高通汉显®

GTHMI

GT5GL64A
GT5GL64U
标准GUI芯片

— 产品规格书 —

V 1.0

2024-06

www.hmi.gaotongfont.cn

版本修订记录

版本号	修改内容	日期	备注
V 1.0	规格书制定	2024-06	

目录

1 概述

GT5GL64系列是一款动态配置HMI资源的GUI芯片，需要与GT-HMI Designer 软件代码生成功能配合使用，快速开发嵌入式交互图形界面。可以简化嵌入式开发步骤，实现低代码开发，降低开发成本，同时提高开发效率，为您的嵌入式系统提供丰富且引人入胜的用户体验，轻松实现嵌入式 GUI 创意的搭建。

GT5GL64系列使用GT-HMI Designer动态配置字库及图片资源，为用户提供64Mb的存储空间，可支持高通全系列字库，例如中文、拉丁文、日文、韩文、希腊文、西里尔文、希伯来文、阿拉伯文、泰文等，用户可根据自身需求选择使用。

1.1 主要特性

- GT-HMI Designer内嵌下位机技术框架编译器，可自动生成下位机代码。
- 提供GT-HMI模块和GUI-LCD开发板，已适配GT-HMI上下位机软件及驱动，可以用于前期开发调试，也可以直接作为显示模块使用；
- 多平台兼容，移植便捷；
- 上位机设计软件GT-HMI Designer：
  1. 可直接生成交互代码，免去写代码的繁琐工作
  2. 模拟器仿真即见即所得
  3. 内置了大量常用的组件，如按钮、文本框、进度条、单选框等
  4. 支持高通全系列灰度/点阵/矢量字库，支持中外文及小语种，多种字号及字体。
  5. 内置中英文及数字输入法
  6. 持续更新的GUI示例库和UI资源库
  7. 支持跟指滑屏操作。
- 下位机技术框架GT-HMI Engine
  1. 纯C语言编写，使用无门槛
  2. 小巧高效，不限平台，最小仅需24K RAM+32K FLASH，可运行在Cortex-ARM M0\M3等小资源平台
  3. 移植便捷，切换平台只需移植定时器、TP和LCD接口，并提供移植教程及示例。
- 丰富详实的例程，配套的开发套件易于上手
- 支持自定义功能开发，可用GT-HMI Engine自定义控件和功能，组合进GT-HMI Designer生成的标准代码。

1.2 芯片规格

- 数据总线：SPI串行总线接口，Quad SPI串行总线接口
- 工作电压：2.7V~3.6V
- 电流：

  工作电流：读电流 20mA(max)

  写电流 30mA(max)

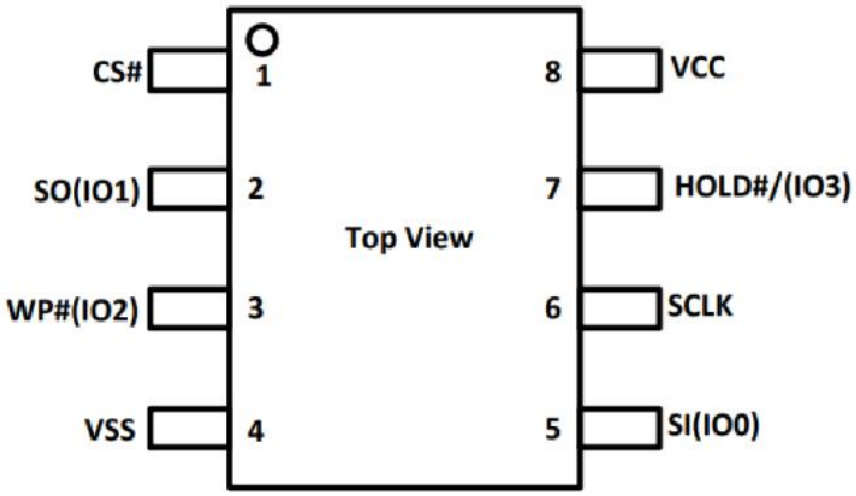
  睡眠电流：15uA(Max)
- 工作温度：-40℃~85℃
- 封装类型：

  GT5GL64A: SOP8-150mil

  GT5GL64U: USON8 (4*3*0.55mm)
- 10万次擦写循环寿命、20年数据记忆保留

1.3 引脚配置

SOP8 / USON8



1.4 引脚描述

NO.	名称	I/O	描述
1	CS#	I	片选输入（Chip enable input）
2	SO（IO1）	I / O	串行数据输出（Serial data output） 4x I/O读取模式输入和输出
3	WP#（IO2）	I / O	写保护低电平有效 4x I/O读取模式输入和输出
4	VSS		地(Ground)
5	SI（IO0）	I / O	串行数据输入（Serial data input） 4x I/O读取模式输入和输出
6	SCLK	I	串行时钟输入（Serial clock input）
7	HOLD#/(IO3)	I / O	总线挂起（Hold, to pause the device without） 4x I/O读取模式输入和输出
8	VCC		电源(+ 3.3V Power Supply)

Note：

1. **SI(IO0) ~ SO(IO1)适用于标准SPI。**
2. **IO0 ~ IO3用于Quad SPI 指令，WP#和HOLD#仅适用于标准SPI。**

2 GUI 工具使用-HMI

2.1 GUI工具概述

GT-HMI 是高通专为 GUI 用户开发的界面设计和编辑工具，包含 GT-HMI Designer 和 GT-HMI Engine 两款软件，上位机 GT-HMI Designer 自带图形库并包含多种控件供开发者使用，可以帮助用户快速创作出富有创意的 UI 图形交互效果。下位机 GT-HMI Engine 是一款高效的嵌入式 UI 引擎，可以帮助用户更快的构建、集成和优化 UI 应用程序；GT-HMI 工具为开发者提供了创新、高效的解决方案，让用户轻松实现自己的图形界面想法，有助于用户节省大量开发时间，提升开发效率，降低开发成本，从而增强产品的竞争力和用户体验，是开发者在使用 GUI 时的第一选择。

2.2 GT5GL64芯片与HMI工具使用

GT5GL64芯片可配合我司GT-HMI Designer上位机使用，用户使用GT-HMI Designer开发好GUI界面交互后，点击编译可生成resource.bin图片和字库资源包，将resource.bin资源包用烧录器烧录到GT5GL64芯片。

使用GT5G系列芯片可以使用GUI-LCD（GUI-LCD上面自带一颗GT5G芯片）直接在我们移植好的示例工程上配合GUI-LCD开发板进行开发，提高开发效率。也可以使用客户的自己的单片机进行开发，后面介绍 HMI 界面移植。

获取 GUI-HMI 工具的方式及视频教程：

1. 软件下载链接：高通字库官方网站高通字库-首页 (https://www.hmi.gaotongfont.cn)
2. 说明书下载： 高通字库-GT-HMI Designer 用户手册 (https://www.hmi.gaotongfont.cn/kfgj)
3. 软件视频教程：Https://space.bilibili.com/3493293474188211/channel/collectiondetail?sid=3159444

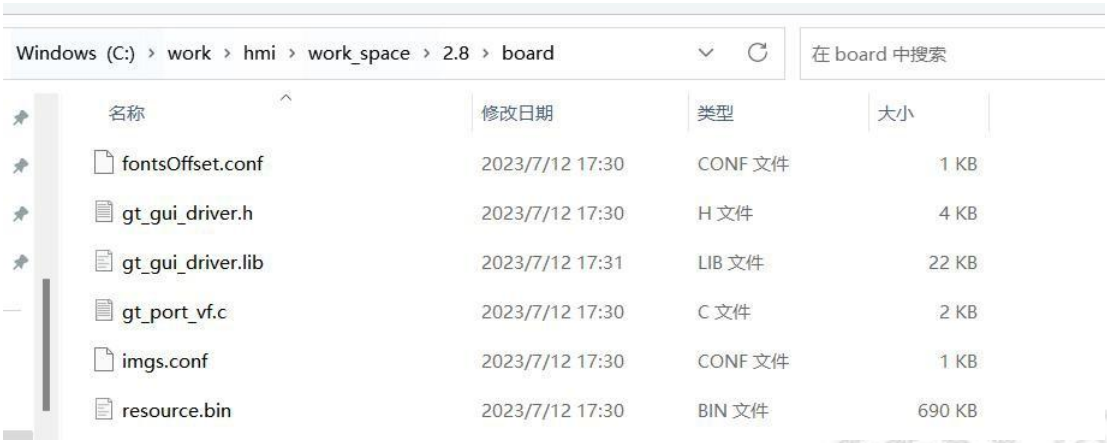
2.3 GT-HMI Designer 上位机代码移植

下位机配合 GT-HMI Designer 上位机设计界面非常快，本小节讲解下位机与上位机配合使用。第一步：首先在 GT-HMI Designer 上新建工程设计界面，设计界面教程可参考“GT-HMI Designer 用户手册”。

第二步：GT-HMI Designer 工程目录下的 screen 目录是每个界面的程序源码，需要把这部分源码添加到 keil5 工程上，在 main 函数里面初始化 gt_ui_init()函数接口。这里还没完，还需要将 board 目录的 gt_port_vf.c，gt_gui_driver.lib 和 gt_gui_driver.h 替换工程 GT-HMI-Engine\driver 下的同名文件。

第三步：将 board 目录下的 resource.bin 烧录到存储芯片里面去。

图 screen 目录下的文件

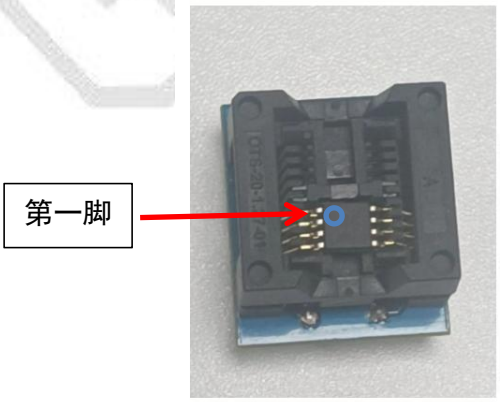


烧录 bin 文件步骤如下：

第一步：备好如下烧录器，在烧录接的是最下面四行， 右边表格表示烧录器接存储芯片的引脚号。

1	8
2	7
3	6
4	5

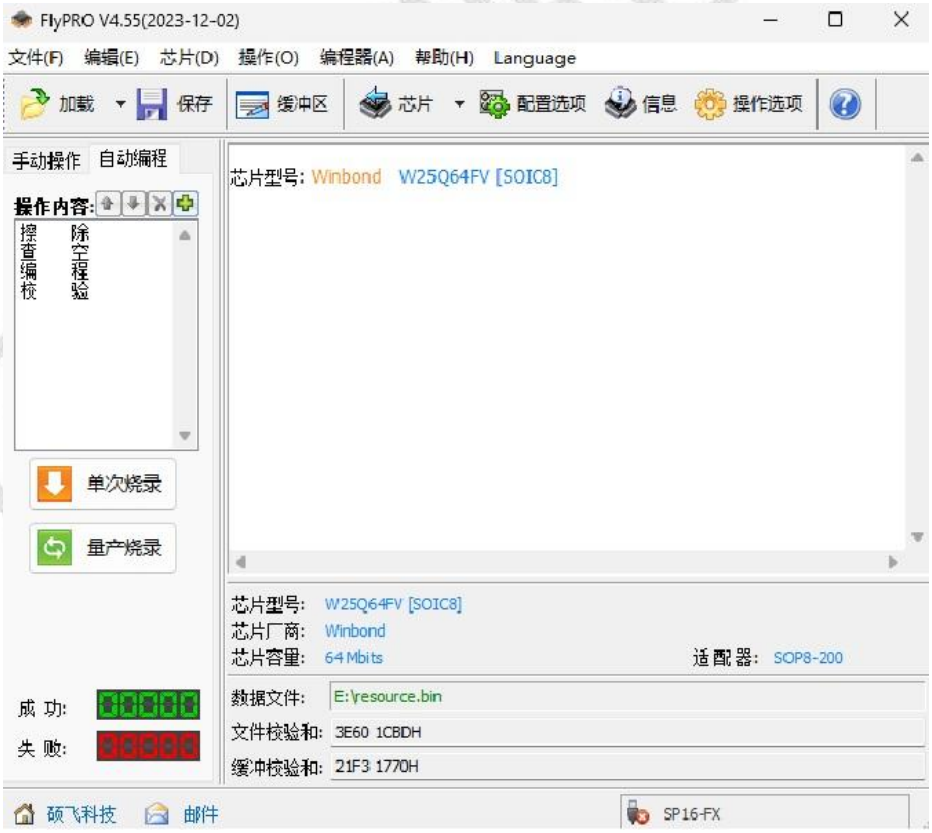
第二步：使用对应烧录座将烧录器和存储芯片引脚连接起来， 右边表格表示座子对应芯片的引脚号。芯片表面的圆点应和右边表格第1脚对应。



