基于LGTSDK Builder

LGT8F690A 快速开发系列教程

第十一篇：SPI接口的使用

本篇为系列教程的第十一篇。如果需要了解教程相关的软件硬件环境，请参考本系列教程的第一篇：《LGT8F690A快速开发系列教程第一篇_急速上手》

LGT8F690A内部集成一个支持主从工作模式的SPI接口控制器。SPI接口非常简单，但确非常的高效。因为是同步传输接口，SPI接口与UART以及IIC接口相比简单易用，也能轻松实现很高的数据吞吐率。SPI接口的缺点是接口占用的引脚资源多一些。完成最基本的主从收发通讯，一般需要4根线。另外，SPI不支持多主机的总线结构，而且当从设备比较多时，也需要更多的引脚资源用于子设备的片选！

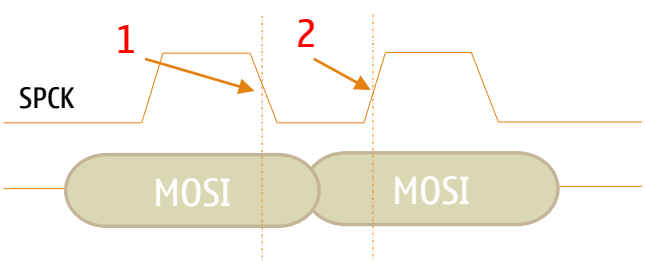
综合SPI的以上特点，它比较适合于简单的一主多从的总线结构，主从之间对数据传输速度有一定的要求。因此，SPI接口比较广泛的用于存储模块，显示模块等外设的接口。

SPI接口为单端同步接口，数据传输同步与时钟的边沿。SPI的时钟信号由主设备提供。考虑到单端同步传输的时钟延迟，主从之间的收发数据，往往需要使用不同的时钟沿，这样可以给从机采样数据平衡的数据建立和保持时间。因此，对于SPI总线，一个非常关键的因素就是定义时钟与数据之间的相位关系。这也是实现SPI高速传输的基本因素。

LGT8F690A的SPI支持四种时钟与数据相位配置，这四种配置分别确定了时钟的起始相位以及数据收发与时钟沿的关系。配置由SPI控制寄存器(SPCR)中的CPOL和CPHA两个位决定：

CPOL : CPHA	SPCK空闲电平	数据收发控制	模式说明
00	低电平	SPCK上升沿采样数据 SPCK下降沿发送数据	SPI模式0 这是最为常用的模式！
01	低电平	SPCK下降沿采样数据 SPCK上升沿发送数据	SPI模式1
10	高电平	SPCK下降沿采样数据 SPCK上升沿发送数据	SPI模式2
11	高电平	SPCK上升沿采样数据 SPCK下降沿发送数据	SPI模式3

右图为模式0的时序图，这种模式最为常用。
1：SPCK空闲为低电平，
　主机在SPCK的下降沿发送数据；
2：从机在SPCK的上升沿采样数据；



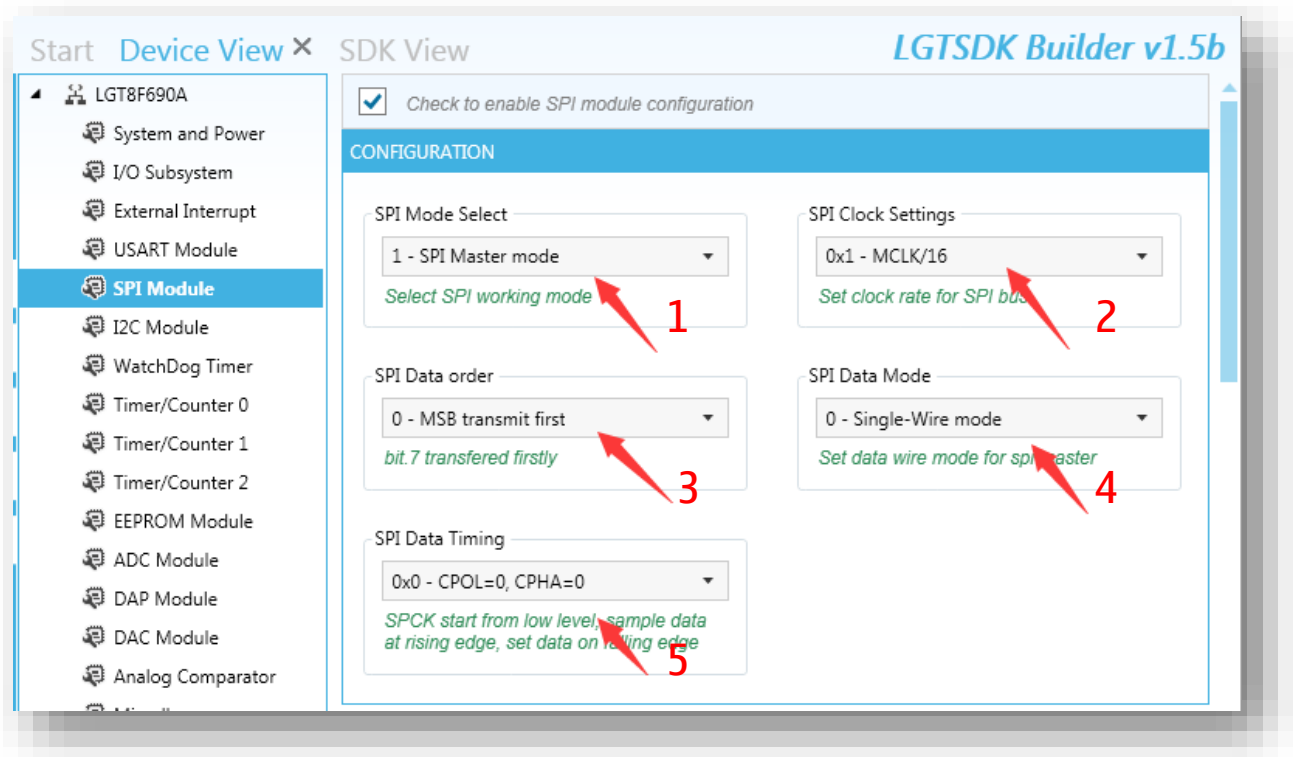
从右图可以看到，1处主机发送数据，到2处从机采样数据，我们有半个SPCK周期的时间可以留给SPCK的线路延迟，也同样有半个时钟的数据保持时间。这样可以最大限度的保证数据传输的正确性。

