点阵 LCD 的驱动显控原理

——小丑、Powerint 2007 年 8 月

V1.0

Mz 出品

1. 一些需在提示您的

1.1. 本书更适合什么样的 LCD 模块?

在本书的开始之处，先将本书将要介绍的 LCD 圈定一个小的范围，即本书所说的 LCD 指的是哪类型的 LCD?
在这里将主要针对单色的点阵液晶屏（LCD）进行介绍，而且是针对 LCD 模块本身就集成有驱动控制 IC 的，以及显存；那些字符型以及段码型的 LCD 不在介绍之例，但也可以在一定程序上参考本书的驱动编程方法来编写这些 LCD 的驱动程序。此外，也有些小规模的（一般是 3.5 寸以下）彩色 TFT LCD 也有内置驱动控制器以及显存的，也可参考这里的介绍来编写它们的驱动程序。

1.2. 这里的 LCD 驱动程序更适合于什么样的 MCU

MCU 即常说的单片机，本书所介绍的通用版 LCD 驱动程序其实是可以用在所有的 MCU 的，只不过，笔者从应用的角度出发，建议在使用本书所介绍的驱动程序时，更适合的 MCU 类型。
目前市面上的 MCU 非常之多，从功能从资源角度来看的话，大概可分为以下几类：

1、 小资源 MCU，类似于传统 51 的 89S51 单片机、PIC 等的小资源单片机等，通常它们的资源都很少，片内的 ROM 少于或等于 4K byte，RAM 少于或等于 128 byte，速度较慢，MIPS 数通常在 1M MIPS 左右；
2、 中资源 MCU，这类 MCU 的涵盖面非常广，在实际的产品设计中应用非常多，大概定义如一些增强型的 51 单片机、中资源的 AVR 单片机、16 位的 MSP430 系列的中等资源单片机、凌阳的 SPCE061A、PIC 的中等资源单片机等等，非常多，甚至包含至 ARM7 核心的 LPC 系列 MCU，如 LPC21XX 系列等；一般来说指的是片内的 ROM 资源在 8K byte 以上，RAM 在 256 byte 以上，MCU 的运行速度较快，片内资源丰富，应用面非常广；
3、 跑 OS 的大资源 MCU，这类的 MCU 其实大部份指 ARM7 和 ARM9 核心或与这些核心同等级的处理器了，通常都会在设计中跑操作系统，也就是现在常说的 32 位嵌入式处理器。

本书所介绍的 LCD 驱动程序更适合于中等资源的 MCU，因为它们有足够的片内资源和运行的速度，而且在应用它们的设计当中往往会涉及到 LCD 的人机界面显示。

其实在此，无非就是在于说明，如果您使用一些类似 2051、89S51 级别的 MCU 的话，没有太多必要使用本书所介绍的驱动程序，因为很有可能连 LCD 驱动中的自带字库都装不下去，不过驱动的方法还是可以参考的。
而如果您使用的是较大资源的嵌入式处理器，而且本身就跑着图形操作系统的话，更没必要了，因为往往这类 MCU 都会自带有 LCD 的控制器，而操作系统也带有完整的图形界面接口。

1.3. 为什么用 C 语言

本书中全部的源码都是以 C 语言为平台的，这点请读者确认在读本书前，自身已经具备 C 语言的编程基础。
现下，中等资源的 MCU 开发如果没有 C 语言的编译器支持的话，可以说是极其少见的了；使用 C 语言对 MCU 进行开发已经形成当前的主流模式，何乐而不为呢！
ASM 的执行效率比 C 语言的代码要好？当然，这是肯定的，不过现在的 MCU 速度和性能已经提升上来了，而且很多优秀的 C 编译器编译的结果就未必比自己使用 ASM 编程的结果效率低。
ASM 的代码短小精悍？从某个角度来说是的，不过时下优秀的 C 编译器对 C 代码的优化比自己使用 ASM 编写的还要好；况且，在编写结构复杂的程序时，C 语言肯定比 ASM 占优势，在编程速度上和可读性以及可移植性上等。

为什么不用 C 语言编程呢？

2. 以 MzL02 LCD 模块为例

这里将介绍铭正同创所销售的 LCD 模块：MzL02-12864，并在后面的大部分例子当中，以它为介绍的对像。
实际上现在市面上的 LCD 种类非常多，各个厂家生产的编号都有所不同，即使是使用一样的玻璃、驱动控制 IC 都有可能存在不同的产品编号；但真正意义上对设计者（软硬件工程师）来说有用的，也就是 LCD 模块当中的驱动控制器型号以及驱动控制器与玻璃的连接方法（也就是生产 LCD 模块时驱动控制器与玻璃引脚的连接，以及一些驱动控制器封装好的特性等）。无论怎么着，LCD 模块总还是大同小异的，这里以 MzL02 LCD 模块为对像介绍，并不代表本书仅适用于该 LCD 模块，其它厂家生产的不同型号的 LCD 模块也可以套用这里的介绍去理解、去掌握 LCD 驱动程序的编程方法，当然仅仅是供您参考。

2.1. LCD 模块的结构

通常我们所见到的 LCD 模块，分为几部分：LCM（玻璃）、背光、PCB 板；而背光和 PCB 板部分其实是可有可无的，视具体的 LCD 模块而定。点阵的 LCD 模块按照驱动控制器的集成方式，大可分为两种：COB 和 COG；COG 其实就是将驱动控制 IC 集成到了玻璃里面，这样的而后面的 PCB 板上其实只是一些驱动控制 IC 无法集成的电容电阻而已；COB 也就是把驱动控制 IC 焊接在 LCD 模块后面的 PCB 板上。

MzL02-12864 为一块 128X64 点阵的 LCD 显示模块，模块上的 LCM 采用 COG 技术将控制（包括显存）、驱动器集成在 LCM 的玻璃上，接口简单、操作方便；为方便用户的使用，在 LCM 的基础上设计了 MzL02-12864 模块，将模块所必需的外围电容电阻集成到模块上，并引出多种形式的引线接口方便用户使用。MzL02-12864 模块与各种 MCU 均可进行方便简单的接口操作。

1. 128 x 64 点阵 FSTN
2. 1/64 占空比 1/9 偏压比
3. 单电源供电对比度编程可调
4. 并行接口为 6800 时序 MPU 接口方式
5. 3.3V 的白色 LED 背光，美观大方
6. 集成 S6B0724 驱动控制 IC

2.1.1. 结构示意

图 1.1 为 MzL02-12864 模块中的 LCM 的结构尺寸示意图。

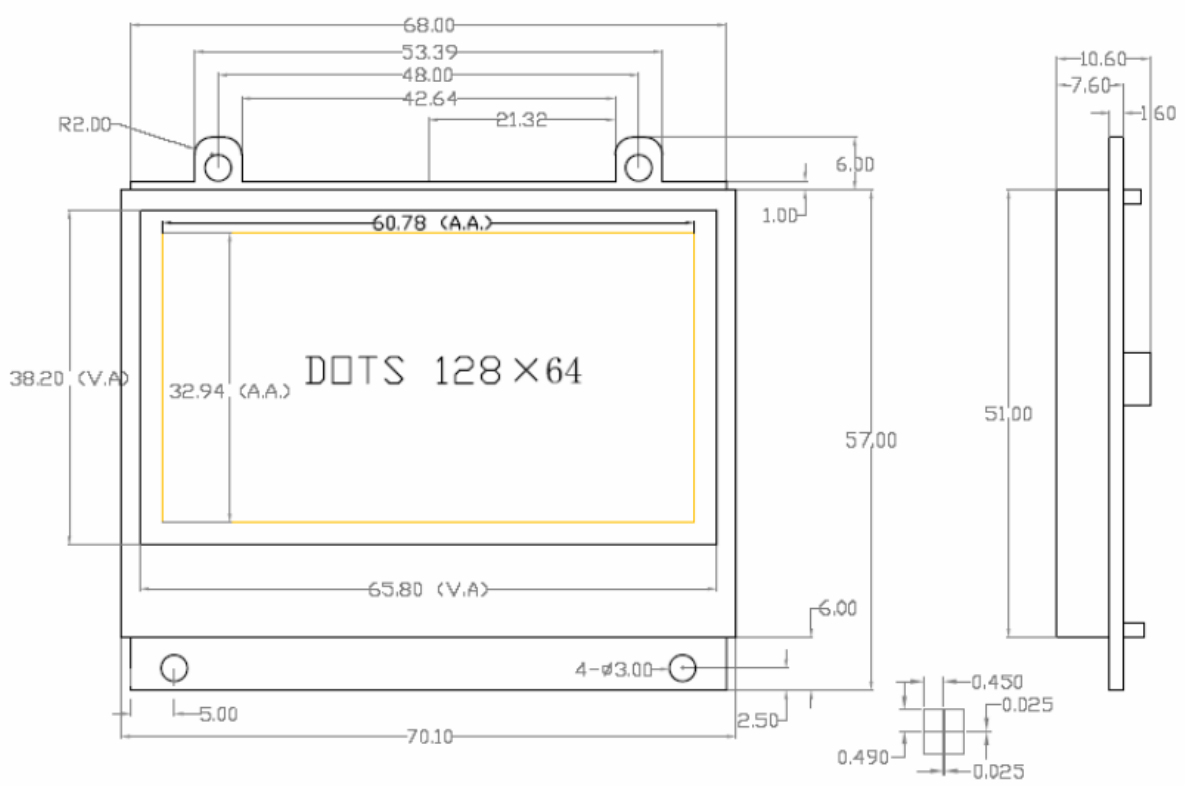


图 2.1 LCM 结构尺寸

2.1.2. 显示 RAM 区映射情况

对于 LCD 模块，了解清楚驱动控制 IC 当中的显存与 LCD 玻璃上的点的对应关系是非常重要的，这是编写 LCD 的驱动程序的基础。

MzL02-12864 液晶显示模块的显示器(玻璃)上的显示点与驱动控制芯片中的显示缓存 RAM 是一一对应的；驱动控制芯片当中共有 65（8 Page x 8 bit+1）X 132 个位的显示 RAM 区。而显示器的显示点阵大小为 64X128 点，所以实际上在液晶显示模块中有用的显示 RAM 区为 64 X 128 个位；按 byte 为单位划分，共分为 8 个 Page，每个 Page 为 8 行，而每一行为 128 个位（即 128 列）。

驱动控制芯片的显示RAM区每个byte的数据对应屏上的点的排列方式为：纵向排列，低位在上高位在下；如 图 2.2 所示：

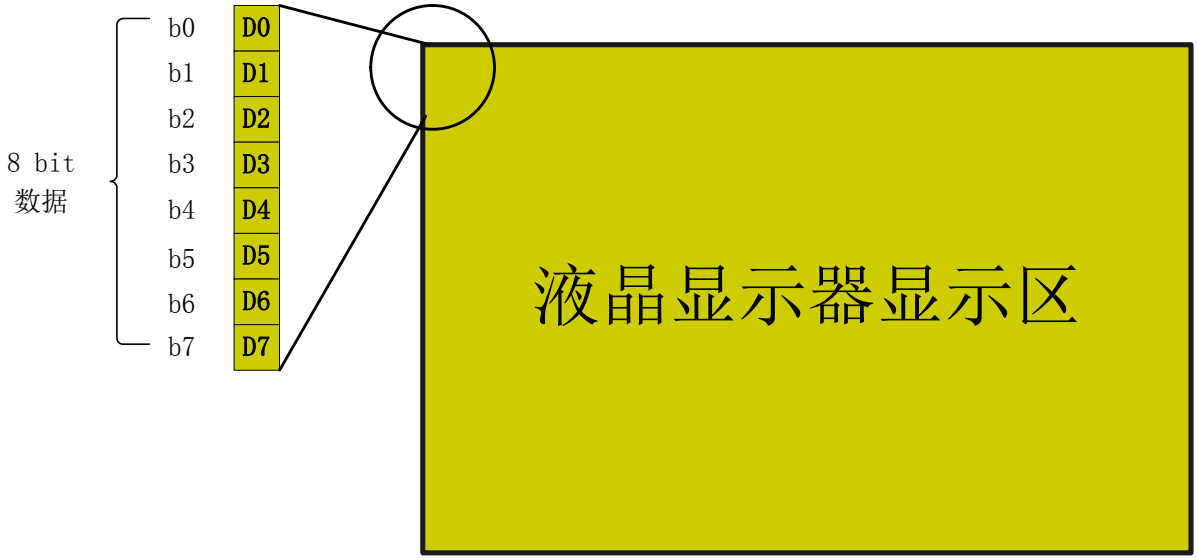


图 2.2 字节数据排列情况

MzL02-12864 液晶显示模块的显示屏上的每一个点都对应有控制器片内的显示缓存RAM中的一个位，显示屏上 64X128 个点分别对应着显示RAM的 8 个Page，每一个Page有 128 个byte的空间对应。因此可知显示RAM区中的一个Page空间对应 8 行的点，而该Page中的一个byte数据则对应一列（8 个点）。图 2.3 为显示RAM区与显示屏的点映射图：

			列 行	LCD 显示器横向坐标（自左至右）								
				0	1	2	3	……	……	125	126	127
LCD 显示器纵向坐标（自上至下）	Page0	8bit 数据	0	bit0	bit0	bit0	bit0	……	……	bit0	bit0	bit0
			1	bit1	bit1	bit1	bit1	……	……	bit1	bit1	bit1
			2	bit2	bit2	bit2	bit2	……	……	bit2	bit2	bit2
			……	……	……	……	……	……	……	……	……	……
			6	bit6	bit6	bit6	bit6	……	……	bit6	bit6	bit6
			7	bit7	bit7	bit7	bit7	……	……	bit7	bit7	bit7
	Page1	8bit 数据	8	bit0	bit0	bit0	bit0	……	……	bit0	bit0	bit0
			9	bit1	bit1	bit1	bit1	……	……	bit1	bit1	bit1
			……	……	……	……	……	……	……	……	……	……
			15	bit7	bit7	bit7	bit7	……	……	bit7	bit7	bit7
	……		……	……	……	……	……	……	……	……	……	……
	……		……	……	……	……	……	……	……	……	……	……
	Page7	8bit 数据	56	bit0	bit0	bit0	bit0	……	……	bit0	bit0	bit0
			……	……	……	……	……	……	……	……	……	……
			59	bit7	bit7	bit7	bit7	……	……	bit7	bit7	bit7
			60	bit0	bit0	bit0	bit0	……	……	bit0	bit0	bit0
			61	bit1	bit1	bit1	bit1	……	……	bit1	bit1	bit1
			62	bit2	bit2	bit2	bit2	……	……	bit2	bit2	bit2
			63	bit3	bit3	bit3	bit3	……	……	bit3	bit3	bit3

图 2.3 显示 RAM 区与显示屏点映射图

2.1.3. 行、列地址

用户如要点亮 LCD 屏上的某一个点时，实际上就是对该点所对应的显示 RAM 区中的某一个位进行置 1 操作；所以就要确定该点所处的行地址、列地址。从上图中可以看出，MzL02-12864 液晶显示模组的行地址实际上就是 Page 的信息，每一个 Page 应有 8 行；而列地址则表示该点的横坐标，在屏上为从左到右排列，Page 中的一个 Byte 对应的是一列（8 行，即 8 个点），达 128 列。

可以根据这样的关系在程序中控制 LCD 显示屏的显示。

注意：MzL02-12864 的显示缓存 RAM 区实际上比模块上的显示器所对应的 RAM 区要大；而 LCD 模块具体设置 Page（有时也称页）时，屏上的位置与驱动控制 IC 当中的哪里的 RAM 区对应，还与驱动控制 IC 与屏的连接有关；所以，实际在使用时，请参考所提供的范例设置（主要是设置 COM 反向扫描、SEG 设置为正向扫描，以此设置方法，则每个 Page 中的前三列以及最后一列是不对应在 LCD 屏幕上的）。

2.1.4. 其它

市面上的 LCD 模块在屏幕点与显存对应关系这块，都不尽相同，这主要跟玻璃的 SEG、COM 与驱动器间的连接关系定的，所以，在编写一块 LCD 模块的驱动程序前，对这块 LCD 模块的显存和显示点对应关系的了解是非常必要的。

2.2. LCD 的接口

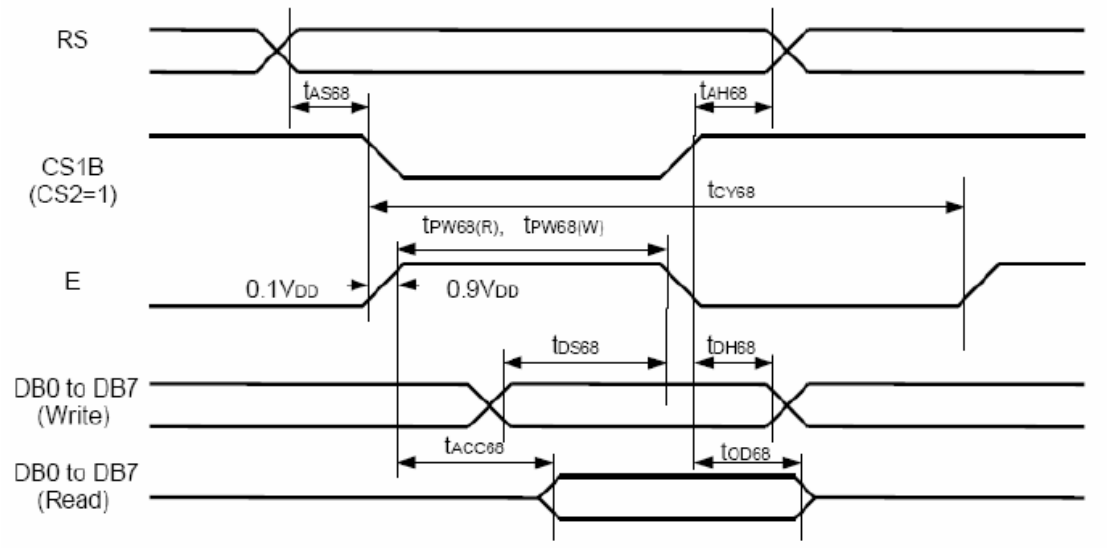
一般来说，LCD 模块（带有驱动控制器）的接口多为总线的接口，不是 6800 就是 8080，或者是串行 SPI（及类 SPI 时序）；除了这些总线的端口外，有的 LCD 模块还引出了一些功能性的端口，如偏压调节输入、负压输出等，当然还有电源输入端口，这些一般与编写驱动程序没有太多的关系，所以这里就不介绍了。

MzL02 LCD 模块的接口如下：

序号	接口引脚名	说明	序号	接口引脚名	说明
1	VDD	LCM 供电	2	LEDA	背光正极输入
3	DB0	8 位数据总线	4	/CS	片选（低电平有效）
5	DB1		6	/RST	复位脚（低电平复位）
7	DB2		8	RS（A0）	数据命令选择脚
9	DB3		10	/WR	6800 系列 MPU 的读/写信号（R/W）
11	DB4		12	EP	6800 系列 MPU 的时钟信号使能脚（EP）
13	DB5		14		
15	DB5		16		
17	DB7		18		
19	VSS	地	20	LEDK	背光负极输入

这是很典型的 6800 总线 LCD 接口，有的 LCD 模块的资料里，也有将 A0 称为 RS，只是字面表达不同而已，一般来说意义是一样的。

它的操作时序图如下：



上图并不是非常准确，一般来说，片选信号 CS 要在整个时序过程当中保持有效（低电平）；而 RS 信号是一种状态的设定，以此来告诉被操作的 LCD 驱动控制器，当前的操作是针对其中的寄存器还是显存的。而数据（包括设置寄存器的数据等，只要是数据线上的信号）的

载入和装载（指的是被操作的 LCD 控制器将要读取的数据装载至数据端口，以供操作者（MCU）读取）的触发由 EP 信号的下降沿触发。

注意：关于 EP 信号的有效触发沿，上图中有很大的可能性有误，经试验实际上是上升沿触发的，以上的描述只是照图分析，上图是 S6B0724 的 PDF 资料当中来的。