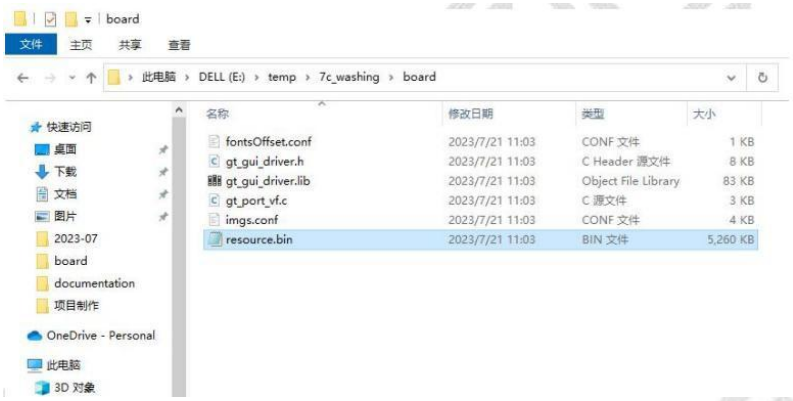
1	8
2	7
3	6
4	5

存储芯片与烧录器连接图

第三步：打开 FlyPRO 烧录软件，选择 FlyPRO 上方的芯片-选择W25Q64FV。



第四步：点击 FlyPRO 软件界面上方的加载，将所需要烧入的 bin 文件加载进去。



第五步：点击烧录软件的自动编程，选择单次烧录。等待程序将 bin 文件烧录进 flash。下图为成功烧录的界面（烧录时如提示“ID校验不通过”，是否继续选择“是”，即可正常烧录）。

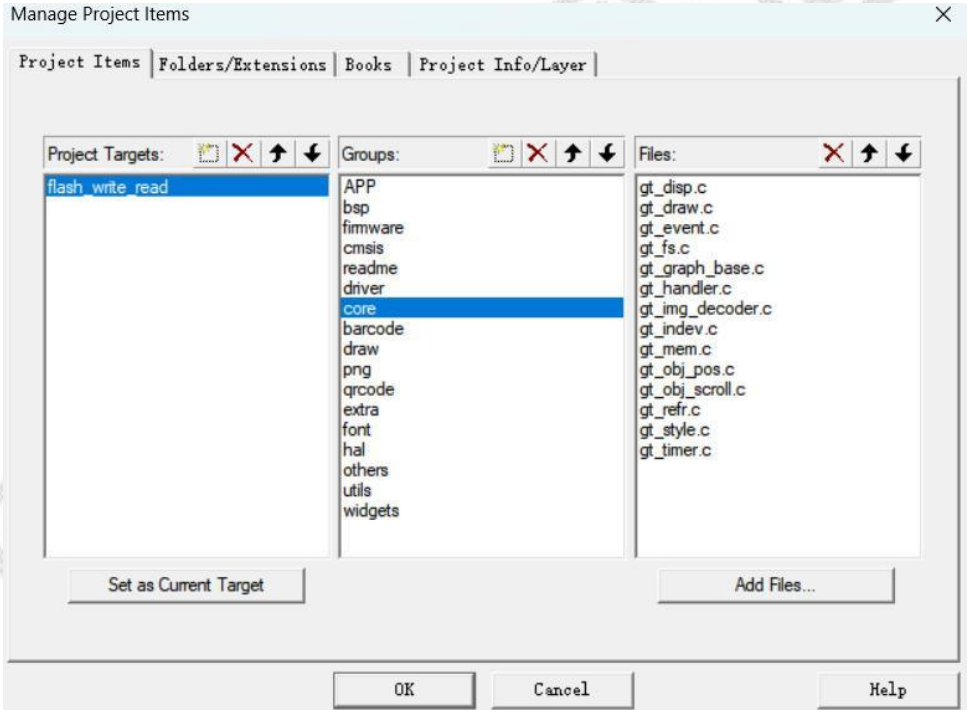
2.4 GT-HMI-Engine 代码移植

GT-HMI-Engine 代码移植很重要，在 hmi 上位机设计的界面需要依赖这些文件才能使用，下面介绍移植步骤（下面介绍的步骤是在 keil5 环境下完成的）。

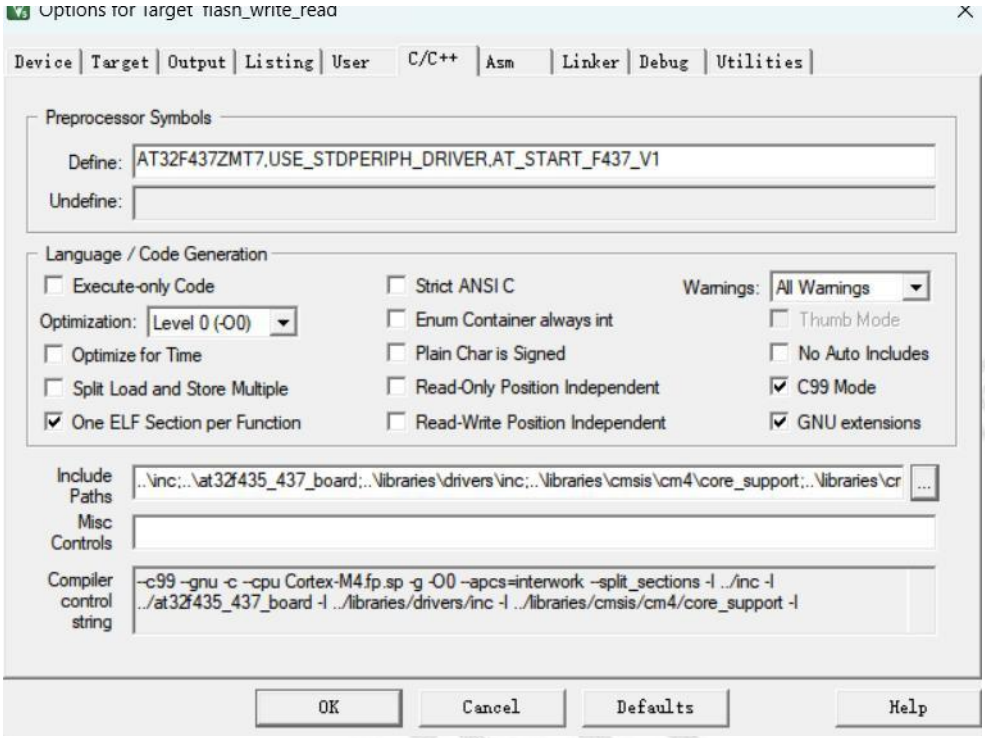
第一步：首先用 git 的下载 GT-HMI-Engine 源代码。在 git bash 上使用"git clone git@gitee.com:genitop/GT-HMI-Engine.git -b develop"命令拉取代码。

```
$ git clone git@gitee.com:genitop/GT-HMI-Engine.git -b develop
Cloning into 'GT-HMI-Engine'...
remote: Enumerating objects: 4939, done.
remote: Counting objects: 100% (4939/4939), done.
remote: Compressing objects: 100% (3385/3385), done.
remote: Total 4939 (delta 3981), reused 1921 (delta 1546), pack-reused 0
Receiving objects: 100% (4939/4939), 1015.83 KiB | 5.58 MiB/s, done.
Resolving deltas: 100% (3981/3981), done.
```

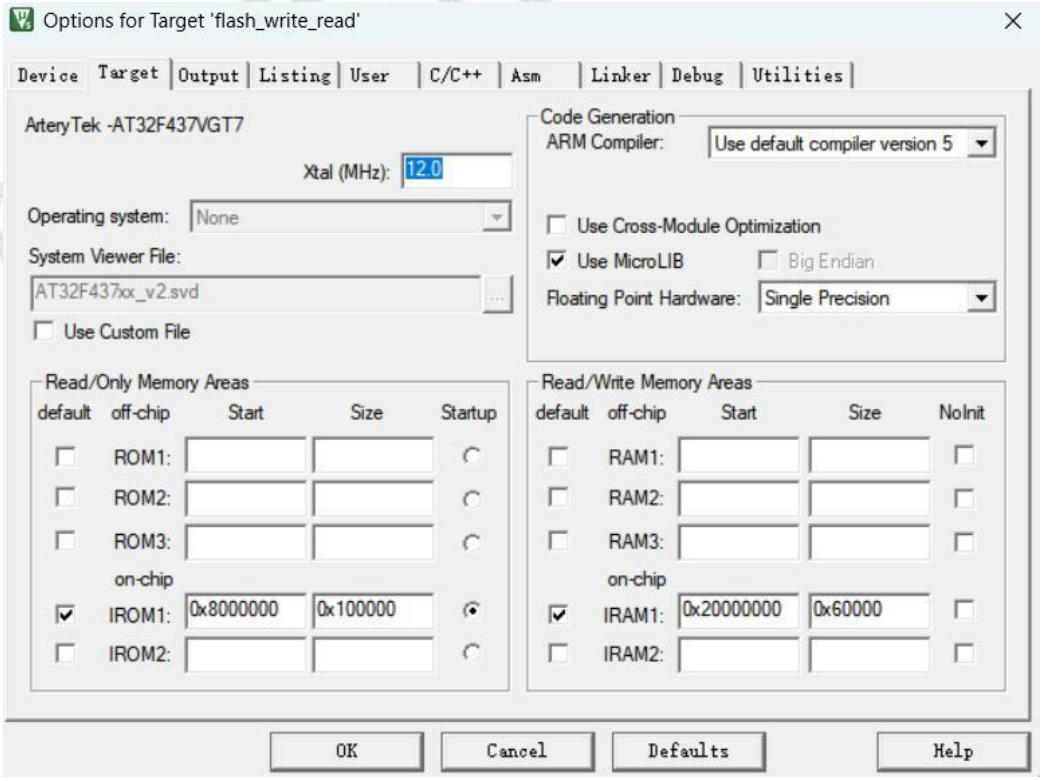
第二步：将 GT-HMI-Engine 源码添加到功能目录里面去，并添加到 keil5 的工程里面去，



第三步：打开 gnu 配置进行编译，下图为参考的配置，不同的 keil5 版本配置 gnu 界面都不一样。这里 gnu 配置必须打开，不然会报很多错误。



下图为 Target 配置，Use MicroLIB 也需要打开，不然也会报错。



第四步：编译通过之后添加 spi_wr、_flush_cb、read_cb、read_cb_btn 四个函数，spi_wr 是读取素材图片的，支持 flash、SD 卡读取，_flush_cb 是刷屏函数接口，read_cb 是触摸上报接口，read_cb_btn是按键上报接口。read_cb_btn 接口根据客户的要求添加，如果用不到，可定义空函数，就是这函数里面什么也没有，防止编译报错。

第五步：初始化 gt_init 函数，while(1)循环添加如下图操作，gt_tick_inc 为 GUI 系统提供 1ms 的心跳。

```
while(1)
{
    gt_tick_inc(1);
    gt_task_handler();
    delay_ms(1);
}
```

做完上面操作可显示一个按键控件看是否能正常显示，如显示不正常请检查_flush_cb 刷屏函数是否正确。

控件能正常使用并能正常触摸，表明 GT-HMI-Engine 已移植成功。

2.5 接口函数代码示例

```
// 定义触摸状态变量
uint8_t touch_status;

// SPI 写入函数，用于向 SPI 设备写入数据并可选地读取数据
// 参数：data_write - 指向写入数据的指针
//      len_write - 写入数据的长度
//      data_read - 指向读取数据的指针
//      len_read - 读取数据的长度
uint32_t spi_wr(uint8_t * data_write, uint32_t len_write, uint8_t * data_read, uint32_t len_read)
{
   // 读取地址的临时变量
   unsigned long ReadAddr;
   unsigned long addr, len;

   // 计算读取地址
   ReadAddr = *(data_write + 1) << 16; // 高八位地址
   ReadAddr += *(data_write + 2) << 8; // 中八位地址
   ReadAddr += *(data_write + 3);     // 低八位地址

   // 从 SPI 闪存读取数据
   spiflash_read(data_read, ReadAddr, len_read);

   // 返回成功标志
   return 1;
}
```

```c
// 刷新回调函数，用于更新显示区域
// 参数：drv - 显示驱动结构体
//      area - 显示区域信息
//      color - 颜色信息
void _flush_cb(struct _gt_disp_drv_s * drv, gt_area_st * area, gt_color_t * color)
{
   gt_size_t x = area->x, y = area->y;
   uint16_t w = area->w, h = area->h;
   int i = 0;

   // 设置 LCD 块
   lcd_setblock(x, y, x + w - 1, y + h - 1);
   for (i = 0; i < w * h; i++)
   {
      // 写入颜色数据
      lcd_wr_data(color->full >> 8);
      lcd_wr_data(color->full & 0xff);
      color++;
   }
}

// 输入设备读取回调函数，用于处理触摸状态
// 参数：indev_drv - 输入设备驱动结构体
//      data - 输入设备数据结构体
void read_cb(struct _gt_indev_drv_s * indev_drv, gt_indev_data_st * data)
{
   if (!touch_status)
   {
      // 如果触摸状态未激活，设置为释放状态
      data->state = GT_INDEV_STATE_RELEASED;
      return;
   }

   // 更新触摸状态
   touch_status = 0;
   data->point.x = tp_dev.point.x;
   data->point.y = tp_dev.point.y;
   data->state = GT_INDEV_STATE_PRESSED;
}
```

