19	18 9	8 0
0x07	format					linestride	Height

参数

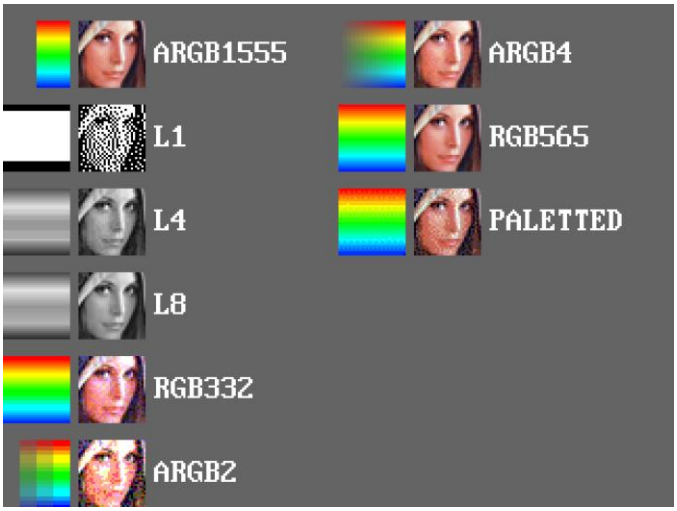
format

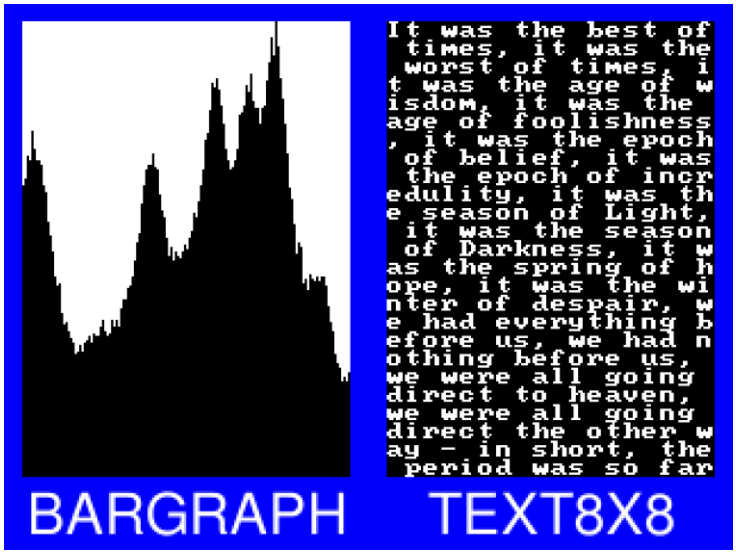
位图像素格式。有效范围是从 0 到 11，按照下表定义。

表 7 BITMAP_LAYOUT 格式清單

NAME	VALUE
ARGB1555	0
L1	1
L4	2
L8	3
RGB332	4
ARGB2	5
ARGB4	6
RGB565	7
PALETTED	8
TEXT8X8	9
TEXTVGA	10
BARGRAPH	11

各种支持的位图格式为：





BARGRAPH - 渲染数据成一个长条图。在一个字节阵列里查询 x 坐标后，如果字节的值小于 y，会给出一个非透明像素。否则，则给出一个透明的像素。这个结果是一个位图数据的长条图。使用 BARGRAPH 格式绘制，最大可画 256x256 大小的位图。图形的方向、宽度、及高度可以利用位图变换矩阵更改。

TEXT8X8 – 在一个固定的 8x8 字体里查找。位图是一个存在于图形 RAM 字节阵列，而每个字节索引到内部的 8x8 CP437 [2]字体(内置字体度句柄 16 & 17 是用来绘制 TEXT8X8 格式)。结果是位图就像字符网格一样。可以绘制单一个位图，此单一位图可以涵盖全部或是部份的显示；在位图数据里的每一个字节对应到一个 8x8 像素的字符单元格。

TEXTVGA – 以 TEXTVGA 语法在一个固定的 8x16 字体里查找。位图是一个存在图形随机存取存储器里，每一个元素索引到一个内部的 8x16 CP437 [2]字体里(内置字型位图句柄 18 & 19 是用来绘制 TEXTVGA 格式，包含控制信息，如背景颜色，前景颜色，及游标等等)。结果是位图就像一个 TEXTVGA 网格一样。可以绘制一个单一位图，而此单一位图可涵盖全部或是部份的显示；在位图里的每一个 TEXTVGA 数据类型对应到一个 8x16 像素的字符单元格。PALETTED - 位图字节是索引到一个调色板的表格。这个调色板表格含有 32-bit 的 RGBA 色产，故利用一个调色板表格可以省去很多存储器。这个 256 色的调色板存在一个专有的 1K(256x4)字节的寄存器 RAM_PAL 里。

linestride

位图的线跨，以字节(byte)为单位。请注意以下所叙述的对齐要求。

height

位图高度，以线为单位。

描述

支持的位图式为 L1、L4、 L8、 RGB332、 ARGB2、 ARGB4、ARGB1555、 RGB565 及 Palette。

对于 L1 来说，线跨必须是 8 字节的整数倍；对而 L4 格式来说，线跨必须是 2 个半字节的整数倍(也就是对齐成一字节的整数数)。

更多有关对齐的细节，可参考下图：

L1 format layout		Byte Order
Pixel 0	Bit 7	Byte 0
Pixel 1	Bit 6	
.....		
Pixel 7	Bit 0	

L4 format layout		Byte Order
Pixel 0	Bit 7-4	Byte 0
Pixel 1	Bit 3-0	

L8 format layout		Byte Order
Pixel 0	Bit 7-0	Byte 0
pixel 1	Bit 15-8	Byte 1
pixel 2	Bit 23-16	Byte 2

图 **9: L1/L4/L8** 的像素格式

ARGB2 format layout		Byte Order
A	Bit 7-6	Byte 0
R	Bit 5-4	
G	Bit 3-2	
B	Bit 1-0	

ARGB1555 format layout		Byte Order
A	Bit 15	Byte 1 Byte 0
R	Bit 14-10	
G	Bit 9- 5	
B	Bit 4-0	

图 **10: ARGB2/1555** 的像素格式

ARGB4 format layout		Byte Order
A	Bit 15-12	Byte 1
R	Bit 11-8	
G	Bit 7-4	Byte 0
B	Bit 3-0	

RGB332 pixel layout		Byte Order
R	Bit 7-5	Byte 0
G	Bit 4-2	
B	Bit 1-0	

RGB565 format layout		Byte Order
R	Bit 15-11	Byte 1 Byte 0
G	Bit 10-5	
B	Bit 4-0	

Palette format layout		Byte Order
A	Bit 31-24	Byte 3
R	Bit 23-16	Byte 2
G	Bit 15-8	Byte 1
B	Bit 7-0	Byte 0

图 **11: ARGB4**、**RGB332**、**RGB565** 的像素格式及调色板

图形上下文

无

请参阅

BITMAP_HANDLE, BITMAP_SIZE, BITMAP_SOURCE

4.8 BITMAP_SIZE

为目前的句柄指定位图的屏幕绘制。

编码

31 24	23 21	20	19	18	17 9	8 0
0x08	4.8.1 保留	filter	wrapx	wrapy	width	height

参数

filter

位图过滤模式，NEAREST 或 BILINEAR 其中一种。

NEAREST 的值为 0，而 BILINEAR 的值为 1。

wrapx

位图 x 方向缠绕模式，REPEAT 或是 BORDER 其中一种。

BORDER 的值为 0 而 REPEAT 的值为 1。

wrapy

位图 y 方向缠绕模式，REPEAT 或是 BORDER 其中一种。

width

已画的位图宽度，单位为像素。

height

已画的位图高度，单位为像素。

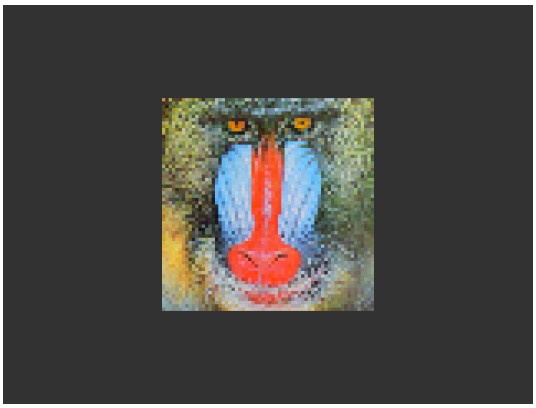
描述

这个指令控制位图的绘制：位图屏幕上的尺寸、缠绕行为、以及过滤功能。请注意如果参数 wrapx 或 wrapy 是 REPEAT，则对应的存储器布局尺寸 (BITMAP_LAYOUT 线跨或高度)必须是 2 的次方，否则结果为未知。

至于宽度及高度参数，若值是 1 到 511 是表示位图以像素为单位的宽度或高度。若值为 0 表示宽度或高度为 512 个像素。

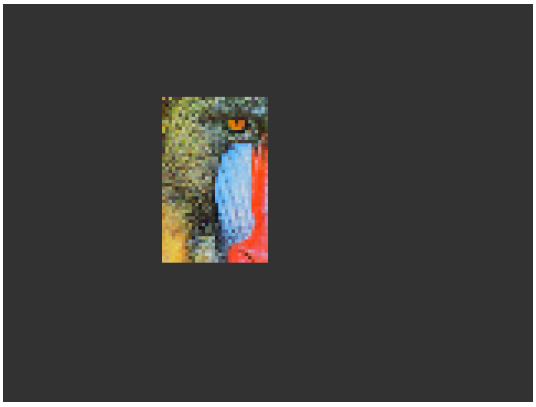
范例

绘制一个 64 x 64 的位图：



```
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) );
```

尺寸缩小成 32 x 50:

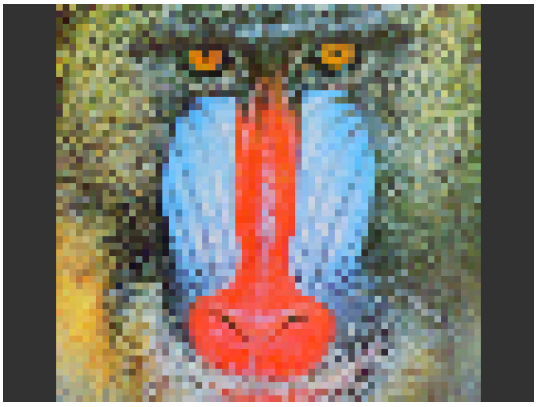


```
dl( BITMAP_SOURCE(0) );
dl( BITMAP_LAYOUT(RGB565, 128, 64) );
dl( BITMAP_SIZE(NEAREST, BORDER, BORDER, 32, 50) );
dl( BEGIN(BITMAPS) );
dl( VERTEX2II(48, 28, 0, 0) );
```

使用循环模式履盖位图：

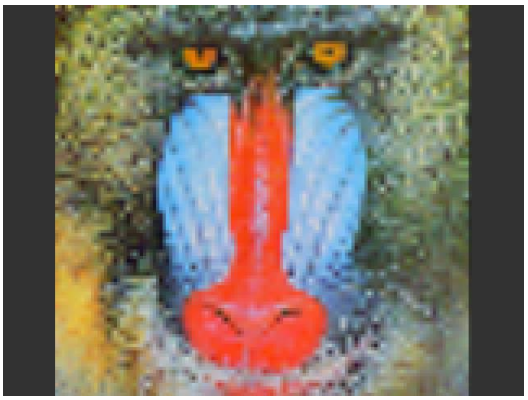
```
dl( BITMAP_SOURCE(0) );
dl( BITMAP_LAYOUT(RGB565, 128, 64) );
dl( BITMAP_SIZE(NEAREST, REPEAT, REPEAT, 160, 120) );
dl( BEGIN(BITMAPS) );
dl( VERTEX2II(0, 0, 0, 0) );
```

放大 4X - 128 X 128 – 使用位图变换：



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

使用双线性过滤器可使放大后的影像较平滑：



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

图形上下文

无

请参阅

BITMAP_HANDLE, BITMAP_LAYOUT, BITMAP_SOURCE

4.9 BITMAP_SOURCE

指定 FT800 图形存储器 RAM_G 里，位图数据的源头地址。

编码

31 24	23 20	19 0
0x01	保留	addr

参数

addr

在 SRAM FT800 里的位图地址，会依据位图格式对齐。

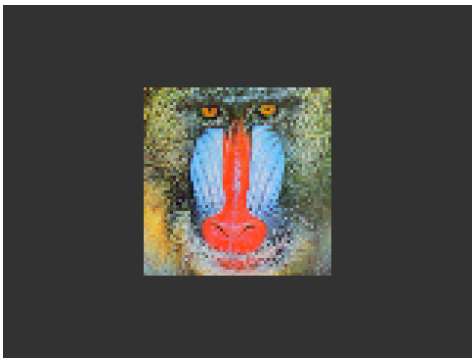
例如，如果位图格式是属于 RGB565/ARGB4/ARGB1555 其中之一，位图源头应该要对齐成 2 字节的整数倍。

描述

位图的源头地址正常来说是在主存储器里，主存储器里是位图图形数据载入的地方。

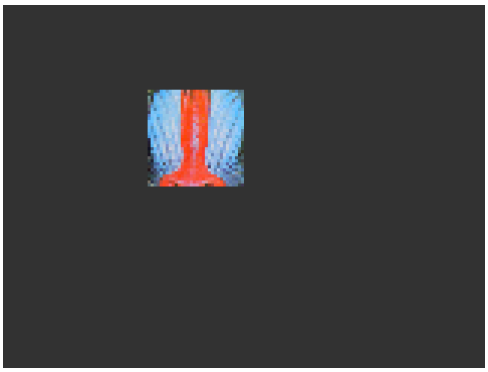
范例

绘制一个 64 x 64 大小的位图，在地址 0 的位置载入：



```
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) );
```

使用同一个图形数据，但源头及尺寸改成只显示出 32 x 32 的部份图形：



```
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) );
```

图形上下文

无

请参阅

BITMAP_LAYOUT, BITMAP_SIZE

4.10 BITMAP_TRANSFORM_A

指定位图变换矩阵的 A 系数

编码

31 24	23 17	16 0
0x15	保留	a

参数

a

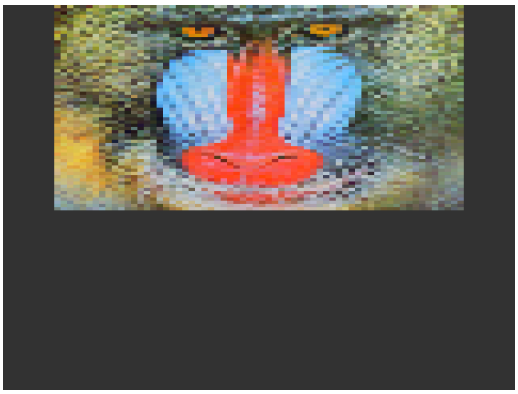
位图变换矩阵的系数 A，形式是有符号的 8.8 bit 定点。初始值是 256。

描述

BITMAP_TRANSFORM_A-F 系数是用来表现位图变换的功能，例如缩放、旋转、及移动。这些功能与 openGL 变换功能相似。

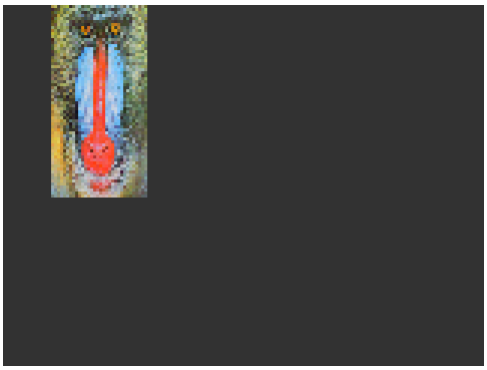
范例

值为 0.5(128)，使得位图宽度变为两倍：



```
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) );
```

值为 2.0(512)，会使得位图宽度变为一半：



```
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) );
```

图形上下文

A 值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

无

4.11 BITMAP_TRANSFORM_B

指定位图变换矩阵的 B 系数

编码

31 24	23 17	16 0
0x16	保留	b

参数

b

位图变换矩阵的系数 B，形式是有符号的 8.8 bit 定点。初始值是 0。

描述

BITMAP_TRANSFORM_A-F 系数是用来表现位图变换的功能，例如缩放、旋转、及移动。这些功能与 openGL 变换功能相似。

图形上下文

B 值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

无

4.12 BITMAP_TRANSFORM_C

指定位图变换矩阵的 C 系数

编码

31 24	23 0
0x17	c

参数

c

位图变换矩阵的系数 C，形式是有符号的 15.8 bit 定点。初始值是 0。

描述

BITMAP_TRANSFORM_A-F 系数是用来表现位图变换的功能，例如缩放、旋转、及移动。这些功能与 openGL 变换功能相似。

图形上下文

C 值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

无

4.13 BITMAP_TRANSFORM_D

指定位图变换矩阵的 D 系数

编码

31 24	23 17	16 0
0x18	保留	d

参数

d

位图变换矩阵的系数 D，形式是有符号的 8.8 bit 定点。初始值是 0。

描述

BITMAP_TRANSFORM_A-F 系数是用来表现位图变换的功能，例如缩放、旋转、及移动。这些功能与 openGL 变换功能相似。

图形上下文

D 值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

无

4.14 BITMAP_TRANSFORM_E

指定位图转换矩阵的 E 系数

编码

31 24	23 17	16 0
0x19	保留	e

参数

e

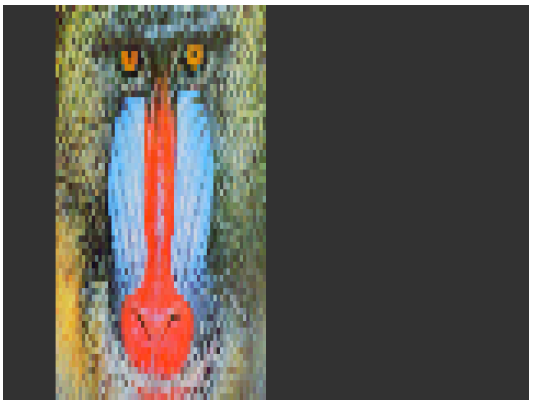
位图变换矩阵的系数 E，形式是有符号的 8.8 bit 定点。初始值是 256。

描述

BITMAP_TRANSFORM_A-F 系数是用来表现位图变换的功能，例如缩放、旋转、及移动。这些功能与 openGL 变换功能相似。

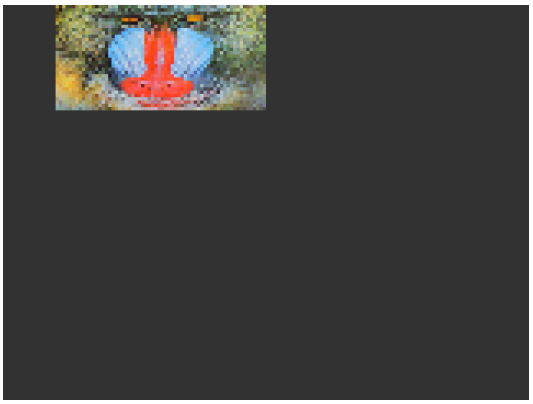
范例

值为 2.0(512)，会使得位图高度变成两倍：



```
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) );
```

值为 2.0(512)，会使得位图高度变为一半：



```
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) );
```

图形上下文

e 值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

无

4.15 BITMAP_TRANSFORM_F

指定位图变换矩阵的 F 系数

编码

31 24	23 0
0x1A	f

参数

f

位图变换矩阵的系数 F，形式是有符号的 15.8 bit 定点。初始值是 0。

描述

BITMAP_TRANSFORM_A-F 系数是用来表现位图变换的功能，例如缩放、旋转、及移动。这些功能与 openGL 变换功能相似。

图形上下文

F 值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

无

4.16 BLEND_FUNC

指定像素算术

编码

31 24	23 6	5 3	2 0
0x0B	保留	src	dst

参数

src

指定源头混合因子如何计算。是下列方法的其中一种：ZERO,ONE、SRC_ALPHA、DST_ALPHA、ONE_MINUS_SRC_ALPHA、或 ONE_MINUS_DST_ALPHA。初始值是 SRC_ALPHA(2).

dst

指定混合因子如何计算。目的地因子跟源头混合因子一样，是上述所列方法的其中一种初始值是 ONE_MINUS_SRC_ALPHA(4)

表 8 BLEND_FUNC 常数值定义

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

描述

混合功能控制新的颜色值如果与已经在颜色缓冲器里的值做结合。若有一个像素值的源头及一个已在颜色缓冲器先前值的目的地，则计算后的颜色为：

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

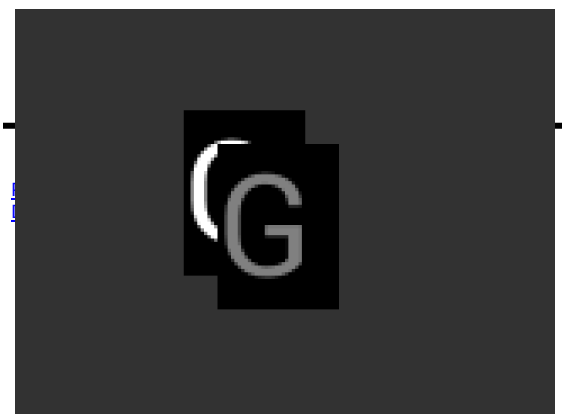
对每一个颜色通道: 红、绿、蓝、及 alpha 透明。

范例

预设的(SRC_ALPHA, ONE_MINUS_SRC_ALPHA)混合功能可利用 alpha 透明值使绘图覆盖目的地：

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

0 值的目的地因子表示目的地像素没有被使用：



```
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) );
```