2.3. LCD 控制器特性

LCD 模块中，用户程序对其进行显示控制时，无非就是通过对 LCD 模块内部的驱动控制器当中的寄存器进行设置操作；最常用的如 LCD 的显示开/关、显存操作地址（行与列地址）的设置等。这些寄存器一般都在 LCD 模块的驱动控制器文档中有详细介绍，所以在编写驱动程序时，有必要拿到一份驱动控制器的文档；不过，一般常使用到的寄存器不会太多（除了一些在上电后需要初始化的寄存器外）。

MzL02-1286 液晶显示模组共有 22 种显示指令，下面分别介绍以下 20 种指令：
注意：下面的指令介绍中，A0P 信号指的是 RS 信号。

1，显示开关指令

A0P	EP RD	RWP WR	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Setting
0	1	0	1	0	1	0	1	1	1	1	Display ON
										0	Display OFF

2，显示起始行设置

这个指令设置了对应显示屏上首行的显示 RAM 行号。有规律的修改该行号，可以实现滚屏功能。

A0P	EP RD	RWP WR	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Line Address
0	1	0	0	1	0	0	0	0	0	0	0
					0	0	0	0	0	1	1
					0	0	0	0	1	0	2
							↓				↓
					1	1	1	1	1	0	62
					1	1	1	1	1	1	63

3，页地址设置

A0P	EP RD	RWP WR	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Page Address
0	1	0	1	0	1	1	0	0	0	0	0
							0	0	0	1	1
							0	0	1	0	2
									↓		↓
							0	1	1	1	7
							1	0	0	0	8

4，设置列地址

由上图可以看出显示 RAM 被分成 9 页每页 132 个字节，当设置了页地址和列地址后，就确

定了显示 RAM 中的唯一单元，该单元由低到高各个数据位对应于显示屏上的某一列的 8 行数据位。

注：在本模组中与 LCD 屏上对应的显示 RAM 仅为 8 页有效（0~7），每页 128 字节，所以每页当中的条一列和最后的三列是不在显示屏上有点对应的，这点用户在使用时请注意，可以参考提供的驱动程序。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	A7	A6	A5	A4	A3	A2	A1	A0	Column Address
0	1	0	0	0	0	1	A7	A6	A5	A4	0	0	0	0	0	0	0	0	0
						0	A3	A2	A1	A0	0	0	0	0	0	0	0	1	1
											0	0	0	0	0	0	1	0	2
															↓				↓
											1	0	0	0	0	0	0	0	130
											1	0	0	0	0	0	1	1	131

列地址的设置需要连续写两次指令，如上图所示，指令数据为 0001XXXXB 和 0000XXXXB，都是用低四位放置有 8 位地址的高低四位数据，而指令的 DB4 指明当前设置的是高四位地址还是低四位地址。

5，读状态

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	BUSY	ADC	ON/OFF	$\overline{RESET}$	0	0	0	0

BUSY	当 BUSY 为 1 时，忙状态；当 BUSY 为 0 时，准备好状态，
ADC	表示行和列的关系 ADC:1 正常输出（n－131==SEGn），ADC:0 为反向输出（131－n ==SEG n）
ON/OFF	表示液晶显示开和关 0：显示打开，1：显示关闭
RESET	0：正常工作状态，1：复位

6，写显示数据

这条指令可以将显示数据（8 位）写到 RAM 中，显示地址自动加一。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	1	0	Write data							

7，读显示数据

这条指令从指定地址中读取显示数据，读取显示数据后，列地址自动加一。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	0	1	Read Data							

8，ADC 选择（Segment 方向选择）

这条命令用于将 Segment 驱动输出反向。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Setting
0	1	0	1	0	1	0	0	0	0	0	Normal
										1	Reverse

9，正向/反向显示

这条命令用于设置显示正向和反向。正向为正常模式，反向时 LCD 屏的显示将反色显示；但执行该指令后，显示 RAM 中的内容不变。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Setting
0	1	0	1	0	1	0	0	1	1	0	RAM Data 'H' LCD ON voltage (normal
										1	RAM Data 'L' LCD ON voltage (reverse)

10，全屏点亮/变暗

这条命令使所有的液晶点被点亮/变暗，无论显示 RAM 中有任何数据。此命令优先于正向/反向显示。当液晶处于显示关闭状态时，执行此命令将会自动进入节电状态。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Setting
0	1	0	1	0	1	0	0	1	0	0	Normal display mode
										1	Display all points ON

11，LCD 偏压设置

这条命令用于液晶显示的偏压设置。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Select Status
0	1	0	1	0	1	0	0	0	1	0	1/9 bias
										1	1/7 bias

12，读/改/写模式设置

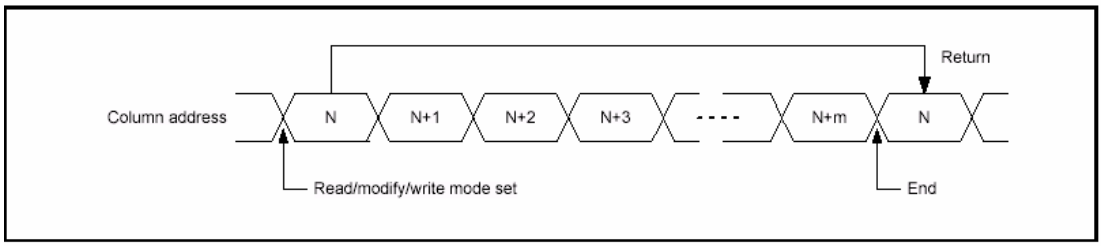
这条指令用到两次结束命令，一旦写入此命令后，读显示数据命令不再修改列地址，但是写显示数据命令还可以使列地址自动加一。当有结束命令输入时，列地址恢复到读/改/写时的列地址。这个命令可用于光标显示。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	0	1	1	1	0	0	0	0	0

13，读/改/写模式结束

这条指令用于结束读/改/写模式。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	0	1	1	1	0	1	1	1	0



14，复位

这条指令初始化显示起始行、起始列地址、起始页地址、正常输出模式。结束读/改/写模式和测试模式。此命令不影响显示 RAM 中的数据。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	0	1	1	1	0	0	0	1	0

15，COM 口扫描方向选择

这条指令用于确定 COM 口扫描的方向。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Select Status SPLC501C
0	1	0	1	1	0	0	0	*	*	*	Normal COM0 --> COM63
							1				Reverse COM63 --> COM0

16，上电控制设置

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Selected Mode
0	1	0	0	0	1	0	1	0			Booster circuit: OFF
								1			Booster circuit: ON
									0		Voltage regulator circuit :OFF
									1		Voltage regulator circuit: ON
										0	Voltage follower circuit: OFF
										1	Voltage follower circuit: ON

17，V5 电压内部电阻调整设置

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Setting
0	1	0	0	0	1	0	0	0	0	0	Small
								0	0	1	
								0	1	0	
									↓		↓
								1	1	0	
								1	1	1	Large

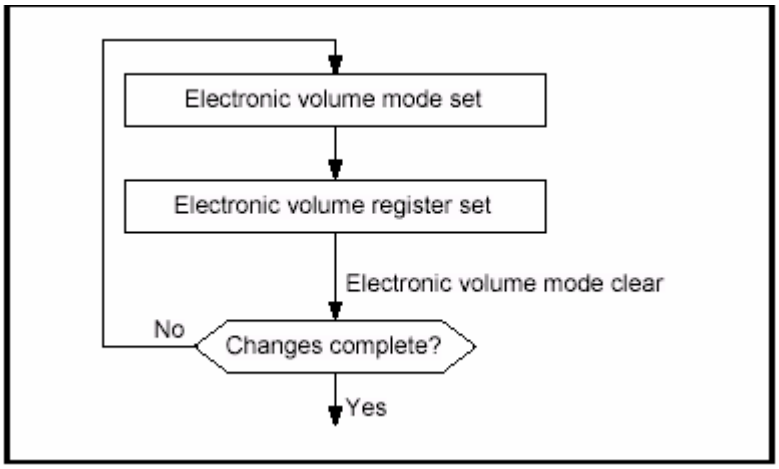
18，电量（electronic Volume）设置模式

这条命令用于调整显示屏的亮度。此命令用到双字节：一个是设置为电量设置模式，另一个是设置电量寄存器设置模式。

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	1	0	1	0	0	0	0	0	0	1

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	| Vs|
0	1	0	*	*	0	0	0	0	0	1	Small
0	1	0	*	*	0	0	0	0	1	0	
0	1	0	*	*	0	0	0	0	1	1	
						↓					↓
0	1	0	*	*	1	1	1	1	1	0	
0	1	0	*	*	1	1	1	1	1	1	Large

流程如下：



19，静态指示器

这条命令用于控制静态驱动指示器显示。为双字节命令。

静态指示器开/关

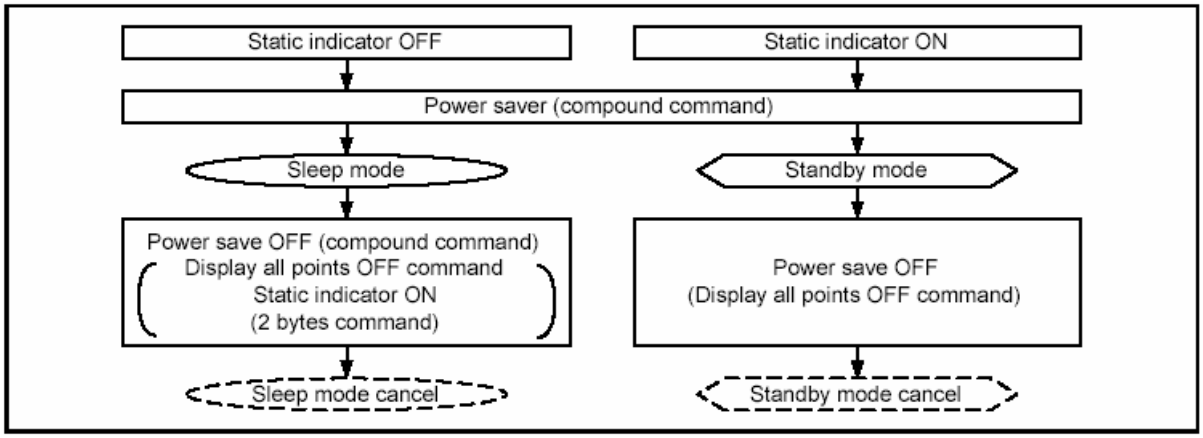
A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Static Indicator
0	1	0	1	0	1	0	1	1	0	0	OFF
										1	ON

静态指示器寄存器设置状态

A0P	EP $\overline{RD}$	RWP $\overline{WR}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Static Indicator
0	1	0	*	*	*	*	*	*	0	0	OFF
			*	*	*	*	*	*	0	1	ON (blinking at approximately 0.5 second intervals)
			*	*	*	*	*	*	1	0	ON (blinking at approximately one second intervals)
			*	*	*	*	*	*	1	1	ON (constantly on)

20，节电模式

当在显示关闭时，设置全屏点亮，则进入节电状态。节电模式有两种状态一个是睡眠模式另一个是备用模式。当静态指示器关闭时，进入睡眠模式。当静态指示器打开时，进入备用模式。在睡眠模式和备用模式时，显示数据保存操作模式时的数据。在这种模式时，MPU 可以访问显示 RAM。



睡眠模式：
在此模式下，除了 MPU 访问显示 RAM 外，停止所有的液晶显示操作。晶振、液晶上电和液晶驱动电路全部暂停。
备用模式：
在此模式下，液晶上电和液晶驱动电路暂停，振荡器继续振荡。在备用模式下，有复位命令时，系统进入睡眠模式。

2.4. LCD 驱动的基本流程

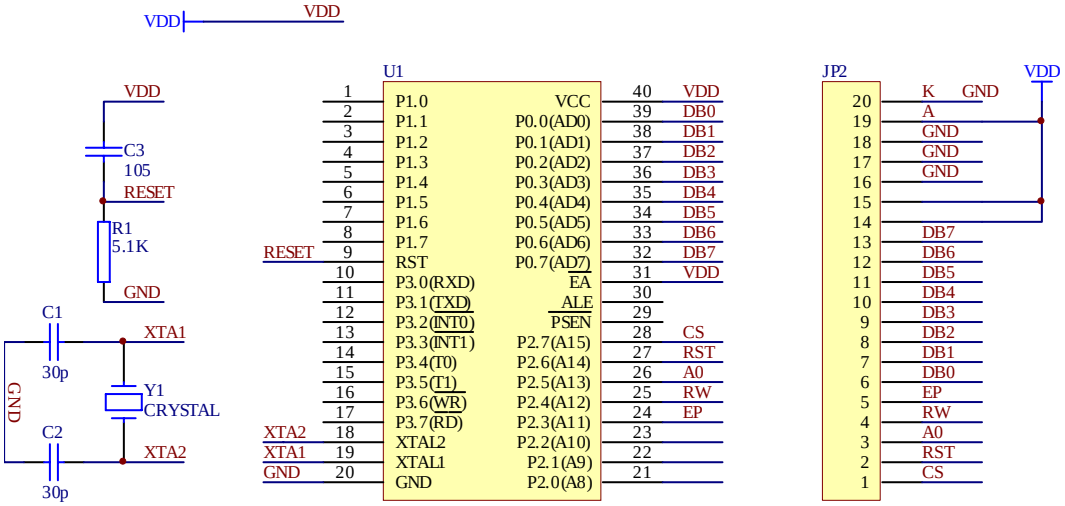
介绍基本的流程控制方法，这里重在介绍方法，从时序的模拟或者是总线的连接，到利用 LCD 的特性来做一些显示的处理，如单色液晶如何显示一个点，彩色 LCD 如何显示一个点的关系；操作地址与 LCD 上点的对应关系；LCD 控制器的特性，如指针自动加/减特性如何利用、滚屏特性如何利用、COM 和 SEG 扫描正相反相的特性利用等等~~~~~

2.4.1. LCD 模块的连接

在很多资料以及书籍当中，通常介绍有两种 LCD 模块的连接方法：直接控制和间接控制；直接控制实际指的是 LCD 模块的总线接口直接与 MCU 端口连接，然后 MCU 通过程序控制端口来模拟 LCD 的总线时序来完成对其的控制操作；而间接控制指的是 MCU 本身就有外部总线拉出，与 LCD 的总线接口对应的连接上，程序中直接操作总线以控制 LCD。
目前有很多 MCU 都把总线藏着掖着了，都没有引出外部总线；所以通常在用 MCU 控制 LCD 模块时，时常会选择直接控制的方式，即利用端口来模拟总线时序；当然了，如果本身就有总线而且也与 LCD 模块的总线配得上的话，肯定会使用总线连接的间控方式。

MzL02 LCD 模块的总线接口为 6800 总线，这与绝大部分引出外部总线的 MCU 的时序是不一样的，再说了本身就有那么多的 MCU 连总线都没有外拉，所以，在这里将采用端口模拟总线的方式对 LCD 模块进行控制。
采用端口模拟时序的方式控制 LCD 模块时，端口的连接非常简单也有很大的自由度；一般来说，会选择 8 个（跟总线宽度相同的个数）连续的端口作为数据端口与 LCD 模块的数据端口连接，而其它的如 CS、RS（A0）、RESET（RST）、WR、EP 之类的引脚连接时就比较随意了，视具体情况而定。当然，有的 LCD 模块还有其它的一些引脚需要连接，不过如前面所述，跟这里所介绍的编写驱动程序的关系不大，就不作说明了，可以直接参考 LCD 模块的文档介绍的连接即可。

下图就为 MCS51 与 MzL02 LCD 的连接电路图，采用直控的端口模拟时序的方式。



注：此图仅供参考，可能与有的资料上不太一样，注意一下 JP2 是指 MzL02 模块上的 FPC20 的接线座。

2.4.2. 控制 LCD 模块显示一个点

点阵 LCD 的特点就是以点的形式呈现用户想要显示的图形，故点阵 LCD 又有称之为图形点阵 LCD；通常在编写一个 LCD 模块的驱动程序时，最基本的功能是绘制一个具体指定点，只有在这样的功能的基础之上，才能通过各个点的组合，呈现出点阵的图形。

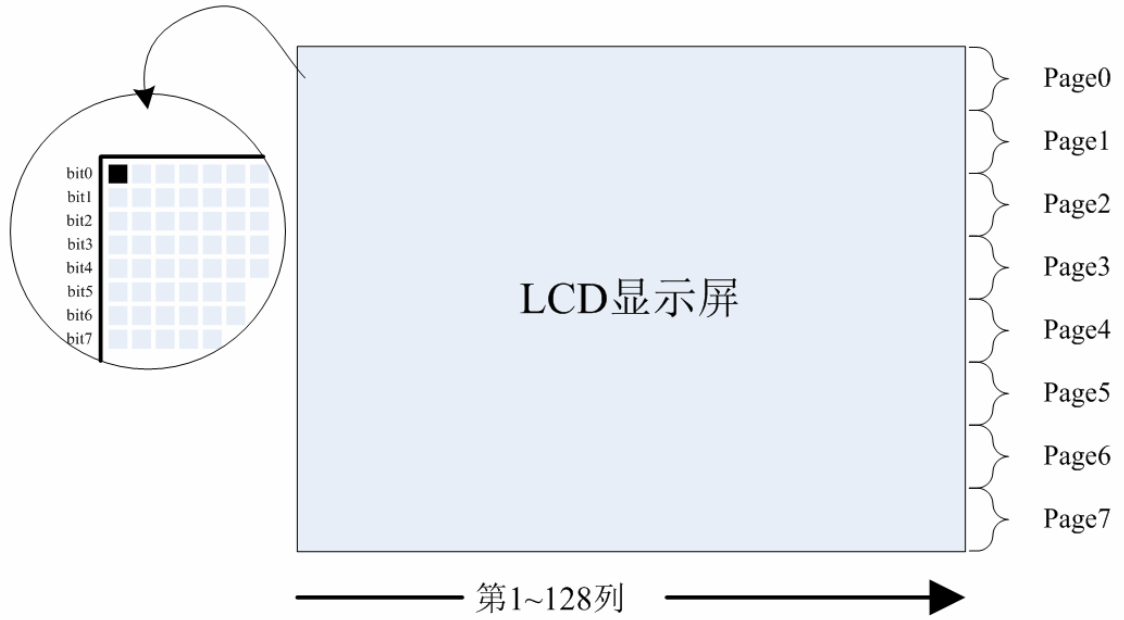
其实，绘制一个指定位置的点，也就是将显存当中的对应该点的数据位进行操作；在前面的 LCD 显示 RAM 区映射介绍当中，可以得知显存当中的数据与 LCD 屏幕上的点的对应关系，这样就可以在程序当中通过简单的换算而有序的控制 LCD 屏上的点的显示了。