注：read_cb_btn 接口函数，如有需求可联系我司技术人员提供。

2.6 HMI函数代码移植

GT-HMI-Engine 移植成功后，就可以实现 HMI 设计的界面移植到板子上，下面讲解一下 HMI 界面移植。

第一步：HMI 的工程目录如下图，board 和 out 目录是工程编译生成的字库库文件，图片素材寻址 C文件，还有素材 bin 文件，我们需要做的是将 gt_gui_driver.lib、gt_gui_driver.h 和 gt_port_vf.c 文件替换 keil 工程 GT-HMI-Engine/driver 原来的文件，向 flash 烧录 bin 文件。screen 目录是存放 HMI 设计的界面 C 文件，将这些界面文件添加到 keil5 工程里面去。

第二步：在 main 函数调用 gt_ui_init 界面初始化接口函数，gt_ui_init 是第一步添加到 keil5 工程界面文件中的函数。

在 GT-HMI-Engine 移植成功的基础上，按上面介绍的步骤操作完，即可实现 HMI 的界面移植，如移植不成功，请检查 bin 文件是否烧写正确。

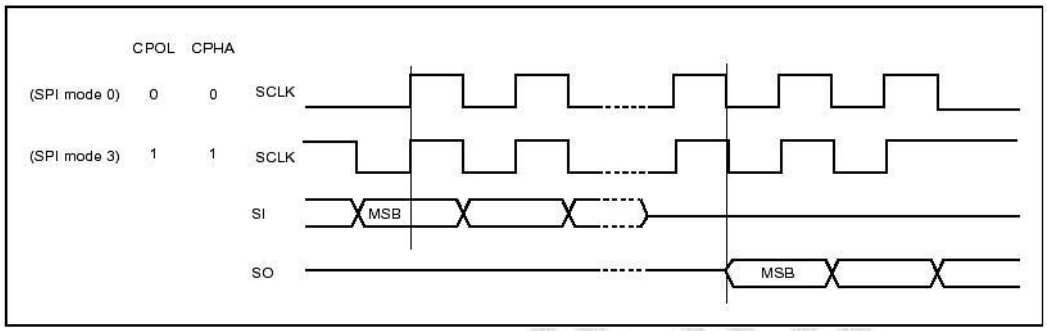
3 标准SPI操作指令

串行数据输出（SO）：该信号用来把数据从芯片串行输出，数据在时钟的下降沿移出。

串行数据输入（SI）：该信号用来把数据从串行输入芯片，数据在时钟的上升沿移入。

串行时钟输入（SCLK）：数据在时钟上升沿移入，在下降沿移出。

片选输入（CS#）：所有串行数据传输开始于CS#下降沿，CS#在传输期间必须保持为低电平，在两条指令之间保持为高电平。

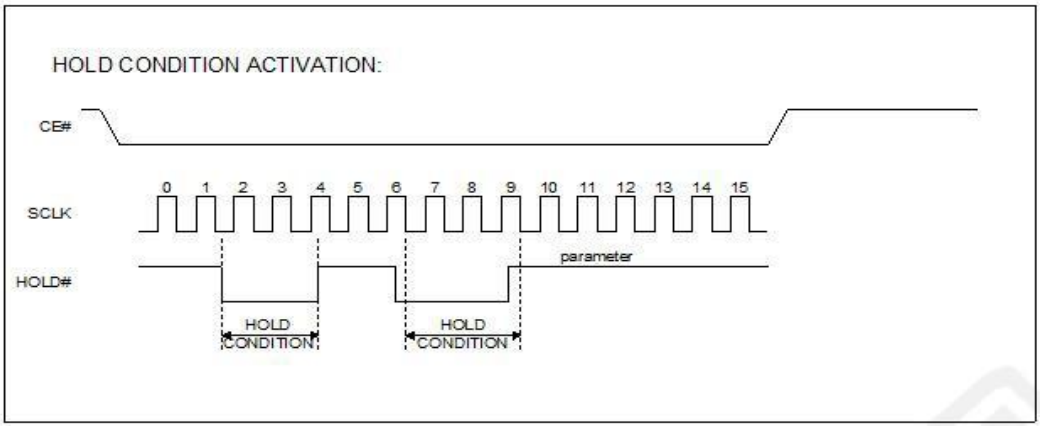


总线挂起输入（HOLD#）：

该信号用于片选信号有效期间暂停数据传输，在总线挂起期间，串行数据输出信号处于高阻态，芯片不对串行数据输入信号和串行时钟信号进行响应。

当HOLD#信号变为低并且串行时钟信号（SCLK）处于低电平时，进入总线挂起状态。

当HOLD#信号变为高并时串行时钟信号（SCLK）处于低电平时，结束总线挂起状态。



3.1 Instruction Parameter(指令参数)

Command set (Standard/Quad SPI)

Commands	Abbr.	Code	ADR Bytes	DMY Cycles	Data Bytes	Function description
Fast Read Array	FREAD	0BH	3	8	1+	n bytes read out until CS# goes high
Normal Read Array	READ	03H	3	0	1+	n bytes read out until CS# goes high
Read Quad Output	QREAD	6BH	3	8	1+	n bytes read out by Quad output
Read 4IO	4READ	EBH	3	6(10)	1+	n bytes read out by 4IO
Sector Erase (4K bytes)	SE	20H	3	0	0	erase selected sector
Page Program	PP	02H	3	0	1+	program selected page
Quad page program	QPP	32H	3	0	1+	quad input to program selected page
Write Enable	WREN	06H	0	0	0	sets the write enable latch bit
Write Disable	WRDI	04H	0	0	0	resets the write enable latch bit
Read Status Register	RDSR	05H	0	0	1	read out status register
Deep Power-down	DP	B9H	0	0	0	enters deep power-down mode
Release Deep Power-down/Read Electronic ID	RDP/RES	ABH	3	0	1	Read electronic ID data

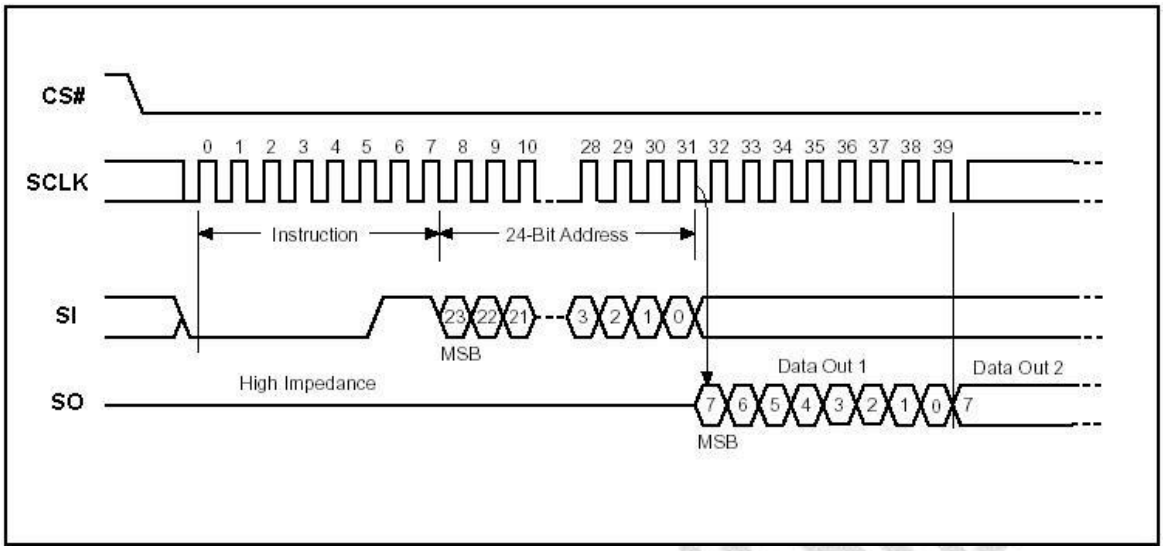
Note:常用操作存储芯片指令，可供客户写驱动做参考。

3.2 Read Data Bytes（一般读取）

Read Data Bytes 需要用指令码来执行每一次操作。READ 指令的时序如下(图)：

- 首先把片选信号（CS#）变为低，紧跟着的是 1 个字节的指令字（03 h）和 3 个字节的地址和通过串行数据输入引脚（SI）移位输入，每一位在串行时钟（SCLK）上升沿被锁存。
- 然后该地址的字节数据通过串行数据输出引脚（SO）移位输出，每一位在串行时钟（SCLK）下降沿被移出。
- 读取字节数据后，则把片选信号（CS#）变为高，结束本次操作。
  如果片选信号（CS#）继续保持为底，则下一个地址的字节数据继续通过串行数据输出引脚（SO）移位输出。

读取数据（READ）指令时序 和数据输出时序图

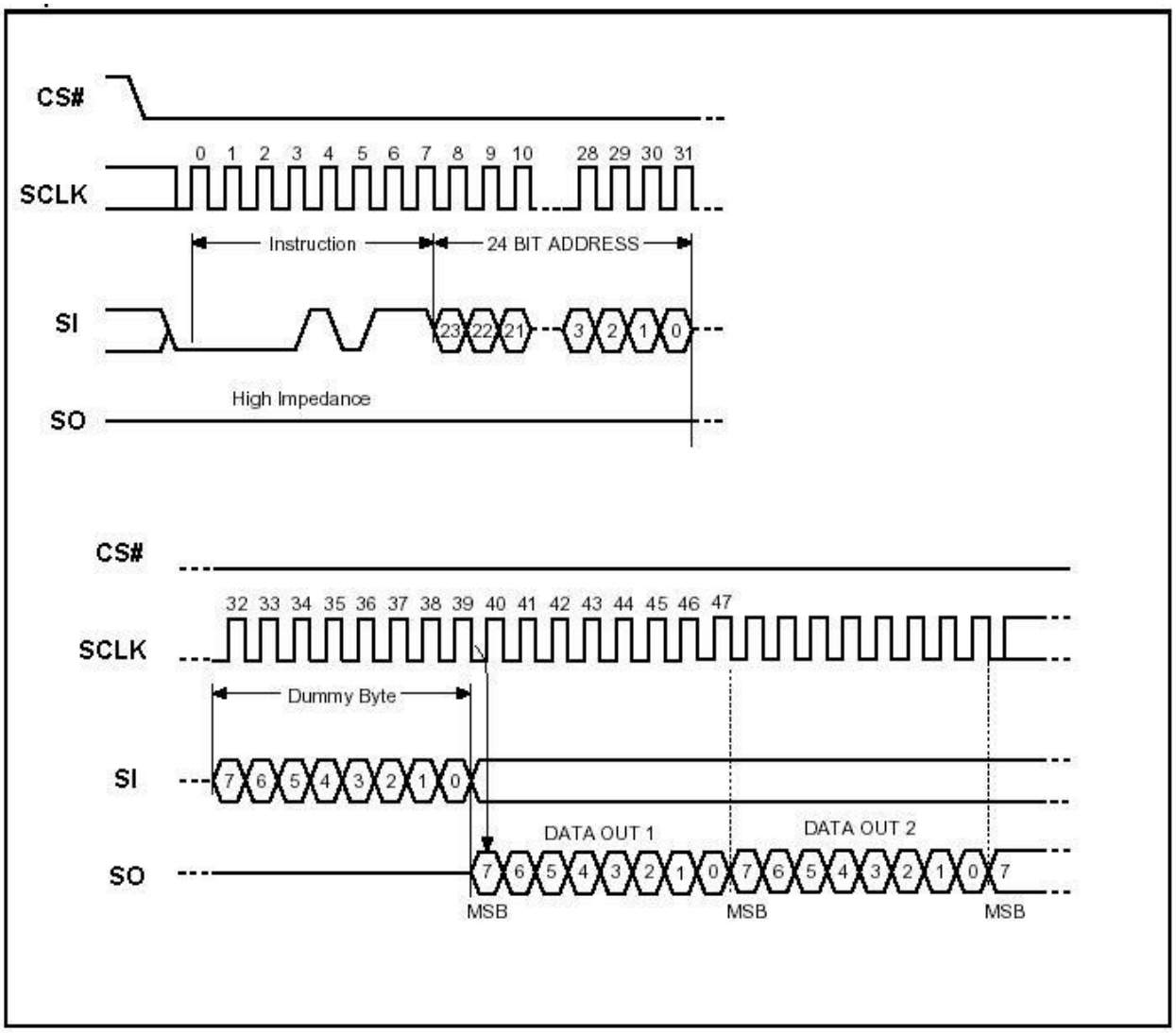


3.3 Read Data Bytes at Higher Speed（快速读取点阵数据）

Read Data Bytes at Higher Speed 需要用指令码来执行操作。READ_FAST 指令的时序如下(图)：

- 首先把片选信号（CS#）变为低，紧跟着的是 1 个字节的指令字（0B h）和 3 个字节的地址以及一个字节 Dummy Byte 通过串行数据输入引脚（SI）移位输入，每一位在串行时钟
  （SCLK）上升沿被锁存。
- 然后该地址的字节数据通过串行数据输出引脚（SO）移位输出，每一位在串行时钟（SCLK）下降沿被移出。
- 如果片选信号（CS#）继续保持为底，则下一个地址的字节数据继续通过串行数据输出引脚（SO）移位输出。例：读取一个 15x16 点阵汉字需要 32Byte，则连续 32 个字节读取后结束一个汉字的点阵数据读取操作。
  如果不需要继续读取数据，则把片选信号（CS#）变为高，结束本次操作。