利用源头的 alpha 透明值控制要保留多少目的地的部份：

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

图形上下文

src 值及 dst 值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

COLOR_A

4.17 CALL

在显示清单里另一个位置执行一序列的指令

编码

31 24	23 16	15 0
0x1D	保留	dest

参数

dest

RAM_DL 里的目的地地址，是显示指令會切換至的位置。而 FT800 有栈储存返回地址，若要返回源头地址的下一个指令，可利用 RETURN 指令。

描述

CALL 及 RETURN 除了目前的指针(pointer)之外，有一个四层的堆栈。任何额外多做的 CALL/RETURN 会导致意料外的运转行为。

图形上下文

无

请参阅

JUMP, RETURN

4.18 CELL

为 VERTEX2F 指令指定位图单元格数量。

编码

31 24	23 7	6 0
0x06	保留	cell

参数

cell

位图单元格数量。 初始值是 0

图形上下文

单元格的值是图形上下文的一部份， 如 **Error! Reference source not found.**节所述。

请参阅

无

4.19 CLEAR

清除缓冲器，回预置值

编码

31 24	23 3	2	1	0
0x26	保留	C	S	T

参数

c

清除颜色缓冲器。设定此 bit 为 1 会清除 FT800 的颜色缓冲器，回预置(preset)值。设为 0 则此 FT800 的颜色缓冲器线持在一个未改变的值。供 RGB 通道使用的预置值定义在 CLEAR_COLOR_RGB 指令，供 alpha 透明通道使用的预置值定义在 CLEAR_COLOR_A 指令。

s

清除模板缓冲器。设定此 bit 为 1 会清除 FT800 的模板缓冲器，回预置值。设为 0 则此 FT800 的模板缓冲器线持在一个未改变的值。这个预置值是在 CLEAR_STENCIL 指令里定义。

t

清除标记缓冲器。设定此 bit 为 1 可清除 FT800 的标记缓冲器，回预置值。设定此 bit 为 0，会使 FT800 的标记缓冲器维持在一个未改变的值。这个预置值是在 CLEAR_TAG 指令里定义。

描述

剪刀测试及缓冲器写入遮盖会影响清除的动作。剪刀限制已清除的矩阵，而缓冲器写入遮盖会限制受影响的缓冲器。Alpha 透明功能、混合功能、及模板印刷不会影响清除的动作。

范例

清除屏幕，设成亮蓝色：

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

利用剪刀矩形清除部份屏幕，部份设为灰色、部份设为蓝色:

```
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) );
```

图形上下文

无

请参阅

CLEAR_COLOR_A, CLEAR_STENCIL, CLEAR_TAG, CLEAR_COLOR_RGB

4.20 CLEAR_COLOR_A

为 alpha 透明通道指定清除值

编码

32 24	23 8	7 0
0x0F	保留	Alpha 透明

参数

alpha

颜色缓冲器清除后，使用的 alpha 透明值。初始值为 0

图形上下文

alpha 透明值是图形上下文的一部份，如 **Error! Reference source not found.**节所示。

请参阅

CLEAR_COLOR_RGB, CLEAR

4.21 CLEAR_COLOR_RGB

为红、绿、蓝通道指定清除值

编码

31 24	23 16	15 8	7 0
0x02	红	蓝	绿

参数

red

颜色缓冲器被清除后的红色值。初始值是 0。

green

颜色缓冲器被清除后的绿色值。初始值是 0。

blue

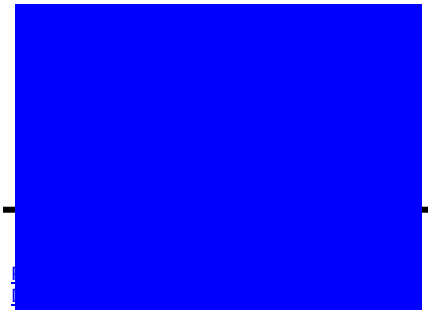
颜色缓冲器被清除后的蓝色值。初始值是 0。

描述

设定 CLEAR 指令后的颜色值。

范例

清除屏幕，设成亮蓝色：



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

利用剪刀矩形清除部份屏幕，部份设为灰色、部份设为蓝色:

```
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) );
```

图形上下文

红色值、绿色值、蓝色值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

CLEAR_COLOR_A, CLEAR

4.22 CLEAR_STENCIL

指定模板缓冲器的清除值。

编码

31 24	23 8	7 0
0x11	保留	**s**

参数

s

模板缓冲器清除后的值，初始值是 0

图形上下文

s 值是图形上下文的一部份，如 **Error! Reference source not found.**所述。

请参阅

CLEAR

4.23 CLEAR_TAG

指定标记缓冲器的清除值。

编码

31 24	23 8	7 0
0x12	保留	t

参数

t

标记缓冲器清除后的值，初始值是 0.

图形上下文

t 值是图形上下文的一部份，如 **Error! Reference source not found.**所述。

请参阅

TAG, TAG_MASK, CLEAR

4.24 COLOR_A

设定目前的 alpha 透明值

编码

31 24	23 8	7 0
0x10	保留	alpha

参数

alpha

目前颜色的 alpha 透明值，初始值是 255。

描述

设定应用在绘图元素上的 alpha 透明值，如点、线、及位图。Alpha 透明值如何影响影像像素是依 BLEND_FUNC 而定；预设的行为是透明混合。

范例

以 3 种 alpha 透明值，画出三个字符:

```
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) );
```

图形上下文

alpha 透明值是图形上下文的一部份，如 **Error! Reference source not found.**所述。

请参阅

COLOR_RGB, BLEND_FUNC

4.25 COLOR_MASK

开启或是关闭颜色成分的写入

编码

31 24	23 4	3	2	1	0
0x20	保留	r	g	b	a

参数

r

开启或是关闭 FT800 颜色缓冲器的红通道更新。初始值是 1，表示开启。

g

开启或是关闭 FT800 颜色缓冲器的绿通道更新。初始值是 1，表示开启。

b

开启或是关闭 FT800 颜色缓冲器的蓝通道更新。初始值是 1，表示开启。

a

开启或是关闭 FT800 颜色缓冲器的 alpha 透明通道更新。初始值是 1，表示开启。

描述

颜色遮盖可控制像素的颜色值是否更新。有时候，它是用来选择性的更新影像的红、绿、蓝、或 alpha 透明通道。但较多情况是当更新标记或模板缓冲器时，完全地关闭颜色更新。

范例

在屏幕的中间画一个数字'8'，然后画一个看不见的 40-像素圆形触摸区域至标记缓冲器：

```
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) );
```

图形上下文

r, g, b, a 值是图形上下文的一部份，如 **Error! Reference source not found.**所述。

请参阅

TAG_MASK

4.26 COLOR_RGB

设定目前红绿蓝的颜色编码值

编码

31 24	23 16	15 8	7 0
0x04	红	蓝	绿

参数

red

目前颜色的红色值，初始值为 255。

green

目前颜色的绿色值， 初始值是 255。

blue

目前颜色的蓝色值，初始值是 255。

描述

设定 FT800 颜色缓冲器的红色、绿色、蓝色值，这些值会应用在下一个绘图操作。

范例

以不同的颜色画三个字符:


```
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) );
```

图形上下文

红色、绿色、蓝色值是图形上下文的一部份，如 **Error! Reference source not found.**所述。

请参阅

COLOR_A

4.27 DISPLAY

结束显示清单，FT800 会忽略这个指令之后的所有指令。

编码

31 24	23 0
0x0	保留

参数

无

图形上下文

无

请参阅

无

4.28 END

绘制图形基元结束。

编码

31 24	23 0
0x21	保留

参数

无

描述

建议每个 BEGIN 指令之后要有一个 END 指令当结尾。但进阶的使用者可以不用 END 指令，以在显示清单里节省一些图形指令。

图形上下文

无

请参阅

BEGIN

4.29 JUMP

在显示清单另一个位置执行指令

编码

31 24	23 16	15 0
0x1E	保留	dest

参数

dest

要跳至的显示清单地址。

图形上下文

无

请参阅

CALL

4.30 LINE_WIDTH

指定 LINES 基元画线的线宽，单位是 1/16 像素。

编码

31 24	23 12	11 0
0x0E	保留	width

参数

width

以 1/16 像素为单位的线宽，初始值是 16。

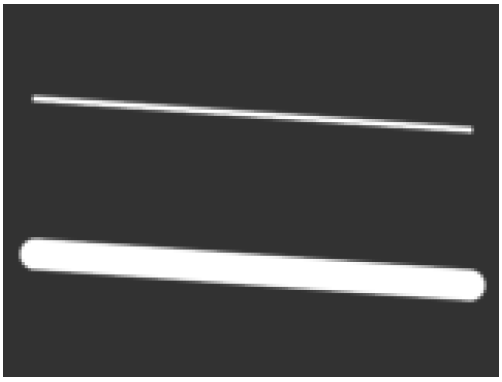
描述

为已画的线设定线宽。宽度定义是线中心到最边缘的像素距离，单位是 1/16 像素。有效范围是从 16 到 4095，以 1/16 像素为单位。

请注意 LINE_WIDTH 指令会影响 LINES, LINE_STRIP, RECTS, EDGE_STRIP_A/B/R/L 基元。

范例

第二条线是以 width 为 80 当半径所画的线，也就是 5 个像素。



```
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) );
```

图形上下文

线的宽度是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

无

4.31 MACRO

从一个宏寄存器(macro register)执行一个单一指令。

编码

31 24	23 1	0
0x25	保留	m

参数

m

要读取的宏寄存器。值为 0 表示 FT800 会从 REG_MACRO_0 拿取指令执行。值为 1 表示 FT800 会从 REG_MACRO_1 拿取指令执行。REG_MACRO_0 或 REG_MACRO_1 应是一个有效的显示清单指令，否则属于未定义的运转状态。

图形上下文

无

请参阅

无

4.32 POINT_SIZE

设定点的半径

编码

31 24	2317	16 0
0x0D	保留	Size

参数

size

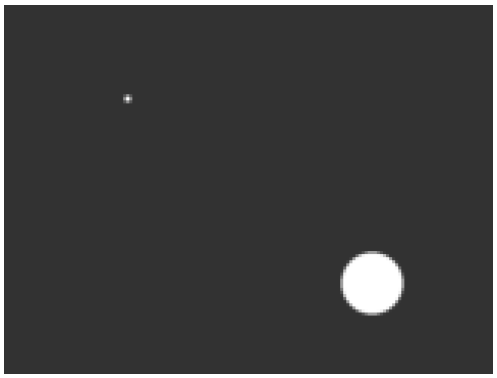
以 1/16 像素为单位的点半径，初始值是 16。

描述

设定已画点的尺寸大小。半径的定义是从点的中心到最边缘的像素，以 1/16 像素为单位。有效范围是从 16 到 8191，单位是 1/16 像素。

范例

第二个点是以半径 160 单位所绘制，也就是 10 像素的半径：



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

图形上下文

尺寸大小的值是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

无

4.33 RESTORE_CONTEXT

从目前的上下文堆栈恢复目前的图形上下文

编码

31 24	23 0
0x23	保留

参数

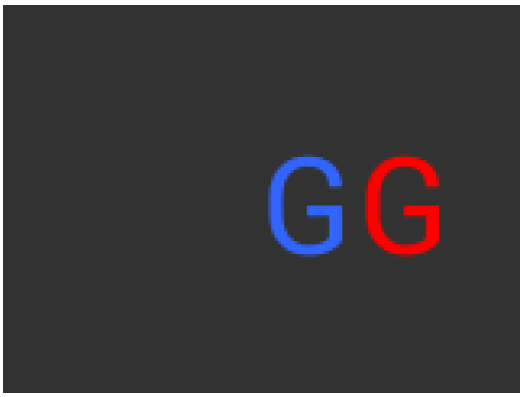
无

描述

恢复目前的图形上下文，如 **Error! Reference source not found.**节所述。FT800 里，SAVE 和 RESTORE 有四个层次可用。任何额外的 RESTORE_CONTEXT 指令会载入预设值至目前的上下文。

范例

储存及恢复上下文表示第二个'G'是用红色绘制，而不是蓝色：




```
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) );
```

图形上下文

无

请参阅

SAVE_CONTEXT

4.34 RETURN

从前面一个 CALL 指令返回。

编码

31 24	23 0
0x24	保留

参数

无

描述

CALL 及 RETURN 除了目前的指针(pointer)之外，有一个四层的堆栈。任何额外多做的 CALL/RETURN 会导致意料外的运转状态。

图形上下文

无

请参阅

CALL

4.35 SAVE CONTEXT

将目前的图形上下文压入上下文栈。

编码

31 24	23 0
0x22	保留

参数

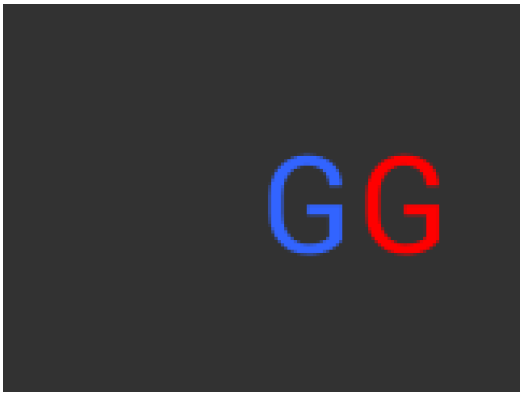
无

描述

保存目前的图形上下文，如 **Error! Reference source not found.**节所述。任何额外的 SAVE_CONTEXT 会仍掉最先保存的上下文。

范例

保存及恢复上下文表示第二个'G'是以红色缩制，而不是蓝色：




```
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) );
```

图形上下文

无

请参阅

RESTORE_CONTEXT

4.36 SCISSOR_SIZE

指定剪刀要修剪的矩形

编码

31 24	23 20	1910	9 0
0x1C	保留	Width	Height

参数

width

剪刀修剪矩形的宽度，以像素为单位。初始值是 512。有效范围是从 0 到 512。

height

剪刀修剪矩形的高度，以像素为单位。初始值是 512。有效范围是从 0 到 512。 描述

设定剪刀修剪矩形的宽度和高度，这样限制绘制的区域。

范例

设定一个 40 x 30 剪刀矩形对位图做修剪和清除：


```
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) );
```

图形上下文

宽度及高度是图形上下文的一部份，如 **Error! Reference source not found.**节所述。

请参阅

无

4.37 SCISSOR_XY

指定剪刀修剪矩形的左上角位置

编码

31 24	23 19	17 9	8 0
0x1B	保留	x	y

参数

x

剪刀修剪矩形的 x 坐标，以像素为单位。初始值为 0

y

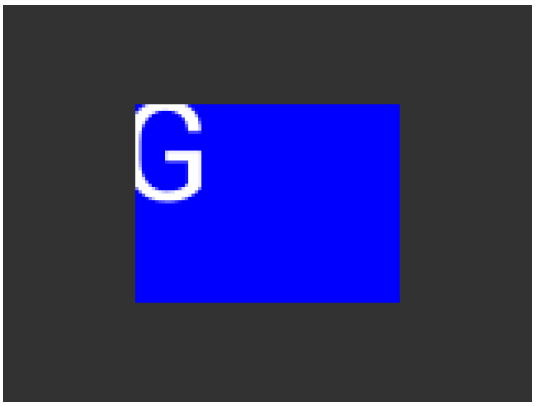
剪刀修剪矩形的 y 坐标，以像素为单位。初始值为 0

描述

设定剪刀修剪矩形的左上角位置，限制绘制区域。

范例

设定一个 40 x 30 剪刀修剪矩形并清除位图绘制：



```
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) );
```

图形上下文

x 及 y 值是图形上下文的一部份，如 **Error! Reference source not found.**所述。

请参阅

无

4.38 STENCIL_FUNC

设定模板测试的功能及参考值。

编码

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

参数

func

指定测试功能，指定下列其中一项： NEVER, LESS, LEQUAL, GREATER,GEQUAL, EQUAL, NOTEQUAL, 或 ALWAYS。 初始值是 ALWAYS。有关这些常数的值，请参考图 8: ALPHA_FUNC

ref

指定模板测试的参考值，初始值是 0

mask

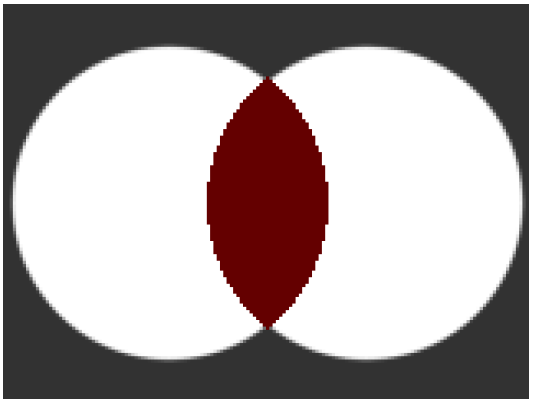
指定一个与参考值及储存的模板值交集的遮盖，初始值是 255

描述

模板测试拒绝或接收像素是依据定义在 func 参数的测试功能结果，与参考值相比，这个测试功能是操作在模板缓冲器里目前的值。

范例

绘制两个点，在每个像素增加模板，然后以红色值为 2 绘制像素。



```
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) );
```

图形上下文

func, ref 及 mask 的值是图形上下文的一部份， 如 **Error! Reference source not found.**节所述。

请参阅

STENCIL_OP, STENCIL_MASK

4.39 STENCIL_MASK

控制在模板平面里，个别 bits 的写入

编码

31 24	23 8	7 0
0x13	保留	**mask**

参数

mask

用来启动写入模板 bits 的遮盖，初始值是 255

图形上下文

遮盖的值是图形上下文的一部份， 如 **Error! Reference source not found.**节所述。

请参阅

STENCIL_FUNC, STENCIL_OP, TAG_MASK

4.40 STENCIL_OP

设定模板测试的动作

编码

31 24	23 6	5 3	2 0
0x0C	保留	**sfail**	**spass**

参数

sfail

指定模板测试失败后的动作，采下面所列的其中一种：KEEP,ZERO, REPLACE, INCR, DECR, 及 INVERT。 初始值是 KEEP(1)。

spass

指定模板测试通过后的动作，与 sfail 一样，是上述所列的其中一种。 初始值是 KEEP(1)

NAME	VALUE
ZERO	0
KEEP	1
REPLACE	2
INCR	3
DECR	4
INVERT	5

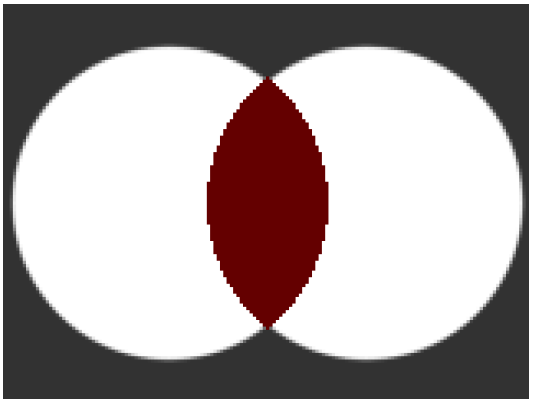
图 12: STENCIL_OP 常数的定义

描述

模板操作指定模板缓冲器如何被更新。选择的操作是依据模板测试是否通过与否。

范例

绘制两个点，在每个像素增加模板，然后以红色值为 2 绘制像素：



```
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) );
```

图形上下文

sfail 与 spass 的值是图形上下文的一部份， 如 **Error! Reference source not found.**节所述。

请参阅

STENCIL_FUNC, STENCIL_MASK

4.41 TAG

为以下画在屏幕上的图形物件附上标记值，初始标记缓冲器的值是 255。

编码

31 24	23 8	7 0
0x03	保留	**s**

参数

s

标记值。有效范围是从 1 到 255。

描述

FT800 的标记缓冲器初始值是由 CLEAR_TAG 指令指定，生效是由 CLEAR 指令。TAG 指令可以指定 FT800 的标记缓冲器的值，当图形物件画在屏幕上，标记缓冲器的值会应用在图形物件上。这个 TAG 值会被分配到下列全部的物件，除非用 TAG_MASK 指令关闭它。以下图形物件一旦被绘制，它们就被成功附加上标记值。当已附加标记值的图形物件被触摸，寄存器 REG_TOUCH_TAG 将以被触摸图形物件的标记值做更新。

假如一个显示清单里没有 TAG 指令，當被显示清单渲染的图形物件被触摸，它们会向寄存器 REG_TOUCH_TAG，报告标记值为 255。

图形上下文

s 值是图形上下文的一部份， 如 **Error! Reference source not found.**节所述。

请参阅

CLEAR_TAG, TAG_MASK

4.42 TAG_MASK

控制标记缓冲器的写入

编码

31 24	23 1	0
0x14	保留	**mask**

参数

mask

允许标记缓冲器更新。初始值是 1，表示 FT800 的标记缓冲器是以 TAG 指令给的值更新。然而，以下的图形物件会被附加到 TAG 指令给的标记值。

值为 0 表示 FT800 的标记缓冲器设定成预设值，而不是显示清单里 TAG 指令给的值。

描述

每一个画在屏幕上的图形物件都被附加上标记值，此标记值是定义在 FT800 的标记缓冲器里。FT800 标记缓冲器可以被 TAG 指令更新。

FT800 标记缓冲器的预设值是由 CLEAR_TAG 和 CLEAR 指令决定。如果显示清单里没有 CLEAR_TAG 指令，则标记缓冲器里的预设值将为 0 。

TAG_MASK 指令决定 FT800 标记缓冲器从 FT800 标记缓冲器的预设值取值，还是从显示清单里的 TAG 指令取值。

图形上下文

遮盖的值是图形上下文的一部份， 如 **Error! Reference source not found.**节所述。

请参阅

TAG, CLEAR_TAG, STENCIL_MASK, COLOR_MASK

4.43 VERTEX2F

在指定的屏幕坐标位置开始图形基元的操作，单位为 1/16 像素。

编码

31 30	29 15	14 0
0b'01	X	Y

参数

x

有符号的 x 坐标，以 1/16 像素为单位

y

有符号的 y 坐标，以 1/16 像素为单位

描述

坐标范围是从-16384 到+16383，单位为 1/16 像素。负的 x 坐标值表示此坐标是在(0,0)左边的虚拟屏幕，而负的 y 坐标值表示此坐标是在(0,0)上面的虚拟屏幕。若是绘制在负坐标的位置上，绘图操作是看不见的。