实际应用中，需要选择哪一种模式，取决于SPI从设备的时序要求。这个我们将在后面的实例中介绍。

SPI通讯数据的收发是分两个独立的通道，因此收发是同时进行。主机在发送数据时，也会同时接收来自从机的数据。从机在获取来自主机的数据时，同时也需要发送数据到主机。因此SPI的主从通讯是对称的。这种特性对于通讯双方需要交互的情况，非常有利。但对于大部分应用，我们可能不需要这样。我们作为主机发送数据时，并不关心从机回传的数据，此时可以简单将数据丢掉；当我们作为主机读从机的数据时，主机也会发送一个数据给从机。从机接收到这个数据后，也是可以将其忽略。通常，主机读从机数据时，主机会将发送数据设置为一个字节的0xFF。从机需要从软件机制上，合理的处理不需要的数据。

了解了以上几点后，我们就可以开始SPI接口的应用了。下面我们就以LGTSDK Builder为例，示例如何使用LGT8F690的SPI控制器实现与其他外设的通讯。

首先，启动LGTSDK Builder，建立一个新的工程，选择目标芯片，输入项目名称：lgt8f690a_spi
因为SPI的配置非常简单，我们首先来看下SPI的配置：



配置说明：

1：选择SPI的主/从工作模式。这里我们选择主机模式，控制一个SPI从设备；
2：选择SPI时钟分频(SPCK)，SPI时钟是由系统工作时钟分频而来的，因此确定SPCK频率前，需要先设置系统工作频率。我们将使用8MHz的系统频率，因此当前的选择，将会产生8MHz/16 = 512KHz SPCK时钟。大多数SPI外设，都应该可以跑到更高的频率。对于当前8MHz的系统时钟，我们可以通过选择分频的倍速模式，产生最高4MHz的SPCK时钟。这个可以留个大家测试！
3：SPI发送数据移位顺序，我们选择先发送最高位。这个选择与从设备的要求相关；
4：SPI数据总线模式，我们选择常用的单线模式。双线模式仅在主设备读从设备时有效，用于一些支持双线传输数据的SPI FLASH。双线模式下，MISO/MOSI同时用于传输来自从机的数据。
5：SPI总线的时序配置。这就是我们在之前首先描述的部分。我们选择常用模式0，这个模式的选择需要兼容从设备的总线时序。我们将在后面介绍SPI从设备时再做说明。

以上是SPI所有相关配置。然后是与SPI相关的SDK函数:

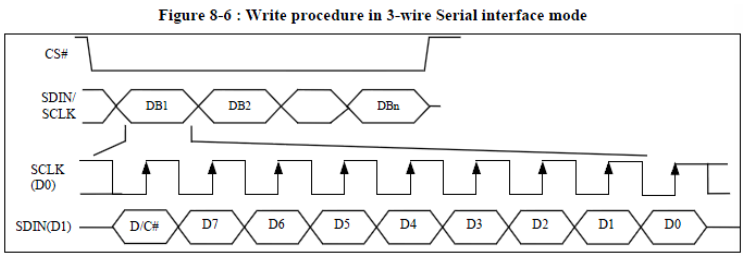
函数名称	功能说明	使用方法
spiTransferByte()	通过SPI总线传输一个字节 传输包含数据的收发	RxData = spiTransferByte(TxData); 发出TxData并同时接收数据到RxData
spiReadBuffer()	从SPI总线读指定长度的数据	spiReadBuffer(buf, length) 读数据时， 会同时发送数据0xFF到SPI总线
spiWriteBuffer()	向SPI总线发送指定长度的数据	spiWriteBuffer(buf, length) 写数据时， 忽略从SPI总线接收到的数据
spiTransferBuffer()	通过SPI总线收发指定长度的数据	spiTransferBuffer(txbuf, rxbuf, length) 从txbuf中， 发送length长度的数据到SPI总线，并同时将接收到的length长度的数据存入rxbuf中

大部分应用中， 我们仅使用spiTransferByte函数即可完成全部的收发工作。 下面的例程中，我们也将仅仅使用这个函数， 完成所有的数据传输和外设控制！

SPI的部分介绍完毕。 下面是一个完整的SPI应用例程。 我们将使用LGT8F690A的SPI接口，连接一个128x64的OLED显示屏， 通过SDK中提供的函数接口，完成OLED的控制和数据显示！

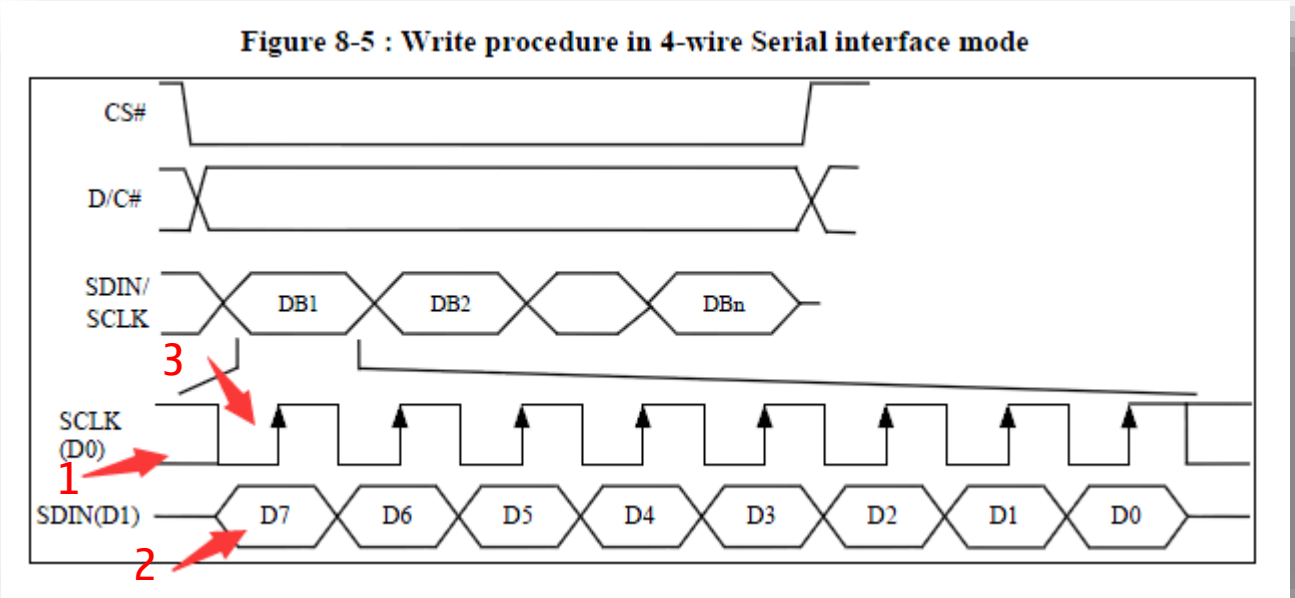
下面来介绍下我们使用的OLED屏。 这里使用的是一款比较常见的0.96寸，128X64点的OLED屏。 驱动芯片为SSD1306。这个屏有显示的GDRAM， 但没有字库。接口方面，这款屏支持IIC，三线SPI和四线SPI模式。可以通过PCB上的0欧电阻选择。我们是要用SPI接口与之通讯，因此需要将它设置为三线或者四线SPI模式。至于是三线模式，还是四线模式， 我们需要了解下他们的区别：