比如，在 MzL02 模块当中，要将坐标位置（0，0）的点点亮时该点对应显存的情况分析如下：

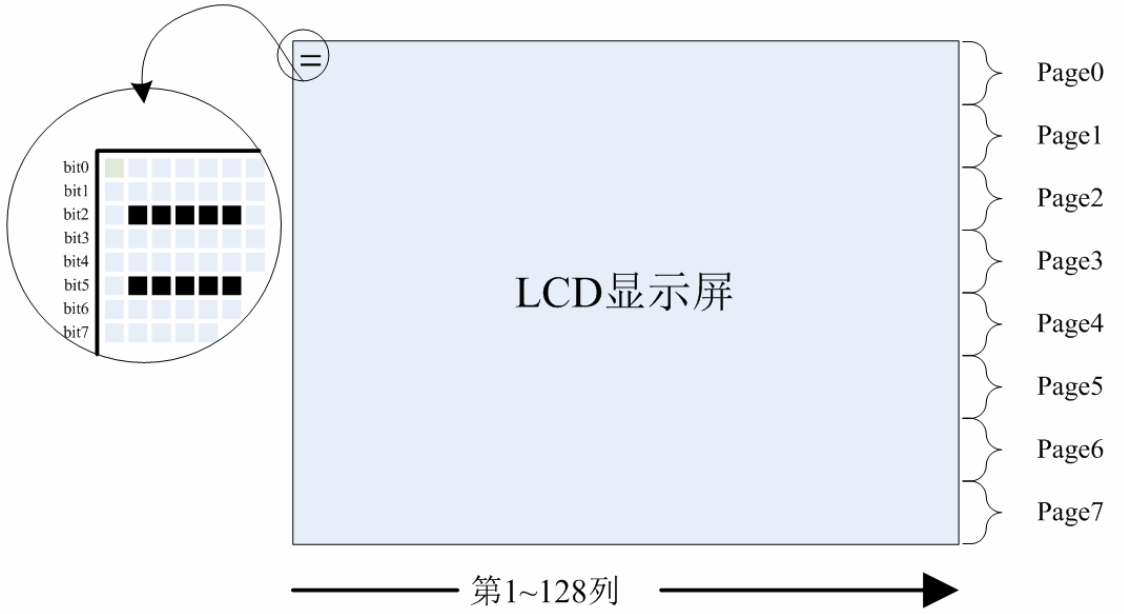
坐标为（0，0）的点我们定义其位于屏幕正向面对我们时的左上角的点，根据前面介绍的内容，可以行知，该点对应为 Page0 的第一个 byte 数据上，而且准确的位置是对应在这个 byte 数据的第 0 位（bit0）上。

如此以来，就可以对显存当中行地址（Page）为 0，列地址为 1（前面寄存器的介绍当中已注明，第 0、129、130、131 列在屏中没有对应的点，所以列地址为 1 时正是屏上点的最左边一列）的显存写入 0x01 的数据以点亮程序中坐标点（0，0）的点。

如下图所示意。



了解在 LCD 屏上绘点的基本原则，就可以灵活的应用于不同的显示操作了，比如：
在 Page0 上的第 0、1、2、3、4、5、6 列分别填入数据：0x00,0x24,024,0x24,0x24,0x24,0x00 后，可以在屏上显示出字符：“=”，如下图显示。实际上就相当于在 Page0 的第 0 列起始显示了一个 8*7（高 8 点、宽 7 点）的字符。

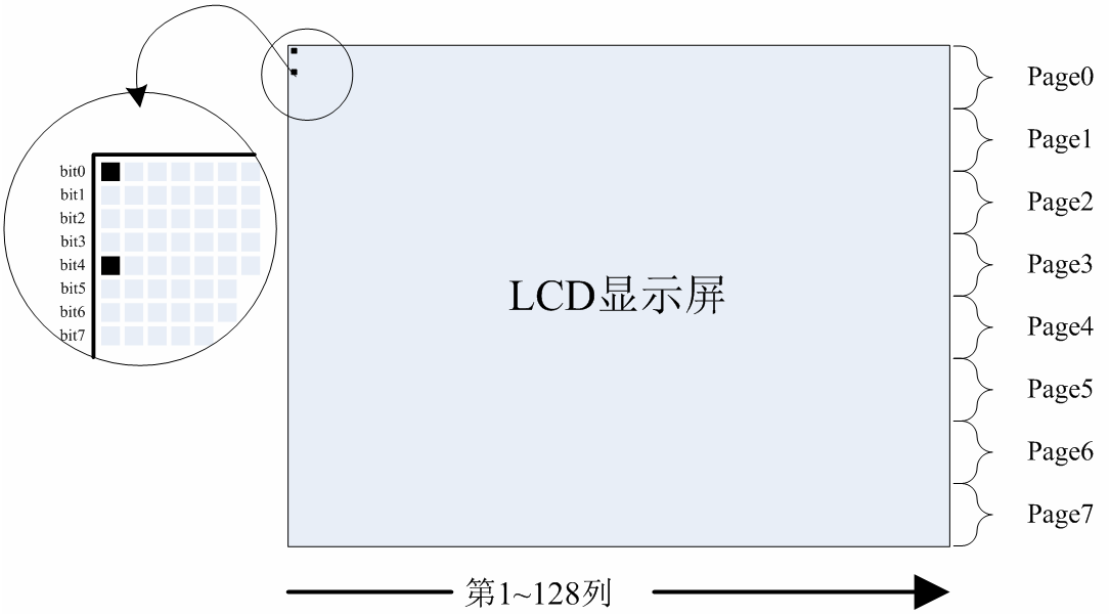


上面所讲的方法是很多人在编写 LCD 的驱动程序时采用的直接填充方式的字符显示方法，不过，这种方法显示字符，在这种单色的 LCD 模块当中是不可以实现任意位置显示字符的。故本书所介绍的字符显示方法并没有采用这样的方法，在接下来的介绍当中会有详细的叙述。

再返回到绘点原理上，这里，建议采用“读—改—写”的方式来完成一个点的显示，即实现本书所介绍的驱动程序架构的最基本的绘图子函数：绘点函数。
实际上，“读—改—写”的绘点方式在单色的点阵 LCD 模块驱动程序当中的思想就是：从要绘制的点所在的显存 byte 读回原来 LCD 屏上显示的点所对应的一个 byte 数据，然后针对要

绘制的点所在该 byte 的实际 bit 位置，改变该 bit 的数据（无非就是 0 或者 1）而保留该 byte 的其它 bit 数据不变，然后再将改完的 byte 数据写回读取它的显存位置。

例如，在 LCD 屏上，已经在坐标（0，0）的位置上有显示了黑点，这时，想在坐标位置（0，4）的点上再绘制一个点；如前面的图所示，该点的所对应的显存数据为 0x01，则将其读回，改变该 byte 的 bit4 值，则数据变为 0x11，再将其写回原位置，就可以显示如下图：



依此方法绘制点，就可以具备以下的优点：

1、 任意位置显示点，而不影响该点以外的点的显示；
2、 在此绘点功能的基础上构建的字符显示/图形显示可以实现任意位置的显示；
3、 在此绘点功能的基础上构建的上层绘图、显控程序会很简练、易理解。

而在彩色的屏幕当中，如 256 色屏、16 位色屏之类的，往往与单色的 LCD 模块不大一样，也就是点与显存的对应关系上而已，256 色的屏中的一个点对应着显存中的一个 byte 数据，16 位色屏当中一个点对应显存中的两个 byte 数据；这样在做绘点操作时就简便得多，可以不必将原先数据读回，直接将要在该点显示的颜色数据写入对应的 byte 位置即可。

2.4.3. 利用 LCD 控制器的特性

在使用 LCD 模块进行显示的控制时，往往都会有些工程师利用 LCD 控制器本身所提供的一些特性，以期实现特殊的显示效果。比如实现纵向/横向的滚屏、坐标轴的翻转等，或者是实现不同图层的叠加显示效果等，这些都与 LCD 控制器所提供的特性有关，也就是看一下 LCD 控制器是否提供了设置这些特性功能的寄存器。

在 MzL02 模块当中，就提供了 COM 显示起始行设置的寄存器，该寄存器除了在初始化时定义显示起始行对应的显存 RAM 中的哪一行之外，还可以依此实现垂直方向的滚屏显示。而 ADC 选择（Segment 方向选择）寄存器以及 COM 口扫描方向设置寄存器可以实现坐标轴的变化，即相对用户的坐标轴零点位于左上角还是右下角，或者左下角、右上角。这些寄存器的使用最好可以自己去进行一下简单的实验，才可更深刻的理解它们的作用；一般来说

很多寄存器的初始化都会有供货商提供的键议的初始化值，在使用时初始化可以按此来即可，如想了解具体每个寄存器设置的含义最好还是通过实验的方式，这里就不作过多的描述。

3. 点阵 LCD 的驱动与显控

在适当的硬件的基础介绍之后，这里将以 MzDesign 所提供的针对 MzL02 的通用版 LCD 驱动程序为对像介绍一种 LCD 驱动程序的设计思想；将以在 LCD 上的绘点功能程序为基础，构建较为完整的驱动程序（包括显示控制）。通过下图来大概了解一下这种驱动程序的结构以及各模块之间的关系。

MzT24-1 模块的基础驱动程序架构如所示：

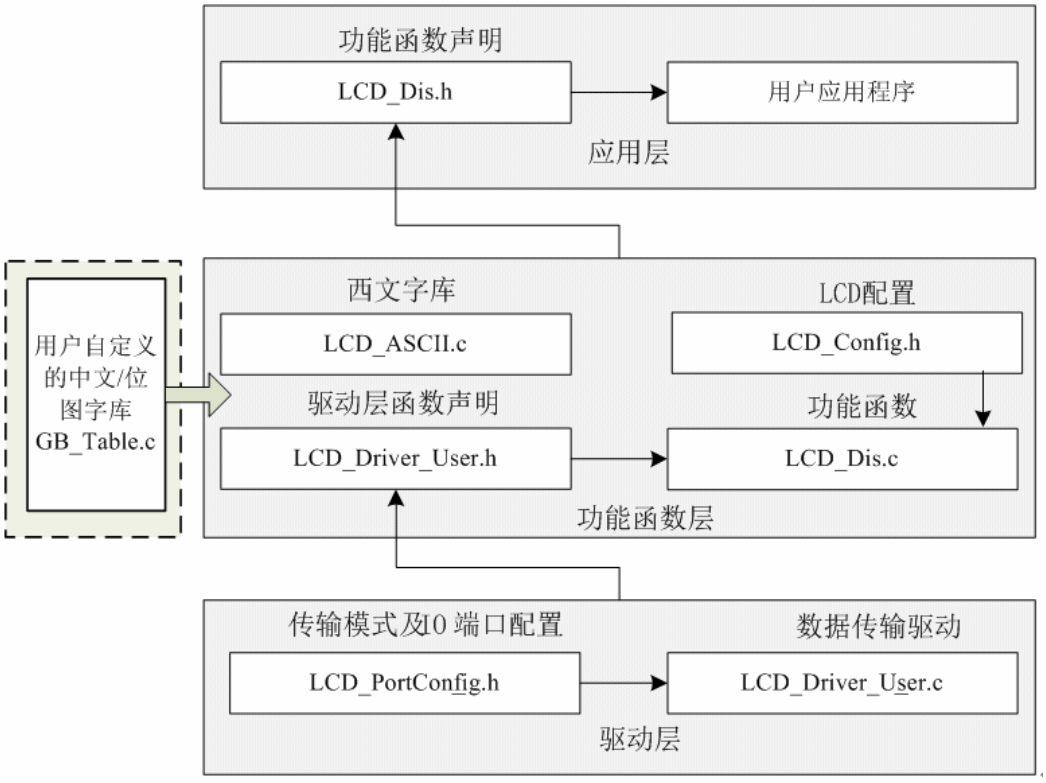


图 3.1 驱动程序架构

基础驱动程序由 8 个文件组成，分别为：底层驱动程序文件 LCD_PortConfig.h、LCD_Driver_User.c、用户 API 功能接口函数文件 LCD_Driver_User.h、LCD_Config.h、LCD_ASCII.c、LCD_Dis.c 以及 GB_Table.c、LCD_Dis.h。

LCD_PortConfig.h：该文件为底层驱动程序的头文件，主要对使用到的端口进行定义以及配置，用户在使用基础驱动程序时，要使端口的分配符合实际硬件的接线。

LCD_Driver_User.c：该文件为底层驱动程序，负责 MCU 与 MzT24-1 模块进行数据传输的任务，主要包括初始化模块、写控制指令、写数据、读数据等函数；这些函数仅供给上一层的 LCD_Dis.c 调用，不建议用户在应用程序中调用这些函数。

LCD_Driver_User.h：该文件为 LCD_Driver_User.c 的对应头文件，里面对应 C 源文件当中的函数进行外部声明。

LCD_Config.h：此文件为 MzT24-1 模块的功能配置文件，实际上此基础驱动程序是通用于各种不同的 LCD 模块的，如果需要将驱动程序应用于其它的 LCD 模块时，可以在该文件里面修改相关的配置；另外，功能配置文件里面提供了软件的坐标轴翻转功能配置（有些版本的代码中将此功能屏蔽了，仅为了让 MCU 对 LCD 的操作更快些）。

LCD_Dis.c：该文件中提供了供用户应用程序调用的 MzT24-1 驱动 API 功能函数，如绘点、绘线、绘矩形、绘圆等绘图函数，以及写字符、字符串等功能。

LCD_ASCII.c：为基础驱动程序的自带西文字库的数据文件。

GB_Table.c：为基础驱动程序的汉字/位图字库的数据文件，为用户自定义使用，另外如果定义了一些有别于基础程序所提供的范例汉定规格的字库，则用户需要自行修改 LCD_Dis.c 当中的 FontSet 函数。

LCD_Dis.h：该文件为 LCD_Driver_User.c 的对应头文件，里面对应 C 源文件当中的函数进行外部声明。

3.1. 基本驱动程序(LCD_Driver_User)

3.1.1. 端口配置头文件 LCD_Portconfig

以 MCS51 单片机版的通用 LCD 驱动程序为例，看看基本驱动程序里是什么东西。
首先，LCD_PortCongif.h 文件里定义了驱动程序当中所占用 MCU 对 LCD 驱动控制的端口，这里是对这些端口进行了重定义，如下：

```
// this file for MCU I/O port or the orther`s hardware config
// for LCD Display
#include "REG51.h"
#include "intrins.h"            //包含此头文件可直接操作内核的寄存器以及一些定义好的宏

#define DAT_PORT      P0

sbit LCD_EP=P2^3;
sbit LCD_RW = P2^4;
sbit LCD_A0=P2^5;
sbit LCD_CS = P2^7;

sbit LCD_RS=P2^6;           //如果您认为有必要在程序里控制 LCD 的复位，就定义它吧~呵呵
```

在上面的程序代码当中，对 P0 端口进行了重定义，即：”#define DAT_PORT　P0”，这样，在其它的代码里面都将使用 DAT_PORT 进行操作 P0 端口；实际上是定义了与 MzL02 的 8

位数据端口连接的 P0 口，如果在其它的应用当中，不再使用 P0 口作数据口的话，如改用了 P1 口之类的，就可以在 LCD_PortConfig.h 当中修改这个重定义就可以了。
其它的端口定义也是类似的意思，LCD_EP 定义到了 P2.3 的端口，即接 MzL02 的 EP 口；LCD_RW 定义为 P2.4 口，接 RW 读写控制；LCD_A0 定义到 P2.5，接 A0 数据命令选择端口；LCD_CS 定义到 P2.7 口，接片选；LCD_RS 定义为 P2.6 口，接 LCD 模块的外部复位端口。

3.1.2. MCU 与 LCD 基本时序控制程序

一般来说，LCD 模块的控制都是通过 MCU 对 LCD 模块的内部寄存器、显存进行操作来最终完成的；在此我们设计了三个基本的时序控制程序，分别是：

- 写寄存器函数（LCD_RegWrite）
- 数据写函数（LCD_DataWrite）
- 数据读函数（LCD_DataRead）

这三个函数需要严格的按照LCD所要求的时序来编写，下面可以看看MzL02模块的时序图：

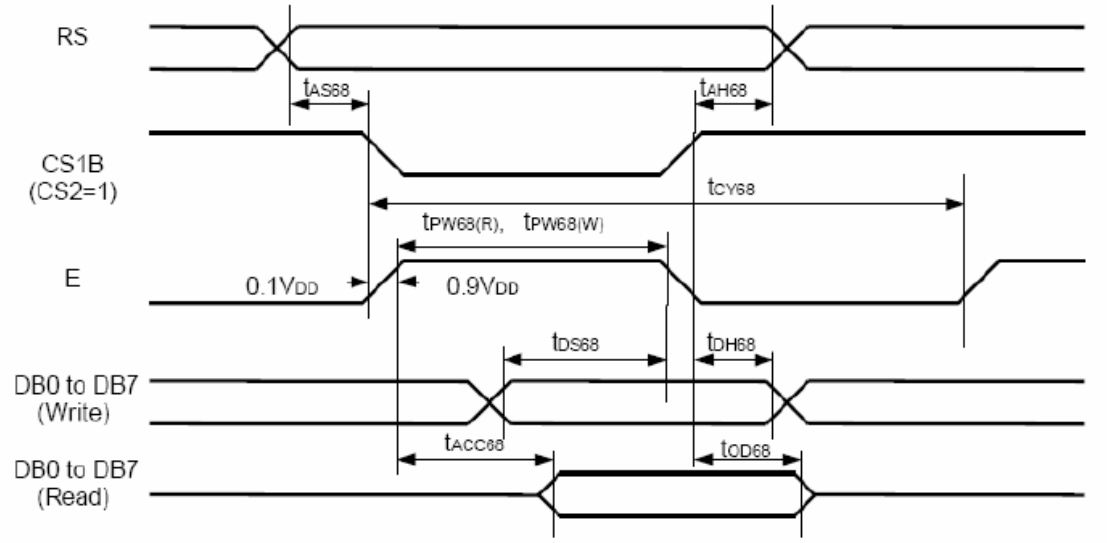


图 3.2 MzL02 模块的 6800 时序示意

注意：上图是该模块的控制 IC 资料中的原版时序图，其实有些示意不是太稳妥（少标出了 RW 线信号的要求），或者说是不太严谨，不过这些不作讨论，请看分析即可；而 EP 的有效触发沿在图中很有可能示意有误，实测为上升沿。

图中 CS1B（CS2）的信号即为片选 CS，RS 即为数据/寄存器的选择端口 A0 信号，E 为 EP；当作写入寄存器数据操作时，首先要将 A0 置低，以通知 LCD 模块即将进行的是对寄存器的操作；而 RW 线需要置低，以示即将要进行的是写入的操作；然后片选 CS 信号置低，装载数据至总线，然后在 EP 线上产生一个上升沿以触发 LCD 模块将总线上的数据最终载入；在前面的操作完成后一般都会将各个信号线的状态恢复。而数据（显存）写入、数据读出的操作时序也比较类似，这里就不多作介绍，直接参考例程即可。

```
//=========================================================================
// 函数: void LCD_RegWrite(unsigned char Command)
// 描述: 写一个字节的数据至 LCD 中的控制寄存器当中
```

```
// 参数: Command        写入的数据，低八位有效（byte）
// 返回: 无
// 版本:
//        2007/01/09        First version
//=============================================================================
void LCD_RegWrite(unsigned char Command)
{
        LCD_A0 = 0;                     //A0 置低，示意进行寄存器操作
        LCD_RW = 0;                     //RW 置低，示意进行写入操作
        LCD_EP = 0;                     //EP 先置低，以便后面产生跳变沿
        LCD_CS = 0;                     //片选 CS 置低
        DAT_PORT = Command;             //装载数据置总线

        LCD_EP = 1;                     //产生有效的跳变沿
        LCD_CS = 1;                     //片选置高
}
```

数据写入以及读出的函数源码如下：

```
//=============================================================================
// 函数: void LCD_DataWrite(unsigned char Dat)
// 描述: 写一个字节的显示数据至 LCD 中的显示缓冲 RAM 当中
// 参数: Data 写入的数据
// 返回: 无
// 版本:
//        2007/01/09        First version
//=============================================================================
void LCD_DataWrite(unsigned char Dat)
{
        LCD_A0 = 1;                     //A0 置高，示意进行显存数据操作
        LCD_RW = 0;                     //RW 置低，示意进行写入操作
        LCD_EP = 0;                     //EP 先置低，以便后面产生跳变沿
        LCD_CS = 0;                     //片选 CS 置低
        DAT_PORT = Dat;                 //装载数据置总线

        LCD_EP = 1;                     //产生有效的跳变沿
        LCD_CS = 1;                     //片选置高
}
//=============================================================================
// 函数: unsigned char LCD_DataRead(void)
// 描述: 从 LCD 中的显示缓冲 RAM 当中读一个字节的显示数据
// 参数: 无
// 返回: 读出的数据，
```

```
// 版本:
//        2007/01/09        First version
//============================================================================
unsigned char LCD_DataRead(void)
{
        unsigned char Read_Data;
        DAT_PORT = 0xff;                    //51 的端口想要输入前，要先给端口全置 1
        LCD_A0 = 1;                         //A0 置高，示意进行显存数据操作
        LCD_RW = 1;                         //RW 置高，示意进行读出操作
        LCD_EP = 0;                         //EP 先置低，以便后面产生跳变沿
        LCD_CS = 0;                         //片选 CS 置低
        LCD_EP = 1;                         //产生有效的跳变沿
        LCD_EP = 0;
        Read_Data = DAT_PORT;               //读出数据
        LCD_CS = 1;                          //片选置高
        return Read_Data;                   //返回读到的数据
}
```