图形上下文

无

请参阅

无

4.44 VERTEX2II

在指定的像素单位坐标上，开始图形基元的操作。

编码

31 30	2921	20 12	11 7	6 0
0b'10	X	Y	handle	cell

参数

x

以像素为单位的 x 坐标，从 0 到 511。

y

以像素为单位的 y 坐标，从 0 到 511。

handle

位图的句柄。有效范围从 0 到 31，16 到 31 的范围的位图句柄是专给 FT800 内置的字体。

cell

单元格索引号。单元格索引号是位图的索引，这些位图有相同位图布局及格式。举例来说，对于句柄 31，单元格 65 表示最大内置字体里的"A"字符。

描述

坐标范围是从-16384 到+16383，单位为像素。句柄及单元参数都被忽略，除非在这个指令之前，图形基元被 BEGIN 指令指定为位图。

图形上下文

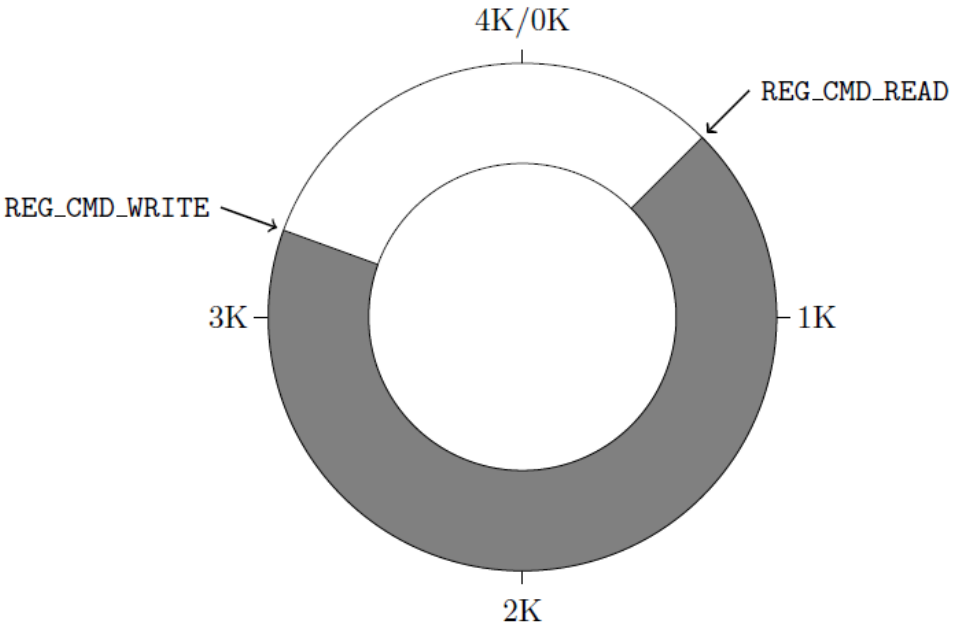
无

请参阅

无

5 协处理器引擎指令

协处理器引擎是透过一个 4K 字节 FIFO 所馈入，此 FIFO 在 FT800 存储器里的寄存器 RAM_CMD 上。MCU 写指令到 FIFO，协处理器引擎读取并执行指令。MCU 更新寄存器 REG_CMD_WRITE 以表示在 FIFO 里有新指令，而协处理器引擎在指令被执行后，更新 REG_CMD_READ。



所以为了计算在 FIFO 里的可用空间，MCU 会计算：

fullness = (REG_CMD_WRITE -REG_CMD_READ) *mod* 4096

freespace = (4096 - 4) -*fullness*;

计算不会报告出有 4096 字节可用空间的情况，以避免使 FIFO 进入循环或是看起来是空的。

假如在 FIFO 里有足够的空间，MCU 会在 FIFO RAM 合适的位置写入指令，然后更新 REG_CMD_WRITE。为了简化 MCU 代码，FT800 硬件自动循环连续的写入，从(RAM_CMD + 4095)回到(RAM_CMD + 0)。

FIFO 的条目总是 4 个字节宽 - REG_CMD_READ 或 REG_CMD_WRITE 如果有非 4 字节整数的值，属于错误的情况。每一个发到协处理器的指令引擎可以会拿取 1 个或多个字(words)：长度是依剧指令本身及任何附加的数据。一些指令会接著多种长度的数据，所以指令大小可能不是 4 字节的整数倍。在这样的情况，协处理器引擎忽略额外的 1、2、或 3 个字节，然后继续在接下来的 4 字节的界限上续续读取下一个指令。

5.1 显示清单指令的协处理器操纵

大部份协处理器引擎的指令写到目前的显示清单。目前显示清单的写入位置是维持在 REG_CMD_DL。当协处理器引擎写入一个字到显示清单，协处理器引擎就是在 REG_CMD_DL 的位置上写入显示清单，然后增加 REG_CMD_DL。特别指令 CMD_DLSTART 设定 REG_CMD_DL 为 0，是为了一个新显示清单的开始。

所有显示清单指令可以写成协处理器引擎的指令。协处理器引擎复制这些指令到目前的显示清单，其位置在 REG_CMD_DL。例如，这个协处理器引擎指令的序列写到写入一个小的显示清单：

```
cmd(CMD_DLSTART); //开始一个新显示清单
    cmd(CLEAR_COLOR_RGB(255, 100, 100)); //设定清除颜色
    cmd(CLEAR(1, 1, 1)); //清除屏幕
    cmd(DISPLAY()); //显示
```

当然，这个显示清单可能已经被写入到 RAM_DL。这个技术的优点是你可以在单一个流(stream)中，混合低水平操作(low-level operations)及高水平协处理器引擎指令(high-level co-processor engine commands)：

```
cmd(CMD_DLSTART);                          // 开始一个新的显示清单
cmd(CLEAR_COLOR_RGB(255, 100, 100));       // 设定清除颜色
cmd(CLEAR(1, 1, 1));                       // 清除屏幕
cmd_button(20, 20,                         // x,y
60, 60,                                    // 以像素为单位的宽度和高度
30,                                        // 字体 30
0,                                         // 预设选项
"OK!");
cmd(DISPLAY());                  // 显示
```

5.2 同步

在某些时候，必须等到协处理器处理完所有未完成的指令。当协处理引擎完成在指令缓冲器里最后一个未完成的指令后，它会举起 INT_CMDEMPTY 中断。其他方法是 MCU 会修剪 REG_CMD_READ，直到 REG_CMD_WRITE 与其相等。

一个需要同步的情况是当 MCU 需要对显示清单做直接的写入，要读出 REG_CMD_DL 的值。在这个情况 MCU 在读出 REG_CMD_DL 时，应该等到协处理器引擎处于空闲状态。

5.3 ROM 及 RAM 字体

图形引擎硬件绘制位图图形，而将这些图形当作字体，对于软件来说是有用处的。

字体度量，例如字体高度及宽度，是在当需要放置字体字符的时候，供软件使用的，对于 ROM 字符位图来说，这些字体字符是在 ROM 里。当以任何一个内置的 16 个 ROM 字体(编号 16-31)绘制文字时，协处理器引擎会使用这些度量。使用者可以载入相似的字体度量到 RAM，因此在位图句柄(handle)0-14 的位置创造额外的使用者定义的字体。句柄 15 则是保留给协处理指令 CMD_Button/CMD_Keys/CMD_Gradient 所使用。

每个 148-字节的字体度量区块都是以下的格式：

表 9 FT800 字体度量区块格式

地址	尺寸大小	值
p + 0	128	每个字体字符的宽度，以像素为单位
p + 128	4	字体度量格式，例如：L1, L4 或 L8
p + 132	4	字体线跨 ，以字节为单位
p + 136	4	字体宽度，以像素为单位
p + 140	4	字体高度，以像素为单位
p + 144	4	存储器里指像字体图形数据的指针

对 ROM 字体来说，这些区块也在 ROM 里，以一个长度为 16 的阵列存在。这个阵列的地址维持在 ROM 里 0xffffc 的位置。例如要找到字体大小 31 的字符'g'(ASCII 0x67)宽度：

从 0xffffc 读 32-bit 指针 p。

宽度 widths = p + (148 * (31 - 16)) (表中字体大小从 16 开始)

从存储器 widths[0x67]的位置读出字节

对 FT800 内置的 ROM 字体，一个位图句柄有效的字符范围是可印刷的 ASCII 代码。例如 32 到 127 都包含在内。对用户化的 RAM 字体有效字符的 ASCII 代码范围是从 1 到 127。

在用户介面物件里使用用户化的字体:

- 从 0 到 14 选一个位图
- 载入字体度量到存储器里
- 使用 BITMAP_SOURCE、BITMAP_LAYOUT、和 BITMAP_SIZE 指令设定位图参数
- 创建及下载一个字体度量区块到 RAM。度量区块的地址应该要对齐到 **4 字节的整数倍**。
- 使用指令 CMD_SETFONT ，以选择的句柄去注册此新字体。
- 在任何协处理器指令的字体属性值里，使用选择的句柄

5.4 警告

对于一些小工具，如果输入的参数值是超过 512 像素的解析度，则产生出来的小工具可能不适当。

如果追踪物件的中心(旋转追踪的情况)或是追踪物件的左上角(线性追踪的情况)是在显示区域的外面，则 CMD_TRACK 的行为不会被定义。

在 CMD_NUMBER(不支持小数部份)的使用上，只有有符号(signed)及无符号(unsigned)的整数有被支持。

假如输入参数值是在有效范围之外，小工具的行为不会被定义。

5.5 错误的情况

有些指令会造成协整理器错误。这些错误的发生是因为协处理引擎无法继续。例如：

- 一个无效的 JPEG 被提供到 CMD__LOADIMAGE
- 一个无效的数据流里被提供到 CMD_INFLATE
- 试图写入超过 2048 个指令到一个显示清单

在错误的情形下，协处理器引擎设定 REG_CMD_READ 到 0xfff(这是一个非法值因为所有的指令缓冲器数据应该要对齐 32-bit)，然后举起 INT_CMDEMPTY 中断，停止接收新的指令。当主机 MCU 认出这个错误的情况，主机 MCU 应该按照以下步骤恢复：

- 将 REG_CPURESET 设定为 1，使协处理器引擎维持在复位的情况
- 将 REG_CMD_READ 和 REG_CMD_WRITE 设为 0
- 将 REG_CPURESET 设定为 0，重启协处理器引擎

5.6 小工具实体尺寸

这个部份包含小工具共同的实体尺寸。

- 所有圆角的半径是小工具字体计算出来的(字符小写’o’的曲度)。半径是以字体高度的 3/16 算出。
- 所有 3D 阴影是以下列方法画出 (1) 照亮物件左上角偏移 0.5 个像素的区域 (2) 投阴影于物件右下角偏移 1.0 个像素的区域。
- 对于进度条、卷动条、及滑块，输出的小工具是垂直的小工具，以防宽度及高度相同的情况。

5.7 小工具颜色设定

协处理器引擎小工具是以先前指令所指派的颜色绘制：CMD_FGCOLOR, CMD_BGCOLOR and COLOR_RGB。根据这些指令，协处理器引擎会判协处理器引擎小工具里的不同区域，会以不同颜色渲染。

通常，假如协处理器引擎小工具互动的区域是设计给互动的 UI 元素使用，CMD_FGCOLOR 指令会影响协处理器引擎小工具的互动区域。例如：CMD_BUTTON、CMD_DIAL。 CMD_BGCOLOR 应用在有背景的协处理器引擎小工具。更多细节请看下面的表格。

表 10 小工具(WIDGET)颜色设定表

Widget	CMD_FGCOLOR	CMD_BGCOLOR	COLOR_RGB
CMD_TEXT	NO	NO	YES
CMD_BUTTON	YES	NO	YES(标签)
CMD_GAUGE	NO	YES	YES(针与标识)
CMD_KEYS	YES	NO	YES(文字)
CMD_PROGRESS	NO	YES	YES
CMD_SCROLLBAR	YES(内部条)	YES(外部条)	NO
CMD_SLIDER	YES(旋扭)	YES(旋扭右边条)	YES(旋扭左边条)
CMD_DIAL	YES(旋扭)	NO	YES(标识)
CMD_TOGGLE	YES(旋扭)	YES(条)	YES(文字)
CMD_NUMBER	NO	NO	YES

CMD_CALIBRATE	YES(动画点)	YES(外部点)	NO
CMD_SPINNER	NO	NO	YES

5.8 协处理器引擎图形状态

协处理器引擎会维持一小部份的内部状态供图形绘制。这个状态在协处理器引擎复位的时候，会被 CMD_COLDSTART 指令设定预设值。状态的值不被 CMD_DLSTART 或 CMD_SWAP 影响，所以应用程式只需要在启动时设定它们一次。

表 11 协处理器引擎图形状态

状态	预设值	指令
背景颜色	暗蓝 (0x002040)	CMD_BGCOLOR
前景颜色	浅蓝 (0x003870)	CMD_FGCOLOR
梯度颜色	白 (0xffffff)	CMD_GRADCOLOR
转盘	*无*	CMD_SPINNER
物件追踪器	*全部关闭*	CMD_TRACK
中断计时器	*无*	CMD_INTERRUPT
位图变换矩阵: $\begin{bmatrix} A & B & C \\ D & E & F \end{bmatrix}$	$\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \end{bmatrix}$	CMD_LOADIDENTITY, CMD_TRANSLATE, CMD_ROTATE, etc.
位图句柄	15	CMD_GRADCOLOR, CMD_KEYS, CMD_BUTTON

5.9 OPTION 参数的定义

下列表格定义这个章节提到的 OPTION 参数。

表 12 OPTION 参数定义

名称	值	描述	指令
OPT_3D	0	5.9.1.1.1.1 协处理器小工具是以 3D 效果绘制。这是预设的选项。	CMD_BUTTON,CMD_CLOCK,CMD_KEYS,CMD_GAUGE,CMD_SLIDER, CMD_DIAL, CMD_TOGGLE,CMD_PROGRESS, CMD_SCROLLBAR
OPT_RGB565	0	解码 JPEG 影像格式成 RGB565 的协处理器选项	CMD_IMAGE
OPT_MONO	1	解码 JPEG 影像格式成 L8 的协处理器选项。例如 monochrome	CMD_IMAGE

名称	值	描述	指令
OPT_NODL	2	没有显示清单指令产生给从 JPEG 影像编码的位图	CMD_IMAGE
OPT_FLAT	256	5.9.1.1.1.2 协处理器小工具以没有 3D 效果的方式绘制	CMD_BUTTON,CMD_CLOCK,CMD_KEYS,CMD_GAUGE,CMD_SLIDER,CMD_DIAL,CMD_TOGGLE,CMD_PROGRESS,CMD_SCROLLBAR
OPT_SIGNED	256	此数字被视为 32bit 有符号的整数	CMD_NUMBER
OPT_CENTERX	512	协处理器小工具水平集中	CMD_KEYS,CMD_TEXT,CMD_NUMBER
OPT_CENTERY	1024	协处理器小工具垂直集中	CMD_KEYS,CMD_TEXT,CMD_NUMBER
OPT_CENTER	1536	协处理器小工具水平及垂直集中	CMD_KEYS,CMD_TEXT,CMD_NUMBER
OPT_RIGHTX	2048	协处理器小工具上的标签向右调整	CMD_KEYS,CMD_TEXT,CMD_NUMBER
OPT_NOBACK	4096	协处理器小工具没有画背景。	CMD_CLOCK, CMD_GAUGE
OPT_NOTICKS	8192	协处理器时钟小工具没有绘制小时刻度。仪表小工具没有画大小刻度。	CMD_CLOCK, CMD_GAUGE
OPT_NOHM	16384	协处理器时钟小工具不画时针、分针、只画秒针	CMD_CLOCK
OPT_NOPOINTER	16384	协处理器仪表不画指针	CMD_GAUGE
OPT_NOSECS	32768	协处理器时钟小工具不画秒针	CMD_CLOCK
OPT_NOHANDS	49152	协处理器时钟小工具不画时针、分针、及秒针	CMD_CLOCK

5.10 协处理器引擎的资源

协处理器引擎不会改变硬件的图形状态。也就是说，像颜色及线宽等图形状态不会被协处理器引擎改变。

然而，小工具会保留一些硬件资源，这是用户程序需要考虑的：

- 位图句柄 15 是被 3D 效果的按扭、按键、及梯度所用。
  一个图形上下文是被物件所用，所以供 SAVE_CONTEXT 及 RESTORE_CONTEXT 使用的有效栈深是 3 层。

5.11 指令组

开始及结束显示清单的指令：

- CMD_DLSTART -开始一个新的显示清单
- CMD_SWAP -交换目前的显示清单

绘制图形物件的指令：

- CMD_TEXT - 绘制文字
- CMD_BUTTON - 绘制一个按扭
- CMD_CLOCK - 绘制一个模拟时钟
- CMD_BGCOLOR - 设定背景颜色
- CMD_FGCOLOR - 设定前景颜色
- CMD_GRADCOLOR -为 CMD_BUTTON 和 CMD_KEYS 的強調(highlight)颜色设定 3D 效果
- CMD_GAUGE - 绘制一个仪表
- CMD_GRADIENT - 绘制一个平滑的颜色梯度
- CMD_KEYS - 绘制一行按键
- CMD_PROGRESS - 绘制一个进度条
- CMD_SCROLLBAR - 绘制一个卷动条
- CMD_SLIDER - 绘制一个滑块
- CMD_DIAL - 绘制一个旋转拨号控制
- CMD_TOGGLE - 绘制一个切换开关
- CMD_NUMBER -绘制一个十进位数字

在存储器上操作的指令：

- CMD_MEMCRC - 为存储器计算一个 CRC-32
- CMD_MEMZERO - 写入 0 到一个存储器区块
- CMD_MEMSET - 以一个字节的值填满存储器
- CMD_MEMWRITE - 写入字节到存储器
- CMD_MEMCPY - 复制一个存储器区块
- CMD_APPEND -附加存储器到显示清单

载入影像数据到 FT800 存储器的指令：

- CMD_INFLATE - 解压缩数据到存储器
- CMD_LOADIMAGE - 载入一个 JPEG 影像

设定位图变换矩阵的指令：

- CMD_LOADIDENTITY - 设定目前的矩阵成单位矩阵
- CMD_TRANSLATE - 应用翻转效果到目前的矩阵
- CMD_SCALE - 应用缩放效果到目前的矩阵
- CMD_ROTATE - 应用旋转效果到目前的矩阵
- CMD_SETMATRIX - 当一个位图变换，写入目前的矩阵
- CMD_GETMATRIX - 取回目前矩阵的系数

其他指令：

- CMD_COLDSTART - 设定协处理器引擎的状态到预设值
- CMD_INTERRUPT - 触发中断 INT_CMDFLAG
- CMD_REGREAD - 读出一个寄存器的值
- CMD_CALIBRATE -执行触屏校正的例行程序

- CMD_SPINNER -开始一个动画转盘
- CMD_STOP - 停止任何转盘、屏幕保护、或是素描功能
- CMD_SCREENSAVER - 开始一个动画的屏幕保护
- CMD_SKETCH - 开始连续的素描更新
- CMD_SNAPSHOT - 对目前的屏幕作截屏
- CMD_LOGO - 拨放芯片 logo 动画

5.12 CMD_DLSTART – 开始一个显示清单

当协处理引擎执行这个指令，它一直等到目前的显示清单被扫描输出，然后设定 REG_CMD_DL 的值为 0。

C 语言原型

void cmd_dlstart( );

指令布局

+0	CMD_DLSTART (0xffffff00)

范例

```
cmd_dlstart();
...
cmd_dlswap();
```

5.13 CMD_SWAP – 交换目前的显示清单

当协处理器引擎执行这个指令，在目前的显示清单被扫描输出后，需要一个立即的显示清单交换。在内部，此协处理器引擎透过写入 REG_DLSWAP 实现这个指令。请参考 REG_DLSWAP 。

这个协处理引擎指令不会产生任何显示清单指令到显示清单存储器 RAM_DL。

C 语言原型

void cmd_swap( );

指令布局

+0	CMD_DLSWAP(0xffffff01)

范例

无

5.14 CMD_COLDSTART -将协处理器引擎的状态设为预设值

这个指令将协处理器引擎的状态设成预设值

C 语言原型

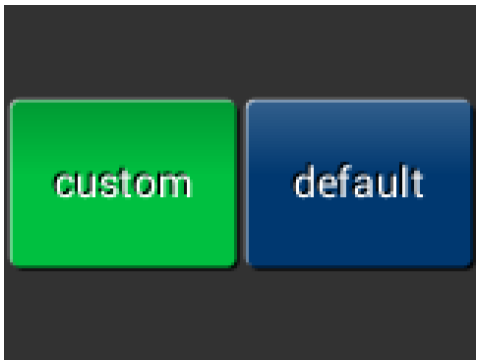
void cmd_coldstart( );

指令布局

+0	CMD_COLDSTART(0xffffff32)

范例

改变成一个定制的颜色主题，然后恢复预设颜色：




```
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");
```

5.15 CMD_INTERRUPT – 触发 INT_CMDFLAG 中断

当协处理器引擎执行这个指令，会触发中断 INT_CMDFLAG。

C 语言原型

void cmd_interrupt(uint32_t ms);

参数

ms

中断触发之前的延迟，以毫秒为单位。在这延迟之前，可以保证中断不会发出。假如 ms 为 0，会立即发出中断。

指令布局

+0	CMD_INTERRUPT(0xffffff02)
+4	ms

范例

在一个 JPEG 结束载入后触发一个中断

```
cmd_loadimage();
...
cmd_interrupt(0); // 载入前一个影像完成，触发中断
```

在 0.5 秒后触发一个中断：

```
cmd_interrupt(500);
...
```

5.16 CMD_APPEND - 附加存储器于显示清单

附加一个存储器区块到目前的显示清单存储器地址，偏移量在 REG_CMD_DL 指示。

C 语言原型

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

参数

ptr

在主存储器里源头指令的开始

num

要复制的字节数量。必须是 4 的整数倍。

指令布局

+0	CMD_APPEND(0xffffff1e)
+4	Ptr
+8	Num

描述

当附加完成后，协处理器引擎会增加 num 到 REG_CMD_DL，以确认显示清单有按照顺序。

范例

```
...
cmd_dlstart();
cmd_append(0, 40); // 从主存储器地址 0 的位置复制 10 个指令
cmd(DISPLAY); // 结束显示清单
cmd_swap();
```

5.17 CMD_REGREAD -读取一个寄存器的值

C 语言原型

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

参数

ptr

要读取的寄存器地址

result

在 ptr 地址读到的寄存器数值

指令布局

+0	CMD_REGREAD(0xffffff19)
+4	Ptr
+8	Result

范例

补捉指令结束的确切时间：

```
uint16_t x = rd16(REG_CMD_WRITE);
cmd_regread(REG_CLOCK, 0);
...
printf("%08x\n", rd32(RAM_CMD + x + 8));
```

5.18 CMD_MEMWRITE -写入字节到存储器

写入下列字节到 FT800 存储器里。这个指令可以用来设定寄存器的值，或是在指定的时间更新存储器的内容。

C 语言原型

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

参数

Ptr

要写入的存储器地址

num

要写入的字节数量

描述

在指令缓冲器里，数据字节应该要立即接在后面。如果字节数量不是 4 的整数倍，则需再附加上 1、2、或 3 个字节以确保下一个指令可以对齐到 4 字节的整数倍，而这些加上的衬垫字节可以是任何的值。此功能是否完成，可透过侦测 REG_CMD_READ 是否等于 REG_CMD_WRITE。

警告: 如果使用这个指令，若使用不当可能会损坏 FT800 的存储器

指令布局

+0	CMD_MEMWRITE(0xffffff1a)
+4	ptr
+8	Num
+12	Byte0
+13	Byte1
..	..
+n	..