OLED所支持的接口协议，可以从它的驱动芯片SSD1306的手册中查到， 下面是它的三线SPI模式时序图：



可以看到， 这个三线模式， 是用SPI发送的第一个位来作为数据类型的标示(D/C#)，从而可以省去一根D/C#连线。这样的话，其实SPI需要发送9位数据。LGT8F690A的SPI是不支持的，因此我们不能用SPI三线模式。只能选择SPI四线模式，使用一根专用的D/C#连线作为发送命令/数据的控制！

下面我们详细分析OLED屏的SPI四线模式， 看看他如何与我们的SPI设置对应起来！



上面是SSD1306的四线SPI时序， 从上图可以确定如下SPI配置：

1： SCLK空闲电平可以高，也可以低， SSD1306对此并无特别要求；

2： 数据是最高位先发送；

3： 数据是在SPCK的上升沿采样， 因此主机应该是需要在SPCK的下降沿发送；因此LGT8F690A/SPI的模式0和模式3都可以用于和用来和它通讯。 我们选择常用的模式0；

到此，我们确定了LGT8F690A和OLED的接口方式：

OLED显示屏	LGT8F690A	说明
CS#	SPSS/RA6	SPI的片选输出，OLED的片选信号。低电平有效
D/C#	RA3	OLED数据/命令控制，低电平选择命令
RST#	RA2	OLED复位信号。低电平复位LED屏
D1	MOSI/RA5	SPI主机数据输出，OLED从机数据输入
D0	SPCK/RA4	SPI时钟线，主机输出
VCC	VCC	OLED电源。5V/3.3V都可以。OLED驱动自带升压
GND	GND	系统地

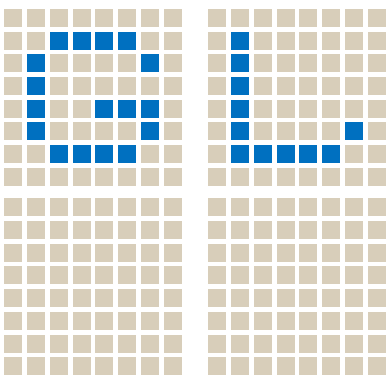
到此，我们确定了LGT8F690A和OLED的接口方式。接下来我们介绍OLED驱动相关的问题。

从SSD1306的手册中可以了解到，当SSD1306工作于串行接口模式(IIC/SPI)时，OLED仅支持单向的写操作，也就是说，串行接口模式下， 我们无法读取OLED相关的寄存器和GDRAM内容。大部分情况下，LED作为显示的只写设备， 这也并不影响我们的正常使用。但也会带来一些像素点操作的问题，类似点阵屏和字符屏的区别。当然，我们也可以通过在主控芯片中开辟一块和显示GDRAM大小相同的RAM作为显存，来解决像素级操作的问题。但这对于LGT8F690A并不现实。因此，这里我们仅仅将展示OLED的字符驱动模式。

驱动OLED屏幕并显示所需内容，需要解决两个问题：OLED的模块驱动与OLED显示字模！
这两个方面分别涉及到OLED驱动控制命令和显示数据。其中OLED模块的驱动部分， 是非常确定的时序。购买OLED屏幕时，也可以索取相关驱动的示例代码。这部分我们稍后稍作介绍。本篇教程附带的例程中，也可以找到OLED的驱动接口部分。我们先详细介绍下字模以及如何产生字库。

字模或者字库，是将字符映射到显示设备上的数据。显示的基本单位是像素，字模就是字符的像素点表达方式。字符在显示设备上通常用一定大小的像素矩形显示，这个矩形的大小，也就是字符的大小。字库是常用或者专用字符的集合， 通常一个字库中的字符大小是相等的。比如， 8x8像素的字库，其中的每一个字符，在显示设备占据了长宽分别为8个像素的矩形区域。

8X8像素的字符如右图所示，字库将显示设备以8x8像素为一个字符单位分隔。显示设备的像素点大小是固定的，因此越大的字符，也将占据更大的显示位置。汉字结构复杂，通常字符需要更大的显示空间，因此字库也就非常大。