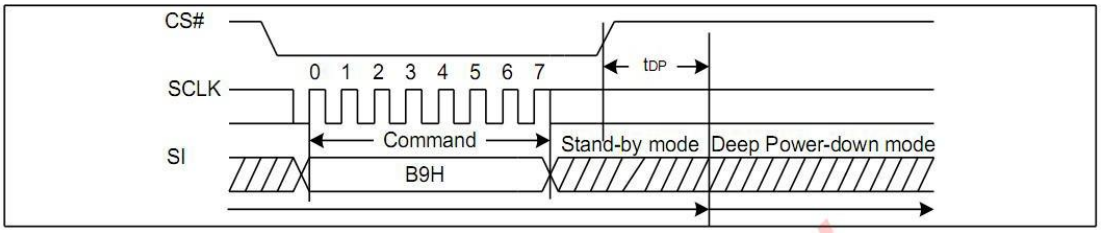
高速读取数据字节 (READ_FAST)指令时序和数据输出时序图



3.4深度睡眠模式指令（B9H）

一旦GT5GL64芯片进入深度睡眠模式，所有的指令将被忽略，除了唤醒深度睡眠模式指令，首先首先 CS#为低电平，输入 B9H 指令，然后然后 CS#变为高电平并持续 TDP 的时间(TDP=25us)，在 TDP 的持续时间内，GT5GL64芯片进入深层关机模式。

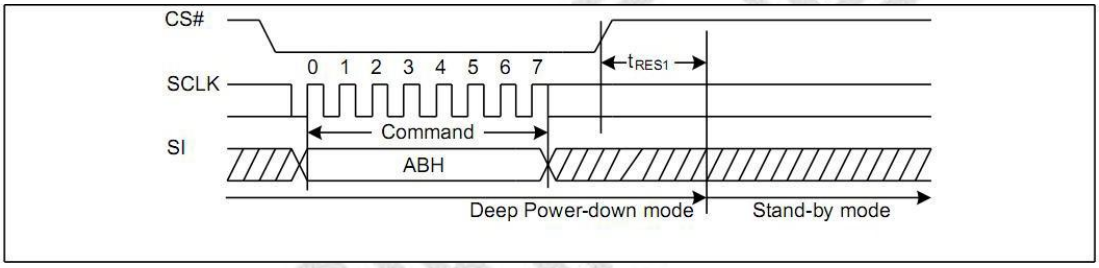
深度睡眠模式指令的时序波形图



3.5唤醒深度睡眠模式指令（ABH）

首先 CS# 为低电平， 向GT5GL64芯片发送 ABH 指令， 然后 CS# 变为高电平并持续 Tres1 的时间(Tres1=25us)，GT5GL64芯片将恢复正常运行，CS#引脚必须在 Tres1 时间内保持高电平。

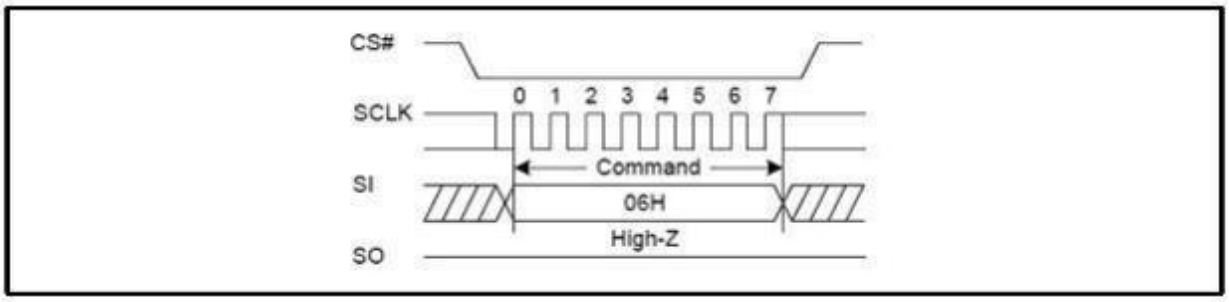
唤醒深度睡眠模式指令的时序波形图



3.6 Write Enable（写使能）

Write Enable 指令的时序如下(图)：

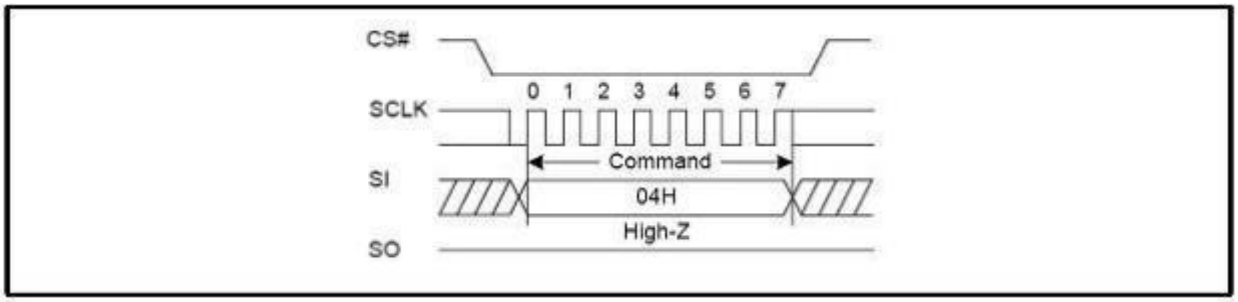
CS#变低- 发送 Write Enable 命令—>CS#变高



3.7 Write Disable（写非能）

Write Enable 指令的时序如下(图)：

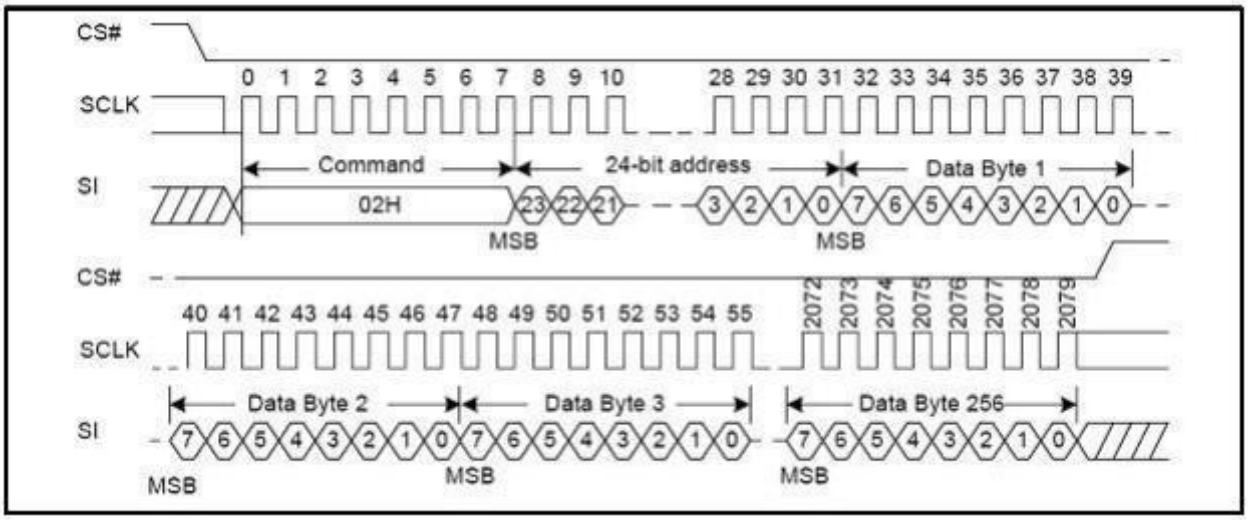
CS#变低- 发送 Write Disable 命令— >CS#变高



3.8 Page Program（页写入）

Page Program 指令的时序如下(图)：

CS#变低- 发送 Page Program 命令 发送 3 字节地址—>发送数据—>CS#变高

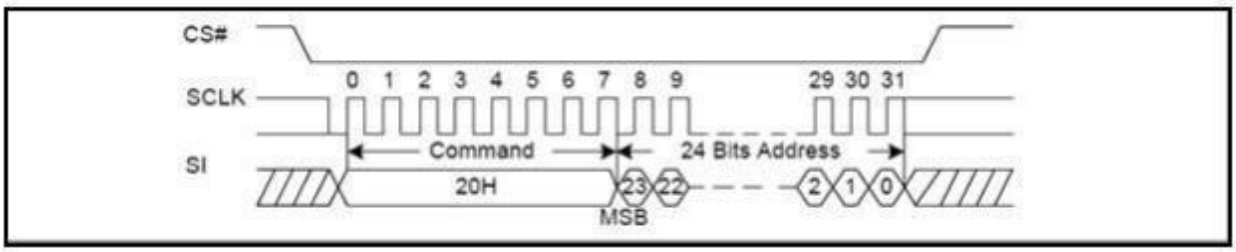


注：写入指令发送 CS#变高后需进行忙状态判断，等待芯片内部完成写入后，才可以对芯片进行下一步操作，判断忙状态请参考该型号相应的库文件，如无库文件请与我司索要。

3.9 Sector Erase（扇区擦除）

Sector Erase 指令的时序如下(图)：

CS#变低- 发送 Sector Erase 命令 发送 3 字节地址— >CS#变高



注：擦除指令发送 CS#变高后需进行忙状态判断，等待芯片内部完成擦除后，才可以对芯片进行下一步操作，判断忙状态请参考该型号相应的库文件，如无库文件请与我司索要。

3.10 芯片状态寄存器以及说明

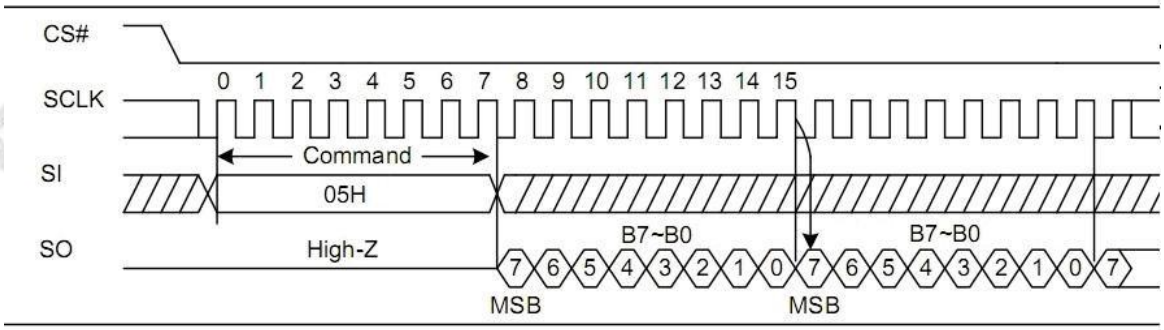
Status Register

B7	B6	B5	B4	B3	B2	B1	B0
BP0	SP4	SP3	SP2	SP1	SP0	WSL	WIP

判断芯片是否在忙状态，使用寄存器 B0，当 B0 位的 WIP 位为 1 的时候，为忙状态，当 WIP 位为 0的时候芯片处于空闲状态。

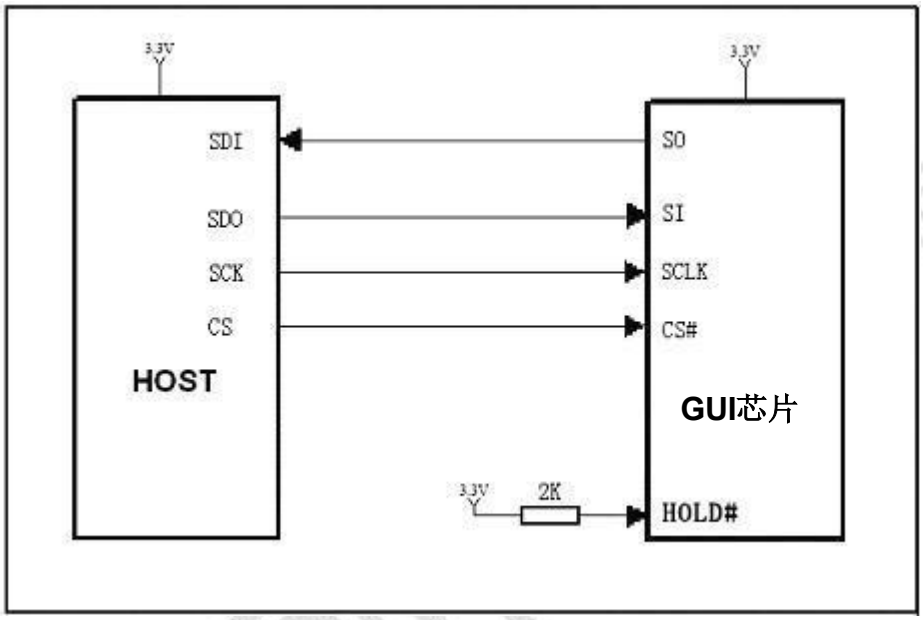
3.11 读取芯片状态寄存器的命令说明

发送命令 05H，然后读取芯片状态寄存器的 B7-B0 位。判断 WIP 位的状态来判断芯片是否在忙状态。



3.12 SPI 接口与主机接口参考电路示意图

SPI 与主机接口电路连接可以参考下图（HOLD#管脚建议接 2K 电阻 3.3V 拉高）。



SPI 接口与主机接口参考电路示意图

4 Quad SPI操作指令

GT5GL64芯片在使用“Quad Output Fast Read”，“Quad I/O Fast Read”（6BH，EBH)指令时支持Quad SPI操作，这些指令允许数据以标准SPI的四倍速率与设备进行传输，使用Quad SPI 指令时，SI和SO引脚变为双向I/O引脚（IO0和IO1），WP#和HOLD#变为IO2和IO3。 Quad SPI指令需要设置状态寄存器中的Quad Enable bit（QE)为1。