范例

为一个特定的屏幕截图改变背光亮度到 64 (一半的强度)：

```
...
cmd_swap(); // 结束这个显示清单
cmd_dlstart(); // 等待直到交换(swap)之后
cmd_memwrite(REG_PWM_DUTY, 4); // 写入到 PWM_DUTY 寄存器
cmd(100);
```

5.19 CMD_INFLATE -解压缩数据到存储器

解压缩以下的压缩数据到 FT800 的存储器 RAM_G。这个数据应该已经被 DEFLATE 演算法压缩，例如 ZLIB 库。ZLIB 压缩对于载入图形数据特别有用。

C 语言原型

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

参数

ptr

目的地地址。在指令缓冲器里，数据字节应该要立即跟在后面。

描述

如果字节数量不是 4 的整数倍，则需再附加上 1、2、或 3 个字节以确保下一个指令可以对齐到 4 字节的整数倍，而这些加上的衬垫字节可以是任何的值。

指令布局

+0	CMD_INFLATE(0xffffff22)
+4	ptr
+8	Byte0
+9	Byte1
..	..
+n	..

范例

载入图形数据到主存储器址址 0x8000：

```
cmd_inflate(0x8000);
...                                    // 以下是 zlib 压缩的数据
```

5.20 CMD_LOADIMAGE - 截入 JPEG 影像

在主存储器里，解压缩后面的 JPEG 影像数据成一个 FT800 的位图。这个影像数据应该是一个规律基线(regular baseline)的 JPEG(JFIF)影像。

C 语言原型

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

参数

ptr

目的地地址

options

预设上，选项 OPT_RGB565 表示载入的位图是 RGB565 的格式。OPT_MONO 选项表示载入的位图是单色(monochrome)的，格式为 L8。此指令附加显示清单指令，以设定源头、布局、以及结果影像的大小。选项 OPT_NODL 是用来避免载入，没有东西会被写入显示清单。OPT_NODL 可以和 OPT_MONO 或 OPT_RGB565 作联集(OR)。

描述

在指令缓冲器里，数据字节应该要立即接在后面。如果字节数量不是 4 的整数倍，则需再附加上 1、2、或 3 个字节以确保下一个指令可以对齐到 4 字节的整数倍，而这些加上的衬垫字节可以是任何的值。

主机处理器上的应用程式必须解析 JPEG 的标头(header)以得到 JPEG 影像的属性并决定做解码。对于非基线 JPEG 影像或是产生的输出数据超过 RAM_G 大小的情况，运转行为是无法预测的。

指令布局

+0	CMD_LOADIMAGE(0xffffff24)
+4	Ptr
+8	Options
+12	Byte0
+13	Byte1
..	..
+n	..

范例

在地址 0 载入一个 JPEG 影像，然后在(10,20) 和 (100,20)绘制位图：

```
cmd_loadimage(0, 0);
...                                    // JPEG 文件数据在后
cmd(BEGIN(BITMAPS))
cmd(VERTEX2II(10, 20, 0, 0));          // 在 (10,20)绘制位图
cmd(VERTEX2II(100, 20, 0, 0));         // 在 (100,20)绘制位图
```

5.21 CMD_MEMCRC – 为存储器记算一个 CRC-32

为 FT800 存储器的一个区块计算一个 CRC-32

C 语言原型

```
void cmd_memcrc( uint32_t ptr,
                 uint32_t num,
                 uint32_t result );
```

参数

ptr

此存储器区块的开始地址

num

存储器区块里字节的数量

result

输出参数；在指令执行后，与 CRC-32 一起写入。此功能是否完成，可透过侦测 REG_CMD_READ 是否等于 REG_CMD_WRITE。

指令布局

+0	CMD_MEMCRC(0xffffff18)
+4	Ptr
+8	Num
+12	Result

范例

为了计算 FT800 存储器第一个 1K 字节的 CRC-32，先纪录 REG_CMD_WRITE 的值，执行指令，等待完成，然后在结果的位置读出此 32-bit 的值：

```
uint16_t x = rd16(REG_CMD_WRITE);
cmd_crc(0, 1024, 0);
...
printf("%08x\n", rd32(RAM_CMD + x + 12));
```

5.22 CMD_MEMZERO -写入零值到一个存储器区块

C 语言原型

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

参数

ptr

此存储器区块的开始地址

num

存储器区块里字节的数量

此功能是否完成，可透过侦测 REG_CMD_READ 是否等于 REG_CMD_WRITE。

指令布局

+0	CMD_MEMZERO(0xffffff1c)
+4	ptr
+8	num

范例

移除主存储器的第一个 1K：

```
cmd_memzero(0, 1024);
```

5.23 CMD_MEMSET -将一个字节的值填入存储器

C 语言原型

```
void cmd_memset( uint32_t ptr,
                 uint32_t value,
                 uint32_t num );
```

参数

ptr

此存储器区块的开始地址

value

要写入存储器的值

num

存储器区块里的字节数量

此功能是否完成，可透过侦测 REG_CMD_READ 是否等于 REG_CMD_WRITE。

指令布局

+0	CMD_MEMSET(0xffffff1b)
+4	ptr
+8	Value
+12	num

范例

将主存储器的第一个 1K 字节写入 0xff：

```
cmd_memset(0,  0xff, 1024);
```

5.24 CMD_MEMCPY -复制一个存储器区块

C 语言原型

```
void cmd_memcpy( uint32_t dest,
                 uint32_t src,
                 uint32_t num );
```

参数

dest

目的地存储器区块的地址

src

源头存储器区块的地址

num

要复制的字节数量

此功能是否完成，可透过侦测 REG_CMD_READ 是否等于 REG_CMD_WRITE。

指令布局

+0	CMD_MEMCPY(0xffffff1d)
+4	dst
+8	src
+12	num

范例

从 0 到 0x8000 复制存储器的 1K 字节：

```
cmd_memcpy(0x8000, 0, 1024);
```

5.25 CMD_BUTTON –绘制一个按钮

C 语言原型

```
void cmd_button( int16_t x,
                 int16_t y,
                 int16_t w,
                 int16_t h,
                 int16_t font,
                 uint16_t options,
                 const char* s );
```

参数

x

左上角按扭的 x 坐标，以像素为单位

y

左上角按扭 y 坐标，以像素为单位

font

位图句柄，以指定用在按扭标签上的字体。参考 ROM 及 RAM 字体。

options

预设上，按扭是以 3D 效果绘制，值为 0。OPT_FLAT 移除 3D 效果。OPT_FLAT 的值是 256。

s

按扭的标签。必须是一个以 null 字符结束的字符串。也就是 C 语言的'\0'。对 FT800 内置的 ROM 字体，s 里的有效字符是可打印的 ASCII 代码，也就是从 32 到 127(包含 32 及 123) 。对于定制的 RAM 字体，s 里有效字符的 ASCII 代码是从 1 到 127。

描述

更多信息可参考 Co-processor engine widgets physical dimensions

指令布局

+0	CMD_BUTTON(0xffffff0d)
+4	X
+6	Y
+8	W
+10	H
+12	Font
+14	Options
+16	S
+17	..
..	..
+n	0

范例

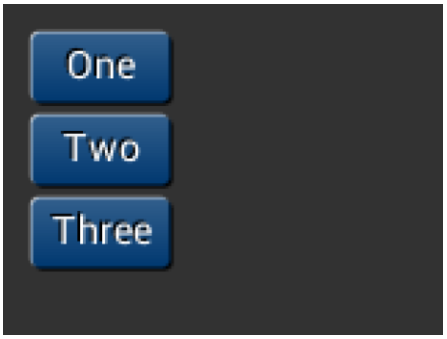
一个有较大文字，大小为 140x00 像素的按扭：

```
cmd_button(10, 10, 140, 100, 31, 0, "Press!");
```

没有 3D 效果的外观：


```
cmd_button(10, 10, 140, 100, 31, OPT_FLAT, "Press!");
```

几个较小的按扭：




```
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");
```

改变按扭的颜色


```
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");
```

5.26 CMD_CLOCK –绘制一个针式台钟

C 语言原型

```
void cmd_clock( int16_t x,
                int16_t y,
                int16_t r,
                uint16_t options,
                uint16_t h,
                uint16_t m,
                uint16_t s,
                uint16_t ms );
```

参数

x

时钟中心的 x 坐标，以像素为单位

y

时钟中心的 y 坐标，以像素为单位

options

预设的时钟拨号盘是以一个 3D 效果绘制而这个选项的名称是 OPT_3D。选项 OPT_FLAT 移除此 3D 效果。如果有选项 OPT_NOBACK，不画背景。如果有选项 OPT_NOTICKS，不画 12 小时刻度。如果有选项 OPT_NOSECS, 不画秒针。如果有选项 OPT_NOHANDS, 不画任何指针。如果有 OPT_NOHM，不画小时针及分针。

h

小时

m

分钟

s

秒

ms

毫秒

描述

实体尺寸的详细信息如下：

- 12 个刻度的标记是放在一个半径为 r*(200/256)的圆上
- 每个刻度是一个全径为 r*(10/256)的点
- 秒针长度为 r*(200/256)、宽度为 r*(3/256)
- 分针长度为 r*(150/256)、宽度为 r*(9/256)
- 时针长度为 r*(100/256)、宽度为 r*(12/256)

更多信息请参考 Co-processor engine widgets physical dimensions 。

指令布局

+0	CMD_CLOCK(0xffffff14)
+4	X
+6	Y
+8	R
+10	Options
+12	H
+14	M
+16	S
+18	Ms

范例

一个半径为 50 像素的时钟，显示时间为 8 点 15 分：

```
cmd_clock(80, 60, 50, 0, 8, 15, 0, 0);
```

设定背景颜色

```
cmd_bgcolor(0x401010);
```

```
cmd_clock(80, 60, 50, 0, 8, 15, 0, 0);
```

没有 3D 效果的外观：

```
cmd_clock(80, 60, 50, OPT_FLAT, 8, 15, 0, 0);
```

时间的设定部份可以有大的数值。下图设定的小时是(7 x 3600s)而分钟是(38 x 60s)，秒是 59。创建出一个时钟面标示时间为 7 点 38 分 59 秒：

```
cmd_clock(
80, 60, 50, 0,
0, 0, (7 * 3600) + (38 * 60) + 59, 0);
```

沒有秒針:

```
cmd_clock(80, 60, 50, OPT_NOSECS, 8, 15,
0, 0);
```

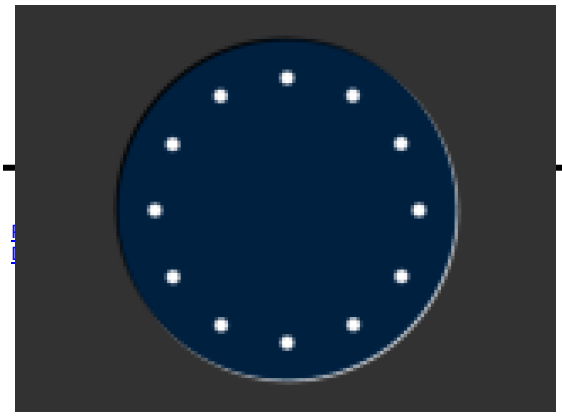
沒有背景:

```
cmd_clock(80, 60, 50, OPT_NOBACK, 8, 15,
0, 0);
```

沒有刻度:

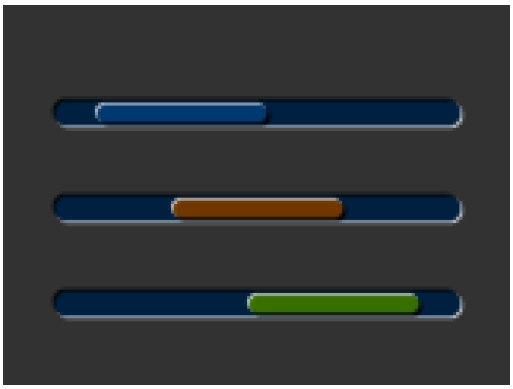
```
cmd_clock(80, 60, 50, OPT_NOTICKS, 8, 15,
0, 0);
```

沒有指針:



cmd_clock(80, 60, 50, OPT_NOHANDS, 8, 15, 0, 0);

5.27 CMD_FGCOLOR - 设定前景颜色



C 语言原型

void cmd_fgcolor(uint32_t c);

参数

c

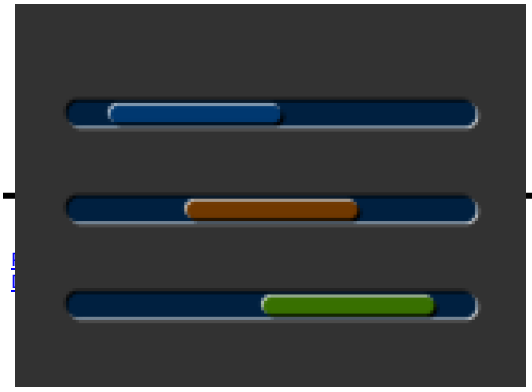
新的前景颜色，格式是一个 24-bit RGB 数字。红色是最高位 8 bits，蓝色是最低位 8 bits。所以 0xff0000 是亮红色。前景颜色可以应用在使用者可以移动的物件，例如把手及按扭。(设计上的"能供性")。

指令布局

+0	CMD_FGCOLOR(0xffffff0a)
+4	C

范例

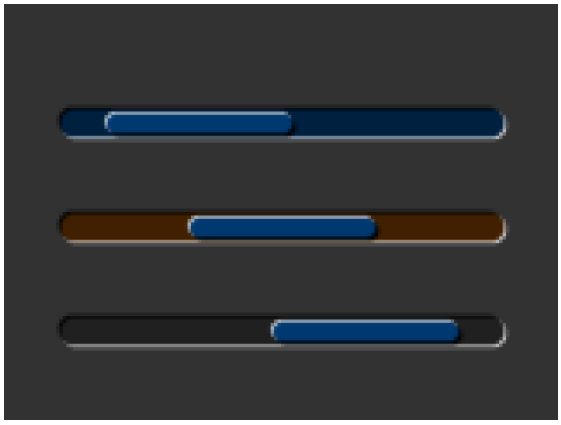
最上面的卷动条使用预设的前景颜色，其他则使用改变过的颜色：



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);
```

5.28 CMD_BGCOLOR -设定背景颜色



C 语言原型

void cmd_bgcolor(uint32_t c);

参数

c

新的背景颜色，格式是一个 24-bit RGB 数字。红色是最高位 8 bits，蓝色是最低位 8 bits。所以 0xff0000 是亮红色。背景颜色可以应用在使用者不能移动的物件，例如仪表后及滑块后。

指令布局

+0	CMD_BGCOLOR(0xffffff09)
+4	C

范例

最上面的卷动条使用预设的背景颜色，其他则使用改变过的颜色：

```
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);
```

5.29 CMD_GRADCOLOR -设定 3D 按键上强调显示的颜色

C 语言原型

```
void cmd_gradcolor( uint32_t c );
```

参数

c

新的强调梯度颜色，格式是一个 24-bit RGB 数字。红色是最高位 8 bits，蓝色是最低位 8 bits。所以 0xff0000 是亮红色。

梯度效果只可供按扭及按键小工具支持。

指令布局

+0	CMD_GRADCOLOR(0xffffff34)
+4	C

范例

改变梯度颜色: 白色 (预设), 红色, 绿色，及蓝色


```
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");
```

梯度颜色也可以供按键使用：


```
cmd_fgcolor(0x101010);
cmd_keys(10, 10, 140, 30, 26, 0, "abcde");
cmd_gradcolor(0xff0000);
cmd_keys(10, 50, 140, 30, 26, 0, "fghij");
```

5.30 CMD_GAUGE –绘制一个仪表

C 语言原型

```
void cmd_gauge( int16_t x,
                int16_t y,
                int16_t r,
                uint16_t options,
```

uint16_t major,
uint16_t minor,
uint16_t val,
uint16_t range);

参数

x

仪表中心的 x 坐标，以像素为单位

y

仪表中心的 y 坐标，以像素为单位

r

仪表半径，以像素为单位

options

预设的时钟拨号盘是以一个 3D 效果绘制而这个选项的值为 0。选项 OPT_FLAT 移除此 3D 效果。如果有选项 OPT_NOBACK，不画背景。如果有选项 OPT_NOTICKS，不画 12 小时刻度。如果有选项 OPT_NOPOINTER, 不画指针。

major

拨号盘上的大刻度区间数量，范围 1-10

minor

拨号盘上的小刻度区间数量，范围 1-10

val

仪表指示的值，介于 0 到 range 值之间，包含 range 值。

range

最大值

描述

实体尺寸的细节：

- 刻度标记放置在一个 270 度的圆弧上，以顺时针方向从西南方位置开始
- 小刻度线宽 r*(2/256)，大刻度 r*(6/256)
- 刻度间距介于 r*(190/256)到 r*(200/256)之间
- 指针线宽为 r*(4/256)，长度为从中心点开始算 r*(190/256)的长度
- 线的另一端与指针方向成 90 度垂直，离中心 r*(3/256)的距离。

更多信息可参考 Co-processor engine widgets physical dimensions

指令布局

+0	CMD_GAUGE(0xffffff13)
+4	X
+6	Y
+8	R
+10	Options
+12	Major
+14	Minor
+16	Value
+18	Range

范例

一个半径为 50 像素的仪表，5 个大刻度区间，每个大刻度区间里有 4 个小刻度区间，图中指示为 30%：

```
cmd_gauge(80, 60, 50, 0, 5, 4, 30, 100);
```

没有 3D 效果的外观:

```
cmd_gauge(80, 60, 50, OPT_FLAT, 5, 4, 30, 100);
```

10 个大刻度区间，每个大刻度区间有 2 个小刻度区间：

```
cmd_gauge(80, 60, 50, 0, 10, 2, 30, 100);
```

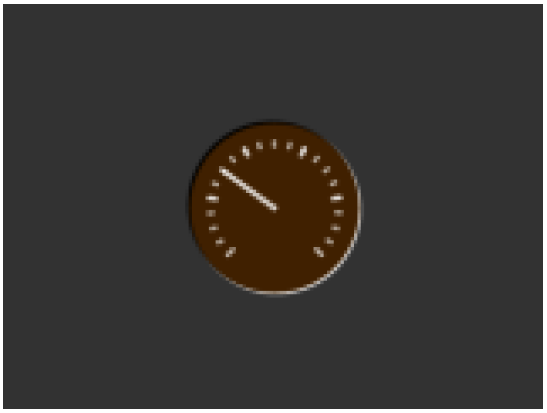
设定小刻度区间为 1，使其没有小刻度区间：

```
cmd_gauge(80, 60, 50, 0, 10, 1, 30, 100);
```

设定大刻度区间为 1，则会只有小刻度区间：

```
cmd_gauge(80, 60, 50, 0, 1, 10, 30, 100);
```

一个褐色背景、较小的仪表：



```
cmd_bgcolor(0x402000);
cmd_gauge(80, 60, 25, 0, 5, 4, 30, 100);
```

数值范围 0-1000，下图指示结果为 1000：

```
cmd_gauge(80, 60, 50, 0, 5, 2, 1000, 1000);
```

数值范围 0-65535，下图指示为 49152：

```
cmd_gauge(80, 60, 50, 0, 4, 4, 49152, 65535);
```

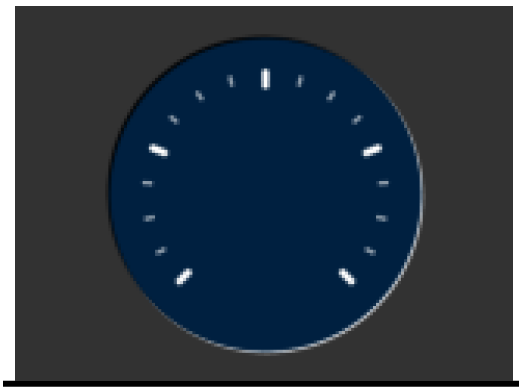
没有背景：