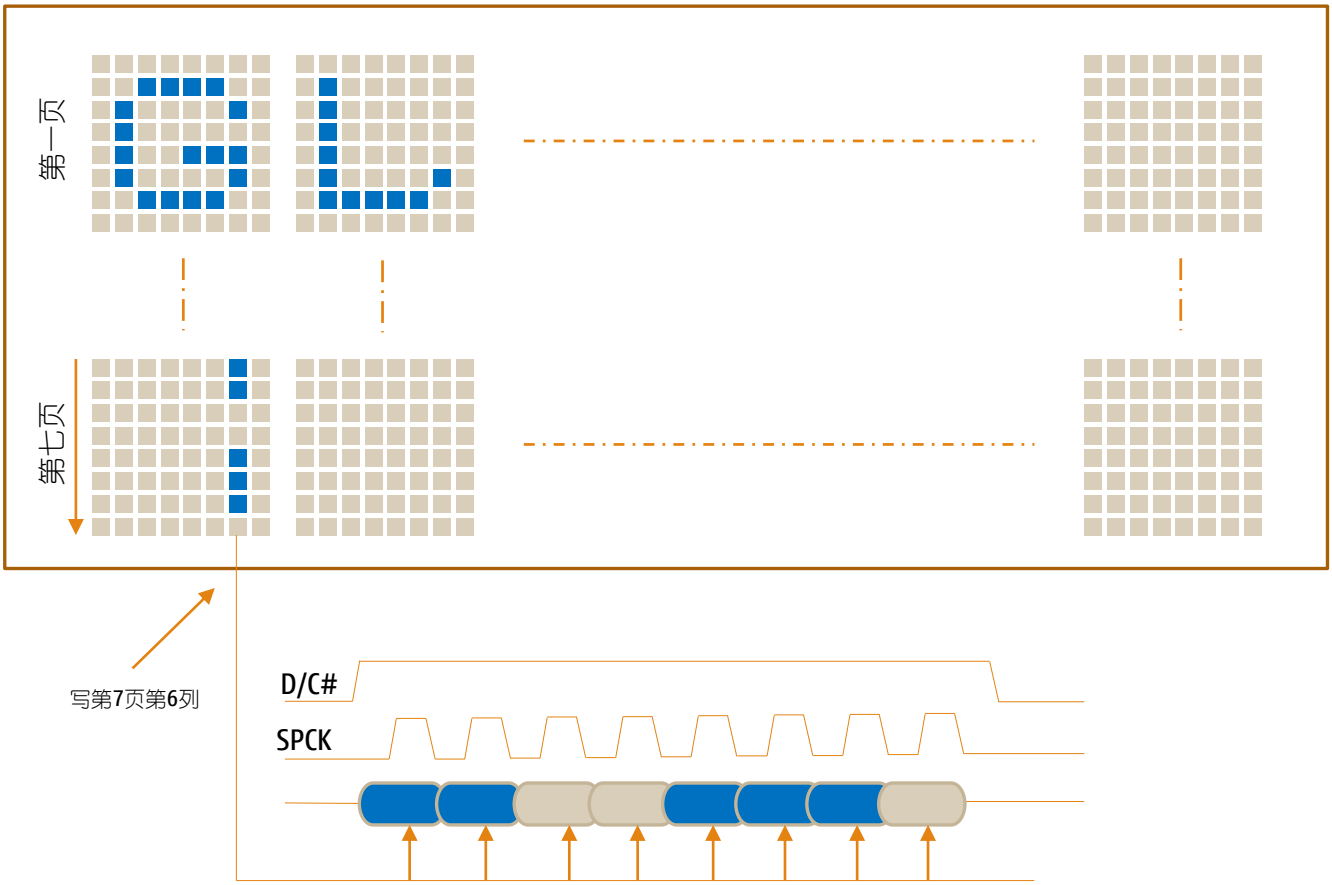


然后我们来关注一个问题，就是像素的大小。像素的大小一般是用字节/位来表示，他将实实在在的与我们需要发送的数据相关。不同的显示设备，像素点的大小是不同的。比如我们的电脑显示器的像素，每个像素都是由三基色和一个透明度的控制字组成，三基色每种颜色一般是由一个字节控制，这样一个像素点需要用4个字节(32位)表示。这4个字节是在内存或显存中，显示驱动将内存或显存中的数据驱动到显示控制器中，进而点亮显示屏上的一个像素。

但对于单色的显示设备，比如我们将要使用的OLED屏，是一种单色的显示屏，像素点只有亮和灭两种状态，亮度的控制由背光驱动实现。因此一个像素点仅由一个位来控制。因此显存中的每一位将会直接映射到显示屏的一个像素上。至于是0点亮还是1点亮，这个是可以控制。默认是1点亮像素，我们也可以通过控制反色实现0为点亮。

虽然我们的显示屏是基于像素的，但不意味着驱动接口就可以直接控制每一个像素。这与显示控制器的显存访问方式以及显示扫描方式有关。我们来看看工作在串行接口模式下的SSD1306是怎么来实现显示控制的。

首先，SSD1306看到的屏幕是基于像素的，但它并不是把它看到的显示屏结构直接通过串口接口暴露出来。显示驱动芯片考虑到计算机通讯以及数据处理是以字节为单位的，因此，它把128x64的像素屏进行了分隔。分隔后的结构更适合字节为单位的通讯控制。一个字节是8位，因此SSD1306把128x64的屏幕分成8个页面，可以理解为8行；每行是128x8像素, 也就是对于128个字节。这样128x64的屏幕就可以理解为8行128列的结构。主机在更新显示数据时，是以字节为单位下发的， 首先主机需要指定操作的页地址，也就是行地址，还需要给出列地址，这样行列对应的就是8个像素的一列像素，我们发的一个字节的数据，将会与这个8个像素直接对应。



如上图，这样我们就很好理解通过SPI接口发送一个字节的数据，将会如何影响到显示屏幕。我们通过串行接口，可以随时更新显示数据，更新数据前，首先指定我们需要更新的页地址和列地址，然后顺序发送显示数据。SSD1306将设置的页地址和列地址作为起始位置，然后每收到一个字节的数据，更新对应列的8个像素，然后自动指向到下一个列，直到本页结束(到达列地址127)。SSD1306并不会在到达最后一列后自动切换到下一页，而是回到本页的开始。换页(行)需要我们发新的页地址。

另外需要注意的是， SSD1306在收到一个字节的显示数据时，是按照从上到下的顺序刷一列的8个像素，分别对应字节的低位到高位。我们需要了解这个刷屏模式，这将会影响创建字模时，如何将字模对应的像素数据转化为字库中的数组。这种在一行中按列的刷屏方式在字模软件中，叫做列行式！

由以上分析可知，SSD1306的一页/行的高度是8个像素，如果我们的字符高度也是8个像素，那么显示字符是非常简单的。比如8X8像素的字符，我们需要连续发8个字节，就完成一个像素的显示。