以上便是要介绍的最基本的时序操作程序，它们几乎是整个 LCD 驱动程序当中与底层硬件打交道的代码了，这样的话，当要改变驱动 LCD 的 MCU 端口时或者换用别的 MCU 来驱动 LCD 时，基本上只需要在这些代码里作一下修改即可。

关于读 LCD 状态

而在一般的 LCD 模块当中，还有一个功能同样重要，就是读 LCD 状态；可以通过此操作获取当前 LCD 模块的忙状态以及一些相关的状态信息，当 LCD 模块正处于忙状态时，则不宜对它进行数据的写入或读出操作（有很多较老式的 LCD 控制器规定在忙的状态下时不允许写入或读出数据）。

所以在很多 LCD 的驱动程序当中，会在寄存器写入、数据写入/读出的操作前加入读取 LCD 状态并判别忙状态的代码；这点可以参考网上流传的很多 LCD 驱动程序。不过，对于 MzL02 这样的较新出的 LCD 控制器来说，已经对忙状态不是很在乎了，或者说影响已经很小甚至没有了；所以我们在前面的代码当中并没有加入这样的代码。

至于有没有必要加读状态判忙的代码，要视具体的 LCD 控制器而定。

关于时序的时间要求

时序的一个非常重要的数据就是类似上图中标出的 t_{AS88} 之类的时间长短要求，只是上图中并没有标出它们的具体最大最小值要求而已；但在编写这类的时序接口程序时它们还是非常重要的，当然还要看 MCU 的端口操作速度以及 MCU 的指令执行速度。打个比方，有的时序里就会有要求某些信号的电平保持最小宽度，而如果 MCU 的指令执行速度以及端口操作速度非常快的话，就需要酌情在连续操作端口的代码之间加入适量的延时（通用用空操作来代替，具体多少个多少时长视具体的 MCU 以及 LCD 控制器而定）以保证该信号的脉冲宽度满足要求。

在本文的所列出的源代码当中，并没有如前所述的为时序的要求而插入空操作或延时处理，因为 MCU 的速度并不是非常快，况且现在的 LCD 控制器的总线速度都挺快的了，没有必

要加入而已。

3.2. LCD 的初始化

LCD 模块的初始化主要就是对 LCD 模块的寄存器进行初始化，也就是对 LCD 控制器当中的寄存器写入要预设的数据，设置好 LCD 的特性。MzL02 模块的初始化如下：

```
//=============================================================================
// 函数: void LCD_Init(void)
// 描述: LCD 初始化程序，在里面会完成 LCD 初始所需要设置的许多寄存器，具体如果
//          用户想了解，建议查看 DataSheet 当中各个寄存器的意义
// 参数: 无
// 返回: 无
// 版本:
//        2006/10/15        First version
//         2007/01/09            V1.2
//=============================================================================
//延时程序
void TimeDelay(int Time)
{
        int i;
        if(Time > 0)
        {
            for(i = 0;i < 800;i++)
            {
            }
            Time --;
        }
}
void LCD_Init(void)
{
        //LCD 驱动所使用到的端口的初始化（如果有必要的话）
    //  LCD_PortInit();
        LCD_RS = 0;
        TimeDelay(200);
        LCD_RS = 1;

        LCD_RegWrite(0xaf);                                    //LCD On
        LCD_RegWrite(0x2f);                                    //设置上电控制模式

        LCD_RegWrite(0x81);                                    //电量设置模式（显示亮度）
        LCD_RegWrite(0x1f);                                    //指令数据 0x0000~0x003f
```

```
        LCD_RegWrite(0x27);                    //V5 内部电压调节电阻设置
        LCD_RegWrite(0xa2);                    //LCD 偏压设置

        LCD_RegWrite(0xc8);                    //Com 扫描方式设置,反向
        LCD_RegWrite(0xa0);                    //Segment 方向选择,正常
        LCD_RegWrite(0xa4);                    //全屏点亮/变暗指令
        LCD_RegWrite(0xa6);                    //正向反向显示控制指令

        LCD_RegWrite(0xac);                    //关闭静态指示器
        LCD_RegWrite(0x00);                    //指令数据

        LCD_RegWrite(0x40 +32);                //设置显示起始行对应 RAM
        LCD_RegWrite(0xe0);                     //设置读写改模式
}
```

在前面的代码当中，可以看到有一行屏蔽掉的代码：LCD_PortInit()；如注释所言，像有的 MCU 的端口在使用前是需要初始化的，这时就需要在 LCD 初始化之前完成对端口的初始化，而 MCS51 的端口是双向口，无需初始其的方向；所以在此将该函数屏蔽。
在对寄存器进行写入操作前，一般会控制端口对 LCD 模块进行一次外部的硬件复位，即给 LCD 模块的复位端口一个低电平的脉冲，这样可以确保 LCD 完成正确的复位。当然，如果在硬件上可以保证 MCU 在对 LCD 进行操作前 LCD 模块已经完全完成了复位的话，也可以不需要这样的代码。此外，从合理性来说，对 LCD 的复位操作最好是放置在 LCD_PortInit 函数当中，因为放在上面的代码当中的那个位置的话，当需要移植驱动程序置另外的 MCU 时，这块的代码也可能会作修改，相对麻烦一些而已。

一般来说，LCD 模块的初始化代码都会由厂商提供参考的，因为有些设置是与集成 LCD 模块时的配置有关，如不告知用户推荐设置的话，往往会让使用者浪费时间在摸索合适的配置上。

在初始化的代码当中，可以看到都是通过之前写好的寄存器写入子函数来对 LCD 进行操作，当需要换用另外的 MCU 来驱 LCD 模块时，是不需要修改这部分的代码的。

如想了解 LCD 屏的一些寄存器的设置的作用，最好的方法是作一些小实验修改一下设置看一下现像。

3.3. 绘点子程序

3.3.1. 基本绘点函数

前面我们已经提到了，MzDesign 所提供的通用版 LCD 驱动程序是基于绘点功能的，下面将介绍这个基本的绘点子程序，源代码如下：

```
//=============================================================================
// 函数: void Write_Dot_LCD(unsigned char x,unsigned char y,unsigned char i)
// 描述: 在 LCD 的真实坐标系上的 X、Y 点绘制填充色为 i 的点
// 参数: x          X 轴坐标
//       y          Y 轴坐标
//       i          要填充的点的颜色
// 返回: 无
// 版本:
//       2006/10/15       First version
//       2007/01/09       V1.2
//=============================================================================
void Write_Dot_LCD(unsigned char x,unsigned char y,unsigned char i)
{
    unsigned char x_low,x_hight;                    //定义列地址的高低位指令
    unsigned char Dot_Mask_Buf=0x01;
    unsigned char y_Page;                           //用于存放要画点的位置所在的 byte 数据位置
    x = x+1;
    x_low = (x&0x0f);                               //定位列地址设置的低位指令
    x_hight = ((x>>4)&0x0f)+0x10;                   //定位列地址设置的高位指令
    switch(y&0x07)
    {
        case 0: Dot_Mask_Buf = 0x01;break;
        case 1: Dot_Mask_Buf = 0x02;break;
        case 2: Dot_Mask_Buf = 0x04;break;
        case 3: Dot_Mask_Buf = 0x08;break;
        case 4: Dot_Mask_Buf = 0x10;break;
        case 5: Dot_Mask_Buf = 0x20;break;
        case 6: Dot_Mask_Buf = 0x40;break;
        case 7: Dot_Mask_Buf = 0x80;break;
    }
    y_Page = (y>>3)+0xb0;                           //Get the page of the byte
    LCD_RegWrite(y_Page);
    LCD_RegWrite(x_low);
    LCD_RegWrite(x_hight);
    y_Page = LCD_DataRead();                        //
    if(i) y_Page |= Dot_Mask_Buf;
    else y_Page &= ~Dot_Mask_Buf;
    LCD_DataWrite(y_Page);                          //
}
```

在第二章，已经介绍过了在 LCD 屏上绘点的程序设计思想了，而从上面的代码当中，更具体的展现了这一基本的程序设计思路。

在该函数执行时，首先会对 x 轴坐标进行加 1 的操作，实际上是因为 LCD 的控制 IC 当中显存的范围为 132*65，而 LCD 玻璃屏上只有 128*64 点；所以会有那么 4 列显存在 LCD 屏上没有对应的点（在上一章已有介绍），而具体哪些列不对在 LCD 屏上则由 LCD 玻璃的封装厂在配置时决定的，通常都是开头的或者是最后的几列。笔者也是通过实验测出 MzL02 模块当中当，Segment 采用正向扫描时第 0 列和最后的 3 列不对应对 LCD 玻璃之上；所以在这里为了使坐标轴归零，才在这个函数当中对 x 轴坐标加 1 操作，这样使用上层的程序看起来，LCD 的显示依然从第 0 列开始对应最左边的一列。
而 x 轴的坐标需要分两个寄存器设置，所以在代码当中对处理过的 x 轴坐标进行高低位数据的分离。

Y 轴的坐标在传递入该函数时是以 0~63 的点范围传递过来的，而在上一章中已有说明，MzL02 模块的 Y 轴实际上是以 page 形式存在的，每个 page 有 8 个点，分别对应一个 byte 的 8 个位；所以也对 y 值进行处理，提取出该点所对应的 page 值以及该点对应的 byte 当中的具体位的位置。

处理完 x 和 y 轴的坐标值后，代码当中调用寄存器写入函数来完成显存数据指针的指向设置操作，以表明即将进行的读写操作是针对于显存中的哪一个数据进行操作。
在读-改-写的操作流程中，可以看到代码里依据 i 的值来对读回的数据进行操作，如下列的片断代码：

```
        ......
            y_Page = LCD_DataRead();                //
            if(i) y_Page |= Dot_Mask_Buf;
            else y_Page &= ~Dot_Mask_Buf;
            LCD_DataWrite(y_Page);                  //
        ......
```

当 i 为 1 时则将要绘点的位进行置位操作，即画出黑点；当 i 为零时，则对其进行清零操作，即清除该点；操作完成之后再将数据写入显存当中，这些操作是不会影响屏上其它点状态的。

以上便是 MzL02 模块的绘点函数，其具体的操作流程是跟 LCD 模块的特性相关的，即 LCD 屏的操作方法相关；所以在移植通用版 LCD 驱动程序至别外一块 LCD 屏时，这个函数是需要进行一定的修改的。

3.3.2. 一些扩展的基础功能函数

除了基本的绘点函数之外，在这版通用版的 LCD 驱动程序当中还编写了一个全屏填充的函数，如下：

```
//=============================================================================
// 函数: void LCD_Fill(unsigned char Data)
// 描述: 会屏填充以 Data 的数据至各点中
// 参数: Data    要填充的颜色数据
// 返回: 无
// 版本:
//        2006/10/15        First version
```

```
//          2007/01/09          V1.2
//=============================================================================
void LCD_Fill(unsigned char Data)
{
        unsigned char i,j;
        unsigned char uiTemp;
        uiTemp = Dis_Y_MAX;
        uiTemp = uiTemp>>3;
        for(i=0;i<=uiTemp;i++)                          //往 LCD 中填充初始化的显示数据
        {
            LCD_RegWrite(0xb0+i);
            LCD_RegWrite(0x01);
            LCD_RegWrite(0x10);
            for(j=0;j<=Dis_X_MAX;j++)
            {
                LCD_DataWrite(Data);
            }
        }
}
```

这个函数用于清屏或者是全屏填充。

3.4. 驱动配置头文件 LCD_Config

在驱动程序当中，还配备了一个头文件：LCD_Config.h；在该文件里对 LCD 模块的点阵数量，以及坐标轴方向作了软件的定义，另外还定义了一些在 LCD 初始化时的 LCD 寄存器命令。代码如下：

```
#define LCD_X_MAX           128-1           //屏幕的 X 轴的物理宽度
#define LCD_Y_MAX           64-1            //屏幕的 Y 轴的物理宽度

#define LCD_XY_Switch       0               //显示时 X 轴和 Y 由交换
#define LCD_X_Rev           0               //显示时 X 轴反转
#define LCD_Y_Rev           0               //显示时 Y 轴反转

#if LCD_XY_Switch == 0
    #define Dis_X_MAX       LCD_X_MAX
    #define Dis_Y_MAX       LCD_Y_MAX
#endif

#if LCD_XY_Switch == 1
    #define Dis_X_MAX       LCD_Y_MAX
    #define Dis_Y_MAX       LCD_X_MAX
```

```
#endif

#define LCD_INITIAL_COLOR      0x00            //定义 LCD 屏初始化时的背景色

//以下定义为针对于 SPLC501 的功能指令进行定义的，局部可修改~
//LCD 供电电平选择
#define       M_LCD_VDD_SET  M_LCD_SETR_4    //3.3V 供电时选此二项
#define       M_LCD_VDD      M_LCD_BIAS_9    //....
//#define     M_LCD_VDD_SET  M_LCD_SETR_4    //5.0V 供电时选此二项
//#define     M_LCD_VDD      M_LCD_BIAS_9    //...
//LCD 指令
//LCD 开关命令
#define       M_LCD_ON       0xaf
#define       M_LCD_OFF      0xae
//设置上电控制模式
#define       M_LCD_POWER_BC     0x2c
#define       M_LCD_POWER_VR     0x2a
#define       M_LCD_POWER_VC     0x29
#define       M_LCD_POWER_ALL    0x2f
//V5 内部电压调节电阻设置……
#define       M_LCD_SETR_0   0x20
#define       M_LCD_SETR_1   0x21
#define       M_LCD_SETR_2   0x22
#define       M_LCD_SETR_3   0x23
#define       M_LCD_SETR_4   0x24
#define       M_LCD_SETR_5   0x25
#define       M_LCD_SETR_6   0x26
#define       M_LCD_SETR_7   0x27
//...end
#define       M_LCD_ELE_VOL  0x81      //电量设置模式（显示亮度）
//偏压设置
#define       M_LCD_BIAS_9   0xa2      //V5 时选此选项设置
#define       M_LCD_BIAS_7   0xa1      //V3 时选此选项设置
//Com  扫描方式设置命令
#define       M_LCD_COM_NOR 0xc0       //正常方式
#define       M_LCD_COM_REV  0xc8      //反相
//Segment 方向选择
#define       M_LCD_SEG_NOR  0xa0      //正常
#define       M_LCD_SEG_REV  0xa1      //反向
//全屏点亮/变暗指令
#define       M_LCD_ALL_LIGNT0xa5      //LCD ALL paint ON
```

```
#define     M_LCD_ALL_LOW   0xa4      //Normal Display mode
//正相反相显示控制指令，RAM 中数据不变
#define     M_LCD_ALL_NOR   0xa6      //正相
#define     M_LCD_ALL_REV   0xa7      //反相
//静态指示器控制指令
#define     M_LCD_STATIC_ON     0xad      //ON
#define     M_LCD_STATIC_OFF 0xac         //OFF
//设置显示起始行对应 RAM 行号
#define     M_LCD_BEGIN_LINE 0x40         //基数，后面可加的尾数可为 0~63
//设置当前页基数
#define     M_LCD_COL_PAGE0xb0            //基数指令，后可加尾数 0~8
//设置当前列基数
#define     M_LCD_COL_LINE_LOW0x04        //基数指令，低四位有效
#define     M_LCD_COL_LINE_HIG  0x10      //基数指令，低四位有效
```

配置文件中对 LCD 的物理尺寸（点数）进行了定义，并在随后定义了软件上进行坐标变换以及坐标轴反转；在 LCD 驱动的功能接口程序当中，将会引用这些定义，来限制对 LCD 的显控操作，以防止操作超出 LCD 的显示范围；不过，在 MzL02 的通用版 LCD 驱动程序当中，并没有开放坐标轴变换的程序，也就是坐标轴变换的代码被屏蔽掉了，所以在此有关坐标轴变换（包括反转）的定义是没有实际意义的，但用户不要去更改它的定义。

在完整版的驱动程序的功能程序当中，程序将会从 Dis_X_MAX 和 Dis_Y_MAX 获得定义的 LCD 屏物理点数，所以 LCD_XY_Switch 的定义将决定 X 轴和 Y 轴的坐标是否对调了，当然在代码当中还会有一些代码根据 LCD_XY_Switch 的定义而有选择性的进行编译，以配合坐标轴的变换。而类似的定义还有 LCD_X_Rev 和 LCD_Y_Rev，都在功能程序当中有相应的宏定义选择对应的代码进行编译。只不过在 MzL02 的驱动程序当中，作了一定的精简，把这部分的功能删减掉了。

再接下来的定义中，定义了 LCD 屏初始化时的背景色，以及 LCD 控制器的寄存器设置命令（实际上就是命令与数据的结合），这些定义在 LCD 的初始化代码当中使用（前面列出的 LCD 初始化代码是作了修改的，只是把宏定义换为数字表示而已），下面可以再看看实际程序中的初始化代码：

```
void LCD_Init(void)
{
        //LCD 驱动所使用到的端口的初始化（如果有必要的话）
    //  LCD_PortInit();
        LCD_RS = 0;
        TimeDelay(200);
        LCD_RS = 1;
        TimeDelay(20);

        LCD_RegWrite(M_LCD_ON);                 //LCD On
        LCD_RegWrite(M_LCD_POWER_ALL);          //设置上电控制模式
```

```
        LCD_RegWrite(M_LCD_ELE_VOL);            //电量设置模式（显示亮度）
        LCD_RegWrite(0x1f);                     //指令数据 0x0000~0x003f

        LCD_RegWrite(M_LCD_VDD_SET);            //V5 内部电压调节电阻设置
        LCD_RegWrite(M_LCD_VDD);                //LCD 偏压设置，V3 时选

        LCD_RegWrite(M_LCD_COM_REV);            //Com 扫描方式设置
        LCD_RegWrite(M_LCD_SEG_NOR);            //Segment 方向选择
        LCD_RegWrite(M_LCD_ALL_LOW);            //全屏点亮/变暗指令
        LCD_RegWrite(M_LCD_ALL_NOR);            //正向反向显示控制指令

        LCD_RegWrite(M_LCD_STATIC_OFF);         //关闭静态指示器
        LCD_RegWrite(0x00);                     //指令数据

        LCD_RegWrite(M_LCD_BEGIN_LINE+32);      //设置显示起始行对应 RAM
        LCD_Fill(LCD_INITIAL_COLOR);
    }
```

3.5. LCD 驱动功能接口程序(LCD_Dis)

LCD 驱动功能接口程序主要包括两大类：

- 基本绘图功能函数：绘点、直线、矩形、矩形框、圆形、圆框等；
- 字符显示功能函数：西文字符显示、西文字符串显示、中文字符显示；

这些功能接口程序都是基于前面介绍的绘点函数的，其实，这样的做法并不是很高效，可以说是利用软件而牺牲了硬件的速度；但主要考虑到普通的人机界面程序要求的并不是非常快，只要在人眼能反映过来的时间里完成显示的刷新就可以了，而且主要考虑了移植性、可读性，才选用了这样的程序架构。

一般 LCD 模块都会提供了这样或者那样的特性，以便于 MCU 对其的显示控制，但通常都是每种 LCD 模块都会有自己的一套特殊功能；所以想要编写最高效的 LCD 驱动程序的，就不心参考本书所介绍的架构了，直接根据每个 LCD 屏的特性来编写；不过现在常用的 MCU 性能已经不同于原始的 MCS51 了，利用一下软件牺牲一下硬件也何偿不可。