```
cmd_gauge(80, 60, 50, OPT_NOBACK, 4, 4,
49152, 65535);
```

没有刻度遮盖：

```
cmd_gauge(80, 60, 50, OPT_NOTICKS, 4, 4,
49152, 65535);
```

没有指针：

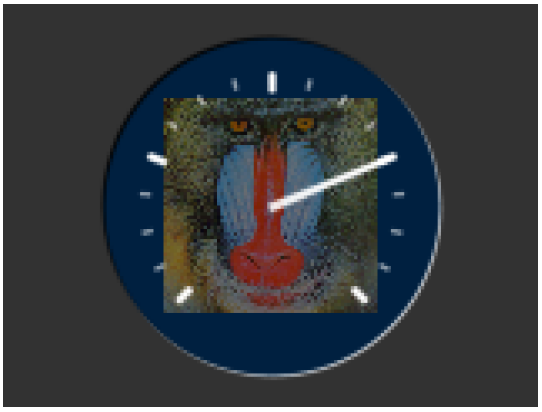


```
cmd_gauge(80,60, 50, OPT_NOPOINTER, 4,
4, 49152, 65535);
```

使用两次指令的方式画仪表，以亮红色为指针颜色：

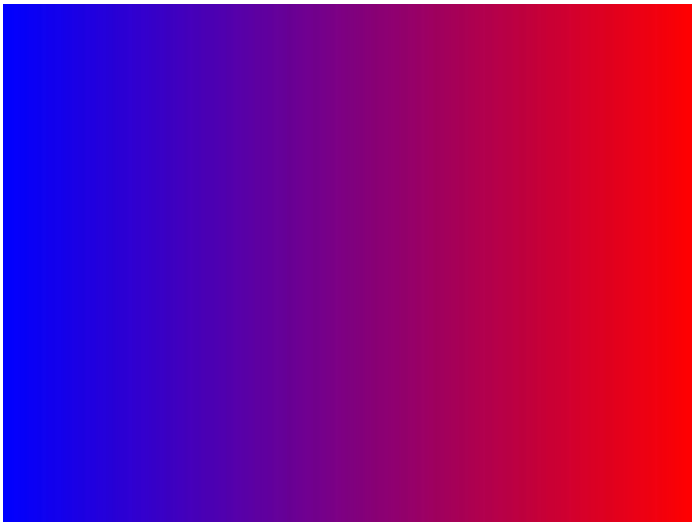
```
GAUGE_0 = OPT_NOPOINTER;
GAUGE_1 = OPT_NOBACK | OPT_NOTICKS;
cmd_gauge(80, 60, 50, GAUGE_0, 4, 4, 49152, 65535);
cmd(COLOR_RGB(255, 0, 0));
cmd_gauge(80, 60, 50, GAUGE_1, 4, 4, 49152, 65535);
```

增加一个定制的图形到仪表：先画背景，再画一个位图，然后再加上前景：



```
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);
```

5.31 CMD_GRADIENT –绘制一个平滑的颜色梯度



C 语言原型

```
void cmd_gradient( int16_t x0,
                   int16_t y0,
                   uint32_t rgb0,
                   int16_t x1,
                   int16_t y1,
                   uint32_t rgb1 );
```

参数

x0

指针 0 的 x 坐标，以像素为单位

y0

指针 0 的 y 坐标，以像素为单位

rgb0

指针 0 的颜色，格式是一个 24-bit RGB 数字。红色是最高位 8 bits，蓝色是最低位 8 bits。所以 0xff0000 是亮红色。

x1

指针 1 的 x 坐标，以像素为单位

y1

指针 1 的 y 坐标，以像素为单位

rgb1

指针 1 的颜色

描述

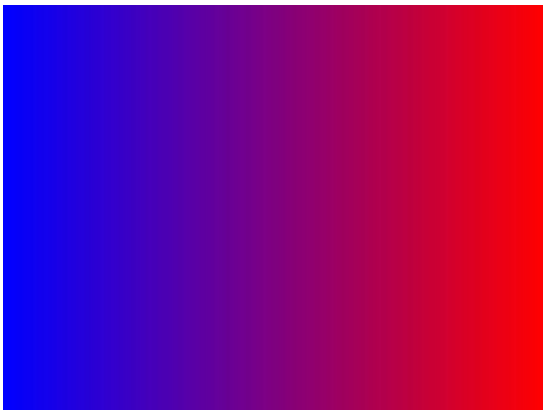
所有颜色的步进值是依据从 RGB0 到 RGB1 参数的内插平滑曲线。此平滑曲线公式是独立的计算三种颜色，此公式为 R0 + t * (R1 - R0)，t 是介于 0 到 1 之间的内插。梯度功能必须与剪刀功能一起用以得到想要的梯度显示效果。

指令布局

+0	CMD_GRAGIENT(0xffffff0b)
+4	X0
+6	Yo
+8	RGB0
+12	X1
+14	Y1
+16	RGB1

范例

一个从蓝到红的水平梯度



```
cmd_gradient(0, 0, 0x0000ff, 160, 0, 0xff0000);
```

一个垂直梯度

```
cmd_gradient(0, 0, 0x808080, 0, 120, 0x80ff40);
```

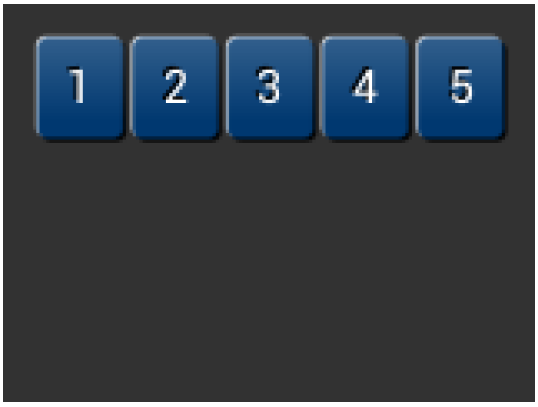
对角线梯度里一样的颜色

```
cmd_gradient(0, 0, 0x808080, 160, 120, 0x80ff40);
```

使用一个剪刀矩形画一个梯度条纹当作一个标题的背景：


```
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");
```

5.32 CMD_KEYS –绘制一行按键



C 语言原型

```
void cmd_keys( int16_t x,
               int16_t y,
               int16_t w,
               int16_t h,
               int16_t font,
               uint16_t options,
               const char* s );
```

参数

x

按键左上角的 x 坐标，以像素为单位

y

按键左上角的 y 坐标，以像素为单位

font

指示用在按键标签字体的位图句柄。有效范围是从 0 到 31

options

预设上，按键是以 3D 效果绘制而此选项的值为 0。OPT_FLAT 移除此 3D 效果。如果有 OPT_CENTER，按键们会以最小尺寸集中画在在 w x h 矩形里。否则，按键们会展开到全部填满可用的空间。如果有指定一个 ASCII 代码，按键会画成'按著的状态'。也就是说，移除任何背景颜色的 3D 效果。

w

按键的宽度

h

按键的高度

s

按键标签，一个按键一个字符。TAG 值是设定到每个按键的 ASCII 值，所以可以利用寄存器 REG_TOUCH_TAG 侦测按键有无按下。

描述

实体尺寸的详细信息为：

- 按键之间的空隙是 3 个像素。
- 对于 OPT_CENTERX 的情况来说，按键是(字体宽 + 1.5)像素宽，否则按键会以填满可用宽度的方式决定尺寸。

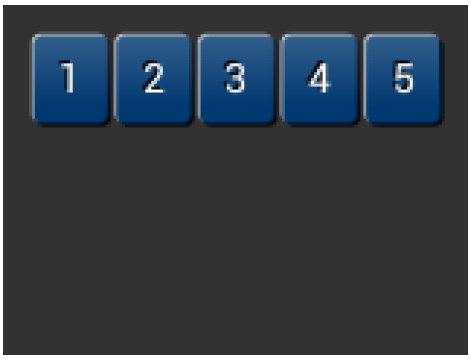
更多信息可参考 Co-processor engine widgets physical dimensions。

指令布局

+0	CMD_KEYS(0xffffff0e)
+4	X
+6	Y
+8	W
+10	H
+12	Font
+14	Options
+16	S
..	..
+n	0

范例

一行按键：

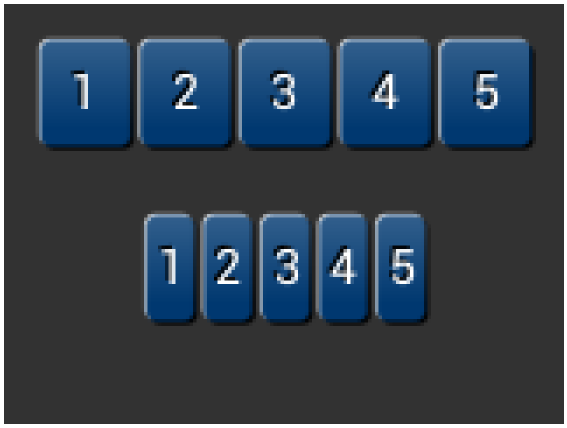


cmd_keys(10, 10, 140, 30, 26, 0, "12345");

没有 3D 效果的外观:

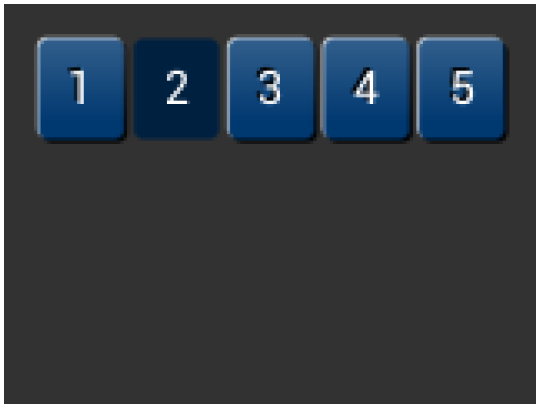
```
cmd_keys(10, 10, 140, 30, 26, OPT_FLAT,
"12345");
```

预设 vs. 集中:




```
cmd_keys(10, 10, 140, 30, 26, 0, "12345");
cmd_keys(10, 60, 140, 30, 26,
OPT_CENTER, "12345");
```

设定选项以使按键'2'呈现'按著的状态'('2'是 ASCII 代码 0x32):



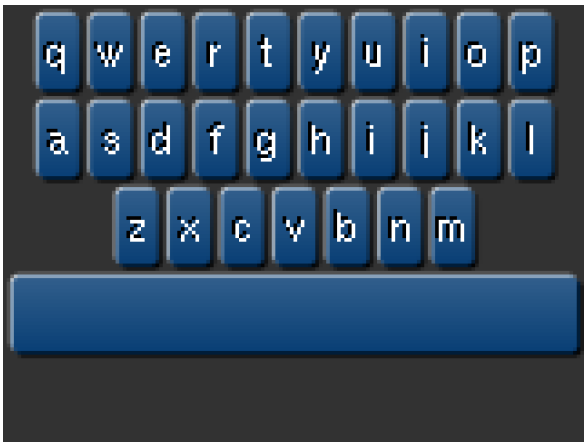

```
cmd_keys(10, 10, 140, 30, 26, 0x32,
"12345");
```

使用字体 29 的计算机风格键盘:


```
cmd_keys(22, 1, 116, 28, 29, 0, "789");
cmd_keys(22, 31, 116, 28, 29, 0, "456");
```

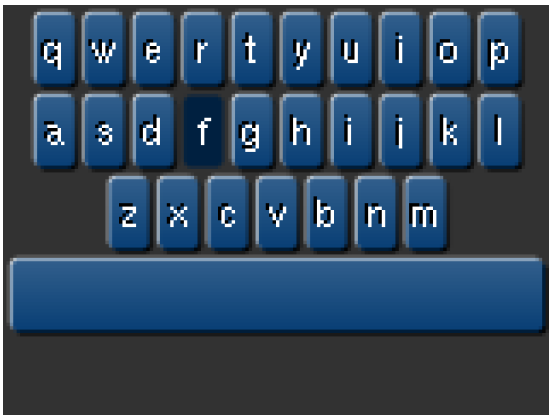
```
cmd_keys(22, 61, 116, 28, 29, 0, "123");
cmd_keys(22, 91, 116, 28, 29, 0, ".");
```

一个以字体 20 绘制的紧密键盘：




```
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, "");
```

使 f(ASCII 0x66)按键表现为按著的状态：




```
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, "");
```

5.33 CMD_PROGRESS –绘制一个进度条

C 语言原型

```
void cmd_progress( int16_t x,
                   int16_t y,
                   int16_t w,
                    int16_t h,
                    uint16_t options,
                    uint16_t val,
                    uint16_t range );
```

参数

x

进度条左上角的 x 坐标，以像素为单位

y

进度条左上角的 y 坐标，以像素为单位

w

进度条的宽度，以像素为单位

h

进度条的高度，以像素为单位

options

预设上，进度条是以 3D 效果绘制，而此选项的值为 0。选项 OPT_FLAT 移除此 3D 效果，其值为 256。

val

进度条的显示值，介于 0 到 range 值之间，包含 range 值

range

最大值

描述

实体尺寸的详细信息为：

- x,y,w,h 定义进度条的外侧尺寸。条的半径(r)是 min(w,h)/2。
- 内侧进度条的半径为 r*(7/8)

更多信息请参考 Co-processor engine widgets physical dimensions。

指令布局

+0	CMD_PROGRESS(0xffffff0f)
+4	X
+6	Y
+8	W
+10	H
+12	options
+14	val
+16	range

范例

一个显示为 50%完成的进度条：

```
cmd_progress(20, 50, 120, 12, 0, 50, 100);
```

没有 3D 效果的外观：

```
cmd_progress(20, 50, 120, 12, OPT_FLAT, 50, 100);
```

一个 4 像素高的进度条，范围 0-65535，背景为褐色：

```
cmd_bgcolor(0x402000);
cmd_progress(20, 50, 120, 4, 0, 9000, 65535);
```

5.34 CMD_SCROLLBAR –绘制一个卷动条

C 语言原型

```
void cmd_scrollbar( int16_t x,
```

```
int16_t y,
int16_t w,
int16_t h,
uint16_t options,
uint16_t val,
uint16_t size,
uint16_t range );
```

参数

x

卷动条左上角的 x 坐标，单位为像素

y

卷动条左上角的 y 坐标，单位为像素

w

卷动条的宽度，单位为像素。如果宽度大于高度，则卷动条以水平方式绘制

h

卷动条的高度，单位为像素。如果高度大于宽度，则卷动条以垂直方式绘制

options

预设上卷动条是以 3D 效果绘制，而此选项的值为 0。选项 OPT_FLAT 可移除此 3D 效果，其值为 256

val

卷动条的显示值，介于 0 到包含的范围内

range

最大值

描述

更多实体尺寸的信息请参考 CMD_PROGRESS。

指令布局

+0	CMD_SCROLLBAR(0xffffff11)
+4	X
+6	Y
+8	W
+10	H

+12	options
+14	val
+16	Size
+18	Range

范例

一个卷动条，指示 10-50%的范围：

```
cmd_scrollbar(20, 50, 120, 8, 0, 10, 40, 100);
```

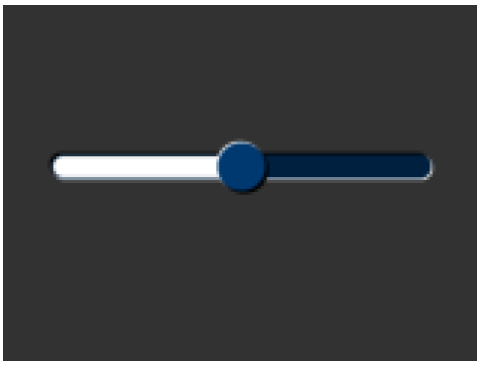
没有 3D 效果的外观:

```
cmd_scrollbar(20, 50, 120, 8, OPT_FLAT, 10, 40, 100);
```

一个褐色主题的垂直卷动条:

```
cmd_bgcolor(0x402000);
cmd_fgcolor(0x703800);
cmd_scrollbar(140, 10, 8, 100, 0, 10, 40, 100);
```

5.35 CMD_SLIDER – 绘制一个滑块



C 语言原型

```
void cmd_slider( int16_t x,
                 int16_t y,
                 int16_t w,
                 int16_t h,
                 uint16_t options,
                 uint16_t val,
                 uint16_t range );
```

参数

x

滑块左上角的 x 坐标，以像素为单位

y

滑块左上角的 y 坐标，以像素为单位

w

滑块的宽度，以像素为单位。如果宽度大于高度，则滑块以水平方式绘制

h

滑块的高度，以像素为单位。如果高度大于宽度，则滑块以垂直方式绘制

options

预设上滑块是以 3D 效果绘制。选项 OPT_FLAT 可移除此 3D 效果。

val

滑块的显示值，介于 0 到 range 值之间，包含 range 值。

range

最大值

描述

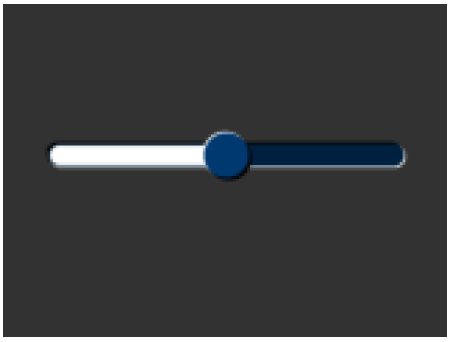
更多实体尺寸的信息可参考 CMD_PROGRESS。

指令布局

+0	CMD_SLIDER(0xffffff10)
+4	X
+6	Y
+8	W
+10	H
+12	options
+14	val
+16	Range

范例

一个设为指示 50%的滑块：

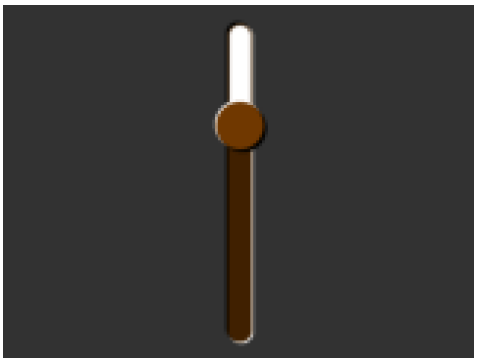


```
cmd_slider(20, 50, 120, 8, 0, 50, 100);
```

没有 3D 效果的外观：

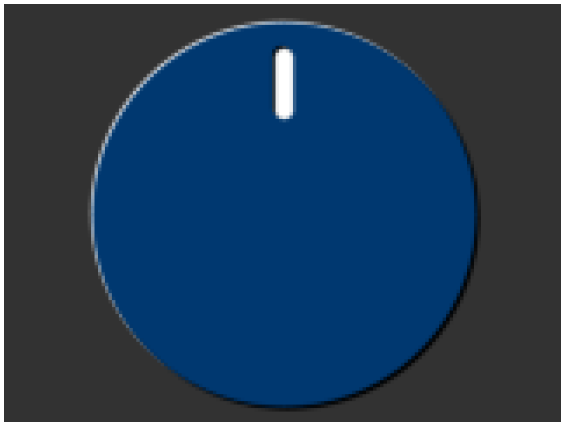
```
cmd_slider(20, 50, 120, 8, OPT_FLAT, 50, 100);
```

一个褐色主题的垂直滑块，范围介于 0 到 65535:



```
cmd_bgcolor(0x402000);
cmd_fgcolor(0x703800);
cmd_slider(76, 10, 8, 100, 0, 20000, 65535);
```

5.36 CMD_DIAL –绘制一个旋转拨号控制



C 语言原型