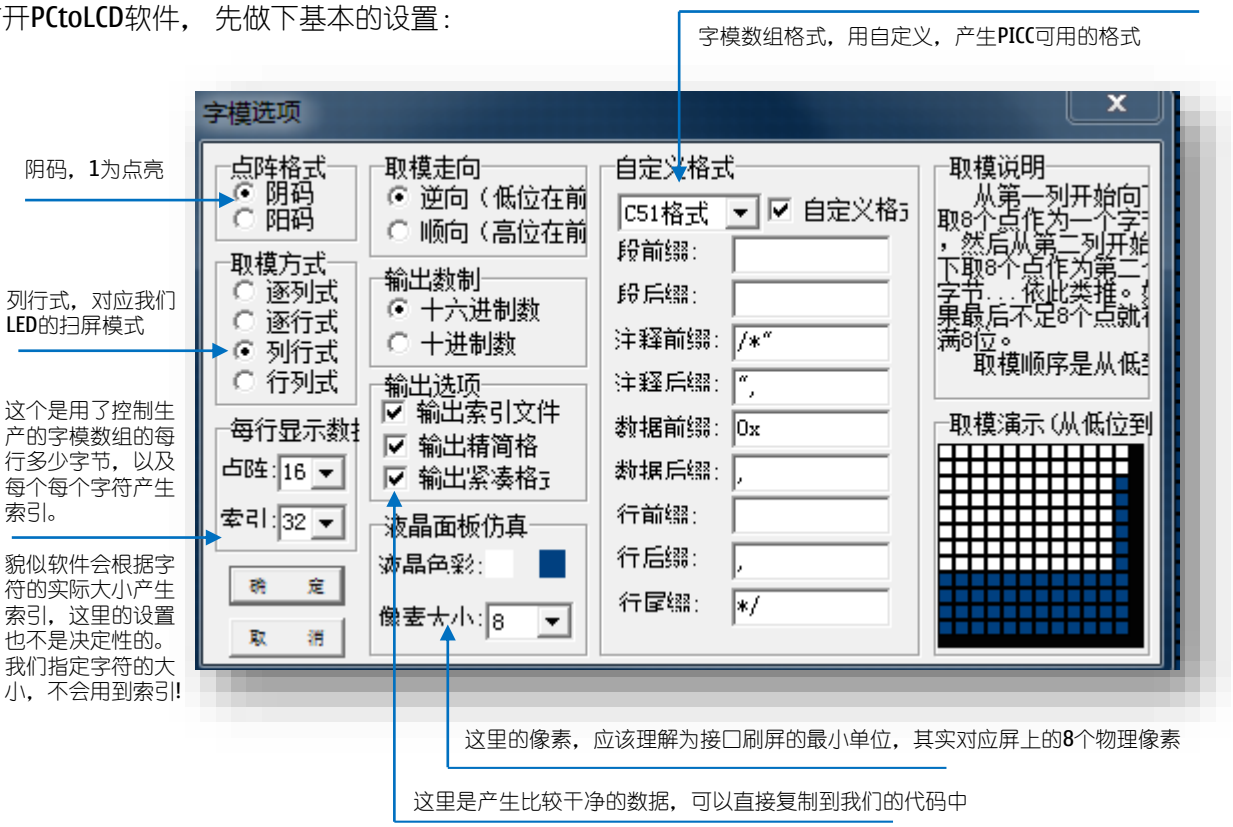
但对于大于较大的字库，字符的高度可能会大于8个像素，也就是说字符会跨几页。这样显示就会稍微复杂些，比如16X8的字符，高度为占用2页，我们要先扫一行的8个像素(宽度)，然后换到下一行，再扫8个像素，这样才能完成一个字符的显示。虽然步骤多了，但其实原理是一样的。

了解了OLED的刷屏方式，就可以设计字模或者字库了。
首先，只要我们知道了字模的产生方式和屏幕的刷屏方式，我们就可以通过软件将任何模式的字模转换到我们的屏幕上。但如果我们按照屏幕的刷屏方式产生字模，软件操作起来将会非常的方便，提高刷屏效率的同时，也可以减少软件的代码量。

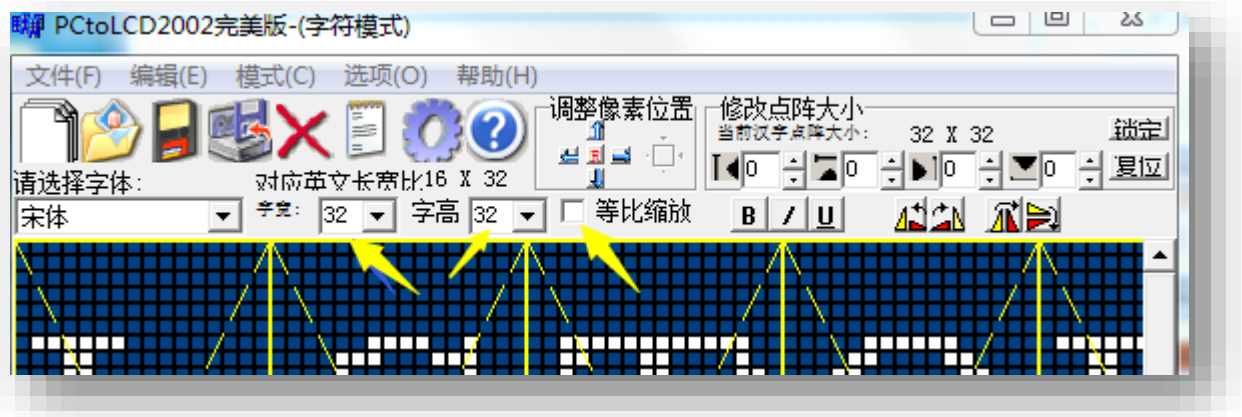
我们先来看看如何设计字模。设计字模有专用的软件，这里我们将以PCtoLCD软件为例，介绍如何制作与我们的OLED最匹配的字模。

首先，我们想在128X64像素的屏幕上完整的显示最大的LGT8F690字样。一共是8个字符，因此每个字符的宽度就是128/8 = 16像素。 高度我们目前无法确定， 因为字符的长宽比和字体有关， 字体基本上确定了字符的正常外形， 虽然我们也可以在改变比例，但一般不建议去改变它。我们将按照字体最合适的设计比例选择宽度， 这往往需要多一些尝试， 找到你觉得满意的字体和比例。