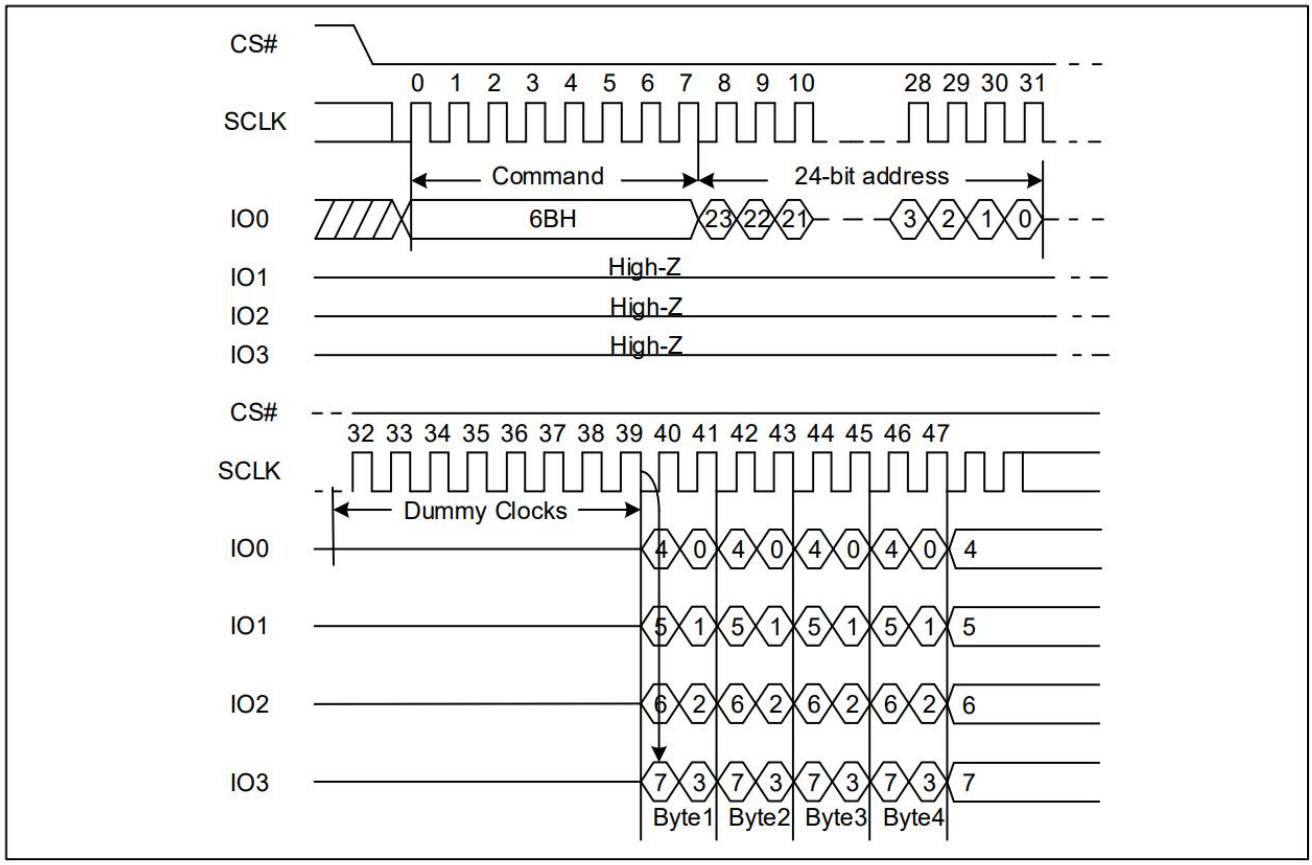
4.1 Quad Read（QREAD）（6BH)指令

QREAD指令在读取模式下启用串行NOR闪存的四倍吞吐量。在发送QREAD指令之前，状态寄存器的Quad Enable（QE）位必须设置为“1”。地址在 SCLK 的上升沿锁存，每四位数据（在 4 个 I/O 引脚上交错）在 SCLK 的下降沿移出，最大频率为 fQ。第一个地址字节可以在任何位置。每个字节数据移出后，地址会自动增加到下一个更高的地址，因此可以在单个QREAD指令中读取整个存储器。当到达最高地址时，地址计数器将滚动到0。一旦写入QREAD指令，下面的数据输出将作为4位执行，而不是以前的1位。

发出QREAD指令的顺序为：CS#变低→发送QREAD指示→3字节SI上的地址→8位伪周期→IO3、IO2、IO1和IO0上的数据输出交错→结束QREAD操作可以在数据输出期间的任何时间使用CS#到高电平。

当编程/擦除/写入状态寄存器周期正在进行时，QREAD指令被拒绝，不会对编程/擦除-写入状态寄存器当前周期产生任何影响。

Quad Read（QREAD）（6BH)指令时序图

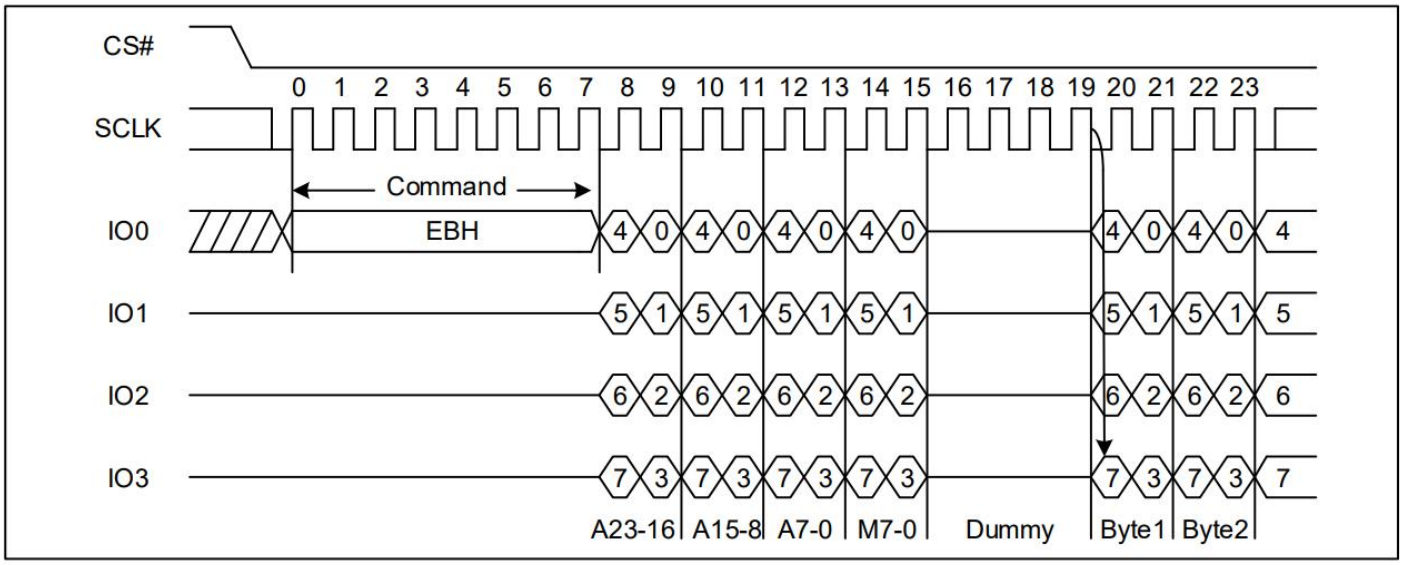


4.2 4I/O Read (4READ) (EBH) 指令

4READ指令在读取模式下启用串行NOR闪存的四倍吞吐量。在发送4READ指令之前，状态寄存器的Quad Enable（QE）位必须设置为“1”。该地址在SCLK的上升沿锁存，每四位数据（在 4 个 I/O 引脚上交错）在 SCLK 的下降沿移出，最大频率为 fQ。第一个地址字节可以在任何位置。每个字节数据移出后，地址会自动增加到下一个更高的地址，因此整个存储器可以在单个4READ指令中读出。当到达最高地址时，地址计数器将滚动到0。一旦写入4READ指令，以下地址/伪地址/数据输出将作为4位执行，而不是以前的1位。

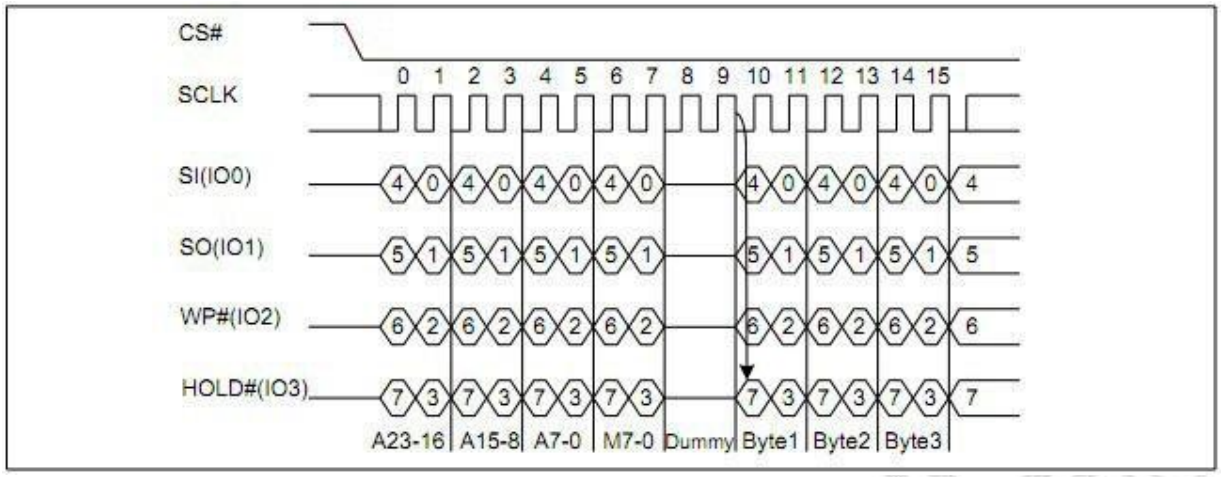
发出4READ指令的顺序是：CS#变低→发送4READ命令→IO3、IO2、IO1和IO0上的24位地址交织→2+4个伪周期→IO3，IO2、IO1和IO0上的数据输出交织→结束4READ操作可以在数据输出期间随时使用CS#变高。