```
void cmd_dial( int16_t x,
               int16_t y,
               int16_t r,
               uint16_t options,
```

uint16_t val);

参数

x

拨号盘中心的 x 坐标，以像素为单位

y

拨号盘中心的 y 坐标，以像素为单位

r

拨号盘的半径，以像素为半径

Options

预设上拨号盘是以 3D 效果绘制，而此选项的值为 0。选项 OPT_FLAT 可移除此 3D 效果，其值为 256

val

透过介于 0 到 65535(包含 65535)的设定值，指定拨号点的位置。0 表示拨号点方向往下，0x4000 表示往左，0x8000 表示往上，0xc000 表示往右。

描述

实体尺寸的预设值为：

- 标示是一条宽度为 r*(12/256)，离中心距离为 r*(140/256)到 r*(210/256)之间的线。

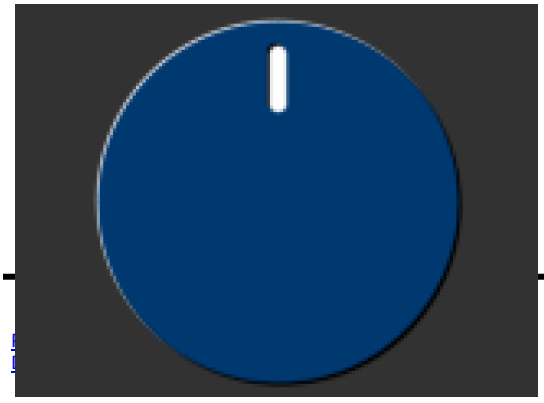
更多信息可参考 Co-processor engine widgets physical dimensions

指令布局

+0	CMD_DIAL(0xffffff2d)
+4	X
+6	Y
+8	r
+10	options
+12	val

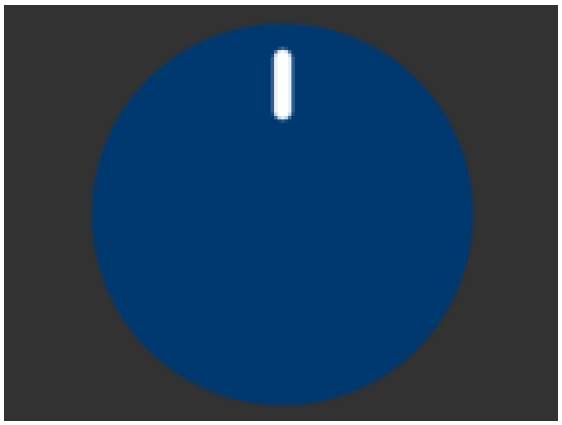
范例

一个设为 50%的拨号盘：



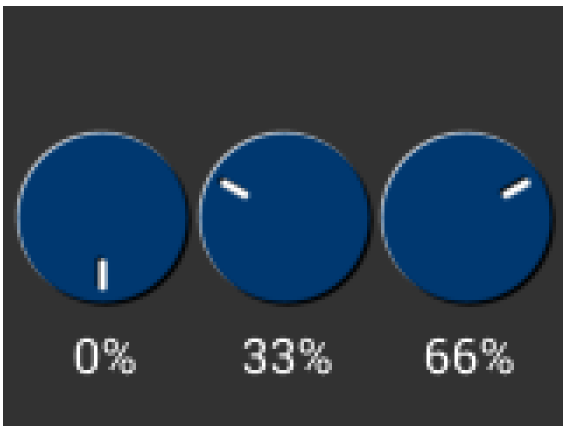
cmd_dial(80, 60, 55, 0, 0x8000);

没有 3D 效果的外观：



cmd_dial(80, 60, 55, OPT_FLAT, 0x8000);

拨号盘设定成 0%、33%、及 66%三种指示结果：



cmd_dial(28, 60, 24, 0, 0x0000);

cmd_text(28, 100, 26, OPT_CENTER, "0%");

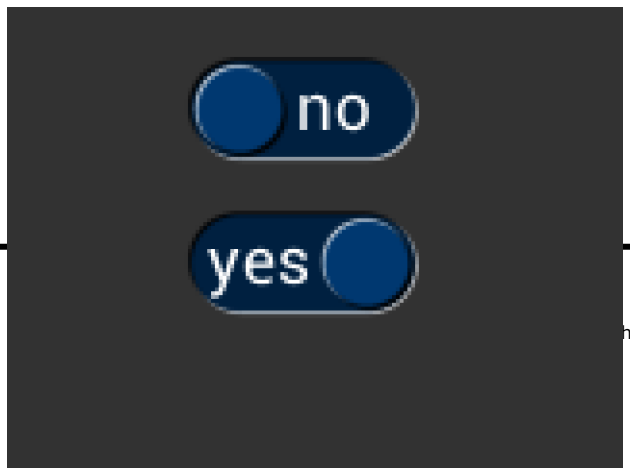
cmd_dial(80, 60, 24, 0, 0x5555);

cmd_text(80, 100, 26, OPT_CENTER, "33%");

cmd_dial(132, 60, 24, 0, 0xaaaa);

cmd_text(132, 100, 26, OPT_CENTER, "66%");

5.37 CMD_TOGGLE –绘制一个切换开关



C 语言原型

```
void cmd_toggle( int16_t x,
                 int16_t y,
                 int16_t w,
                 int16_t font,
                 uint16_t options,
                 uint16_t state,
                 const char* s );
```

参数

x

切换开关左上角的 x 坐标，以像素为单位

y

切换开关左上角的 y 坐标，以像素为单位

w

切换开关的宽度，以像素为单位

font

用于文字的字体。参考 ROM 及 RAM 字体

options

预设上切换开关是以 3D 效果绘制，而此选项的值为 0。选项 OPT_FLAT 可移除此 3D 效果，其值为 256

state

切换开关的状态：0 为关，65535 为开。

S

切换开关的字符串标签。字符值 255(在 C 语言可以写成\xff)可以分开两个标签。

描述

实体尺寸的详细信息为：

- 外围长条半径是字体高*(20/16)

- 旋扭半径是 r-1.5

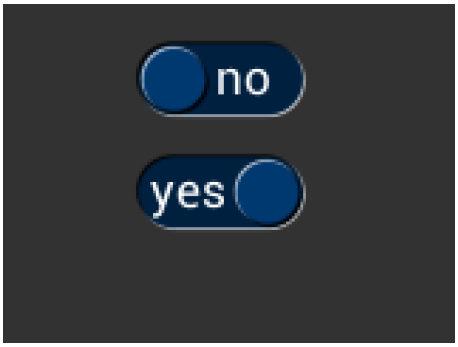
更多信息请参考 Co-processor engine widgets physical dimensions

指令布局

+0	CMD_TOGGLE(0xffffff12)
+4	X
+6	Y
+8	W
+10	Font
+12	Options
+14	State
+16	S
..	..
..	0

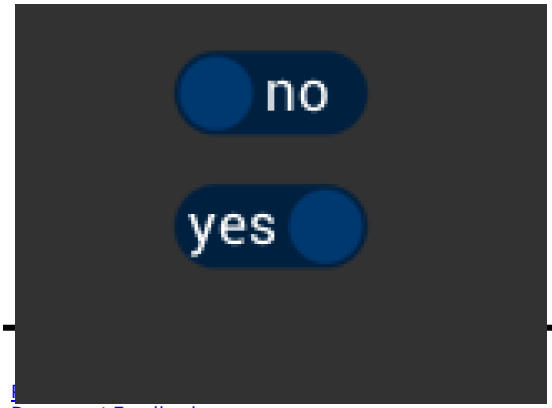
范例

使用中型字体，两种状态的结果



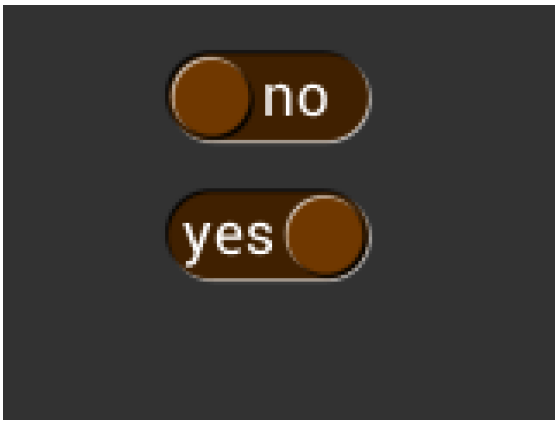

```
cmd_toggle(60, 20, 33, 27, 0, 0,"no" "\xff""yes");
cmd_toggle(60, 60, 33, 27, 0, 65535, "no" "\xff" "yes");
```

没有 3D 效果的外观




```
cmd_toggle(60, 20, 33, 27, OPT_FLAT, 0, "no" "\xff" "yes");
cmd_toggle(60, 60, 33, 27, OPT_FLAT, 65535,"no" "\xff" "yes");
```

前景与背景使用不同的色彩：




```
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”);
```

5.38 CMD_TEXT – 绘制文字

C 语言原型

```
void cmd_text( int16_t x,
               int16_t y,
               int16_t font,
               uint16_t options,
               const char* s );
```

参数

x

文字基底的 x 坐标，以像素为单位

y

文字基底的 y 坐标，以像素为单位

font

文字使用的字体，字体范围 0-31，参考 ROM 和 RAM 字体

options

预设(x,y)是文字左上角的像素，选项的预设值为 0。选项 OPT_CENTERX 水平集中文字，选项 OPT_CENTERY 垂直集中。OPT_CENTER 则同时针对垂直及水平两个方向集中。OPT_RIGHTX 向右调整像素位置，所以 x 变成最右边的像素。OPT_RIGHTX 的值是 2048。

文字字符串

文字字符串本身应测该要以一个 null 字符结束

指令布局

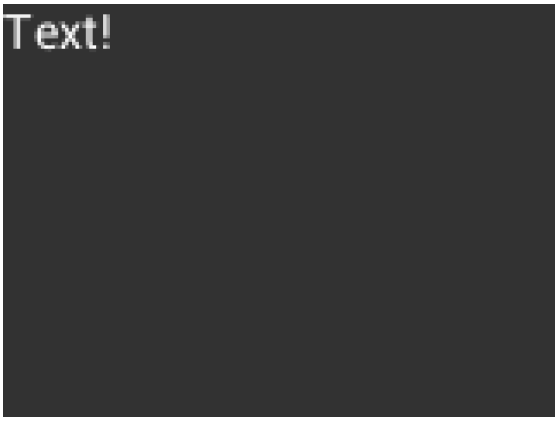
+0	CMD_TEXT(0xffffff0c)
+4	X
+6	Y
+8	Font
+10	Options
+12	S
..	..
..	0(null 字符以结束字符串)

范例

在坐标(0,0)的大字体纯文字：


```
cmd_text(0, 0, 31, 0, “Text!”);
```

使用小一点的字体：

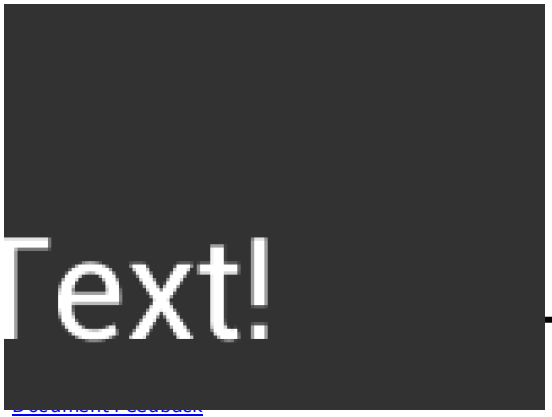


cmd_text(0, 0, 26, 0, “Text!”);

水平集中：

cmd_text(80, 60, 31, OPT_CENTERX, “Text!”);

向右调整像素位置：

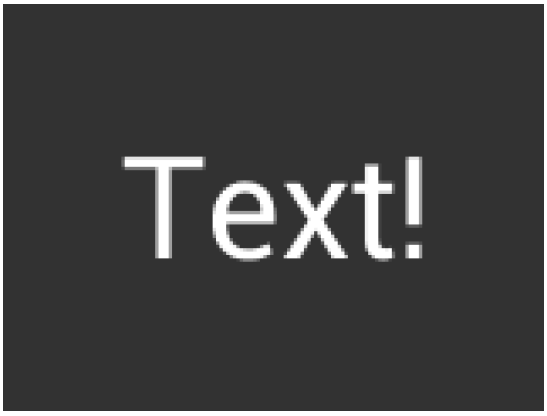


cmd_text(80, 60, 31, OPT_RIGHTX, “Text!”);

垂直集中：

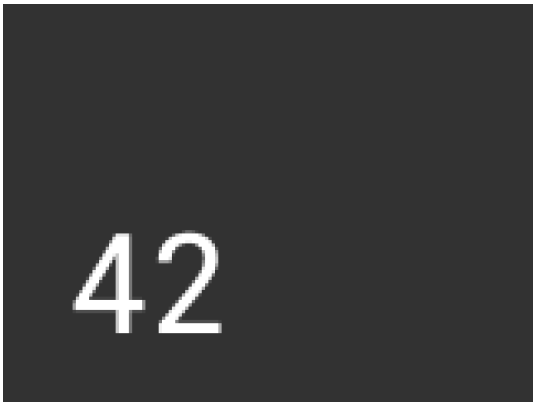
cmd_text(80, 60, 31, OPT_CENTERY, “Text!”);

同时向水平及垂直两个方向集中：



cmd_text(80, 60, 31, OPT_CENTER, “Text!”);

5.39 CMD_NUMBER – 绘制一个十进位数字



C 语言原型

```
void cmd_number( int16_t x,
                 int16_t y,
                 int16_t font,
                 uint16_t options,
                 int32_t n );
```

参数

x

文字基底的 x 坐标，以像素为单位

y

文字基底的 y 坐标，以像素为单位

font

供文字用的字体，介于 0-31。可参考 ROM 及 RAM 字体

options

预设上，(x,y)是文字左上角的像素。选项 OPT_CENTERX 水平集中文字，选项 OPT_CENTERY 垂直集中。OPT_CENTER 则同时针对垂直及水平两个方向集中。OPT_RIGHTX 向右调整像素位置，所以 x 变成最右边的像素。预设上数字显示没有前零串，但若在选项里指定一个 1-9 范围的宽度，则会视需要加上补垫的前零串，以符合给定的宽度。如果有选项 OPT_SIGNED，则数字被视为有符号的，若值为负，会前缀一个负号。

n

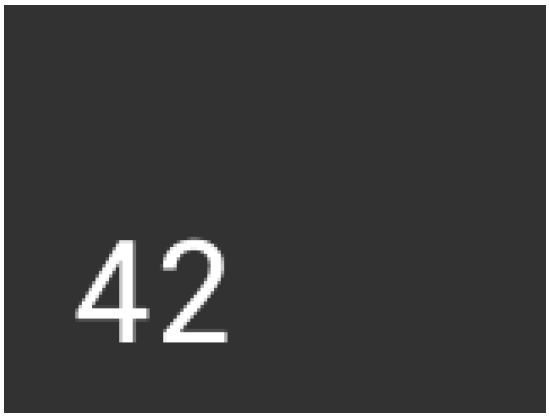
要显示的数字，格式是无符号的或有符号的 32-bit

指令布局

+0	CMD_NUMBER(0xffffff2e)
+4	X
+6	Y
+8	Font
+10	Options
+12	n

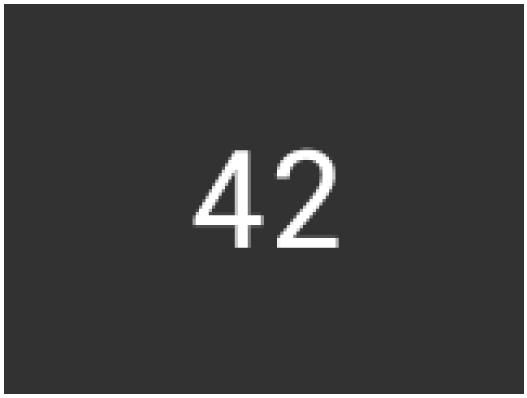
范例

一个数字：



cmd_number(20, 60, 31, 0, 42);

集中：



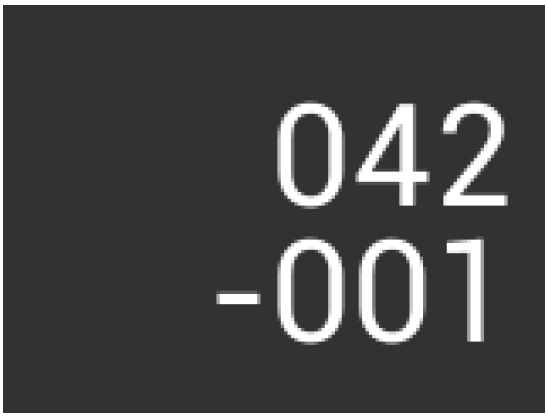
cmd_number(80, 60, 31, OPT_CENTER, 42);

正数与负数的有符号输出：

cmd_number(20, 20, 31, OPT_SIGNED, 42);

cmd_number(20, 60, 31, OPT_SIGNED, -42);

强迫宽度成 3 位，位置向右调整



cmd_number(150, 20, 31, OPT_RIGHTX | 3, 42);

cmd_number(150, 60, 31, OPT_SIGNED | OPT_RIGHTX | 3, -1);

CMD_LOADIDENTI–Y - 设定目前的矩阵成为单位矩阵。这个指令指示 FT800 的协处理器引擎将目前的矩阵设成单位矩阵，如此协处理器引擎才能依下列指令的要求形成新的矩阵：MD_SCALE、 CMD_ROTATE、CMD_TRANSLATE。更多有关单位矩阵的信息，请参考位图变换矩阵的章节。

C 语言原型

void cmd_loadidentity( );

指令布局

+0	CMD_LOADIDENTITY(0xffffff26)

5.40 CMD_SETMATRIX-写入目前的矩阵到显示清单

协处理器引擎透藉由产生显示清单指令 BITMAP_TRANSFORM_A-F，分派目前矩阵的值到图形引擎的位图变换矩阵。在这个指令之后，接著的位图渲染操作会被新的变换矩阵影响。

C 语言原型

void cmd_setmatrix( );

指令布局

+0	CMD_SETMATRIX(0xffffff2a)

Parameter

无

5.41 CMD_GETMATRIX -检索目前的矩阵系数

要在协处理器引擎的上下文检索目前的矩阵。请注意，协处理器引擎的上下文里目前的矩阵不会应用到位图变换，直到目前的矩阵透过 CMD_SETMATRIX 通过图形引擎。

C 语言原型

```
void cmd_getmatrix( int32_t a,
                    int32_t b,
                    int32_t c,
                    int32_t d,
                    int32_t e,
                    int32_t f );
```

参数

a

输出参数；以矩阵系数 a 写入。格式请参考指令 BITMAP_TRANSFORM_A 的参数 a。

b

输出参数；以矩阵系数 b 写入。格式请参考指令 BITMAP_TRANSFORM_B 的参数 b。

c

输出参数；以矩阵系数 c 写入。格式请参考指令 BITMAP_TRANSFORM_C 的参数 c。

d

输出参数；以矩阵系数 d 写入。格式请参考指令 BITMAP_TRANSFORM_D 的参数 d。

e

输出参数；以矩阵系数 e 写入。格式请参考指令 BITMAP_TRANSFORM_E 的参数 e。

f

输出参数；以矩阵系数 f 写入。格式请参考指令 BITMAP_TRANSFORM_F 的参数 f。

指令布局

+0	CMD_GETMATRIX(0xffffff33)
+4	A
+8	B
+12	C
+16	D
+20	E
+24	F

5.42 CMD_GETPTR –取得解压缩数据的结束存储器地址

C 语言原型

```
void cmd_getptr( uint32_t result
                 );
```

参数

result

CMD_INFLATE 解压缩数据的结束地址。

解压缩数据开始地址，由 CMD_INFLATE 指令指定，而解压缩数据的结束地址也可以利用这个指令取得。

这是一个输出的参数，可被 CMD_GETPTR 指令输入到 RAM_CMD。

指令布局

+0	CMD_GETPTR (0xffffff23)
+4	result

范例

```
cmd_inflate(1000);//解压缩数据到  RAM_G + 1000
......//接著zlib压缩数据
While(rd16(REG_CMD_WRITE)!=  rd16(REG_CMD_READ));//等待直到压缩完成

uint16_t x = rd16(REG_CMD_WRITE);
uint32_t ending_address =0;
cmd_getptr(0);
ending_address = rd32(RAM_CMD +  x +4);
```

源代码片段 13 CMD_GETPTR 指令范例

5.43 CMD_GETPROPS -取得 CMD_LOADIMAGE 解压缩的影像特性

C 语言原型

void cmd_getprops(uint32_t &ptr, uint32_t &width, uint32_t &height);

参数

ptr

在 RAM_G 里，由这个指令之前最后一个 CMD_LOADIMAGE 指令解压缩影像的地址。这是一个输出参数。

width

由这个指令之前最后一个 CMD_LOADIMAGE 指令解压缩影像的宽度。这是一个输出参数。

height

由这个指令之前最后一个 CMD_LOADIMAGE 指令解压缩影像的宽度。这是一个输出参数。

指令布局

+0	CMD_GETPROPS (0xffffff25)
+4	ptr
+8	width
+12	Height

描述

这个指令用来检索 CMD_LOADIMAGE 解压缩影像的属性。当这个指令成功执行之后，所有参数会被一一输出。

范例

请参考 CMD_GETPTR

5.44 CMD_SCALE –对目前的矩阵做一个缩放

C 语言原型

```
void cmd_scale( int32_t sx,
                int32_t sy );
```

参数

sx

x 坐标缩放因子，形式是有符号的 16.16 bit 定点

sy

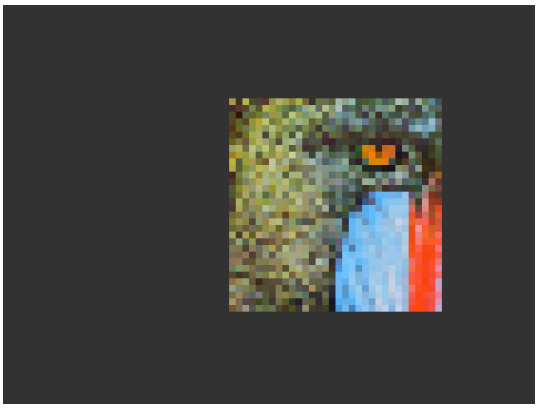
y 坐标缩放因子，形式是有符号的 16.16 bit 定点

指令布局

+0	CMD_SCALE(0xffffff28)
+4	sx
+8	sy

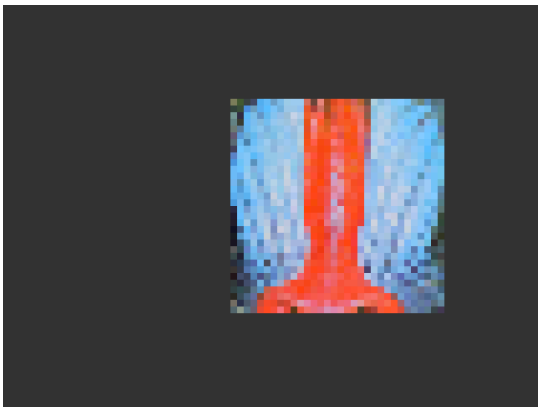
范例

将一个位图放大成 2X:



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_scale(2 * 65536, 2 * 65536);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

在一个位图中心点放大成 2X：