打开PCtoLCD软件， 先做下基本的设置：



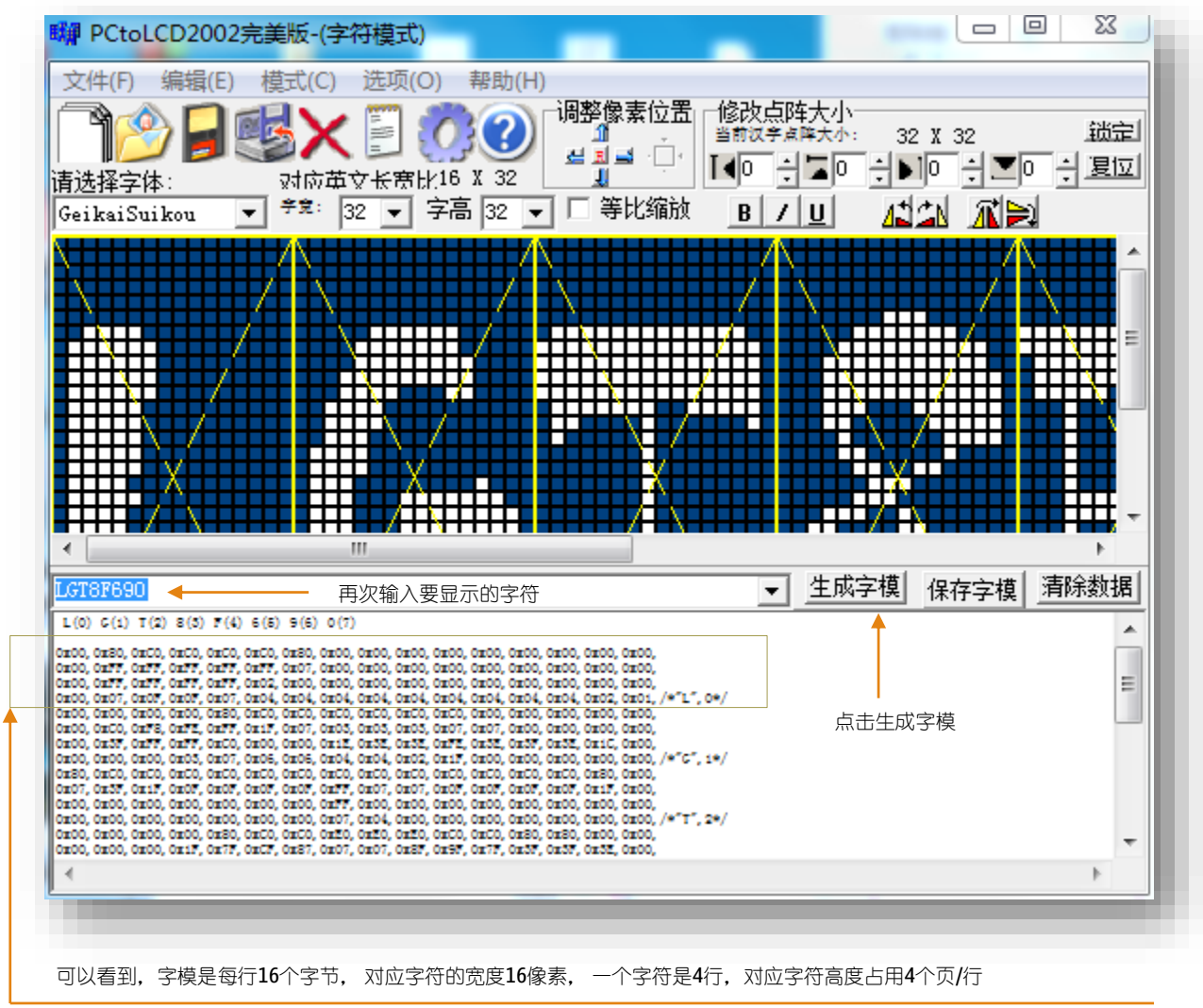
基本的设置完成后，然后就是回到主界面， 设置下字符的大小和输入字符，产生字模数组！



在主界面选择字符的宽和高。 宽度是我们之前确定的16像素，需要注意我们用的是英文字体，软件是以汉字点阵为标准， 要选择宽度32，对应的英文字体才是16像素的宽度。

字模高度的选择，我们可以根据宽度来设置， 一般建议设置8的倍数， 这样产生的字模数组比较整齐，方便软件定位字符。结合我们使用的屏，我们选择32像素的高度， 一行高度是8个像素，因此一个字符将占用4个行，也就是4个页。这也意味这我们显示一个字符，也设置4次页/行地址。

字体的高度的调整自由度比宽度要大， 我们可以通过更换字体， 来观察字符显示是否合适。如果不合适，我们可以尝试选择其他字体。



可以看到，字模是每行16个字节， 对应字符的宽度16像素， 一个字符是4行，对应字符高度占用4个页/行

这样，我们需要的字模数据就准备好了。 可以直接将数据复制到我们的源文件中， 稍做修改产生编译器能够识别的数组。由于8位PICC编译器对数组大小的限制，我们还需要将这个大数组拆分成几个不超过256字节的小数组。这个稍后再做说明。

简单说下如何显示我们的字模数据。 产生的字模数据是和我们的OLED刷屏方式完全匹配的， 我们需要处理的是在合适的位置换行显示。比如，第一个 “L” 字符， 我们首先确定显示的行地址，因为我们的OLED一共有8行， 字符高度是4行，因此如果要居中显示，应该是放到第3行上。列地址是0， 我们这8个字符将会占满128列。 首先通过SPI接口设置好页/列地址，然后发数据， 发16个字节后，换到下一行，继续发接下来的16个字节， 直到发完4行数据。 我们就完成了一个字符的显示！！

显示一个16x32像素字符的接口函数如下：

```
52
53  // draw code of 32x16 (heigh x width)
54  // only for print banner "LGT8F690"
55  void oled_draw32x16(u8 *ptr, u8 page, u8 column)
56  {
57      u8 i, j, ofs;
58
59      for(i = 0; i < 4; i++)          <-- 要刷4页/行数据，对应字符高度
60      {
61          ofs = axu_fmul8x8(i, 16);   <-- 每行16个字节，计算偏移地址
62          // point start
63          oled_setPosition((i + page), column);   <-- 设置行列地址
64          for(j = 0; j < 16; j++)
65          {
66              oled_data(ptr[ofs + j]);    <-- 每行数据16个字节，对应字符宽度
67          }
68      }
69  }
70
```

下面说说关于PICC编译器对常量数组的限制问题。8位PIC 微控制器内核是通过PCL来实现常量数组的寻址。PCL只有一个字节的， 因此可以寻址到256个字节范围内的数组。 这就要求单个数组的长度不能超过256。 而且数组内的地址， 也不能跨越256的地址范围。 因此，我们必须对大于256的数组进行拆分。

我们刚刚产生的字模数组， 大小是：16x4 x 8 = 512字节， 正好是2个256的大小。我们需要将它分离到两个数组中， 每个的长度是256字节。 由于数组内地址不能跨256， 我们还需要将数组的起始地址定位到256对齐的地址上。比如：

```
// ================================================================================
// code sets for banner: LGT8F690
// seperate into two smaller (<= 256bytes) parts to comply with PICC compiler
// ================================================================================
const u8 _lgt8[] @0x200 = {     <-- “LGT8”四个字符的字模数据，起始地址定位到256对齐的地址：0x200
0x00,0x80,0xC0,0xC0,0xC0,0xC0,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0xFF,0xFF,0xFF,0xFF,0xFF,0x07,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0xFF,0xFF,0xFF,0xFF,0x02,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x07,0x0F,0x0F,0x07,0x04,0x04,0x04,0x04,0x04,0x04,0x04,0x04,0x02,0x01,/*"L",0*/
0x00,0x00,0x00,0x00,0x80,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0x00,0x00,0x00,0x00,0x00,
0x00,0xC0,0xF8,0xFE,0xFF,0x1F,0x07,0x03,0x03,0x03,0x07,0x07,0x00,0x00,0x00,0x00,
0x00,0x3F,0xFF,0xFF,0xC0,0x00,0x00,0x1E,0x3E,0x3E,0xFE,0x3E,0x3F,0x3E,0x1C,0x00,
0x00,0x00,0x00,0x03,0x07,0x06,0x06,0x04,0x04,0x02,0x1F,0x00,0x00,0x00,0x00,0x00,/*"G",1*/
0x80,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0xC0,0x80,0x00,
0x07,0x3F,0x1F,0x0F,0x0F,0x0F,0x0F,0xFF,0x07,0x07,0x0F,0x0F,0x0F,0x0F,0x1F,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x07,0x04,0x00,0x00,0x00,0x00,0x00,0x00,0x00,/*"T",2*/
0x00,0x00,0x00,0x00,0x80,0xC0,0xC0,0xE0,0xE0,0xE0,0xC0,0xC0,0x80,0x80,0x00,0x00,
0x00,0x00,0x00,0x1F,0x7F,0xCF,0x87,0x07,0x07,0x8F,0x9F,0x7F,0x3F,0x3F,0x3E,0x00,
0x40,0xE0,0x18,0x0C,0x04,0x06,0x03,0x03,0x0D,0x38,0xE0,0xC0,0x00,0x00,0x00,0x00,
0x00,0x01,0x03,0x03,0x06,0x06,0x06,0x06,0x06,0x06,0x06,0x07,0x03,0x00,0x00,0x00,/*"8",3*/
};
```