当编程/擦除/写入状态寄存器周期正在进行时，4READ指令被拒绝，而不会对编程/擦除-写入状态寄存器当前周期产生任何影响。



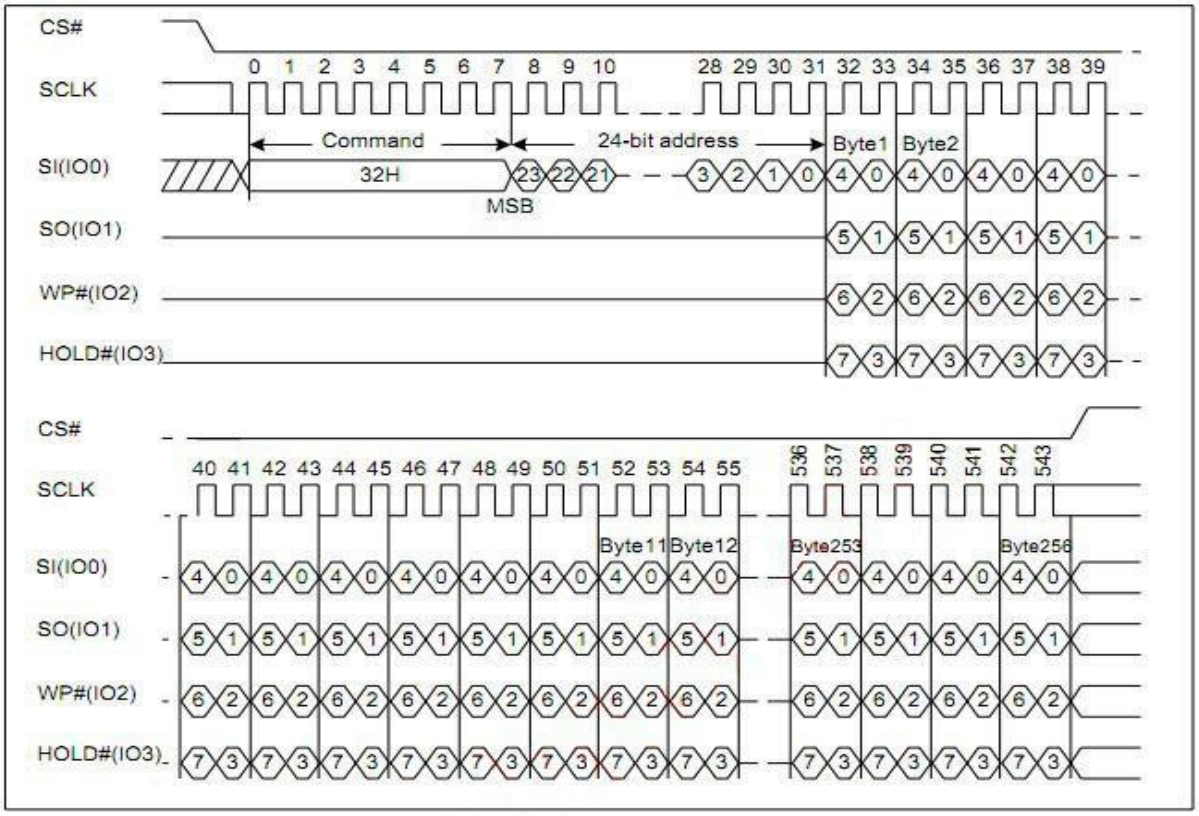
410 Read (4READ) (EBH) 指令时序图

四线 I / O 字快速读取序列图（M7-0 = AXH）



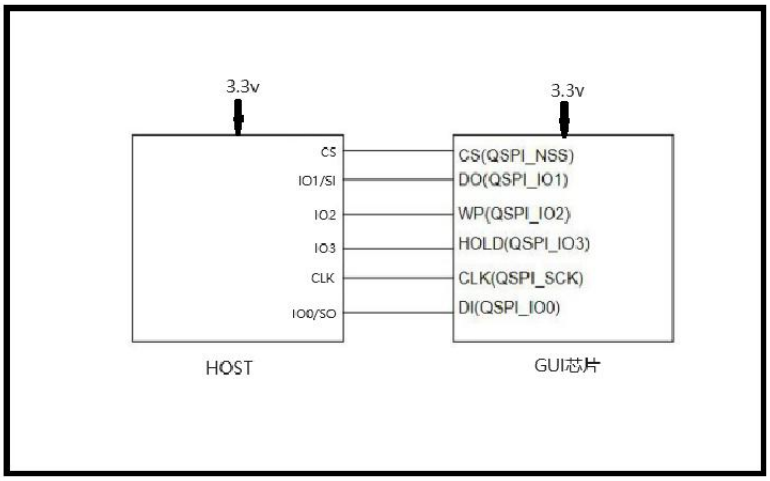
4.3 四线页写入指令Quad Page Program （32H）

四线页写入命令指令用于将内存编程为“0”。在发送四页程序（QPP）指令之前，必须执行写使能（WREN）指令以设置写使能锁存（WEL）位，并且四使能（QE）位必须设置为“1”。四页编程使用四个引脚：IO0、IO1、IO2 和 IO3 作为数据输入，这可以提高程序员的性能和应用的效果。QPP 操作频率支持尽可能快的 fQPP。其他功能描述与标准页程序相同。 发出 QPP 指令的顺序是：CS# 低电平→发送 QPP 指令代码→在 IO0 上的 3 字节地址→在 IO [3:0] 上至少 1 字节数据→CS# 高电平。



Quad 页面程序序列图

4.4 Quad SPI 模式与主机接口参考电路示意图



5 示例程序

5.1 SPI 驱动程序

```
//注：此代码是在雅特力 AT32F437VGT7 实现的代码，仅供参考，其他 MCU 请参考各自 MCU 厂商提供的 SDK 编写 SPI 驱动。
/**
* @brief Write a byte to flash
* @param data: Data to write
* @retval Flash return data
*/
uint8_t spi_byte_write(uint8_t data)
{
   uint8_t brxbuff;
   spi_i2s_dma_transmitter_enable(SPI4, FALSE); // 禁用 DMA 传输
   spi_i2s_dma_receiver_enable(SPI4, FALSE); // 禁用 DMA 接收
   spi_i2s_data_transmit(SPI4, data); // 发送数据
   while (spi_i2s_flag_get(SPI4, SPI_I2S_RDBF_FLAG) == RESET); // 等待接收缓冲区非空
   brxbuff = spi_i2s_data_receive(SPI4); // 接收数据
   while (spi_i2s_flag_get(SPI4, SPI_I2S_BF_FLAG) != RESET); // 等待传输完成
   return brxbuff; // 返回接收到的数据
}

/**
* @brief Read a byte from flash
* @param none
* @retval Flash return data
*/
uint8_t spi_byte_read(void)
{
   return (spi_byte_write(FLASH_SPI_DUMMY_BYTE)); // 发送虚拟字节并返回接收到的数据
}

/**
* @brief Read data from flash
* @param pbuffer: Pointer to data buffer
* @param read_addr: Address where the data is read
* @param length: Buffer length
* @retval none
*/
void spiflash_read(uint8_t *pbuffer, uint32_t read_addr, uint32_t length)
{
   FLASH_CS_LOW(); // 片选拉低
   spi_byte_write(0x03); // 发送读取指令 0x03
   spi_byte_write((uint8_t)((read_addr) >> 16)); // 发送 24 位地址
   spi_byte_write((uint8_t)((read_addr) >> 8));
   spi_byte_write((uint8_t)read_addr);
   // spi_byte_write((uint8_t)0xff); // 使用 0x0B 快速读取时需额外发送一个字节
   spi_bytes_read(pbuffer, length); // 读取数据
   FLASH_CS_HIGH(); // 片选拉高
}

/**
* @brief Erase a sector of data
* @param erase_addr: Sector address to erase
```

```
* @retval none
*/
void spiflash_sector_erase(uint32_t erase_addr)
{
   spiflash_write_enable(); // 使能写操作
   spiflash_write_enable(); // 再次使能写操作
   spiflash_wait_busy(); // 等待设备空闲
   FLASH_CS_LOW(); // 片选拉低
   spi_byte_write(0x20); // 发送擦除指令 0x20
   spi_byte_write((uint8_t)((erase_addr) >> 16)); // 发送 24 位地址
   spi_byte_write((uint8_t)((erase_addr) >> 8));
   spi_byte_write((uint8_t)erase_addr);
   FLASH_CS_HIGH(); // 片选拉高
   spiflash_wait_busy(); // 等待擦除完成
}

/**
* @brief Read data continuously
* @param pbuffer: Buffer to save data
* @param length: Buffer length
* @retval none
*/
void spi_bytes_read(uint8_t *pbuffer, uint32_t length)
{
   while (length--)
   {
      while (spi_i2s_flag_get(SPI4, SPI_I2S_TDBE_FLAG) == RESET); // 等待发送缓冲区空
      spi_i2s_data_transmit(SPI4, 0xa5); // 发送任意值
      while (spi_i2s_flag_get(SPI4, SPI_I2S_RDBF_FLAG) == RESET); // 等待接收缓冲区非空
      *pbuffer = spi_i2s_data_receive(SPI4); // 接收数据
      pbuffer++; // 指针后移
   }
}

/**
* @brief Read device ID
* @param none
* @retval Device ID
*/
uint16_t spiflash_read_id(void)
{
   uint16_t wreceivedata = 0;
   FLASH_CS_LOW(); // 片选拉低
   spi_byte_write(0x90); // 发送读取 ID 指令 0x90
   spi_byte_write(0x00); // 发送三个字节的 0x00
   spi_byte_write(0x00);
   spi_byte_write(0x00);
   wreceivedata |= spi_byte_read() << 8; // 读取高 8 位 ID
   wreceivedata |= spi_byte_read(); // 读取低 8 位 ID
   FLASH_CS_HIGH(); // 片选拉高
   return wreceivedata; // 返回设备 ID
}
```