```
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));
```

5.45 CMD_ROTATE -将目前矩阵做旋转

C 语言原型

void cmd_rotate(int32_t a);

参数

a

顺时针旋转角度, 单位为(360/65536)度

指令布局

+0	CMD_ROTATE(0xffffff29)
+4	a

范例

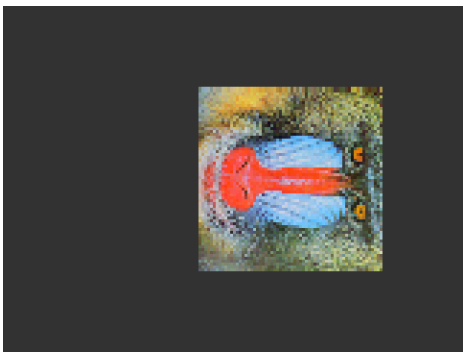
以位图左上角为准，顺时针旋转位图 10 度：

```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_rotate(10 * 65536 / 360);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

以位图左上角为准，逆时针旋转位图 33 度：

```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_rotate(-33 * 65536 / 360);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

旋转一个 64 x 64 位图，以其中心为准：



```
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));
```

5.46 CMD_TRANSLATE -将目前矩阵做一个转移

C 语言原型

```
void cmd_translate( int32_t tx,
                    int32_t ty );
```

参数

tx

x 坐标转移因子，形式为有符号的 16.16 bit 定点。

ty

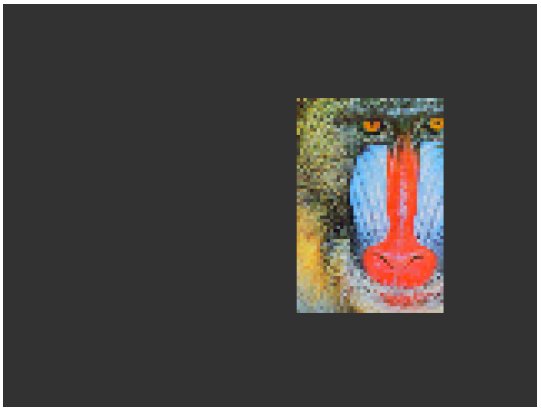
y 坐标转移因子，形式为有符号的 16.16 bit 定点。

指令布局

+0	CMD_TRANSLATE(0xffffff27)
+4	Tx
+8	Ty

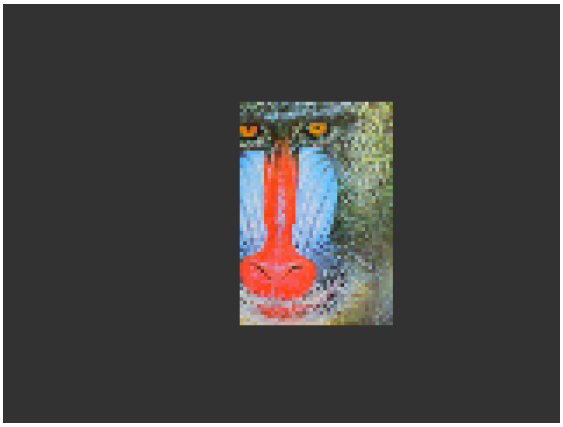
范例

将一个位图向右转移 20 个像素：



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_translate(20 * 65536, 0);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

将一个位图向左转移 20 个像素：



```
cmd(BEGIN(BITMAPS));
cmd_loadidentity();
cmd_translate(-20 * 65536, 0);
cmd_setmatrix();
cmd(VERTEX2II(68, 28, 0, 0));
```

5.47 CMD_CALIBRATE -执行一个触屏校正的例行工作

校正的程序从屏幕收集三个触摸点，然后计算并载入一个适当的矩阵到 REG_TOUCH_TRANSFORM_A-F。为了使用它，创建一个显示清单然后使用 CMD_CALIBRATE。协处理器引擎叠加触摸目标到目前的显示清单上，收集校正的输入并更新 REG_TOUCH_TRANSFORM_A-F。

C 语言原型

```
void cmd_calibrate( uint32_t result );
```

参数

result

输出参数；若是失败的校正，会写入 0。

此功能是否完成，可透过侦测 REG_CMD_READ 是否等于 REG_CMD_WRITE。

指令布局

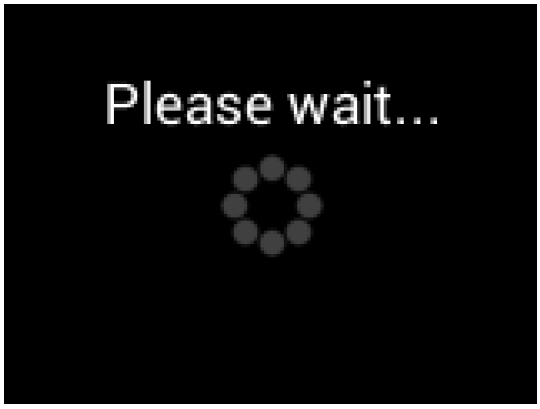
+0	CMD_CALIBRATE(0xffffff15)
+4	result

范例

```
cmd_dlstart();
cmd(CLEAR(1,1,1));
cmd_text(80,30,27, OPT_CENTER,"Please tap on the dot");
cmd_calibrate();
```

源代码片段 14 CMD_CALIBRATE 范例

5.48 CMD_SPINNER – 开始一个动态转盘



转盘是一个动态的叠加，可向使用者表现某个任务仍在继续。要触发转盘，要创建一个显示清单然后用指令 CMD_SPINNER。协处理器引擎叠加转盘到目前的显示清单上，交换显示清单使其变为可视，然后动画会一直持续到 CMD_STOP。REG_MACRO_0 及 REG_MACRO_1 寄存器可用于表示动画类的效果。点移动的频率是依据配置的显示帧而定。

通常，480x272 显示屏幕的显示速率大约是 60fps。风格 0 且速率 60fps，点会在 2 秒内一直重覆序列。风格 1 且速率 60fps，点会在 1.25 秒内一直重覆序列。风格 2 且速率 60fps，时钟指针会在 2 秒内一直重覆序列。风格 3 且速率 60fps，移动点会在 1 秒内一直重覆序列。

注意，在一个时间点，CMD_SKETCH、CMD_SCREENSAVER、或 CMD_SPINNER 这三个指令只能有一个有作用。

C 语言原型

```
void cmd_spinner( int16_t x,
                  int16_t y,
                  uint16_t style,
                  uint16_t scale );
```

指令布局

+0	CMD_SPINNER(0xffffff16)

+4	X
+6	Y
+8	Style
+10	Scale

参数

X

转盘左上角的 x 坐标

Y

转盘左上角的 y 坐标

Style

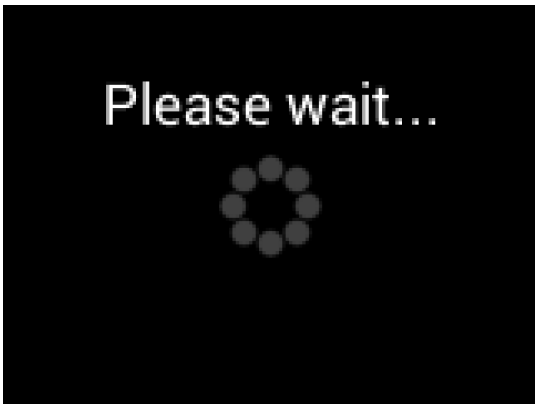
转盘的风格。有效范围是从 0 到 3。

Scale

转盘的缩放系数。0 表示没有缩放。

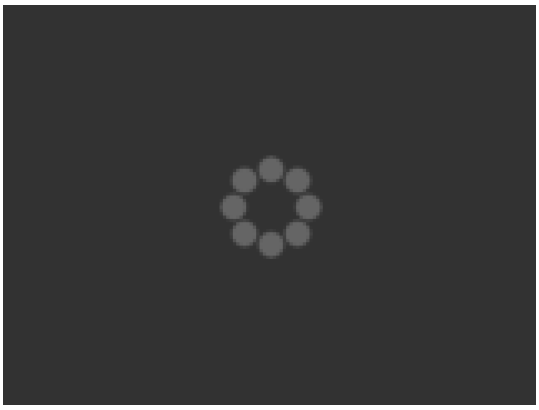
范例

创建一个显示清单，然后开始转盘效果：



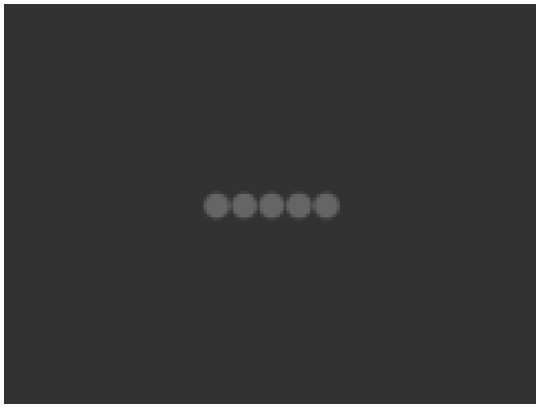

```
cmd_dlstart();
cmd(CLEAR(1,1,1));
cmd_text(80, 30, 27, OPT_CENTER, "Please wait...");
cmd_spinner(80, 60, 0, 0);
```

转盘风格 0，点绕成的圆：



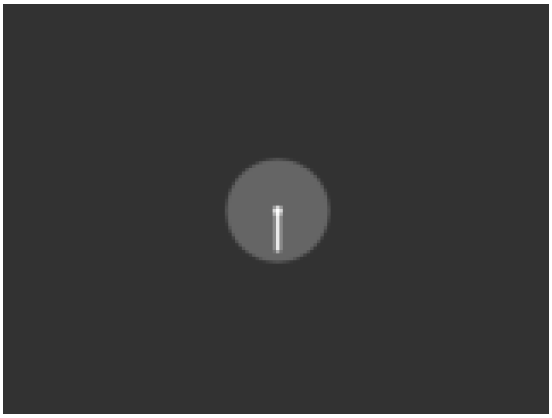
cmd_spinner(80, 60, 0, 0);

风格 1，点构成的线：



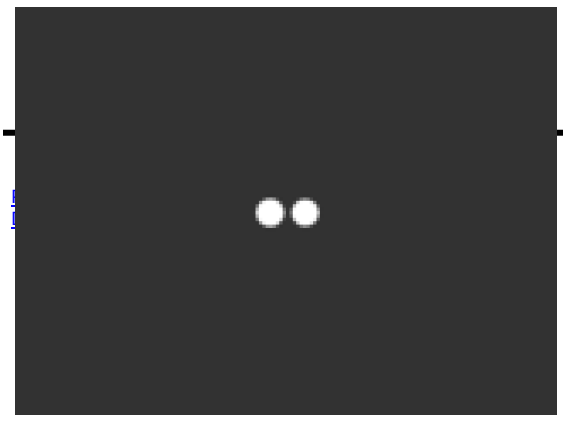
cmd_spinner(80, 60, 1, 0);

风格 2，一个转动的时钟指针：



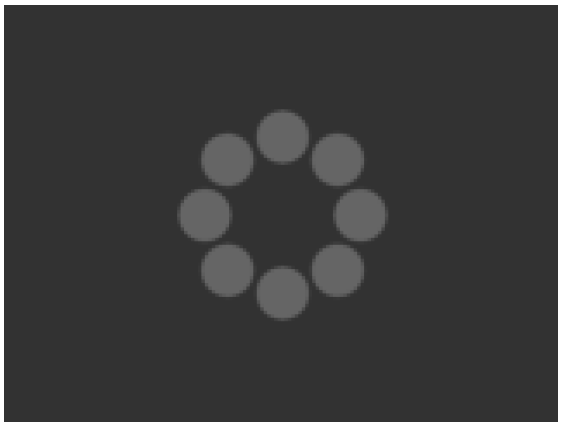
cmd_spinner(80, 60, 2, 0);

风格 3，两个点绕轨运行：



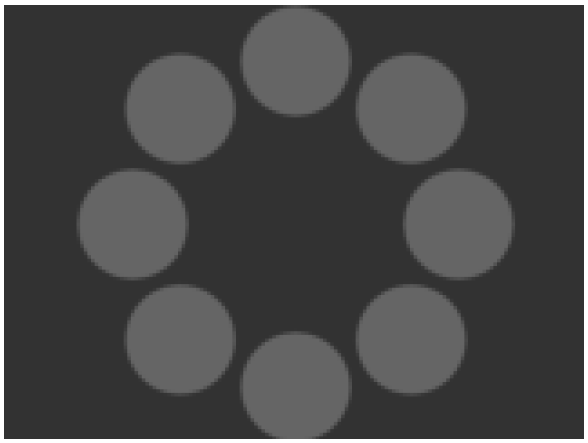
```
cmd_spinner(80, 60, 3, 0);
```

半屏幕，缩放 1：



```
cmd_spinner(80, 60, 0, 1);
```

全屏幕，缩放 2:



```
cmd_spinner(80, 60, 0, 2);
```

5.49 CMD_SCREENSAVER -开始一个动态屏幕保护程序

在屏幕保护指令后，协处理器引擎持续地以 VERTEX2F 指令将变化中的(x,y)坐标更新到寄存器 REG_MACRO_0。这会使得位图在屏幕上绕转，而不用任何 MCU 的工作。CMD_STOP 指令停止这个更新过程。

注意，在一个时间点，CMD_SKETCH、CMD_SCREENSAVER、或 CMD_SPINNER 这三个指令只能有一个有作用。

C 语言原型

```
void cmd_screensaver( );
```

描述

REG_MACRO_0 是依据显示帧的频率更新(取决于显示寄存器的配置)。通常地，对于 480x272 大小的显示器，帧速率大约是每秒 60 个帧。

指令布局

+0	CMD_SCREENSAVER(0xffffff2f)

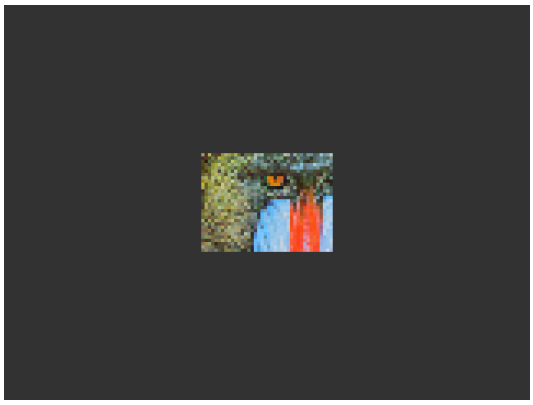
范例

要开始屏幕保护，用 MACRO 指令创建一个显示清单 - 协处理器引擎会持续更新：

```
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());
```

源代码片段 15 CMD_SCREENSAVER 范例

这是结果:



5.50 CMD_SKETCH -开始一个连续草图更新

执行素描指令后，协处理器引擎会持续地取样触摸输入然后根据触摸的(x,y)坐标，将像素描绘到位图。这表示不需要任何 MCU 的工作，使用者的触摸输入即可描绘到位图里。CMD_STOP 指令可停止这个素描过程。

注意，在一个时间点，CMD_SKETCH、CMD_SCREENSAVER、或 CMD_SPINNER 这三个指令只能有一个有作用。

这个指令可以应用在 FT800 及 FT801。建议使用者使用 CMD_SKETCH 指令，因为已经针对电容式触摸作了最佳化。

C 语言原型

```
void cmd_sketch( int16_t x,
                 int16_t y,
                 uint16_t w,
                 uint16_t h,
                 uint32_t ptr,
                 uint16_t format );
```

参数

x

素描区域左上角的 x 坐标，以像素为单位

y

素描区域左上角的 y 坐标，以像素为单位

w

素描区域的宽度，以像素为单位

h

素描区域的高度，以像素为单位

ptr

素描位图的基底地址

format

素描位图的格式，亦或是 L1 或是 L8

描述

请注意在图形存储器里，位图数据的更新频率是决定于 FT800 的内置电路 - 模数转换器的取样频率，此取样频率可以达到 1000 赫兹。

指令布局

+0	CMD_SKETCH(0xffffff30)

+4	X
+6	Y
+8	W
+10	H
+12	Ptr
+16	Format

范例

开始素描到一个 480x272 L1 位图里：

```
cmd_memzero(0,480*272/8);
cmd_sketch(0,0,480,272,0, L1);

//然后显示位图
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));

//最后，停止素描更新
cmd_stop();
```

源代码片段 16 CMD_SKETCH EXAMPLE

5.51 CMD_STOP -停止任何动态转盘、屏幕保护、草图

这个指令是用来通知协处理器引擎停止由 CMD_SKETCH、CMD_SPINNER、或 CMD_SCREENSAVER 触发的周期性操作。

C 语言原型

void cmd_stop( );

指令布局

+0	CMD_STOP(0xffffff17)

参数

无

描述

对于指令 CMD_SPINNER 及 CMD_SCREENSAVER，寄存器 REG_MACRO_0 及 REG_MACRO_1 会停止更新。

对于 CMD_SKETCH 及 CMD_CSKETCH，RAM_G 里的位图数据会停止更新。

范例

参考 CMD_SKETCH,CMD_CSKETCH、CMD_SPINNER、CMD_SCREENSAVER

5.52 CMD_SETFONT -设定一个定制的字体

CMD_SETFONT 是用来注册一个自定义的位图字体到 FT800 的协处理器引擎里。在注册后，FT800 协处理器引擎就能利用其协处理器指令使用此位图字体。

有关如何建立一个定制的字体，请参考 ROM 及 RAM 字体。

C 语言原型

```
void cmd_setfont( uint32_t font,
                  uint32_t ptr );
```

指令布局

+0	CMD_SETFONT(0xffffff2b)
+4	font
+8	ptr

参数

font

从 0 到 14 的位图句柄。位图句柄 15 可以持续地使用。请参考 4.6 节

ptr

在 RAM 的度量区块地址。要求对齐到 4 字节的整数。

范例

一个合适的字体度量区块已经载入到 RAM 里的地址 1000，要建立起来当作字体 7，与物件一起使用：

```
cmd_setfont(7,1000);
cmd_button(20,20,                          // x,y
       120,40,                             //宽度、高度，单位为像素
       7,                                  // 字体7，刚载入
       0,                                  // 预设选项，3D风格
       "customfont!");
```

源代码片段 17 CMD_SETFONT

5.53 CMD_TRACK -追踪触摸以供图形物件使用

这个指令能使协处理器引擎分派一个标记值追踪特定图形物件的触摸。然后，协处理器引擎会周期性以 LCD 显示板的帧速率更新寄存器 REG_TRACKER。

协处理器引擎以旋转追踪模式及线性追踪模式追踪图形物件：

- 旋转追踪模式 - 追踪触摸点与标记值指的图形物件中心点之间的角度。其值是一个 360 度圆的 1/65536。以中心点为准，0 表示角度往下，0x4000 表示角度往左，0x8000 表示角度往上，0xC000 表示角度往右。
- 线性追踪模式 - 如果参数 w 比 h 还大，追踪触摸点到标记值指定图形物件宽度的相对距离。如果参数 w 不比 h 大，追踪触摸点到标记值指定图形物件高度的相对距离。单位是图形物件的宽度或高度的 1/65536。触摸点距离是指从物件左上角像素到触摸点坐标的距离。

C 语言原型

```
void cmd_track( int16_t x,
                int16_t y,
                int16_t w,
                int16_t h,
                int16_t tag );
```

参数

x

对于线性追踪功能，是指追踪区域左上角的 x 坐标，以像素为单位。

对于旋转追踪功能，是指追踪区域中心点的 x 坐标，以像素为单位。

y

对于线性追踪功能，是指追踪区域左上角的 y 坐标，以像素为单位。

对于旋转追踪功能，是指追踪区域中心点的 y 坐标，以像素为单位。

w

追踪区域的宽度，以像素为单位

h

追踪区域的高度，以像素为单位

请注意：

(1,1)的 w 和 h 表示追踪为旋转追踪模式，且会在 REG_TRACKER 回报一个角度值。(0,0)的 w 和 h 关闭协处理器引擎的追踪功能。

tag

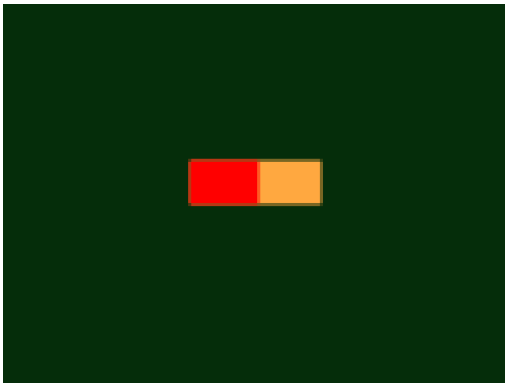
要追踪的图形物件标记, 范围是 1-255

指令布局

+0	CMD_TRACK(0xffffff2c)
+4	X
+6	Y
+8	W
+10	h
+12	tag

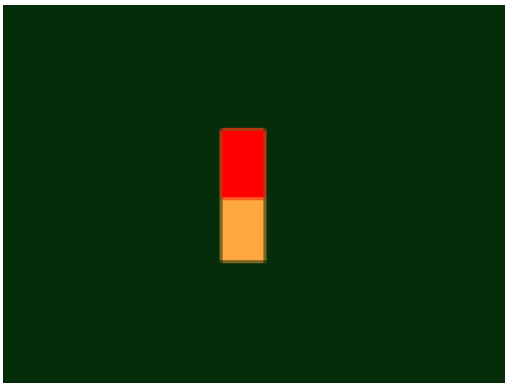
范例

一个尺寸 40x12 像素矩形的水平追踪，而目前的触摸是在 50%：



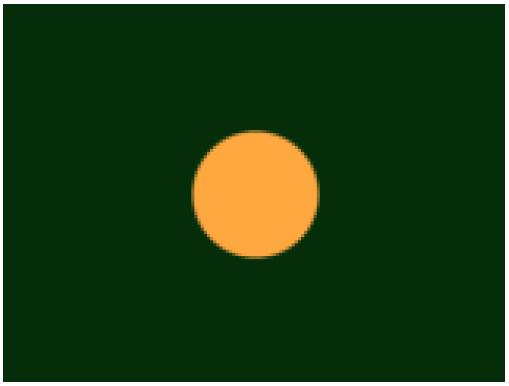
```
dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1 ,1 ,1) );
dl( BEGIN(RECTS) );
dl( VERTEX2F(60 * 16,50 * 16) );
dl( VERTEX2F(100 * 16,62 * 16) );
dl( COLOR_RGB(255, 0, 0) );
dl( VERTEX2F(60 * 16,50 * 16) );
dl( VERTEX2F(80 * 16,62 * 16) );
dl( COLOR_MASK(0 ,0 ,0 ,0) );
dl( TAG(1) );
dl( VERTEX2F(60 * 16,50 * 16) );
dl( VERTEX2F(100 * 16,62 * 16) );
cmd_track(60 * 16, 50 * 16, 40, 12, 1);
```

一个尺寸 12x40 像素矩形的垂直追踪，而目前的触摸是在 50%：



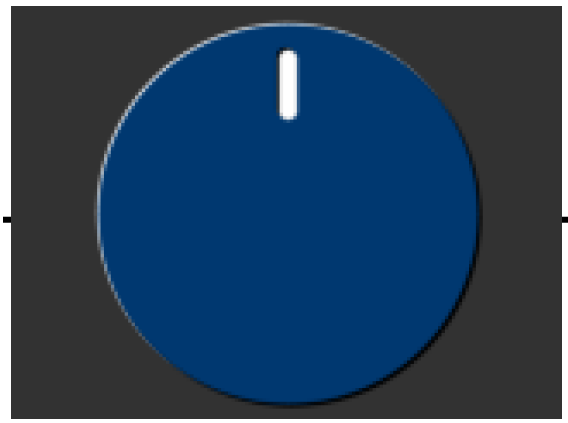
```
dl( CLEAR_COLOR_RGB(5, 45, 110) );
dl( COLOR_RGB(255, 168, 64) );
dl( CLEAR(1 ,1 ,1) );
dl( BEGIN(RECTS) );
dl( VERTEX2F(70 * 16,40 * 16) );
dl( VERTEX2F(82 * 16,80 * 16) );
dl( COLOR_RGB(255, 0, 0) );
dl( VERTEX2F(70 * 16,40 * 16) );
dl( VERTEX2F(82 * 16,60 * 16) );
dl( COLOR_MASK(0 ,0 ,0 ,0) );
dl( TAG(1) );
dl( VERTEX2F(70 * 16,40 * 16) );
dl( VERTEX2F(82 * 16,80 * 16) );
cmd_track(70 * 16, 40 * 16, 12, 40, 1);
```

以(80,60) 的显示位置为中心的圆形追踪