这样，显示完整的”LGT8F690”时， 我们也需要分别进行处理：

```
71  // display a big banner : LGT8F690
72  void oled_drawLGT8F690(u8 page, u8 column)
73  {
74      u8 r_ofs, c_ofs;
75
76      for(u8 i = 0; i < 4; i++)
77      { // "LGT8"
78          r_ofs = axu_fmul8x8(i, 64);      // page offset
79          c_ofs = axu_fmul8x8(i, 16);      // column offset
80          oled_draw32x16(&_lgt8[r_ofs], page, (c_ofs + column));
81      }
82
83      for(u8 i = 0; i < 4; i++)
84      { // "F690"
85          r_ofs = axu_fmul8x8(i, 64);          // page offset
86          c_ofs = axu_fmul8x8(i, 16) + 64;     // column offset
87          oled_draw32x16(&_f690[r_ofs], page, (c_ofs + column));
88      }
89  }
90
```

显示前4个字符 “LGT8”

计算每个字符在字模数组对应的起始地址

数组每行16字节，对应宽度每个字符4行，共64字节

显示后4个字符 “F690”

基本的字模显示原理已介绍完毕。 为了更方便的打印显示， 我们通常是需要制作一个常用字符集的字库，配合一个通用的显示接口， 可以非常直观的打印我们所需的信息。这就需要制作字库。字库的制作原理和我们之前介绍的字模的制作是完全一样的。 区别是， 字库是一组常用的字符，可以直接通过字符本身索引到字符对应的字模数据的地址。

这里，我们介绍下常用的常用的ASCII编码英文字符的字库制作方式。常用显示的ASCII码字符被编排到0x20开始到0x74结束的一段连续的区域。 0x20对应空格字符， 0x74对应”~”字符，中间包含了阿拉伯数字字符，英文字符大小写以及其他常用的运算和标点符号。 我们可以直接用字符的编码减去0x20得到字符的索引， 这对于实现字符的显示非常方便，下面是常用ASCII字符：

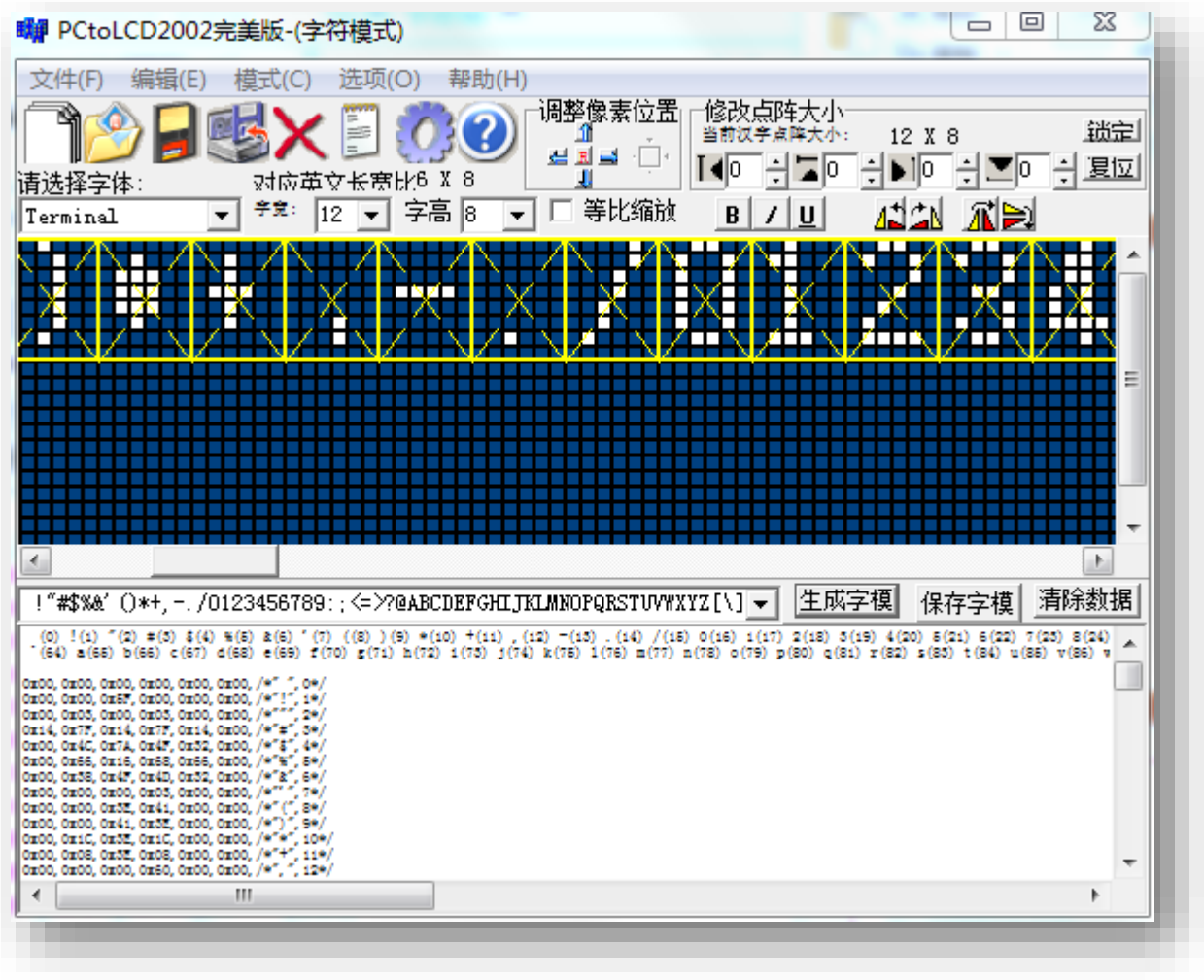
```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

有了字符，制作字库就很简单。 我们还需要确定下字体和字符大小。 为了节省内存和方便显示，我们可以选择比较小的字符。英文字符可比较小， 比如8X8像素， 或者更小的6X8像素。 注意，我们的OLED屏幕一行是8个像素的高度，这也是我们通过SPI接口能够控制的最小刷新单位。因此我们的字符最小高度也是8个像素。 但宽度可以小于8个像素， 这样我们一行就可以显示更多的字符。 6X8像素的字符， 高度是8，宽度是6，这样我们一行就可以显示出：128/6 = 21个字符。

但是，6X8像素的字符是非常挑字体的，大部分字体无法用6X8的像素正常显示。 我们这里提供的一种字体，可以很好的用6X8显示， 这种英文字体叫：Terminal。

显示字符的大小这个是应用决定的。 大部分人不习惯非常小的字体。 我们这里也提供另外一种8X16像素的字库， 高度为16，宽度为8。字体为: System。 这些都是PC上的常用字体， 也比较适合OLED显示。

下面简单介绍下6X8字库的制作过程：



PCtoLCD工具配置不变， 只需要修改下字符的宽度和高度。 之前有说明， 因为是英文字符，我们要得到6像素宽度的字符，需要设置宽度为12。 字符高度设置为8， 对应我们OLED一行的高度。

下面的编辑框里复制我们上面的ASCII字符， 然后产生字模！！
可以看到， 每个字符占一行， 对应字符的高度是一行。 每行6个字节， 对应字符的宽度。

这样简单的字库数组， 也是非常便于打印输出的。 我们首先根据字符计算出字符的对应数据的索引，然后将6个字节输出到指定到行列地址即可。字符的索引可以将字符减去0x20， 得到一个偏移量，用这个偏移量乘以6就可以得到字符对应在字库中数据的起始地址了。

同样， 这个字库的数组也需要拆分，分成数个256字节大小的小数组， 然后字符打印程序中， 根据字符的索引，选择对应的数组。每个字符是6个字节，字库大小一共：95x6 = 570字节， 需要分三个数组。数组比较大，这里就不详细列出，请参考本篇教程附带的实例代码”oled_spi.c”中的定义。

下面是基于6X8字库的字符串打印代码：

字库设计