5.2 Quad SPI 通信程序

```
//注：此代码是在雅特力 AT32F437VGT7 实现的代码，仅供参考，其他 MCU 请参考各自 MCU 厂商提供的 SDK 编写 QSPI 驱动。

#include "qspi_flash.h" // 假设有一个头文件包含必要的定义和声明

qspi_cmd_type qspi_flash_cmd_config;

#define QSPI_FLASH_FIFO_DEPTH (32 * 4)
#define QSPI_FLASH_PAGE_SIZE 256
#define QSPI_FLASH_QSPIx QSPI1

/**
* @brief qspi flash 写使能
* @param none
* @retval none
*/
void qspi_flash_write_enable(void) {
   // 写使能配置
   qspi_flash_cmd_wren_config(&qspi_flash_cmd_config);
   qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config);

   // 等待命令完成
   while (qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
   qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);
}

/**
* @brief qspi_flash 擦除命令配置
* @param qspi_cmd_struct: qspi_cmd_type 参数的指针
* @param addr: 擦除地址
* @retval none
*/
void qspi_flash_cmd_erase_config(qspi_cmd_type *qspi_cmd_struct, uint32_t addr) {
   qspi_cmd_struct->pe_mode_enable = FALSE; // 性能增强模式关闭
   qspi_cmd_struct->pe_mode_operate_code = 0;
   qspi_cmd_struct->instruction_code = 0x20; // 擦除指令
   qspi_cmd_struct->instruction_length = QSPI_CMD_INSLEN_1_BYTE; // 命令长度
   qspi_cmd_struct->address_code = addr; // 擦除地址
   qspi_cmd_struct->address_length = QSPI_CMD_ADRLEN_3_BYTE; // 地址长度 3 字节
   qspi_cmd_struct->data_counter = 0;
   qspi_cmd_struct->second_dummy_cycle_num = 0;
   qspi_cmd_struct->operation_mode = QSPI_OPERATE_MODE_111;
   qspi_cmd_struct->read_status_config = QSPI_RSTSC_HW_AUTO;
   qspi_cmd_struct->read_status_enable = FALSE;
   qspi_cmd_struct->write_data_enable = TRUE;
}

/**
* @brief qspi_flash 读状态寄存器命令配置
* @param qspi_cmd_struct: qspi_cmd_type 参数的指针
* @retval none
*/
void qspi_flash_cmd_rdsr_config(qspi_cmd_type *qspi_cmd_struct) {
```

```
    qspi_cmd_struct->pe_mode_enable = FALSE;
    qspi_cmd_struct->pe_mode_operate_code = 0;
    qspi_cmd_struct->instruction_code = 0x05; // 读状态寄存器指令
    qspi_cmd_struct->instruction_length = QSPI_CMD_INSLEN_1_BYTE; // 命令长度
    qspi_cmd_struct->address_code = 0;
    qspi_cmd_struct->address_length = QSPI_CMD_ADRLEN_0_BYTE;
    qspi_cmd_struct->data_counter = 0;
    qspi_cmd_struct->second_dummy_cycle_num = 0;
    qspi_cmd_struct->operation_mode = QSPI_OPERATE_MODE_111;
    qspi_cmd_struct->read_status_config = QSPI_RSTSC_HW_AUTO;
    qspi_cmd_struct->read_status_enable = TRUE;
    qspi_cmd_struct->write_data_enable = FALSE;
}

/**
* @brief qspi flash 检查忙状态
* @param none
* @retval none
*/
void qspi_flash_busy_check(void) {
    qspi_flash_cmd_rdsr_config(&qspi_flash_cmd_config);
    qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config);

    // 等待命令完成
    while (qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
    qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);
}

/**
* @brief qspi flash 擦除数据
* @param sec_addr: 擦除扇区地址
* @retval none
*/
void qspi_flash_erase(uint32_t sec_addr) {
    qspi_flash_write_enable(); // 写使能
    qspi_flash_cmd_erase_config(&qspi_flash_cmd_config, sec_addr); // 擦除结构体配置
    qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config); // 执行擦除操作

    // 等待命令完成
    while (qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
    qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);
    qspi_flash_busy_check(); // 检查忙状态
}

/**
* @brief qspi_flash 读命令配置
* @param qspi_cmd_struct: qspi_cmd_type 参数的指针
* @param addr: 读取起始地址
* @param counter: 读取数据计数
* @retval none
*/
void qspi_flash_cmd_read_config(qspi_cmd_type *qspi_cmd_struct, uint32_t addr, uint32_t counter) {
    qspi_cmd_struct->pe_mode_enable = FALSE; // 性能增强模式关闭
```

```
    qspi_cmd_struct->pe_mode_operate_code = 0;
    qspi_cmd_struct->instruction_code = 0xEB; // 四线读指令
    qspi_cmd_struct->instruction_length = QSPI_CMD_INSLEN_1_BYTE; // 命令长度
    qspi_cmd_struct->address_code = addr; // 读取地址
    qspi_cmd_struct->address_length = QSPI_CMD_ADRLEN_3_BYTE; // 地址长度 3 字节
    qspi_cmd_struct->data_counter = counter; // 读取数据长度
    qspi_cmd_struct->second_dummy_cycle_num = 6; // 第二个虚拟状态周期数 6
    qspi_cmd_struct->operation_mode = QSPI_OPERATE_MODE_144; // 指令单线，地址 4 线，数据 4 线
    qspi_cmd_struct->read_status_config = QSPI_RSTSC_HW_AUTO; // 硬件读取
    qspi_cmd_struct->read_status_enable = FALSE;
    qspi_cmd_struct->write_data_enable = FALSE;
}

/**
* @brief qspi_flash 读命令配置
* @param qspi_cmd_struct: qspi_cmd_type 参数的指针
* @param addr: 读取起始地址
* @param counter: 读取数据计数
* @retval none
*/
void qspi_flash_cmd_read_config_6B(qspi_cmd_type *qspi_cmd_struct, uint32_t addr, uint32_t counter) {
    qspi_cmd_struct->pe_mode_enable = FALSE; // 性能增强模式关闭
    qspi_cmd_struct->pe_mode_operate_code = 0;
    qspi_cmd_struct->instruction_code = 0x6B; // 四线读指令
    qspi_cmd_struct->instruction_length = QSPI_CMD_INSLEN_1_BYTE; // 命令长度
    qspi_cmd_struct->address_code = addr; // 读取地址
    qspi_cmd_struct->address_length = QSPI_CMD_ADRLEN_3_BYTE; // 地址长度 3 字节
    qspi_cmd_struct->data_counter = counter; // 读取数据长度
    qspi_cmd_struct->second_dummy_cycle_num = 8; // 第二个虚拟状态周期数 6
    qspi_cmd_struct->operation_mode = QSPI_OPERATE_MODE_114; // 指令单线，地址单线，数据 4 线
    qspi_cmd_struct->read_status_config = QSPI_RSTSC_HW_AUTO; // 硬件读取
    qspi_cmd_struct->read_status_enable = FALSE;
    qspi_cmd_struct->write_data_enable = FALSE;
}

/**
* @brief qspi flash 读取数据
* @param addr: 读取地址
* @param total_len: 读取长度
* @param buf: 读取数据的指针
* @retval none
*/
void qspi_flash_data_read(uint32_t addr, uint8_t* buf, uint32_t total_len) {
    uint32_t i, len = total_len;

//    qspi_flash_cmd_read_config_6B(&qspi_flash_cmd_config, addr, total_len);
    qspi_flash_cmd_read_config(&qspi_flash_cmd_config, addr, total_len);
    qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config);

    // 读取数据
    do {
        if (total_len >= QSPI_FLASH_FIFO_DEPTH) {
            len = QSPI_FLASH_FIFO_DEPTH;
```

```
    } else {
        len = total_len;
    }

    while (qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_RXFIFORDY_FLAG) == RESET);

    for (i = 0; i < len; ++i) {
        *buf++ = qspi_byte_read(QSPI_FLASH_QSPIx);
    }

    total_len -= len;

  } while (total_len);

  // 等待命令完成
  while (qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
  qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);
}

/**
* @brief qspi flash 写入数据
* @param addr: 写入地址
* @param total_len: 写入长度
* @param buf: 写入数据的指针
* @retval none
*/
void qspi_flash_data_write(uint32_t addr, uint8_t* buf, uint32_t total_len) {
  uint32_t i, len;

  do {
    qspi_flash_write_enable();

    // 每次最多发送 256 字节，并且只能在一个页面内
    len = (addr / QSPI_FLASH_PAGE_SIZE + 1) * QSPI_FLASH_PAGE_SIZE - addr;
    if (total_len < len) {
        len = total_len;
    }

    qspi_flash_cmd_write_config(&qspi_flash_cmd_config, addr, len);
    qspi_cmd_operation_kick(QSPI_FLASH_QSPIx, &qspi_flash_cmd_config);

    for (i = 0; i < len; ++i) {
        while (qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_TXFIFORDY_FLAG) == RESET);
        qspi_byte_write(QSPI_FLASH_QSPIx, *buf++);
    }

    total_len -= len;
    addr += len;

    // 等待命令完成
    while (qspi_flag_get(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG) == RESET);
    qspi_flag_clear(QSPI_FLASH_QSPIx, QSPI_CMDSTS_FLAG);
    qspi_flash_busy_check();
```

```
    } while (total_len);
}
```

6 电气特性

6.1绝对最大额定值

Symbol	Parameter	Min.	Max.	Unit	Condition
T_{OP}	Operating Temperature	-40	85	℃	
T_{STG}	Storage Temperature	-65	150	℃	
VCC	Supply Voltage	2.7	3.6	V	
V_{IN}	Input Voltage	-0.3	VCC+0.3	V	

6.2 DC 特性

Symbol	Parameter	Min.	Typ.	Max.	Unit	Condition
I_{DD}	VDD Supply Current(active)	-	-	30	mA	
I_{SB}	VDD Standby Current	-	-	40	uA	/CS=VDD, VIN=VDD orVSS
I_{cc2}	Deep Power-Down Current	-	-	15	uA	/CS=VDD, VIN=VDD or VSS
V_{IL}	Input LOW Voltage	-0.5	-	0.16VCC	V	VDD=2.7~3.6 V
V_{IH}	Input HIGH Voltage	0.8VCC	-	VCC+0.4	V	
V_{OL}	Output LOW Voltage		-	0.2 (I_{OL}=1.6mA)	V	
V_{OH}	Output HIGH Voltage	VDD-0.2 (I_{OH}=-100uA)	-		V	
I_{LI}	Input Leakage Current	-	-	±2	uA	
I_{LO}	Output Leakage Current	-	-	±2	uA	

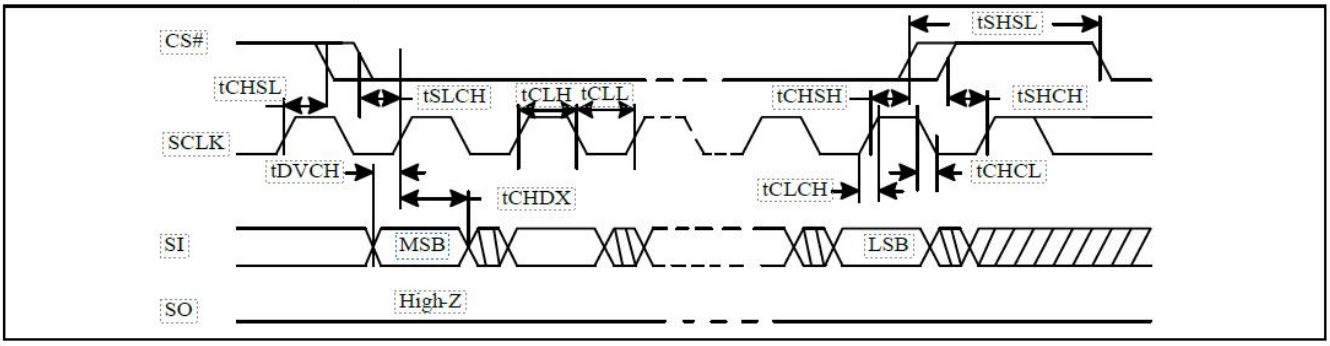
Condition: T_{OP} =-40℃ to 85℃, GND=0V

Note: I_{IL}: Input LOW Current, I_{IH}: Input HIGH Current,
I_{OL}: Output LOW Current, I_{OH}: Output HIGH Current,

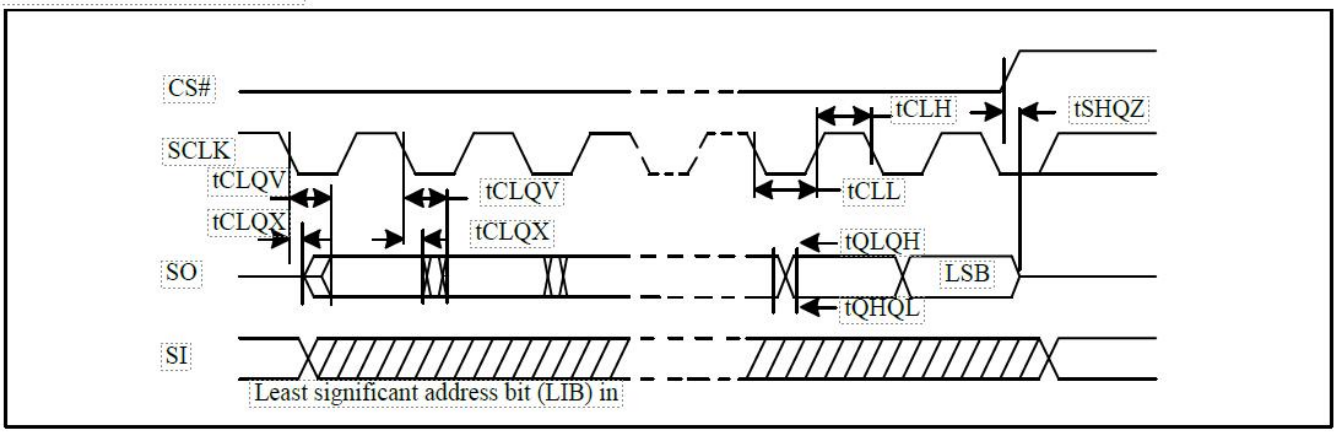
6.3 AC 特性

Symbol	Alt.	Parameter	Min.	Typ.	Max.	Unit
Fc	Fc	Clock Frequency for all instructions except for special marking	D.C.		104	MHz
Fc1	Fc1	Clock Frequency for READ instructions			45	MHz
Fc2	Fc2	Clock Frequency for 4IO WORD READ instructions			80	MHz
tCH	tCLH	Clock High Time	5.5			ns
tCL	tCLL	Clock Low Time	5.5			ns
tCLCH		Clock Rise Time(peak to peak)	0.2			V/ns
tCHCL		Clock Fall Time (peak to peak)	0.2			V/ns
tSLCH	tCSS	CS# Active Setup Time (relative to SCLK)	5			ns
tCHSL		CS# Not Active Hold Time (relative to SCLK)	5			ns
tDVCH	tDSU	Data In Setup Time	2			ns
tCHDX	tDH	Data In Hold Time	2			ns
tCHSH		CS# Active Hold Time (relative to SCLK)	5			ns
tSHCH		CS# Not Active Setup Time (relative to SCLK)	5			ns
tSHSL	tCSH	CS# Deselect Time	20			ns
tSHQZ	tDIS	Output Disable Time			8	ns
tCLQV	tV	Clock Low to Output Valid			7.5	ns
tCLQX	tHO	Output Hold Time	2			ns
tHLCH		HOLD# Setup Time (relative to SCLK)	5			ns
tCHHH		HOLD# Hold Time (relative to SCLK)	5			ns
tHHCH		HOLD Setup Time (relative to SCLK)	5			ns
tCHHL		HOLD Hold Time (relative to SCLK)	5			ns
tHHQX	tLZ	HOLD to Output Low-Z			7	ns
tHLQZ	tHZ	HOLD# to Output High-Z			12	ns