```
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 * 16, 60 * 16, 1, 1, 1);
```

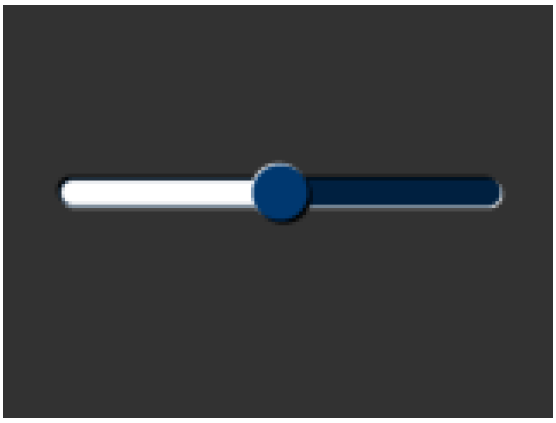
绘制一个标记 33、中心点在(80, 60)的拨号盘，可用触摸调整：



```
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 = trackeI> 16;
...
}
```

创建一个标记 34 的可调整滑块：



```
uint16_t val = 0x8000;
cmd_track(20, 50, 120, 8, 34);
Ile (1) {
...
cmd(TAG(34));
cmd_slider(20, 50, 120, 8, val, 65535);
...
uint32_t tracker = rd32(REG_TRACKER);
if ((tracker & 0xff) == 33)
val = tracker >> 16;
...
}
```

_

5.54 CMD_SNAPSHOT -对目前的屏幕截图

这个指令使协处理器引擎针目前的屏幕截图，然后将结果写入 RAM_G，当作一个 ARGB4 的位图。这个位图的尺寸是屏幕的尺寸，值由寄存器 REG_HSIZE 及 REG_VSIZE 提供。

在截图的过程，要把 REG_PCLK 设定为 0 关闭显示，以避免显示故障。

因为协处理器引擎需要将结果写入目的地址址，目的地址址一定不能被图形引擎使用或是参考。

C 语言原型

```
void cmd_snapshot( uint32_t ptr );
```

参数

ptr

截图的目的地地址，在 RAM_G 里。

指令布局

+0	CMD_SNAPSHOT(0xffffff1f)
+4	ptr

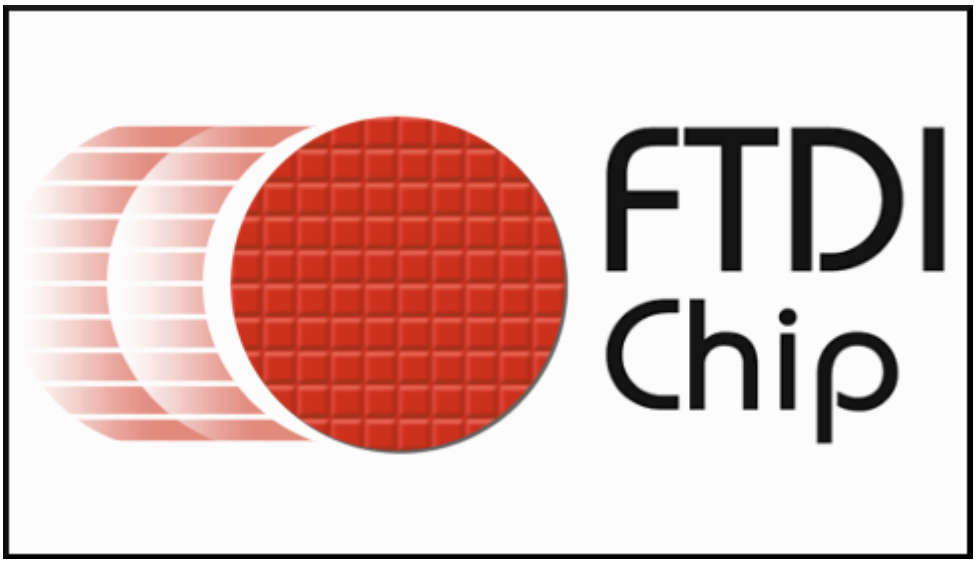
范例

对目前 160 x 120 屏幕作截图，并在新的显示清单里，当作一个位图使用：

```
wr(REG_PCLK,0);//关闭PCLK
wr16(REG_HSIZE,120);
wr16(REG_WSIZE,160);

cmd_snapshot(0);//截屏

wr(REG_PCLK,5);//开启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));
```

源代码片段 18 CMD_SNAPSHOT 160X120-屏幕

5.55 CMD_LOGO –播放 FTDI 标志的动画



标志指令使协处理器引擎播发一个段 FTDI 标志的动画。在标志播放的过程中，MCU 不应该使用任何 FT800 的资源。在经过 2.5 秒后，协处理器引擎对 REG_CMD_READ 及 REG_CMD_WRITE 写入零，然后等待指令。在这个指令完成后，MCU 应写入下一个指令到 RAM_CMD 的开始地址。

C 语言原型

void cmd_logo( );

指令布局

+0	CMD_LOGO(0xffffff31)

范例

播放标志动画：

```
cmd_logo();
delay(3000);          // 选择性的等待
While((0!=rd16(REG_CMD_WRITE))&&
(rd16(REG_CMD_WRITE)!=  rd16(REG_CMD_READ)));//等待直到读出&写入两个指针寄存器都等于零
```

源代码片段 19 CMD_LOGO 指令范例

6 FT801 操作

6.1 FT801 介绍

FT800 和 FT801 有完全一样的图形及音频功能集。FT800 芯片的触摸能力是设计给电阻屏上的控制触摸，而 FT801 是给电容触摸并可允许达到 5 个触摸点。然而，FT801 有与 FT800 不一样的触摸引擎及触摸控制寄存器组。所有名字以 "REG_TOUCH"开头的寄存器已经被分派成新的名字, 叫" REG_CTOUCH"。

6.2 FT801 触控引擎

FT801 有新的电容触屏引擎(CTSE)，内置以下功能：

- 连到电容触屏模组(CTPM)的 I^2C 介面
- 同时可支持 5 个触摸点
- 与 Focaltech FT5x06 系列或 Azotech IQS5xx 系列驱动芯片支持 CTPM
- 相容模式和延伸模式

预设上，FT801 触摸引擎工作在相容模式，只能侦测一个触摸点。在延伸模式下，FT801 触摸引擎可以同时侦测 5 个触摸点。

6.3 FT801 触控寄存器

FT801 已经再次定义 FT800 的触摸寄存器，如下所示：

寄存器定义 76 REG_CTOUCH_MODE 定义

REG_CTOUCH_MODE 定义	
保留	读/写

31 2 1 0

地址: 0x1024F0 复位值: 0x3

Bit 0 - 1: 主机可以设定这两个 bits 控制 FT80 触摸引擎的触摸引擎取样模式，如下：00: 关闭模式。没有取样发生。
01: 无定义。
10: 无定义。
11: 开启模式。

Bit 2 – 31: 保留

寄存器定义 77 REG_CTOUCH_EXTENDED 定义

REG_CTOUCH_EXTEND 定义	
保留	读/写

31 1 0

地址: 0x1024F4 复位值: 0x1

Bit 0:这个 bit 控制 FT801 触摸引擎的侦测模式，如下：
0:延伸模式，多触摸侦测模式，最多可侦测 5 个触摸点
1:相容模式，单触摸侦测模式

寄存器定义 78 REG_CTOUCH_TOUCH0_XY 定义

REG_CTOUCH_TOUCH0_XY 定义	
只读	只读

31 16 15 0

地址: 0x102510 复位值: 0x80008000

Bit 0 -15:这些 bits 的值是第一个触摸点的 Y 坐标
Bit 16 -31:这些 bits 的值是第一个触摸点的 X 坐标

注意：这个寄存器是可供延伸模式及相容模式应用。对相容模式，这个寄存器反应唯一触控点的位置。

寄存器定义 79 REG_CTOUCH_TOUCH1_XY 定义

REG_CTOUCH_TOUCH1_XY 定义	
只读	只读
31 16	15 0

地址: 0x102508 复位值: 0x80008000

Bit 0 -15:这些 bits 的值是第二个触摸点的 Y 坐标
Bit 16 -31:这些 bits 的值是第二个触摸点的 X 坐标

注意：这个寄存器是只供延伸模式应用。

寄存器定义 80 REG_CTOUCH_TOUCH2_XY 定义

REG_CTOUCH_TOUCH2_XY 定义	
只读	只读
31 16	15 0

地址: 0x102574 复位值: 0x80008000

Bit 0 -15:这些 bits 的值是第三个触摸点的 Y 坐标
Bit 16 -31:这些 bits 的值是第三个触摸点的 X 坐标

注意：这个寄存器是只供延伸模式应用。

寄存器定义 81 REG_CTOUCH_TOUCH3_XY 定义

REG_CTOUCH_TOUCH3_XY 定义	
只读	只读
31 16	15 0

地址: 0x102578 复位值: 0x80008000

Bit 0 -15:这些 bits 的值是第四个触摸点的 Y 坐标
Bit 16 -31:这些 bits 的值是第四个触摸点的 X 坐标

注意：这个寄存器是只供延伸模式应用。

寄存器定义 82 REG_CTOUCH_TOUCH4_X 定义

REG_CTOUCH_TOUCH4_X 定义	
只读	
31	0

地址: 0x102538 复位值: 0x8000

Bit 0 -15:这些 bits 的值是第五个触摸点的 Y 坐标

注意：这个寄存器是只供延伸模式应用。

寄存器定义 83 REG_CTOUCH_TOUCH4_Y 定义

REG_CTOUCH_TOUCH4_Y 定义	
只读	
31	0
地址: 0x10250C	复位值: 0x8000
Bit 0 -15:这些 bits 的值是第五个触摸点的 Y 坐标	
注意：这个寄存器是只供延伸模式应用。	

- **REG_CTOUCH_TRANSFORM_A 定义**

REG_CTOUCH_TRANSFORM_A 与 REG_TOUCH_TRANSFORM_A 有一样的定义。
更多细节请参考 REG_TOUCH_TRANSFORM_A

- **REG_CTOUCH_TRANSFORM_B 定义**

REG_CTOUCH_TRANSFORM_B 与 REG_TOUCH_TRANSFORM_B 有一样的定义。
更多细节请参考 REG_TOUCH_TRANSFORM_B

- **REG_CTOUCH_TRANSFORM_C 定义**

REG_CTOUCH_TRANSFORM_C 与 REG_TOUCH_TRANSFORM_C 有一样的定义。
更多细节请参考 REG_TOUCH_TRANSFORM_C

- **REG_CTOUCH_TRANSFORM_D 定义**

REG_CTOUCH_TRANSFORM_D 与 REG_TOUCH_TRANSFORM_D 有一样的定义。
更多细节请参考 REG_TOUCH_TRANSFORM_D

- **REG_CTOUCH_TRANSFORM_E 定义**

REG_CTOUCH_TRANSFORM_E 与 REG_TOUCH_TRANSFORM_E 有一样的定义。
更多细节请参考 REG_TOUCH_TRANSFORM_E

- **REG_CTOUCH_TRANSFORM_F 定义**

REG_CTOUCH_TRANSFORM_F 与 REG_TOUCH_TRANSFORM_F 有一样的定义。
更多细节请参考 REG_TOUCH_TRANSFORM_F

注意: 校准应只能在相容模式(预设)下执行，对于电阻式显示板也是如此。

- **REG_CTOUCH_RAW_XY 定义**

REG_CTOUCH_RAW_XY 定义	
只读	只读
31 16	15 0
地址: 0x102508	复位值: 0xFFFFFFFF
Bit 0 -15:这些 bits 的值是触摸点的 Y 坐标，但是指经过变换矩阵之前 Bit 16 -31:这些 bits 的值是触摸点的 X 坐标，但是指经过变换矩阵之前 注意：这个寄存器是只供延伸模式应用。	

- **REG_CTOUCH_TAG 定义**

这个寄存器在两种模式都可用。在延伸模式，在延伸模式，只有第一个触摸点，也就是 REG_CTOUCH_TOUCH0_XY 可用来询问标记值并以结果更新寄存器。这个寄存器与 REG_TOUCH_TAG 有相同的定义。

6.4 寄存器概要

表 13 触摸寄存器映射表格

FT801- C Mode	FT801 – E Mode	Default Value (C Mode)	Default Value (Extend Mode)	FT800	Default Value	Address	Bit width
REG_CTOUCH_EXTEND	REG_CTOUCH_EXTEND	0x1	0x0	REG_TOUCH_ADC_MODE	0x01	1058036	4 bytes
REG_CTOUCH_TOUCH0_XY	REG_CTOUCH_TOUCH0_XY	0x80008000	0x80008000	REG_TOUCH_SCREEN_XY	0x80008000	1058064	4 bytes
REG_CTOUCH_RAW_XY	REG_CTOUCH_TOUCH1_XY	0xFFFFFFFF	0x80008000	REG_TOUCH_RAW_XY	0xFFFFFFFF	1058056	4 bytes
NA	REG_CTOUCH_TOUCH2_XY	0x0	0x80008000	REG_TOUCH_DIRECT_XY	0x0	1058164	4 bytes
NA	REG_CTOUCH_TOUCH3_XY	NA	0x80008000	REG_TOUCH_DIRECT_Z1Z2	NA	1058168	4 bytes
NA	REG_CTOUCH_TOUCH4_X	0x0	0x8000	REG_ANALOG	0x0	1058104	2 bytes
NA	REG_CTOUCH_TOUCH4_Y	0x7FFF	0x8000	REG_TOUCH_RZ	0x7FFF	1058060	2 bytes
REG_CTOUCH_TRANSFORM_A	REG_CTOUCH_TRANSFORM_A	0x10000	0x10000	REG_TOUCH_TRANSFORM_A	0x10000	1058076	4 bytes
REG_CTOUCH_TRANSFORM_B	REG_CTOUCH_TRANSFORM_B	0x0	0x0	REG_TOUCH_TRANSFORM_B	0x0	1058080	4 bytes
REG_CTOUCH_TRANSFORM_C	REG_CTOUCH_TRANSFORM_C	0x0	0x0	REG_TOUCH_TRANSFORM_C	0x0	1058084	4 bytes
REG_CTOUCH_TRANSFORM_D	REG_CTOUCH_TRANSFORM_D	0x0	0x0	REG_TOUCH_TRANSFORM_D	0x0	1058088	4 bytes
REG_CTOUCH_TRANSFORM_E	REG_CTOUCH_TRANSFORM_E	0x10000	0x10000	REG_TOUCH_TRANSFORM_E	0x10000	1058092	4 bytes
REG_CTOUCH_TRANSFORM_F	REG_CTOUCH_TRANSFORM_F	0x0	0x0	REG_TOUCH_TRANSFORM_F	0x0	1058096	4 bytes
REG_CTOUCH_TAG	REG_CTOUCH_TAG	0x0	0x0	REG_TOUCH_TAG	0x0	1058072	4 bytes
Note: C Mode: Compatibility Mode, default mode after FT801 reset E Mode: Extented Mode							

6.5 校正

CMD_CALIBRATE 启始的校正过程只可用在相容模式。然而，校正过程的结果在相容模式及延伸模式都可应用。因此，建议使用者在进入延伸模式之前完成校正过程。

在校正过程做好之后，寄存器 REG_CTOUCH_TRANSFORM_A~F 会被更新为变换矩阵的系数

6.6 CMD_CSKETCH–电容式触摸的特定草图

这个指令与 CMD_SKETCH 指令有相同功能，差别在 CMD_CSKETCH 有针对电容式触屏作最佳化。因为电容式触屏以较低的取样频率(大约 100 赫兹)报告坐标，相对于电阻式触屏，描述功能以较低的频率作更新。CMD_CSKETCH 引入了线性内插的演算法，当在绘制输出线条的时候，可以有较平滑的效果。

请注意这个指令不能应用在 FT800 芯片。

C 语言原型

```
void cmd_csketch( int16_t x,
                  int16_t y,
                  uint16_t w,
                  uint16_t h,
                  uint32_t ptr,
                  uint16_t format,
                  uint16_t freq);
```

指令布局

+0	CMD_CSKETCH(0xffffff35)
+4	X
+6	Y
+8	W
+10	H
+12	Ptr
+16	Format
+18	Freq

参数

x

素描区域左上角的 x 坐标，以像素为单位

y

素描区域左上角的 y 坐标，以像素为单位

w

素描区域的宽度，以像素为单位

h

素描区域的高度，以像素为单位

ptr

素描位图的基底地址

format

素描位图的格式，是 L1 或是 L8

freq

超取样的频率，典型值是 1500 以确保线能够平滑地连接。若值为 0 表示没有超取样操作。

描述

这个指令只会对 FT801 芯片有效。FT801 协处理器会以'freq'频率超取样由电容屏报告的坐标，然后以较平滑的效果形成线条。

范例

参考 CMD_SKETCH 范例

附录 A –参考文件

1) FT800 Datasheet: DS_FT800_Embedded_Video_Engine
2) OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.4
3) FT801 Datasheet: DS_FT801
4) Application note of FT800 FT801 Internal Clock Trimming:
   AN_299_FT800_FT801_Internal_Clock_Trimming

附录 B –首字母缩略词及缩写

Terms	描述
CS	Chip select
DL/dl	Display list
EVE	Embedded Video Engine
GPIO	General Purpose Input/output
Hz/KHz/MHz	Hertz/Kilo Hertz/Mega Hertz
I2C	Inter-Integrated Circuit
LSB	least significant bit
MCU	Micro controller unit
MSB	most significant bit
OS	operating system
PWM	Pulse-width modulation
PWR	Power
RAM	Random access memory
RGB	Red Blue Green
SPI	Serial Peripheral Interface
USB	Universal Serial Bus
USB-IF	USB Implementers Forum
RO	Read only
fps	Frame per second

附录 C –存储器映射

开始地址	结束地址	尺寸	名称	描述
00 0000h	03 FFFFh	256kB	RAM_G	主要图形 RAM
0C 0000h	0C 0003h	4 B	ROM_CHIPID	FT800 芯片识别及修订信息： Byte [0:1] 芯片 ID: “0800” Byte [2:3] 版本 ID: “0100” FT801 芯片识别及修订信息： Byte [0:1] 芯片 ID: “0801” Byte [2:3] 版本 ID: “0100”
0B B23Ch	0F FFFBh	275 kB	ROM_FONT	字体表和位图
0F FFFCh	0F FFFFh	4 B	ROM_FONT_ADDR	字体表指针地址
10 0000h	10 1FFFh	8 kB	RAM_DL	显示清单 RAM
10 2000h	10 23FFh	1 kB	RAM_PAL	模板 RAM
10 2400h	10 257Fh	380 B	REG_*	寄存器
10 8000 h	10 8FFFh	4 kB	RAM_CMD	图形引擎指令缓冲器
1C 2000 h	1C 27FFh	2 kB	RAM_SCREENSHOT	截图读出缓冲器

注意 1: 这个表格之外的地址是被保留的，不该做读写动作，除非有另外说明。

注意 2: ROM_CHIPID 从 ROM_FONT 地址空间使用了一部份的阴影地址。

附录 D –修订历史

文件标题: FT800 Series Programmer Guide (Simplified Chinese)

文件参考号码: FT_000793

放行号码.: FTDI#349

产品页面: http://www.ftdichip.com/FTProducts.htm

文件反馈意见: Send Feedback

修订	修改	日期
0.1	初稿释出	2012-08-01
2.0	增加 FT801 内容	2016-03-23