3.5.1. 基本绘图功能函数

在我们提供的MzL02的通用版LCD驱动程序当中，基本绘图功能函数一共提供了如下几个：

绘点：PutPixel

绘直线：Line

绘矩形（框）：Rectangle

而鉴于圆形（框）绘制的功能极少用到，将这块的代码裁减掉了，感兴趣的朋友可以参考 MzDesign 为其它的 LCD 模块所提供的驱动程序。

绘点程序的代码如下：

```
//=============================================================================
// 函数: void PutPixel(int x,int y)
// 描述: 在 x、y 点上绘制一个前景色的点
// 参数: x   X 轴坐标        y   Y 轴坐标
// 返回: 无
// 备注: 使用前景色
// 版本:
//        2006/10/15        First version
//=============================================================================
void PutPixel(unsigned char x,unsigned char y)
{
          Write_Dot_LCD/*Writ_Dot*/(x,y,BMP_Color);
}
```

绘点程序非常简单，就是直接调用了在前些节中介绍的绘点函数；这里只不过重新作一下包装，统一下一函数接口。

不过，在代码中可以看到有一个屏蔽掉的函数：Writ_Dot，它就是我们前面所说的实现坐标变换功能的软绘点子程序，这个函数中与 LCD 驱动配置头文件里的坐标轴相关的定义有在联，实现了在软件上的坐标轴变换处理。只是这版的驱动当中删减了这项功能，将其屏蔽而已。它的源代码如下，仅供参考：

```
//=============================================================================
// 函数: void Writ_Dot(int x,int y,unsigned int Color)
// 描述: 填充以 x,y 为坐标的象素
// 参数: x   X 轴坐标        y   Y 轴坐标        Color   像素颜色
// 返回: 无
// 备注: 这里以及之前的所有 x 和 y 坐标系都是用户层的，并不是实际 LCD 的坐标体系
//          本函数提供可进行坐标变换的接口
// 版本:
//        2006/10/15        First version
//=============================================================================
void Writ_Dot(int x,int y,unsigned int Color)
{
#if  LCD_XY_Switch == 0
          #if (LCD_X_Rev == 0)&&(LCD_Y_Rev == 0)
               Write_Dot_LCD(x,y,Color);
          #endif
          #if (LCD_X_Rev == 1)&&(LCD_Y_Rev == 0)
               Write_Dot_LCD(LCD_X_MAX - x,y,Color);
          #endif
          #if (LCD_X_Rev == 0)&&(LCD_Y_Rev == 1)
               Write_Dot_LCD(x,LCD_Y_MAX - y,Color);
```

```
        #endif
        #if (LCD_X_Rev == 1)&&(LCD_Y_Rev == 1)
            Write_Dot_LCD(LCD_X_MAX - x,LCD_Y_MAX - y,Color);
        #endif
    #endif
    #if  LCD_XY_Switch == 1
        #if (LCD_X_Rev == 0)&&(LCD_Y_Rev == 0)
            Write_Dot_LCD(y,x,Color);
        #endif
        #if (LCD_X_Rev == 1)&&(LCD_Y_Rev == 0)
            Write_Dot_LCD(y,LCD_Y_MAX - x,Color);
        #endif
        #if (LCD_X_Rev == 0)&&(LCD_Y_Rev == 1)
            Write_Dot_LCD(LCD_X_MAX - y,x,Color);
        #endif
        #if (LCD_X_Rev == 1)&&(LCD_Y_Rev == 1)
            Write_Dot_LCD(LCD_X_MAX - y,LCD_Y_MAX - x,Color);
        #endif
    #endif
    }
```

绘直线程序是基于绘点函数的，依据的是直线起始坐标位置算出的斜率，然后从起点开始一个点一个点的绘制。源码如下：

```
    //=============================================================================
    // 函数: void Line(unsigned char s_x,unsigned char s_y,unsigned char e_x,unsigned char e_y)
    // 描述: 在 s_x、s_y 为起始坐标，e_x、e_y 为结束坐标绘制一条直线
    // 参数: x   X 轴坐标        y   Y 轴坐标
    // 返回: 无
    // 备注: 使用前景色
    // 版本:
    //        2006/10/15          First version
    //=============================================================================
    void Line(unsigned char s_x,unsigned char s_y,unsigned char e_x,unsigned char e_y)
    {
        char Offset_x,Offset_y,Offset_k = 0;
        char Err_d = 1;
        if(s_y>e_y)
        {
            Offset_x = s_x;
            s_x = e_x;
            e_x = Offset_x;
```

```
        Offset_x = s_y;
        s_y = e_y;
        e_y = Offset_x;
    }
    Offset_x = e_x-s_x;
    Offset_y = e_y-s_y;
    Write_Dot_LCD/*Writ_Dot*/(s_x,s_y,BMP_Color);
    if(Offset_x<=0)
    {
        Offset_x = s_x-e_x;
        Err_d = -1;
    }
    if(Offset_x>Offset_y)
    {
        Offset_k += (Offset_y-Offset_x);
        while(s_x!=e_x)
        {
            if(Offset_k>0)
            {
                s_y+=1;
                Offset_k += (Offset_y-Offset_x);
            }
            else Offset_k += Offset_y;
            s_x+=Err_d;
            if(s_x>Dis_X_MAX||s_y>Dis_Y_MAX) break;
            Write_Dot_LCD/*Writ_Dot*/(s_x,s_y,BMP_Color);
        }
    }
    else
    {
        Offset_k += (Offset_x-Offset_y);
        while(s_y!=e_y)
        {
            if(Offset_k>=0)
            {
                s_x+=Err_d;
                Offset_k += (Offset_x-Offset_y);
            }
            else Offset_k += Offset_x;
            s_y+=1;
            if(s_x>=Dis_X_MAX||s_y>=Dis_Y_MAX) break;
            Write_Dot_LCD/*Writ_Dot*/(s_x,s_y,BMP_Color);
        }
```

```
            }
        }
```

在绘直线的子程序里面，首先会对起始点和结束点的坐标值进行判断，以便统一从 x 轴由小到大的方向绘直线上的点；然后计算直线的起始点和结束点相对的偏移值，并绘制起始点。假设直线的相对斜率 K=offset_x/offset_y，同在后续的代码当中会判断 K 值是否大于 1，并据此选择以 X 轴还是以 Y 轴为基准绘制直线；在绘制直线的过程中，实际上就是从起始点开始一个点一个点的绘制，每个绘制的点的位置都是紧随着上一个绘制的点的位置依据 K 值的计算而得出的，程序中使用的是简化的算法，如果不太理解的话，建议自行将假设的起始点和结束点的值代入程序中，心算一下。

在上面的代码当中，利用 LCD 驱动配置头文件中的定义 Dis_X_MAX 和 Dis_Y_MAX 限制 X 和 Y 轴的范围，以防止绘直线过程中直线超出 LCD 的显示范围。

矩形绘制程序则是基于绘制直线函数的，代码如下：

```
//=============================================================================
// 函数: void Rectangle(unsigned char left, unsigned char top, unsigned char right,
//                                  unsigned char bottom, unsigned char Mode)
// 描述: 以 x,y 为圆心 R 为半径画一个圆(mode = 0) or 圆面(mode = 1)
// 参数: left - 矩形的左上角横坐标，范围 0 到 127
//          top - 矩形的左上角纵坐标，范围 0 到 63
//          right - 矩形的右下角横坐标，范围 1 到 127
//          bottom - 矩形的右下角纵坐标，范围 1 到 63
//          Mode - 绘制模式，可以是下列数值之一：
//                    0:   矩形框（空心矩形）
//                    1:   矩形面（实心矩形）
// 返回: 无
// 备注: 使用前景色
// 版本:
//       2007/01/21        First version
//=============================================================================
    void Rectangle(unsigned char left, unsigned char top, unsigned char right, unsigned char bottom,
unsigned char Mode)
    {
            unsigned char uiTemp;

            if(Mode==0)
            {
                Line(left,top,left,bottom);
                Line(left,top,right,top);
                Line(right,bottom,left+1,bottom);
                Line(right,bottom,right,top+1);
            }
```

```
        else
        {
            if(left>right)
            {
                uiTemp = left;
                left = right;
                right = uiTemp;
            }
            if(top>bottom)
            {
                uiTemp = top;
                top = bottom;
                bottom = uiTemp;
            }
            for(uiTemp=top;uiTemp<=bottom;uiTemp++)
            {
                Line(left,uiTemp,right,uiTemp);
            }
        }
    }
```

绘制矩形的程序较简单，这里不作过多的分析了。

调用该程序时需要传递进来矩形的左上角的坐标值以及右下角的坐标值，还有要绘制的类形：Mode，包括矩形框和矩形块。

绘图设置子程序

在前面介绍的三个绘图功能接口程序当中，实际上有一个全局的变量是很重要的：BMP_Color；它管控着绘图的前景色，其实对于单色的 LCD 模块无非就是黑与白的区分，只不过为了扩展通用版的驱动程序应用于彩色 LCD 模块上，所以使用了一个变量来表示，它可以通过一个接口函数来进行设置，如下：

```
    //=============================================================================
    // 函数: void SetPaintMode(int Mode,unsigned int Color)
    // 描述: 绘图模式设置
    // 参数: Mode  绘图模式      Color   像素点的颜色,相当于前景色
    // 返回: 无
    // 备注: Mode 无效
    // 版本:
    //        2006/10/15        First version
    //=============================================================================
    void SetPaintMode(unsigned char Mode,unsigned char Color)
    {
        Mode = Mode;//Plot_Mode = Mode;        //仅仅是为了保持与其它驱动的一至性，绘图模
                                               //式在该版驱动中未用
        BMP_Color = Color;
```

```
    }
```

在整个 LCD 的驱动程序当中，Mode 并没有使用到，仅仅是为了保留功能的扩展。

关于数据类型

从前面的程序可看出，使用的基本上是 unsigned char 型数据；其实是为了针对 MCS51 的系列 MCU 而修改的，仅仅是为了减少 RAM 的占用；所以这版的 LCD 通用版驱动程序会与其它的稍稍有些不太一样，希望读者理解，反正本书所介绍的并不在于某一个驱动程序，而真正的目的在于介绍一种编程的思想和方法。

3.5.2. 字符显示功能函数

MzL02 模块的通用版 LCD 驱动程序提供了 ASCII 西文字符显示以及中文字符的显示控制函数接口，在驱动程序当中直接集成了 ASCII 西文字符库，而中文字符需要用户在使用时自行利用字模提取工具按需提取。在这一节，将只简单介绍有关字符显示的三个函数，在下一节当中会从字符显示的原理出发分析点阵 LCD 的字符显示原理。

驱动程序当中，提供了三个有关字符显示控制的函数，如下：

- 字体选择设置函数：FontSet；
- 单个字符显示函数：PutChar；
- 字符串显示函数：PutString；

以上三个函数当中，字符串显示函数只能实现 ASCII 码即西文字符串的显示，对中文是无效的；而 PutChar 函数即可用于 ASCII 西文字符显示，也可用于用户定义好的中文字符显示，或者是图形显示。

字符的显示实际上还需要有字符的字模库才可以实现，在驱动程序当中配备了两个字库文件，分别是 LCD_ASCII（ASCII 码西文字符库字模集）、GB_Table（留给用户进行中文字符字模定义用的）。

有关字符显示的功能函数，将在下一节作出详细的分析。

3.6. 字符显示原理

3.6.1. 字符与字模

驱动程序当中，字符库（也就是字模的集全）的数据采用了与一般的单色点阵 LCD 的数据组成方式，即字模当中的一个位代表 LCD 显示中的一个像素点，取点方式为**从左到右，自上到下**的顺序。对于这点，驱动中自带的 ASCII 码西文字库的字模和用户可自定义的中文字库中的字模是一样的。

字模采用了以 Byte 为单位的位流结构，即当一行取点不为 8 的整数倍时，补齐数据至 8 位，无用位填零。

下面用几个字符取字模的例子来解释一下其对应的关系。

所有的字符都可以将其栅格化，化成一个点的阵列来表示，比如下图：

上图为取字模工具中输入字符’A’后显示的栅格情况，字体选择较小，整个字用 8*12 的点阵表示。其实在取这个字符的字模时，只需要取 6*12 的就够了；但如前面所述的，为了补齐 8 位的 byte 数据，才用了 8*12 的点阵规模。

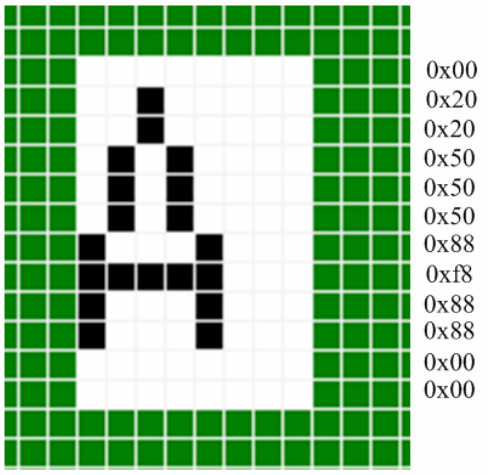
至于数据位补齐的原则，与取模的方向有关，本驱动当中的字模取向为：从左到右，自上到下；是以横向为基准的，所以要在横向满足最小存储单元的位数倍数要求，才补齐字模为 8*12。如果取字模的方向为：自上至下、从左到右（纵向优先），则补齐字模就会为 6*16 的了。

字模数据的补齐只是针对于字模将要存储在 MCU 的存储空间的，实际在显示字符时仍然可以根据字符最合适的尺寸来显示，比如在本驱动当中的 ASCII 码西文字库的字模就为 6*10 的，但占用的数据位数为 8*10。

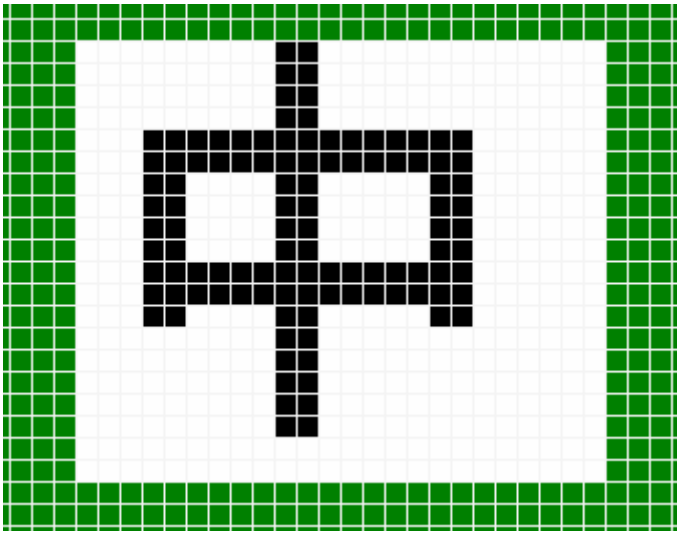
采用从左到右，自上至下的顺序取的字模数据如下所示：

```
/*--  文字:  A  --*/
/*--  MS Gothic9;  此字体下对应的点阵为：宽 x 高=6x12    --*/
/*--  宽度不是 8 的倍数，现调整为：宽度 x 高度=8x12  --*/
0x00,0x20,0x20,0x50,0x50,0x50,0x88,0xF8,0x88,0x88,0x00,0x00
```

将上面的数据配入前面的图中，其实就很简单了，图中的一行栅格点对应一个 byte 的数据，左边的点对应高位，右边的点对应低位，如下图所示：



再取一个点阵大一点的汉字字模，如下图：

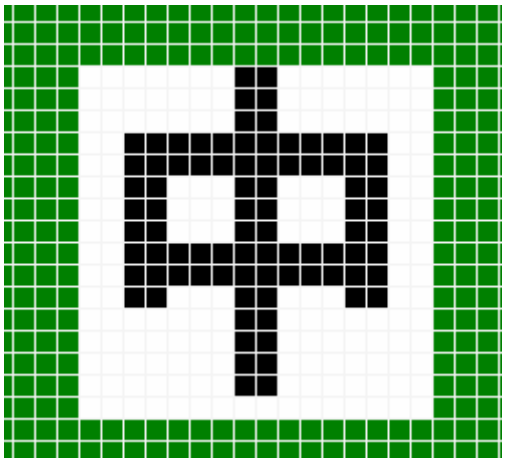


从图中，可以看出，这次取的汉字“中”字的横向点数依然不为 8 的整倍，所以也是需要进行对齐调整的；取出的字模数据如下：

```
/*--  文字:  中  --*/
/*--  黑体 15;  此字体下对应的点阵为：宽 x 高=20x20  --*/
/*--  宽度不是 8 的倍数，现调整为：宽度 x 高度=24x20  --*/
0x00,0x60,0x00,0x00,0x60,0x00,0x00,0x60,0x00,0x00,0x60,0x00,0x1F,0xFF,0xC0,0x1F,
0xFF,0xC0,0x18,0x60,0xC0,0x18,0x60,0xC0,0x18,0x60,0xC0,0x18,0x60,0xC0,0x1F,0xFF,
0xC0,0x1F,0xFF,0xC0,0x18,0x60,0xC0,0x00,0x60,0x00,0x00,0x60,0x00,0x00,0x60,0x00,
0x00,0x60,0x00,0x00,0x60,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
```

每一行的点利用 3 个 byte 的数据表示，最后不足 8 整倍数的，在后面补齐 0；所以上面的“中”字的字模数据所占用的存储单元（byte）数为 3*20，共和 60 个 byte。

我们再取一个小一点字号的“中”字字模，如下图：



利用工具所取得的数据为：

```
/*--  文字:  中  --*/
/*--  黑体 12;  此字体下对应的点阵为：宽 x 高=16x16  --*/
0x01,0x80,0x01,0x80,0x01,0x80,0x3F,0xFC,0x3F,0xFC,0x31,0x8C,0x31,0x8C,0x31,0x8C,
0x3F,0xFC,0x3F,0xFC,0x31,0x8C,0x01,0x80,0x01,0x80,0x01,0x80,0x01,0x80,0x00,0x00
```

3.6.2. 字模与字库

字库就是字模数据的集合，比如 ASCII 码西文字库就是根据 ASCII 码的编码顺序将 127 个 ASCII 码字符的字模数据组成一个数组；当然字库与字模一样也是要有规定每个字模的尺寸大小的，同一字库当中的字模都是一样的点阵大小的，这样才能在字库当中方便的检索到所需要的字符的字模数据。