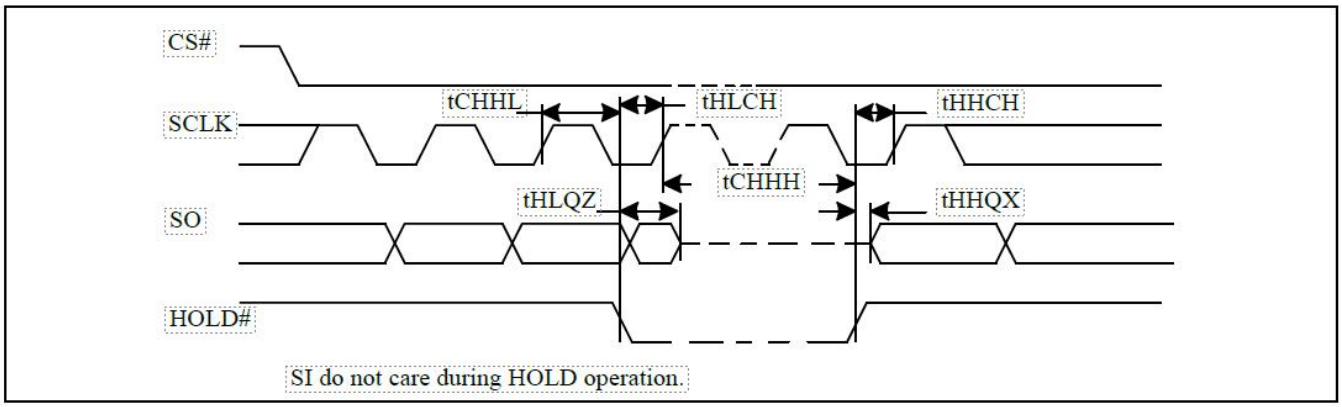
Serial Input Timing



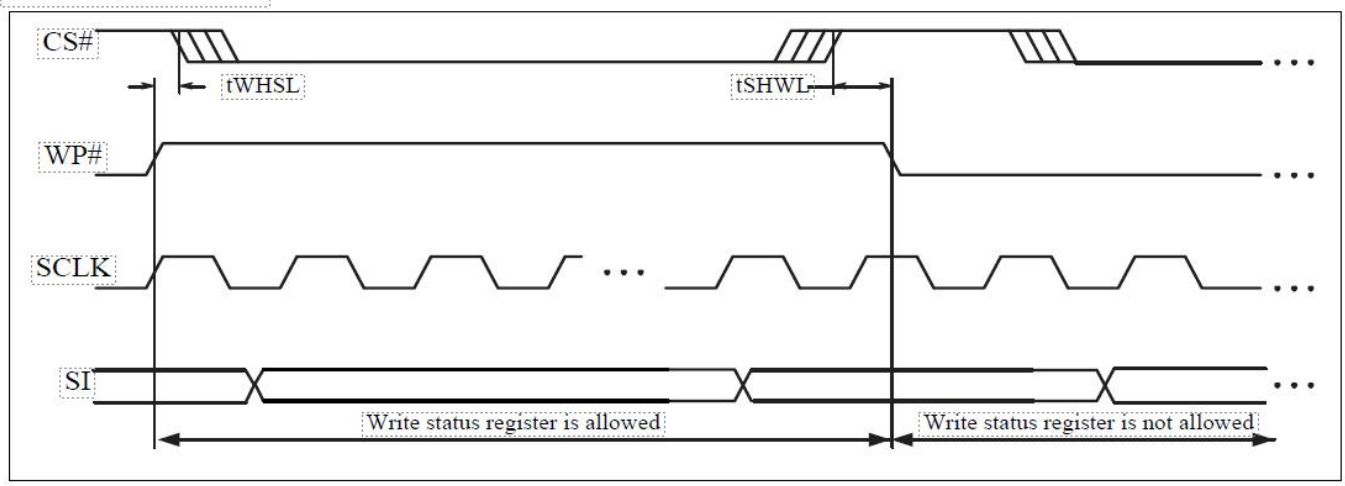
Output Timing



Hold Timing



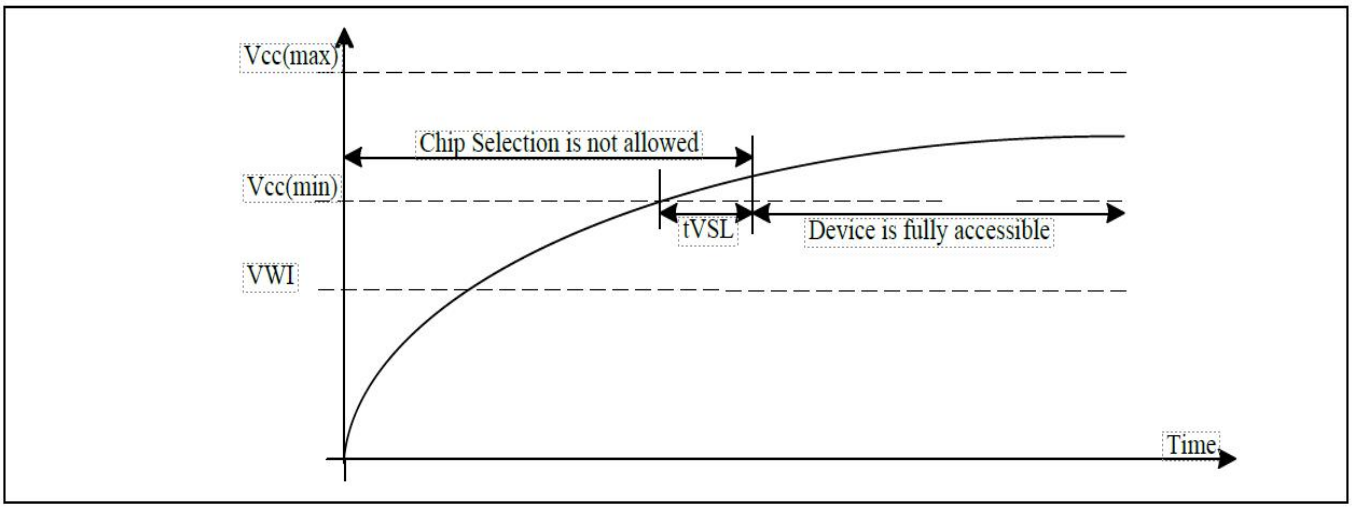
WP Timing



6.4 上电时序

PARAMETER	SYMBOL	TYPE		UNIT
		MIN	MAX	
VCC (min) to CS# Low	t_{VSL}(1)	2.5	-	ms
Write Inhibit Threshold Voltage	V_{WI}(1)	1	2	V

上电时序



7 存储组织描述

7.1 存储组织

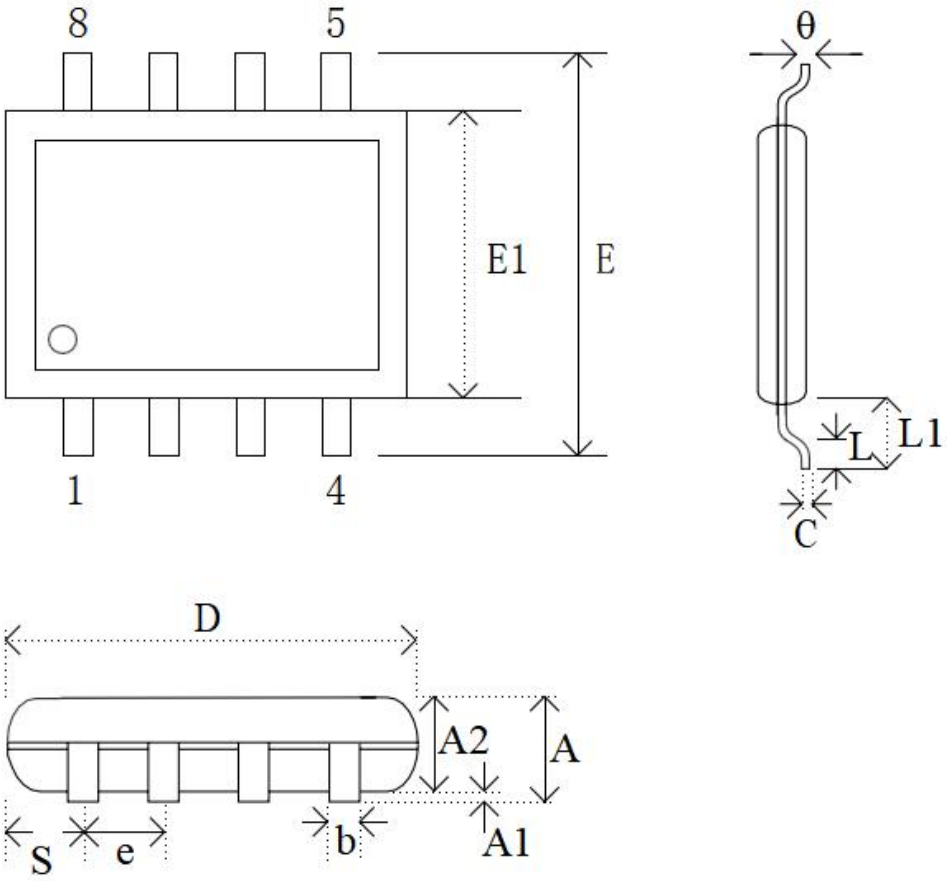
每设备	每块	每扇区	每页	
1M	64K	4K	256	字节
4K	256	16		页
256	16			扇区
16				块

7.2 存储块扇区结构

Block/ Security Register/SFDP	Sector	Address range	
Block 127	2047	7FF000H	7FFFFFH
	.....	.....	.....
	2032	7F0000H	7F0FFFH
Block 126	2031	7EF000H	7EFFFFH
	.....	.....	.....
	2016	7E0000H	7E0FFFH
.....	.....	.....	.....
	.....	.....	.....
	.....	.....	.....
.....	.....	.....	.....
	.....	.....	.....
	.....	.....	.....
Block 2	47	02F000H	02FFFFH
	.....	.....	.....
	32	020000H	020FFFH
Block 1	31	01F000H	01FFFFH
	.....	.....	.....
	16	010000H	010FFFH
Block 0	15	00F000H	00FFFFH
	.....	.....	.....
	0	000000H	000FFFH

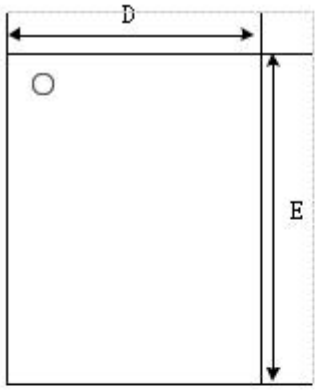
8 封装尺寸

封 装 类 型	封装尺寸
SOP8-A	4.90mmX3.90mm（193milX154mil）

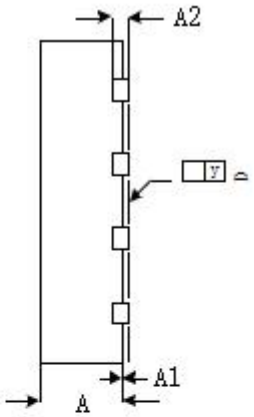


Symbol / Unit		A	A1	A2	b	C	D	E	E1	e	L	L1	S	θ
mm	Min		0.10	1.35	0.36	0.15	4.77	5.80	3.80		0.46	0.85	0.41	0
	Nom		0.15	1.45	0.41	0.20	4.90	5.99	3.90	1.27	0.66	1.05	0.54	5
	Max	1.75	0.20	1.55	0.51	0.25	5.03	6.20	4.00		0.86	1.25	0.67	8
Inch	Min		0.004	0.053	0.014	0.006	0.188	0.228	0.150		0.018	0.033	0.016	0
	Nom		0.006	0.057	0.016	0.008	0.193	0.236	0.154	0.05	0.026	0.041	0.021	5
	Max	0.069	0.008	0.061	0.020	0.010	0.198	0.244	0.158		0.034	0.049	0.026	8

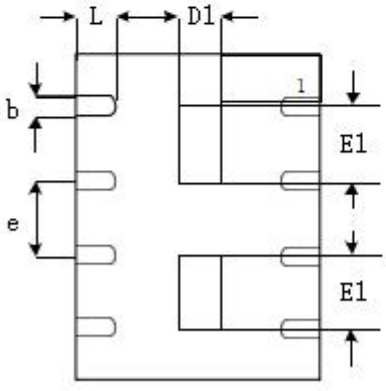
封 装 类 型	封装尺寸
USON4*3	4*3*0.55mm



Top View



Side



Bottom

Symbol / Unit		A	A1	A2	b	D	D1	E	E1	e	L
mm	Min	0.50	0.00		0.25	2.90	0.10	3.90	0.70		0.50
	Nom	0.55		0.15	0.30	3.00	0.25	4.00	0.80	0.80BSC	0.60
	Max	0.60	0.05		0.35	3.10	0.40	4.10	0.90		0.70
Inch	Min	0.020	0.000		0.010	0.114	0.004	0.153	0.027		0.020
	Nom	0.022		0.006	0.012	0.118	0.010	0.157	0.031	0.031BSC	0.024
	Max	0.024	0.002		0.014	0.122	0.016	0.161	0.035		0.028

高通汉显® | GTHMI

深圳 OFFICE

地址： 广东省深圳市福田区沙头街道泰然九路金润大厦 12C

电话： 0755-83453881 83453855

Email： sales@genitop.com