```
347
348   // print string to panel
349   // row : start row (0 ~ 7)
350   // col : start colume (0 ~ 20)
351   // return new line if over a line
352   void oled_sprintf(u8 *buf, u8 row, u8 col)
353   {
354       u8 pos, ch;
355       u8 *pcs;
356
357       while((ch = *buf++) != '\0')
358       {
359           pos = ch - 0x20;
360           if(pos > 83) { pos -= 84; pcs = _ansi6x8_3; }
361           else if(pos > 41) { pos -= 42; pcs = _ansi6x8_2; }
362           else pcs = _ansi6x8_1;
363
364           pcs += axu_fmul8x8(pos, 6);
365
366           oled_draw6x8(pcs, row, col++);
367
368           if(col > 20) { // continue to next line
369               col = 0;
370               row += 1;
371           }
372       }
373   }
```

读字符串中的字符

减去0x20得到字符索引

根据索引计算所在数组

计算字模数据在数组中的起始地址

打印6个字节的字模数据

一行最多21个字符，超出换行处理

字模和字库的制作就介绍到这里。 本篇教程附带的实例中， 有完整的字库和接口输出驱动。

OLED接口驱动

我们所用OLED的屏幕的驱动芯片为SSD1306。主机通讯接口配置为四线SPI模式。对OLED屏幕的控制，是通过SPI接口以及D/C#向SSD1306发送命令字和数据实现。命令字和数据使用D/C#来进行选择。命令字用于设置SSD1306的工作模式，数据就是写到GDRAM用于驱动显示的数据。

另外，SD1306还需要一根复位控制线RST#， 主要用于当显示异常或者需要重新初始化显示设备时，通过RST#将SD1306复位到初始状态。因此，正常的完成与SSD1306的通讯，我们需要另外两个I/O分别用于实现D/C#和RST#的控制。 本篇例程之前已有说明，我们使用LGT8F690A的RA2/3两个I/O来实现。

SSD1306上电后，默认状态下是不能直接显示数据的。我们需要对SSD1306进行系列的配置。SSD1306的启动和配置有一个固定的流程， 可以参考附件代码中的LED初始化函数”oled_init()”的实现，这里我们简单的列出相关操作， 首先是上电的准备流程：

1. 通过VCC上电
2. 通过拉低SPSS，SSD1306选择并处于有效工作状态；
3. 通过拉低RST#，延时后拉高RST#，产生一次复位，让SSD1306回到初始状态；

接下来是一组显示驱动相关的流程，通过发送命令字(D/C# = 0)，配置SSD1306的工作状态：

4. 首先关闭显示，避免初始化过程中看到花屏状态；
5. 设置刷屏周期，一般选择默认的100帧/秒；
6. 设置COM口的复用比例，SSD1306的COM输出可复用为SEG，我们用的是64COM的屏，这里需要设置位默认的0x3F, 64个COM全部使用。
7. 设置显示区与GDRAM的映射关系。一把是正常的一对一直接映射；
8. 开启SSD1306电荷渠，产生用于驱动屏的高压(7.5V)；
9. 设置GDRAM的寻址模式，我们设置为常用的页寻址模式；
10. 设置SEG/COM的扫描模式， 这部分细节请参考SD1306数据手册；
11. 设置对比度， 对于单色屏，相当于控制亮度。一般用默认值即可；
12. 设置于充电周期，
13. 设置COM在无效状态的的电压，选择默认值(0.77VCC)；
14. 设置屏的显示模式；
15. 设置完成，使能显示输出；

SSD1306初始化完成后， 就可以正常的接收显示数据了。 显示数据时，需要首先通过发命令设置显示的位置，就是设置当前数据写操作的GDRAM的位置， 也对应最终的显示位置。对于页寻址模式，就是设置页地址和列地址。 地址设置后， 可以顺序发送以字节为单位的数据， 每个字节数据对应当前页的一列像素点的显示状态。

OLED屏驱动接口驱动，以及字库，字库驱动等代码因为比较多，这里就不列出，请参考附带例程！
下面简单介绍下我们的主程序部分：