比如，在 MzL02 的 MCS51 版通用 LCD 驱动程序当中，集成在驱动程序里面的 ASCII 码西文字库统一为 6*10 的点阵，而针对 byte 进行对齐后，每个字模占用的点阵位数为 8*10；此外，为了节省空间的占用，这里并没有把全部的 127 个 ASCII 码字符的字模都用上，而是把前 32 个编号的字符去掉了，只取后面的。该字库定义如下：

```
code unsigned char Asii0610[] =
{
//--  MS Gothic8;  此字体下对应的点阵为：宽 x 高=6x10    --
//--  宽度不是 8 的倍数，现调整为：宽度 x 高度=8x10   --
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x40,0x40,0x40,0x40,0x40,0x00,0x60,0x00,0x00,
0xA0,0xA0,0xA0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x28,0x28,0xFC,0x50,0xFC,0x50,0x50,0x00,0x00,
0x20,0x70,0xA8,0xA0,0x70,0x28,0xA8,0x70,0x20,0x00,
0x00,0xC8,0xD0,0xD0,0x20,0x58,0x58,0x98,0x00,0x00,0x00,0x20,0x50,0x20,0x60,0x98,0x90,0x68,0x00,0x00,
0x40,0x40,0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x10,0x20,0x40,0x40,0x40,0x40,0x40,0x20,0x10,0x00,
0x80,0x40,0x20,0x20,0x20,0x20,0x20,0x40,0x80,0x00,0x00,0x20,0xA8,0x70,0x70,0xA8,0x20,0x00,0x00,0x00,
0x00,0x00,0x20,0x20,0x70,0x20,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xC0,0x40,0x80,0x00,
0x00,0x00,0x00,0x00,0xF0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xC0,0x00,0x00,
0x10,0x10,0x20,0x20,0x40,0x40,0x80,0x80,0x00,0x00,0x00,0x60,0x90,0x90,0x90,0x90,0x90,0x60,0x00,0x00,
0x00,0x20,0x60,0x20,0x20,0x20,0x20,0x20,0x00,0x00,0x00,0x00,0x60,0x90,0x10,0x20,0x40,0x80,0xF0,0x00,0x00,
0x00,0x60,0x90,0x10,0x60,0x10,0x90,0x60,0x00,0x00,0x00,0x00,0x30,0x30,0x50,0x50,0x90,0xF8,0x10,0x00,0x00,
0x00,0xF0,0x80,0xE0,0x10,0x10,0x90,0x60,0x00,0x00,0x00,0x00,0x60,0x90,0x80,0xE0,0x90,0x90,0x60,0x00,0x00,
0x00,0xF0,0x10,0x20,0x20,0x40,0x40,0x40,0x00,0x00,0x00,0x00,0x60,0x90,0x90,0x60,0x90,0x90,0x60,0x00,0x00,
0x00,0x60,0x90,0x90,0x70,0x10,0x90,0x60,0x00,0x00,0x00,0x00,0x00,0x60,0x00,0x00,0x00,0x00,0x60,0x00,0x00,
0x00,0x00,0x60,0x00,0x00,0x00,0x60,0x20,0x40,0x00,0x00,0x00,0x10,0x20,0x40,0x80,0x40,0x20,0x10,0x00,0x00,
0x00,0x00,0x00,0xF0,0x00,0xF0,0x00,0x00,0x00,0x00,0x00,0x00,0x80,0x40,0x20,0x10,0x20,0x40,0x80,0x00,0x00,
0x00,0x60,0x90,0x10,0x20,0x20,0x00,0x20,0x20,0x00,0x00,0x00,0x70,0x88,0xB8,0xA8,0xB8,0x80,0x70,0x00,0x00,
0x00,0x60,0x60,0x90,0x90,0xF0,0x90,0x90,0x00,0x00,0x00,0x00,0xE0,0x90,0x90,0xE0,0x90,0x90,0xE0,0x00,0x00,
0x00,0x60,0x90,0x80,0x80,0x80,0x90,0x60,0x00,0x00,0x00,0x00,0xE0,0x90,0x90,0x90,0x90,0x90,0xE0,0x00,0x00,
0x00,0xF0,0x80,0x80,0xE0,0x80,0x80,0xF0,0x00,0x00,0x00,0x00,0xF0,0x80,0x80,0xE0,0x80,0x80,0x80,0x00,0x00,
0x00,0x60,0x90,0x80,0xB0,0x90,0x90,0x70,0x00,0x00,0x00,0x00,0x90,0x90,0x90,0xF0,0x90,0x90,0x90,0x00,0x00,
0x00,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0x00,0x00,0x00,0x00,0x10,0x10,0x10,0x10,0x10,0x90,0x60,0x00,0x00,
0x00,0x90,0x90,0xA0,0xC0,0xA0,0x90,0x90,0x00,0x00,0x00,0x00,0x80,0x80,0x80,0x80,0x80,0x80,0xF0,0x00,0x00,
0x00,0x88,0x88,0xD8,0xD8,0xA8,0xA8,0xA8,0x00,0x00,0x00,0x00,0x90,0x90,0xD0,0xD0,0xB0,0xB0,0x90,0x00,0x00,
0x00,0x60,0x90,0x90,0x90,0x90,0x90,0x60,0x00,0x00,0x00,0x00,0xE0,0x90,0x90,0x90,0xE0,0x80,0x80,0x00,0x00,
0x00,0x60,0x90,0x90,0x90,0xD0,0xA0,0x50,0x00,0x00,0x00,0x00,0xE0,0x90,0x90,0xE0,0x90,0x90,0x90,0x00,0x00,
0x00,0x60,0x90,0x80,0x60,0x10,0x90,0x60,0x00,0x00,0x00,0x00,0xF0,0x40,0x40,0x40,0x40,0x40,0x40,0x00,0x00,
```

```
0x00,0x90,0x90,0x90,0x90,0x90,0x90,0x60,0x00,0x00,0x00,0x88,0x88,0x88,0x50,0x50,0x20,0x20,0x00,0x00,
0x00,0xA8,0xA8,0xA8,0xA8,0x50,0x50,0x50,0x00,0x00,0x00,0x90,0x90,0x60,0x60,0x60,0x90,0x90,0x00,0x00,
0x00,0x88,0x88,0x50,0x20,0x20,0x20,0x20,0x00,0x00,0x00,0xF0,0x10,0x20,0x20,0x40,0x80,0xF0,0x00,0x00,
0x30,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x30,0x00,0x00,0x88,0x50,0x20,0xF0,0x20,0xF0,0x20,0x00,0x00,
0xC0,0x40,0x40,0x40,0x40,0x40,0x40,0x40,0xC0,0x00,0x40,0xA0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xF8,0x00,0x80,0x40,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0xE0,0x10,0x70,0x90,0x70,0x00,0x00,0x00,0x80,0x80,0xE0,0x90,0x90,0x90,0xE0,0x00,0x00,
0x00,0x00,0x00,0x60,0x90,0x80,0x90,0x60,0x00,0x00,0x00,0x10,0x10,0x70,0x90,0x90,0x90,0x70,0x00,0x00,
0x00,0x00,0x00,0x60,0x90,0xF0,0x80,0x70,0x00,0x00,0x00,0x60,0x40,0xF0,0x40,0x40,0x40,0x40,0x00,0x00,
0x00,0x00,0x00,0x50,0xA0,0xA0,0x40,0xB0,0x90,0x60,0x00,0x80,0x80,0xE0,0x90,0x90,0x90,0x90,0x00,0x00,
0x00,0x20,0x00,0x20,0x20,0x20,0x20,0x20,0x00,0x00,0x00,0x20,0x00,0x20,0x20,0x20,0x20,0x20,0xE0,0x00,
0x00,0x80,0x80,0x90,0xA0,0xE0,0x90,0x90,0x00,0x00,0x00,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x00,0x00,
0x00,0x00,0x00,0xD0,0xA8,0xA8,0xA8,0xA8,0x00,0x00,0x00,0x00,0x00,0x00,0xE0,0x90,0x90,0x90,0x90,0x00,0x00,
0x00,0x00,0x00,0x60,0x90,0x90,0x90,0x60,0x00,0x00,0x00,0x00,0x00,0x00,0xE0,0x90,0x90,0xE0,0x80,0x80,0x00,
0x00,0x00,0x00,0x70,0x90,0x90,0x70,0x10,0x10,0x00,0x00,0x00,0x00,0xA0,0xC0,0x80,0x80,0x80,0x00,0x00,
0x00,0x00,0x00,0x70,0x80,0x60,0x10,0xE0,0x00,0x00,0x00,0x40,0x40,0xF0,0x40,0x40,0x40,0x60,0x00,0x00,
0x00,0x00,0x00,0x90,0x90,0x90,0x90,0x60,0x00,0x00,0x00,0x00,0x00,0x00,0x90,0x90,0x90,0x60,0x60,0x00,0x00,
0x00,0x00,0x00,0xA8,0xA8,0xA8,0x50,0x50,0x00,0x00,0x00,0x00,0x00,0x00,0x90,0x60,0x60,0x90,0x90,0x00,0x00,
0x00,0x00,0x00,0x90,0x90,0x50,0x20,0x20,0x40,0x00,0x00,0x00,0x00,0xF0,0x20,0x40,0x80,0xF0,0x00,0x00,
0x30,0x20,0x20,0x20,0x40,0x20,0x20,0x20,0x30,0x00,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,0x20,
0xC0,0x40,0x40,0x40,0x20,0x40,0x40,0x40,0xC0,0x00,0x50,0xA0,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
};
```

字库定义为一个 unsigned char 的数组，数组的名称 Asii0610 即为该字库的首地址；所以当要检索字库中的某一个字符的字模数据时，可以利用它来找到具体的地址。

如要检索字符’A’的字模数据，则该字符的字模数据首地址等于如下代码计算的结果：

‘A’字符的首地址= Asii0610-（10*32）+’A’*10

减去 10*32 是因为该字库删减掉了前 32 个字符的字模，而每个字符的字模所占用的 byte 数为 10 个；’A’的 ASCII 码值为 0x41。

驱动程序中自带的 ASCII 码西文字库用户不需要改动，此外驱动程序中还为用户定义了一个汉字字库的数组，放置在文件 GB_Table 当中，用户需要显示中文字符或自提取的图像字模时可以将数据放置在这处，或者另起一个字库数组。

与 ASCII 码西文字库一样，放置在同一个中文字库中的字模也是需要尺寸是相同的才可；而需要索引这些字模时，用户需要知道自己在定义这个它库时每个汉字（或者图像）字模在数组中的偏移位置。

如下面的定义，即为驱动程序当中所提供的几个汉字字模：

```
code unsigned char GB1716[] =
{
/*--  文字:  铭  --*/
/*--  幼圆 12;  此字体下对应的点阵为：宽 x 高=17x16    --*/
/*--  宽度不是 8 的倍数，现调整为：宽度 x 高度=24x16   --*/
```

```
0x00,0x00,0x00,0x00,0x00,0x00,0x30,0x60,0x00,0x3E,0x7E,0x00,0x60,0xC6,0x00,0x61,
0xC6,0x00,0x7E,0x66,0x00,0x18,0x3C,0x00,0x18,0x18,0x00,0x18,0x30,0x00,0xFF,0xFE,
0x00,0x19,0xC3,0x00,0x18,0xC3,0x00,0x18,0xC3,0x00,0x1E,0xC3,0x00,0x1C,0xFF,0x00,
/*--  文字:  正  --*/
/*--  幼圆 12;  此字体下对应的点阵为：宽 x 高=17x16    --*/
/*--  宽度不是 8 的倍数，现调整为：宽度 x 高度=24x16  --*/
0x00,0x00,0x00,0x00,0x00,0x00,0x3F,0xFF,0x00,0x00,0xC0,0x00,0x00,0xC0,0x00,0x00,
0xC0,0x00,0x18,0xC0,0x00,0x18,0xC0,0x00,0x18,0xFE,0x00,0x18,0xC0,0x00,0x18,0xC0,
0x00,0x18,0xC0,0x00,0x18,0xC0,0x00,0x18,0xC0,0x00,0x18,0xC0,0x00,0xFF,0xFF,0x00,
/*--  文字:  同  --*/
/*--  幼圆 12;  此字体下对应的点阵为：宽 x 高=17x16    --*/
/*--  宽度不是 8 的倍数，现调整为：宽度 x 高度=24x16  --*/
0x00,0x00,0x00,0x00,0x00,0x00,0x3F,0xFE,0x00,0x60,0x03,0x00,0x6F,0xFB,0x00,0x60,
0x03,0x00,0x67,0xF3,0x00,0x6C,0x1B,0x00,0x6C,0x1B,0x00,0x6C,0x1B,0x00,0x6C,0x1B,
0x00,0x6C,0x1B,0x00,0x6F,0xFB,0x00,0x60,0x03,0x00,0x60,0x03,0x00,0x60,0x3E,0x00,
/*--  文字:  创  --*/
/*--  幼圆 12;  此字体下对应的点阵为：宽 x 高=17x16    --*/
/*--  宽度不是 8 的倍数，现调整为：宽度 x 高度=24x16  --*/
0x00,0x00,0x00,0x00,0x00,0x00,0x0E,0x03,0x00,0x1B,0x1B,0x00,0x19,0x9B,0x00,0x30,
0xDB,0x00,0xFF,0x7B,0x00,0x31,0x9B,0x00,0x31,0x9B,0x00,0x31,0x9B,0x00,0x31,0x9B,
0x00,0x37,0x9B,0x00,0x30,0x7B,0x00,0x30,0x63,0x00,0x30,0x63,0x00,0x1F,0xDE,0x00
};
```

在该字库中，每个汉字的字模数据为 3*16 个 byte，所以这四个汉字我们可以为它们定下一个查旬的序号就可以检索到它们的字模数据的首地址了；分别是：铭——0，正——1，同——2，创——3。

当然，这里所说的中文字库与 GB 的二级汉字库之类的不是一个概念的，二级汉字库有换算的算法，可以根据汉字的 GB 码值以及该字库的字符尺寸大小来检索到汉字的字模数据。这里就不多作解释。

在驱动程序当中有一个函数与定义的字库是息息相关的，就是 FontSet 函数，用户如果要定义新的字库的话，就需要在该函数修改相关的代码，同时在 LCD_Dis.c 当中对自定义的字库数组进行外部声明。下面来看看一些相关的定义：

在 LCD_Dis.c 文件当中有如下的定义和声明：

```
extern code unsigned char Asii0610[];          //6X10 的 ASII 字符库
extern code unsigned char GB1716[];            //17*16 自定义的汉字库

unsigned char X_Witch;                         //字符写入时的宽度
unsigned char Y_Witch;                         //字符写入时的高度
unsigned char Font_Wrod;                       //字体的每个字模占用多少个存储单元数
unsigned char *Char_TAB;                       //字库指针
unsigned char BMP_Color;
unsigned char Char_Color;
```

在前面的两个外部声明，就是分别在 LCD_ASCII.c 和 GB_Table.c 中定义好的字库；如果用户自己又额外定义了别的字库的话，同样也需要在这里加上它的外部声明。
而接下来的两个变量的定义：X_Witch 和 Y_Witch，分别是存放当前设置的字符类型的宽和高的点数大小，而 Font_Wrod 保存的是当前设置的字符的每个字模所占用的存储单元数量，指针 Char_TAB 则保存当前设置的字库数组的首地址。
BMP_Color 和 Char_Color 这两个变量分别保存当前设置的绘图前景色和字符色，其实对于黑白的单色 LCD 屏也就是零和非零的区别。

下面来看看 FontSet 函数中的设置：

```
//=============================================================================
// 函数: void FontSet(unsigned char Font_NUM,unsigned char Color)
// 描述: 文本字体设置
// 参数: Font_NUM 字体选择,以驱动所带的字库为准
//            Color   文本颜色,仅作用于自带字库
// 返回: 无
// 版本:
//        2006/10/15          First version
//=============================================================================
void FontSet(unsigned char Font_NUM,unsigned char Color)
{
          switch(Font_NUM)
          {
               case 1: Font_Wrod = 10;  //ASII 字符 B
                         X_Witch = 6;
                         Y_Witch = 10;
                         Char_Color = Color;
                         Char_TAB = (unsigned char *)(Asii0610 - (32*10));
               break;
               case 2: Font_Wrod = 48;  //汉字 A
                         X_Witch = 17;
                         Y_Witch = 16;
                         Char_Color = Color;
                         Char_TAB = (unsigned char *)GB1716;
               break;
               default: break;
          }
}
```

FontSet 函数的两个参数分别是表示要选择的字符类型、设置字符色；字符类型的选择实际上就看代码当中的 switch 分支怎么分配了，如上代码，设置为 1 时表示选择 ASCII 西文字符，为 2 时则为 GB_Table 中定义的自定义汉字库；当然如果用户还要另外上自定义的字库的话，可以多增加几个 case 选项。
在上面的代码中的 case 1 分支项里面，对选择的字库进行了参数的设置，跟前面我们分析过的 ASCII 码西文字库中的一样，每个字模所占用的存储单元数量为 10 个 byte，所以

Font_Wrod 设置为 10，X_Witch 和 Y_Witch 分别设置字符的实际宽度和高度，即 6*10；而设置 Char_TAB 值时，这里对驱动中删减掉的前 32 个 ASCII 字符进行了补齐，以便于在后面的字符显示程序中计算字模地址利用。

3.6.3. 用点来绘制字符

如前面所言，本版驱动程序的字符显示程序将以绘点函数为基础，简化字符显示程序的结构，使之更容易被读懂，同时也将为彩色的 LCD 模块驱动相兼容打下基础。
以绘点来显示字符，实际上就是将一个字符的字模数据的每个位都进行里判断，如果查询到哪个位上的值非零则对该位所对应的点的位置进行绘点操作。可以看看以下代码：

```
//=============================================================================
// 函数: void PutChar(unsigned char x,unsigned char y,char a)
// 描述: 写入一个标准字符
// 参数: x   X 轴坐标        y   Y 轴坐标
//           a   要显示的字符在字库中的偏移量
// 返回: 无
// 备注: ASCII 字符可直接输入 ASCII 码即可
// 版本:
//        2006/10/15         First version
//         2007/01/11            V1.1
//=============================================================================
void PutChar(unsigned char x,unsigned char y,char a)
{
        unsigned char i,j;              //数据暂存
        unsigned char *p_data;
        unsigned char Temp;
        unsigned char Index = 0;
        p_data = Char_TAB + a*Font_Wrod; //要写字符的首地址
        j = 0;
        while((j ++) < Y_Witch)
        {
            if(y > Dis_Y_MAX) break;
            i = 0;
            while(i < X_Witch)
            {
                if((i&0x07)==0)
                {
                    Temp = *(p_data+Index);
                    Index++;
                }
                if((Temp & 0x80) > 0) Write_Dot_LCD/*Writ_Dot*/(x+i,y,Char_Color);
                Temp = Temp << 1;
```

```
                    if((x+i) >= Dis_X_MAX)
                    {
                        Index += (X_Witch-i)>>3;
                        break;
                    }
                    i++;
                }
                y ++;
            }
        }
```

函数的一开始便利用前面介绍的 Char_TAB（保存有当前选择的字库的首地址）、Font_Wrod（每个字符的字模数据长度）以及传递进来的参数 a 计算出要显示的字符的字模数据首地址，并保存于指针当中。

随后 while((j ++) < Y_Witch)的循环意思就很明了了，也就是指明了要一行一行的显示，Y_Witch 在前面已经介绍过了。在循环里面每次都会判断是否超出了 Y 轴的最大值，如超出则跳出循环。

while(i < X_Witch)的小循环里则是对一行的点进行扫描，即对字模当中的一行的数据位进行扫描判断；循环里面 if((i&0x07)==0)则是调整指向字模数据的指针，每个数据位数为 8 位表示 8 个点，也就是每 8 个点的扫描后要使指针指向下一个数据，而 Index 变量中保存的是针对于当前字符字模数据的偏移量。

接下来的对字模数据从高位开始进行判断，如果非零则进行绘点的操作（由此可了解到，在我们的驱动程序当中，字符的显示会与之前在该位置上的已经显示了的图形进行重叠的）。调用的绘点函数为：Write_Dot_LCD (x+i,y,Char_Color)，参数 x 和 y 是传递进来的要显示字符的位置，即左上角的坐标值，当然 y 值在每一行的扫描显示结束后会自加 1，i 的变量为当前点在当前行中的列偏移。

字符的显示控制程序其实原理非常简单，无非就是将字模数据对应的位图点矩阵从左上角开始扫描，一行一行的扫描，直到结束，其间如要绘点的地方则绘制点就可以。

在字符显示程序的基础之上，又设计了一个字符串显示的子程序，很简单，代码如下：

```
//=============================================================================
// 函数: void PutString(unsigned char x,unsigned char y,char *p)
// 描述: 在 x、y 为起始坐标处写入一串标准字符
// 参数: x   X 轴坐标        y   Y 轴坐标
//           p   要显示的字符串
// 返回: 无
// 备注: 仅能用于自带的 ASCII 字符串显示
// 版本:
//        2006/10/15         First version
//=============================================================================
void PutString(unsigned char x,unsigned char y,char *p)
{
        while(*p!=0)
```

```
        {
            PutChar(x,y,*p);
            x += X_Witch;
            if((x + X_Witch) > Dis_X_MAX)
            {
                x = 0;
                if((Dis_Y_MAX - y) < Y_Witch) break;
                else y += Y_Witch;
            }
            p++;
        }
    }
```

3.6.4. Mz 的驱动中提供的字符显示

要在 MzL02 的通用版驱动程序的基础上显示字符，操作非常简单，只需要在显示字符之前设置好选择的字符类型（即选择字库），然后再调用单个字符显示程序（PutChar）或者字符串显示程序（PutString）即可。

要显示驱动当中自带的 ASCII 码西文字符的话，则要去了解驱动当中定义 ASCII 码西文字库的类型序号（要给 FontSet 函数的参数）；如果显示用户自行提取的其它字符的字库，则要按上前面所介绍的规则去提取字模并将字模数据加入程序当中形成正确的字库，并作好相关的代码修改，然后调用的方法与 ASCII 码西文字符的显示是差不多的。

如在 MzL02 的通用版 LCD 驱动程序当中就定义 ASCII 码西文字符库的序号为 1，而定义四个汉字“铭正同创”的自定义汉字库序号为 2，显控的示意代码如下：

```
    ……
    FontSet(1,1);           //选择 ASCII 码西文字库,并设字符色为黑色
    PutChar(10,10,'A');   //在坐标位置 10,10 的地方为左上角开始显示字符'A'
    PutString(10,20,"MzDesign!");    //在坐标位置 10,20 的地方为左上角开始显示字符串"MzDesign!"
    FontSet(2,1);           //选择自定义的汉字字库，并设字符色为黑色
    PutChar(20,30,0);       //在坐标位置 20,30 的地方为左上角开始显示字符“铭”
    PutChar(40,30,1);       //在坐标位置 40,30 的地方为左上角开始显示字符“正”
    PutChar(60,30,2);       //在坐标位置 60,30 的地方为左上角开始显示字符“同”
    PutChar(80,30,3);       //在坐标位置 80,30 的地方为左上角开始显示字符“创”
    ……
```

注：以上代码当中，汉字“铭正同创”的字模数据在该字库中的序号分别为 0~3。

如果用户需要到的话，可以自行提取要显示的其它字符的字模，并制成字库存放在 MCU 当中（本驱动仅针对于字库存放在 MCU 内部的存储器当中，当然如果字库存放在外存的话，也是可以在一定程度上参考的，其实也就是读取字模数据的地方作一下修改即可），然后定义好相关的变量修改相关的设置代码，就可以跟驱动当中自带的字库一样去调用显示了。具体的操作方法这里就不多介绍了，希望读者在前面的介绍的基础之上，去理解显示字符的原理，这样的话是无需教条主义式的按照一条一条一例一例的跟随做法去做；在理解的基础之

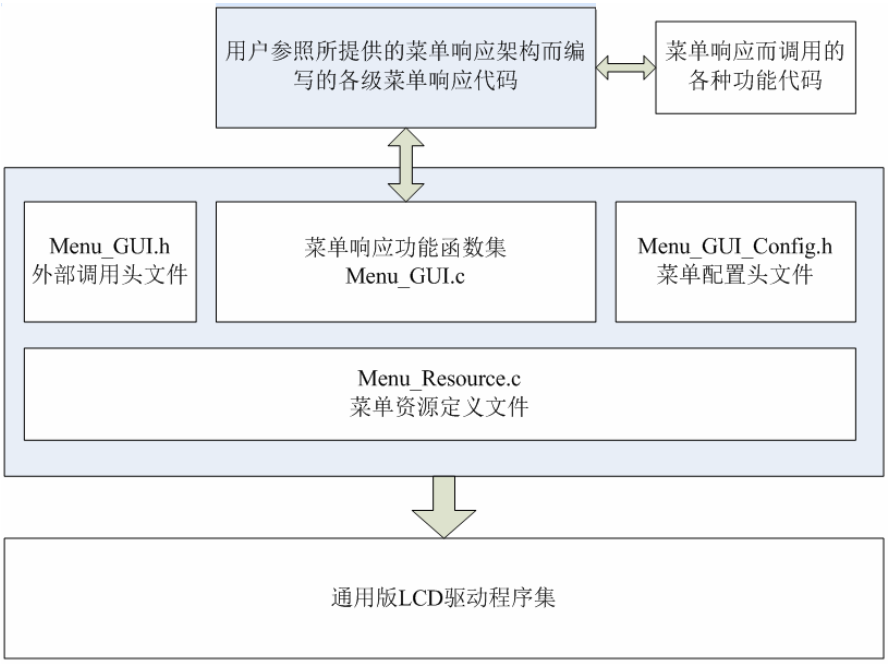
上做实践操作，这才是目的所在。

4. Mz_MenuGUI 菜单应用

4.1. Mz_MenuGUI

在一些带有点阵 LCD 显示界面的产品当中，通常会涉及到一些菜单界面的应用，特别是一些带有设置功能的仪器仪表产品；结合自己的设计经验，设计了一个小巧精致的菜单 GUI 代码，以期将其应用在中小资源的 MCU 上，并充分考虑了接不同的 MCU 或 LCD 模块的扩展能力。

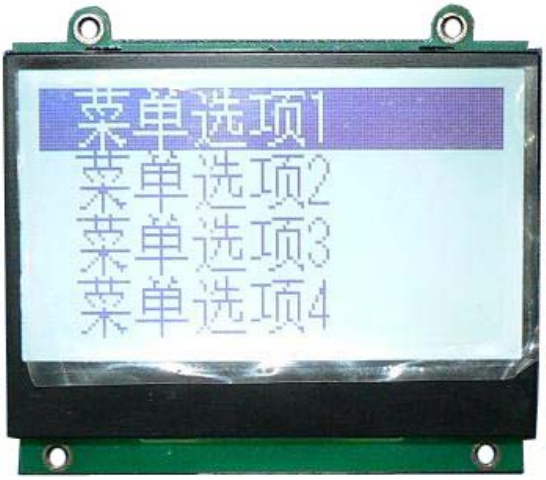
Mz_MenuGUI 实际上就是将菜单显示、响应刷新的代码综合到一起，做成几个精致的通用函数，以减小菜单显示方面的代码量，为小资源的 MCU 应用打下基础；而菜单的确切控制还留给用户来做，但这里提供了参考的框架，使用时往里面填代码就可以了。另外，Mz_MenuGUI 是基于前面所介绍的通用 LCD 驱动程序的，所以当要换用不同的 LCD 模块实现这样的简单菜单功能时，跟 LCD 驱动程序一样非常方便就可以实现移植了；下面了解一下 Mz_MenuGUI 的架构情况吧：



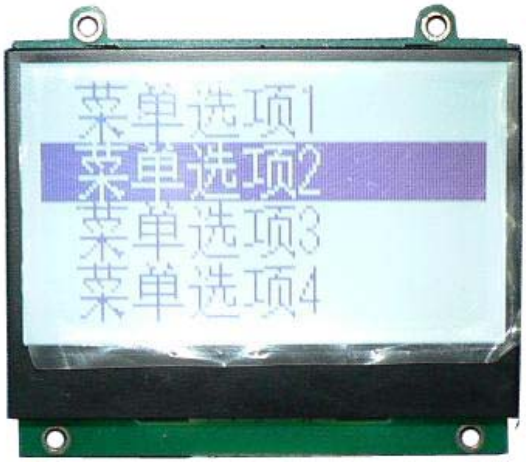
Mz_MenuGUI 共有四个文件提供，其中只有一个 Menu_GUI.c 是正经八百的 C 语言函数代码，其它的三个基本上用于定义与外部声明用；而 Menu_Resource.c 当中是定义菜单资源的代码，其实也就是定义几个数组，里面的内容定义好菜单当中显示什么字符以及菜单字符的一些特性定义，这里的定义是要跟 LCD 的驱动程序当中的字库相关的定义有联系的，这点会在后面的代码介绍里面分析。

而菜单响应控制方面的代码需要用户自行编写，当然是在这里提供的架构参考的基础之上了，其实也就是做一个按键响应的 switch 分支，分别定义什么按键时菜单作什么样的更新操作；这点，我们提供了参考的架构，当然也可以省省事，直接拿来使用也可以。

Mz_MenuGUI 做的菜单界面可以参考以下几张图片：



菜单显示效果（MzL02），当前活动菜单项处于第一项；



菜单显示效果（MzL02），当前活动菜单项处于第二项。

4.2. Mz_MenuGUI 的源码分析

4.2.1. Menu_Resource.c 菜单资源定义

Mz_MenuGUI 的菜单系统当中，在 Menu_Resource.c 文件里完成了所有的菜单字符资源以及菜单项的定义，主要定义有三类如下，可以从源码中看出：

```
//===========================================================================
// 文件名: Menu_Resource.c
// 作　者: Xinqiang Zhang(email: Xinqiang@Mzdesign.com.cn)
//              www.Mzdesign.com.cn
// 日　期: 2007/03/24
// 描　述: 菜单应用范例程序--UI 显示资源定义文件
//              此代码适用于无 byte 操作的 16 位 MCU，如凌阳的 unsp 系列 MCU
//              有关汉字字库的资源请参考铭正同创网站上有关 LCD 显示中文的文章，或
//              直接参考铭正同创（Mzdesign）提供的 LCD 通用版基本驱动程序
```

```
//
// 参　考:
// 版　本:
//          2007/03/24          First version       Mz Design
//          2007/07/26          V1.01                   Mz Design
//
//=========================================================================
//定义单条菜单项内容,格式有两种,一种为支持汉字库 LCD 的纯汉字菜单项,一种是西文字符与自定
//义汉字库的混合菜单项的如下：
//一:直接用汉字的字串即可,不同的编译器可能在汉字的 GB 码数据类型上有所不一样
//二:菜单项字符数,第一个字符在字库中的序号,第二个字符,....
//注：在第二种情况下,为了区分自定义汉字与 ASCII 码，特定将自定义汉字库中的汉字编码前加上
//   128 作为标识
code unsigned char Menu_String01[]={5,0+0x80,1+0x80,2+0x80,3+0x80, 4+0x80};
code unsigned char Menu_String02[]={5,0+0x80,1+0x80,2+0x80,3+0x80, 5+0x80};
code unsigned char Menu_String03[]={5,0+0x80,1+0x80,2+0x80,3+0x80, 6+0x80};
code unsigned char Menu_String04[]={5,0+0x80,1+0x80,2+0x80,3+0x80, 7+0x80};
code unsigned char Menu_String05[]={5,0+0x80,1+0x80,2+0x80,3+0x80, 8+0x80};
code unsigned char Menu_String06[]={5,0+0x80,1+0x80,2+0x80,3+0x80, 9+0x80};
//定义某一组菜单的配置，格式如下：
//{该组菜单的菜单项数目,该组菜单中汉字所选用的字符类型,该组菜单中 ASCII 码所选用的类型,
//该组菜单中每条菜单项所占用的 Y 轴大小,该组菜单中菜单项显示的 X 轴偏移位}
code unsigned char Menu_List01_Config[]={6,3,1,16,10};
//定义一组菜单的菜单项，格式如下：
//{该组菜单所对应的配置,第一条菜单项,第二条菜单项.....}
//注：菜单组列表中菜单项的数目要与相应的配置里一至哦！
code unsigned char *Menu_List01[]=//
{(unsigned char *)Menu_List01_Config,(unsigned char *)Menu_String01,
(unsigned char *)Menu_String02,(unsigned char *)Menu_String03,
(unsigned char *)Menu_String04,(unsigned char *)Menu_String05,
(unsigned char *)Menu_String06};
```

代码前面的地方，定义了单条菜单项的内容，代码的注释里已有格式的简单说明了，用户需要参考前介绍通用 LCD 驱动程序当中关于字库的介绍，因为这里的定义是与驱动函数中字库的定义相关的。比如，菜单项 Menu_String01 的定义里，采用的是西文字库（ASCII 码）与自定义汉字库相结合的类型，第一个量的意义为该菜单项中有几个字符，第二个量之后的数据为字符在字库中的序号，如果字符是 ASCII 码西文字符的话，可以直接用'A'之类的定义即可，编译时会取该字符的 ASCII 码值，如果是自定义汉字库的话则要在其序号的基础之上加上 0x80（即 128），上面的代码中就是例子。（在下面，会介绍这里的菜单 GUI 代码所应用的 LCD 驱动程序中的字库定义情况）

而菜单项要定义多少个就由用户自行选择了，在上面的代码当中，共定义了 6 个菜单项，跟显示效果里的一样，分别是：

“菜单选项一”、“菜单选项二”、“菜单选项三”、“菜单选项四”、“菜单选项五”、“菜单选项六”。

菜单中使用的汉字字符在驱动里都有了定义，它们在字库里的序号分别是：

“菜”——0

“单”——1

“选”——2

“项”——3

“一”——4

“二”——5

“三”——6

“四”——7

“五”——8

“六”——9

在 GB_Table.c 文件当中定义了它们的字库，如下：

```
code unsigned char GB1616[] =
{
/*--  文字:  菜  --*/
0x00,0x00,0x0C,0x70,0x0C,0x70,0xFF,0xFF,0x0C,0x70,0x00,0xFC,0x3F,0xCC,0x19,0x98,
0x1D,0xF8,0x0D,0xF0,0xFF,0xFF,0x03,0xC0,0x07,0xE0,0x1D,0xB8,0x39,0x9E,0xE1,0x87,
/*--  文字:  单  --*/
0x00,0x00,0x18,0x18,0x1C,0x38,0x0C,0x70,0x7F,0xFC,0x61,0x8C,0x61,0x8C,0x7F,0xFC,
0x61,0x8C,0x61,0x8C,0x3F,0xFC,0x01,0x80,0xFF,0xFF,0x01,0x80,0x01,0x80,0x01,0x80,
/*--  文字:  选  --*/
0x00,0x00,0x00,0x00,0x73,0x60,0x37,0x60,0x3F,0xFE,0x1E,0x60,0x0C,0x60,0x00,0x60,
0xFF,0xFF,0x39,0xB8,0x39,0xB8,0x39,0xBF,0x3B,0x3F,0x3F,0x3E,0x7E,0x1E,0xEF,0xFF,
/*--  文字:  项  --*/
0x00,0x00,0x00,0x00,0xFF,0xFF,0x3C,0x60,0x33,0xFC,0x37,0x06,0x37,0x66,0x37,0x66,
0x37,0x66,0x37,0x66,0x37,0x66,0x37,0xE6,0x3F,0xC6,0xF1,0xF8,0x03,0x8E,0x0E,0x07,
/*--  文字:  一  --*/
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0xFF,0xFF,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
/*--  文字:  二  --*/
0x00,0x00,0x00,0x00,0x00,0x00,0x7F,0xFE,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0xFF,0x00,0x00,0x00,
/*--  文字:  三  --*/
0x00,0x00,0x00,0x00,0x00,0x00,0x7F,0xFE,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x3F,0xFC,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xFF,0xFF,0x00,
/*--  文字:  四  --*/
0x00,0x00,0x00,0x00,0x7F,0xFE,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,0x66,
0x66,0x66,0x66,0x66,0x6E,0x66,0x78,0x7E,0x60,0x06,0x60,0x06,0x60,0x06,0x7F,0xFE,
/*--  文字:  五  --*/
0x00,0x00,0x00,0x00,0xFF,0xFE,0x03,0x00,0x07,0x00,0x07,0x00,0x07,0x00,0x7F,0xFC,
```

```
0x06,0x0C,0x06,0x0C,0x06,0x0C,0x06,0x0C,0x0E,0x0C,0x0E,0x0C,0x0C,0x0C,0xFF,0xFF,
/*--  文字:  六  --*/
0x00,0x00,0x00,0x00,0x03,0x80,0x01,0x80,0x01,0xC0,0x00,0xC0,0xFF,0xFF,0x00,0x60,
0x0E,0x70,0x0C,0x38,0x0C,0x18,0x1C,0x1C,0x18,0x0C,0x38,0x0E,0x70,0x06,0xE0,0x07
};
```

定义好字库后同时也要在 LCD_Dis.c 当中为新的字库修改一定的代码，在该文件的前面加上对这个字库的外部声明，如下：

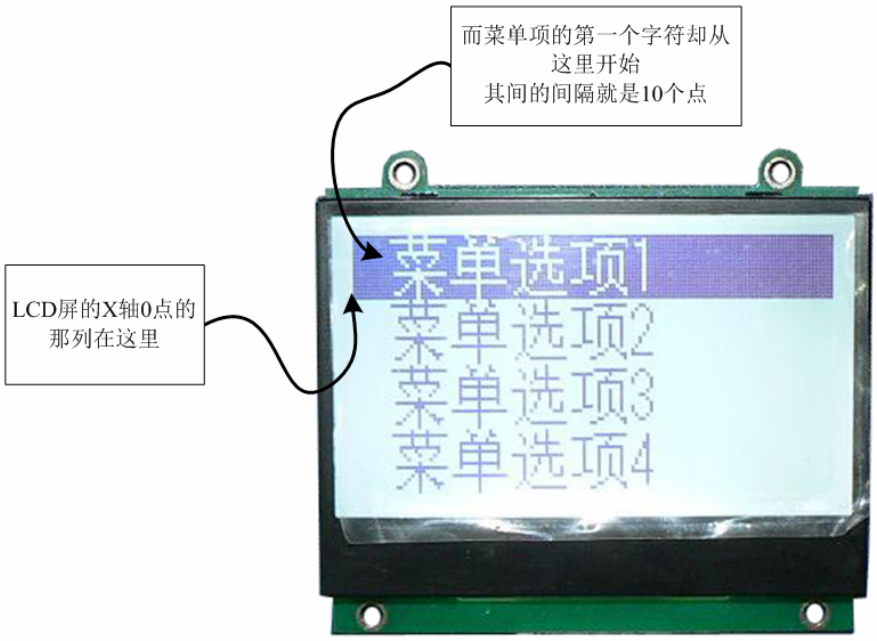
```
extern code unsigned char Asii0610[];          //6X10 的 ASII 字符库
extern code unsigned char GB1616[];            //16X16 的自定义字符库
extern code unsigned char GB1716[];                 //17*16 自定义的汉字库
```

然后在 FontSet 函数中加上该字库所对应的配置代码，如下：

```
void FontSet(unsigned char Font_NUM,unsigned char Color)
{
        switch(Font_NUM)
        {
            case 1: Font_Wrod = 10;  //ASII 字符 B
                    X_Witch = 6;
                    Y_Witch = 10;
                    Char_Color = Color;
                    Char_TAB = (unsigned char *)(Asii0610 - (32*10));
            break;
            case 2: Font_Wrod = 48;  //汉字 A
                    X_Witch = 17;
                    Y_Witch = 16;
                    Char_Color = Color;
                    Char_TAB = (unsigned char *)GB1716;
            break;
            case 3: Font_Wrod = 32;  //汉字 B
                    X_Witch = 16;
                    Y_Witch = 16;
                    Char_Color = Color;
                    Char_TAB = (unsigned char *)GB1616;
            break;
            default: break;
        }
}
```

接下来定义了一个数组，用于一组菜单的配置，也就是说可以在程序里面定义多组菜单，然后每一组菜单都可以拥有自己的特性配置。下面看看这个配置的数组里具体定义的含意。Menu_List01_Config[]={6,3,1,16,10}其中定义了 5 个数据，第一个数据为声明该组菜单当中共有多少个菜单项；比如在前面的介绍中得知，这个菜单 Demo 里面共定义了六个菜单项，所以在这里将该数值设置为 6；接下来的一个数据代表混合字符菜单项里的汉字字库序号，能过前面介绍的代码可知，定义为 3；接着便是 ASCII 码西文字库的序号，为 1。

而其中的数据 16 表示该组菜单当中每一个菜单项所占用的 Y 轴的点数，通常取该组菜单当中的字符里 Y 轴方面最大的值；这里可知前面定义的菜单项里面采用的是 16*16 点阵的字符，所以该值定义为了 16。最后的一个数据表示菜单项显示时，对 X 轴原点的固定偏移数；可以从下图中看出在数组中定义的 10 的意义：



如上图所示意的，如果想要菜单项的字符从 X 轴为 0 的地方开始显示的话，只需要将菜单配置数组中的最后一个数设为 0 即可。

最后一个定义是对一组菜单的定义了，用户在菜单响应的控制程序里会使用到该数组，一组菜单定义对应一个菜单界面，如果有多级菜单的话，可以定义多组菜单，格式参考代码里的定义即可。这组数据当中，第一个量为该组菜单的配置，接下去的数据为该组菜单当中的每个菜单项的定义。

注意：可能不同的编译器对指针的编译会有所不同，包括常量的定义，这点请用户注意，要移植该代码时可能要作出适量的修改。

4.2.2. Menu_GUI_Config.h 菜单 GUI 配置头文件

菜单 GUI 的配置头文件里面定义了菜单的字符色等，下面是源码：

```
//    write your header here
typedef unsigned int UINT;
typedef unsigned char UCHAR;
#define COLOR_1                 0x0001          //黑色
#define COLOR_2                 0x0000          //白色

#define MENU_BACK_COLOR         COLOR_2             //定义菜单系统当中的背景色
#define MENU_FONT_COLOR         COLOR_1             //定义菜单系统当中的文字色
#define MENU_SELE_COLOR         COLOR_1//定义菜单系统当中被选择的菜单项背景色
```

```
#define MENU_SELF_COLOR          COLOR_2//定义菜单系统当中被选择的菜单项文字色

//#define Hz_Lib_II      1       //使用二级汉字库的定义,如果LCD的驱动中支持二级字库
                                 //则使该定义有效,否则将其屏蔽
```

最前面用了两个 typedef 对两个数据类型进行了重定义；然后用 define 定义了两种颜色（该菜单 GUI 代码是可以适应彩色的 LCD 的，所以这里的定义为将来的扩展打基础），再跟着下来是重定义了几个常量，它们在代码中都有注释，这些颜色的重定义将会在 Menu_GUI.c 的代码当中使用。

最后的一个定义在代码当中屏蔽掉了，它是针对于带字库的 LCD 模块而作的扩展定义。

4.2.3. Menu_GUI.c 菜单接口函数

先看看两个跟配置有关的函数：

```
//=========================================================================//
//函数：UCHAR GetMLNum(UCHAR ** Menu_List)
//描述：获取菜单资源的菜单项个数函数
//参数：  Menu_List      菜单资源链表指针
//返回：菜单项个数
//注意：无
//=========================================================================//
UCHAR GetMLNum(UCHAR **Menu_List)
{
        UCHAR uiTemp;
        uiTemp = (unsigned char)**Menu_List;
        return uiTemp-1;
}
//=========================================================================//
//函数：UCHAR GetMLiNum_Page(UCHAR ** Menu_List)
//描述：获取菜单资源在一屏可以显示的菜单项个数函数
//参数：  Menu_List      菜单资源链表指针
//返回：菜单项个数
//注意：无
//=========================================================================//
UCHAR GetMLiNum_Page(UCHAR** Menu_List)
{
        UCHAR uiTemp;
        UCHAR *Menu_Config;
        Menu_Config = (UCHAR *)(Menu_List[0]);
        uiTemp = *(Menu_Config+3);
        uiTemp = (Dis_Y_MAX+1)/uiTemp;              //
        return uiTemp;
}
```

如注释所述，UCHAR GetMLNum(UCHAR ** Menu_List)函数是获取一组菜单资源里面的菜

单项个数的，传递进该函数的参数就是上一小节当中所说的一组菜单的定义数组，返回的是该组菜单资源当中的菜单项个数；不过要注意的是跟前面介绍的定义有点不一样，在前面介绍时，如果菜单项个数为 6 个的话，在定义里面就直接将菜单资源配置数组里的第一个量置为 6 了，这里的返回值也是读取了该数据，但减去了 1，因为菜单项的序号从 0 开始。
UCHAR GetMLiNum_Page(UCHAR** Menu_List)函数就与 LCD 驱动程序当中的配置有关了，它的功能是获取一屏 LCD 显示当中共能显示菜单项数量，代码中使用了 Dis_Y_MAX 的配置，并除以菜单项的 Y 轴点数，得到的是一屏 LCD 显示能容下的菜单项数。

在菜单显示的代码里面，定义了几个全局的变量，用于更新显示时使用，如下：

```
#include "./Driver/LCD_Driver/LCD_Dis.h"
#include "./Driver/LCD_Driver/LCD_Config.h"
#include "./MzMenu_GUI/Menu_GUI_config.h"

UCHAR Y_WIDTH_MENU=16;
UCHAR X_SPACE_FRONT=10;

UCHAR Dis_Menu_Num=0;

UCHAR Font_GB=0;
UCHAR Font_String=0;
UCHAR First_Index_old=0xff;
UCHAR y_Index_old = 0xff;
```

Y_WIDTH_MENU 存放有当前菜单组的一个菜单项占用 Y 轴点数，与菜单组配置数组当中的定义是一样的意义；只不过可能有多个菜单组资源存在于用户的应用程序当中，而这个变量保存的是当前选择的，也就是当前显示在 LCD 当中的菜单组的。
X_SPACE_FRONT 存放着当前菜单组在显示菜单项时，菜单项字符偏移 X 轴原点的点数。
Dis_Menu_Num 则保存着当前的菜单组能在一屏 LCD 当中显示的个数。
Font_GB 中为当前的菜单组当中选用的自定义汉字库序号，Font_String 则是 ASCII 码西文字库的序号。

First_Index_old 变量保存的是菜单当前显示在 LCD 屏上的首行菜单项的序号；由于菜单组可能有多个菜单项，而菜单项个数大于一屏能显示的菜单项个数时，首行的菜单项就有可能会在用户操作菜单时发生变化；此变量将在菜单的显示刷新当中使用。
y_Index_old 则保存关当前菜单项当中，处于活动的那项序号，也就是选择了哪一项菜单。
First_Index_old 和 y_Inded_old 变量初始化它们的值为 0xff，而在代码中也利用了该数值（0xff）在下面的代码当中会有介绍。

接下来，有两个控制函数，功能差不多，只是有细微的区别，在用户编写菜单响应控制函数时有用：

```
//=========================================================================//
//函数：void Redraw_Menu(UCHAR First_Index,UCHAR Menu_Index, UCHAR ** Menu_List)
//描述：刷新整屏菜单显示函数
//参数：First_Index  当面显示页的第一条菜单号
```

```
//          Menu_Index     当前处于选用的菜单项
//          Menu_List      菜单资源链表指针
//返回：
//注意：无
//=========================================================================//
void Redraw_Menu(UCHAR First_Index,UCHAR Menu_Index,UCHAR** Menu_List)
{
        UCHAR *Menu_Config;
        First_Index_old=0xff;
        Menu_Config = (UCHAR *)(*Menu_List);
        Font_GB = *(Menu_Config+1);
        Font_String = *(Menu_Config+2);
        Y_WIDTH_MENU = *(Menu_Config+3);
        X_SPACE_FRONT = *(Menu_Config+4);
        Dis_Menu_Num = (Dis_Y_MAX+1)/Y_WIDTH_MENU;
        UpDate_Menu(First_Index,Menu_Index,Menu_List);
}
//=========================================================================//
//函数：void Initial_Menu(UCHAR ** Menu_List)
//描述：刷新整屏菜单显示函数
//参数：Menu_List      菜单资源链表指针
//返回：
//注意：无
//=========================================================================//
void Initial_Menu(UCHAR** Menu_List)
{
        UCHAR *Menu_Config;
        First_Index_old=0xff;
        y_Index_old = 0xff;
        Menu_Config = (UCHAR *)(*Menu_List);
        Font_GB = *(Menu_Config+1);
        Font_String = *(Menu_Config+2);
        Y_WIDTH_MENU = *(Menu_Config+3);
        X_SPACE_FRONT = *(Menu_Config+4);
        Dis_Menu_Num = (Dis_Y_MAX+1)/Y_WIDTH_MENU;
        UpDate_Menu(0,0,Menu_List);
}
```

Redraw_Menu 和 Initial_Menu 函数的功能相似，可以从代码中看出来，在函数里面都要对前面介绍的全局变量进行设置，并调用 UpData_Menu 函数进行重绘菜单。

而在 Initial_Menu 函数当中，将 First_Index_old 和 y_Index_old 重置为 0xff，则是要全部重新的绘制菜单，使当前显示屏中第一列菜单项为该菜单组中的第一项，而当前选择的菜单项初始化为该菜单组中的第一项；函数中传递进来的 Menu_List 参数为要选用的菜单资源链表

（也就是二维数组）。

Redraw_Menu 函数有三个参数，分别在注释中已有说明；在该函数里与 Inialt_Menu 的区别也就是对 y_Index_old 的初始化，其实也就是 Redraw_Menu 用于重绘菜单，但菜单组中显示在 LCD 屏上的第一项菜单项并不一定是菜单组中的首项，而是由 Fist_Index 指定；而当前处于选定状态的菜单项由参数 Menu_Index 指定。此函数一般使用在菜单响应后，用户程序进入了另外的显示界面的，当返回时重绘菜单使用。

菜单显示控制函数主要只有两个，无论用户定义了多少组菜单资源，都是这两个函数来完成显示。下面分别看一下它们的代码：

UpDate_Menu 菜单更新函数：

```
//=====================================================================//
//函数：void UpDate_Menu(UCHAR First_Index,UCHAR Menu_Index, UCHAR ** Menu_List)
//描述：刷新整屏菜单显示函数
//参数：First_Index   当面显示页的第一条菜单号
//          Menu_Index     当前处于选用的菜单项
//          Menu_List      菜单资源链表指针
//返回：无
//注意：无
//=====================================================================//
void UpDate_Menu(UCHAR First_Index,UCHAR Menu_Index,UCHAR** Menu_List)
{
        UINT y_width,y_Index;
        UCHAR List_Num,i;
        List_Num = (UINT)**Menu_List;
        y_width = Y_WIDTH_MENU;
        y_Index = 0;
        while(First_Index>List_Num) First_Index -= List_Num;          ①
        if(List_Num>Dis_Menu_Num) List_Num = Dis_Menu_Num;
        if(First_Index_old!=First_Index)                              ②
        {
            SetPaintMode(1,MENU_BACK_COLOR);
            ClrScreen(0);                                             //清屏
            SetPaintMode(1,MENU_SELE_COLOR);
            if(y_Index_old==0xff)                                     ③
            {
                Rectangle(0,0,Dis_X_MAX, y_width-1,1);
                i=First_Index+1;
                ShowMenu_Item(y_Index,(UCHAR *)Menu_List[i++],MENU_SELF_COLOR);
                y_Index = y_Index+Y_WIDTH_MENU;
                for(;i<List_Num+1;i++)
                {
                    ShowMenu_Item(y_Index,(UCHAR *)Menu_List[i],
```

```
                                    MENU_FONT_COLOR);
                        y_Index = y_Index+Y_WIDTH_MENU;
                    }
                }
                else                                                    ④
                {
                    y_Index_old = Menu_Index-First_Index;               ⑤
                    Rectangle(0,y_Index_old*Y_WIDTH_MENU,Dis_X_MAX,
                                    y_Index_old*Y_WIDTH_MENU+Y_WIDTH_MENU-1,1);
                    i=First_Index+1;
                    while(List_Num)
                    {
                        if((i-1)==Menu_Index) ShowMenu_Item(y_Index,(UCHAR *)Menu_List[i],
                                    MENU_SELF_COLOR);                   ⑥
                        else ShowMenu_Item(y_Index,(UCHAR *)Menu_List[i],
                                    MENU_FONT_COLOR);                   ⑦
                        y_Index = y_Index+Y_WIDTH_MENU;
                        i++;
                        List_Num--;
                    }
                }
                First_Index_old = First_Index;                          ⑧
            }
            else                                                        ⑨
            {
                y_Index = y_Index_old-First_Index_old;
                y_Index = y_Index*Y_WIDTH_MENU;
                SetPaintMode(1,MENU_BACK_COLOR);
                Rectangle(0,y_Index,Dis_X_MAX, y_Index+Y_WIDTH_MENU-1,1);
                ShowMenu_Item(y_Index,(UCHAR *)Menu_List[y_Index_old+1],
                                    MENU_FONT_COLOR);
                y_Index = Menu_Index-First_Index;
                y_Index = y_Index*Y_WIDTH_MENU;
                SetPaintMode(1,MENU_SELE_COLOR);
                Rectangle(0, y_Index,Dis_X_MAX, y_Index+Y_WIDTH_MENU-1,1);
                ShowMenu_Item(y_Index,(UCHAR *)Menu_List[Menu_Index+1],
                                    MENU_SELF_COLOR);
            }
            y_Index_old = Menu_Index;
        }
```

菜单更新函数较为复杂，下在分段分析代码。

函数多次调用到的子函数 ShowMenu_Item 是菜单项绘制函数，将在接下去的代码中介绍，该函数有两个参数，一个是要绘制的菜单项，一个是菜单项的颜色。

①：
判断传递进来的参数 Fisrt_Index 是否超出了选用的菜单组的菜单项个数，如果超出了则处理一下；接着对 List_Num 进行处理，该变量的意义为一屏 LCD 能显示菜单项的个数，如果当前选用的菜单组的菜单项个数大于一屏 LCD 能显示的菜单个数，则取后者的值；如果等于或小于，则取当前菜单组的菜单项个数。
②：
判断 First_Index_old 是否等于传递进来的参数 First_Index，如果不相等则表示当前 LCD 屏显示当中的第一列菜单项发生较之前发生了变化，需要重绘整个菜单。
要重绘全部菜单时，首先要将屏幕全部清屏。
③：
此处判断 y_Index_old 是否为 0xff，如果是 0xff，则认定为初始化菜单时的重绘菜单，程序会将当前屏的 LCD 显示的第一列菜单项选择为菜单组中的第一项菜单项，而处于选择状态的菜单项为菜单组资源当中的第一项菜单项。
重绘菜单时，先绘制处于选择状态的反色条，与前面图样里屏幕上与背景的白色反色的黑色条一样，当然对于彩色的 LCD 是可以选择很多的不同颜色的。然后在同样的位置上绘制出该条处于选择状态的菜单项，也就是菜单组资源中的第一项菜单项了。随后绘制其它的不处于选择状态的菜单项。
④：
跟随着前面的判断，如果 y_Index_old 不为 0xff 的话，则是正常的重绘菜单，也就是通过 Redraw_Menu 函数调用进来的，或者是 LCD 屏上首行显示的菜单项发生了变化。
⑤：
这时，也是首先绘制处于选择状态的反色条，它的位置由 Menu_Index-First_Index 计算出来，Menu_Index 是传递进来的参数，代表当前菜单当中处于选择状态的菜单项是菜单组资源当中的第几项，而 First_Index 则是当前菜单项当中处于 LCD 屏显示的首行菜单项是菜单组资源当中的第几项，两者相减便得出当前处于选择状态的菜单项应该在 LCD 屏当中的具体位置了；这里使用 y_Index_old 变量暂时保存一下位置数据。
然后，根据计算出来的当前处于选择状态的菜单项的位置绘制反色条，最后依次绘制一屏 LCD 可以显示的多个菜单项。
⑥：
当然在绘制菜单项时会判断是否为处于选择状态的，如是则选用配置的选择菜单项字符色来绘制。
⑦：
否则就使用正常的菜单项字符色绘制菜单项字符。
⑧：
这里将更新菜单前的 LCD 屏显示首行菜单项序号更新，以便在下次调用显示时使用。
⑨：
最后，如果不必要对全屏进行菜单刷新的话，就会进入该分支进行对菜单的刷新了，也就是 y_Index_old 和 First_Index_old 都没有被初始化为 0xff，而且 First_Index_old 与传递进来的参数 First_Index 是相同的，也即说明 LCD 屏上的首行菜单项没有发生变化。
然后，计算出当前 LCD 屏上显示的处于选择状态的菜单项位置，将该位置的显示清零，重绘上该位置的菜单项字符；再计算刷新后处于选择状态的菜单项的位置，在该位置绘制反色条，最后再绘制处于选择状态的菜单项。

ShowMenu_Item 菜单项绘制函数前面介绍的 LCD 驱动程序当中的字库定义以及菜单项的定义有很紧密的关系，下面是它的代码：

```
//=====================================================================//
//函数：UCHAR ShowMenu_Item(UCHAR y,UINT* Menu_String,UCHAR Font_Color)
//描述：显示菜单项子函数
//参数：space_front   显示缩进值
//        y             Y 轴坐标
//        Menu_String   菜单项链表的首地址指针
//返回：显示溢出情况   0：溢出      1：无溢出
//注意：无
//=====================================================================//
UCHAR ShowMenu_Item(UCHAR y,UCHAR* Menu_String,UCHAR Font_Color)
{
        UCHAR *uiTemp;
        UCHAR uiTemp1;
        UCHAR i,x,Char_Nmb;
        x = X_SPACE_FRONT;                          //Menu show front space....
        Char_Nmb = (UCHAR)Menu_String[0];
        if(Char_Nmb<0xA1)
        {
            for(i=1;i<=Char_Nmb;i++)
            {
                uiTemp = (UCHAR*)(Menu_String+i);
                uiTemp1 = (UCHAR)*uiTemp;
                if(uiTemp1>128)
                {
                    FontSet(Font_GB,Font_Color); //选择汉字字库
                    uiTemp1 = uiTemp1-128;
                }
                else
                {
                    FontSet(Font_String,Font_Color);//选择 ASCII 码字库
                }
                PutChar(x,y,uiTemp1);
                x = x+X_Witch;//GetASIIX();

                if(x>=Dis_X_MAX) return 0;              //横坐标溢出，返回零
            }
        }
#ifdef Hz_Lib_II
        else
        {
```

```
                FontSet_cn(Font_String,Font_Color);
                PutString_cn(x,y,(unsigned short *)Menu_String);
            }
    #endif
            return 1;
    }
```

局部变量 x 首先会置为 X_SPACE_FRONT 的值，而 X_SPACE_FRONT 也就是在菜单组资源当中定义的配置数组中的菜单项字符偏移 X 轴原点的点数。
获取要绘制的菜单项的第一个节字，存放于 Char_Nmb 变量当中。
这里作一下简单的说明，在国标的二级汉字库中，每个汉字的 GB 码值的高八位和低八位值都是大于 0xA1 的；所以在此会判断 Char_Nmb 值是否大于 0xA1，如小于则表明当前的菜单 GUI 使用的是自定义汉字库和 LCD 驱动中自带的 ASCII 码西文字库的字符。
绘制菜单项当中的自定义汉字库或者 ASCII 码字符时，Char_Nmb 的值代表该菜单项的字符个数，并以该数值作一个 for 循环，依次将该菜单项的字符绘制完毕。在前面已介绍过，在定义菜单项时，如使用自定义的汉字库字符，则在定义时在该字符的基础上加上 128；所以在代码当中可看到从菜单项的定义数组中读出字符序号数据后会判断是否大于 128，如果大于则表示该字符为自定义的汉字库当中的字符，调用 FontSet 函数选择该字库作为当前字符类型，并调整字符序号值；如小于则选用 LCD 驱运中的 ASCII 码西文字库作为当前字符类型。最后调用 PutChar 函数绘制字符，并对 X 轴的坐标作出调整，以便显示下一个字符。
在前面介绍 Menu_GUI_Config.h 文件时，已经介绍过 Hz_Lib_II 的宏定义表示所使用的 LCD 是否为自带汉字库的 LCD，如果是的话，则 Hz_Lib_II 有定义，则“#def Hz_Lib_II”和“#endif”之间的代码将会被编译，作为选用自带汉字库的 LCD 时显示菜单中的汉字的显控代码。

注意：Mzdesign 的自带汉字库的 LCD 在驱动程序里提供有 FontSet_cn 和 PutString_cn 这两个函数，而本书所介绍的 LCD 通用驱动程序当中是没有这两个函数的，这点请读者参考 Mzdesign 的代码。

4.3. 定制自己的 Menu 菜单界面

4.3.1. 参考的 GUI 响应控制代码

为了方便使用，配合着 Mz_MenuGUI 代码提供了一份参考的响应控制代码，用户可以参照它的架构编写自己需要的控制代码。
当然，菜单的使用一般离不开键盘，在这里就使用了键盘扫描程序，但仅供参考，用户可以根据自己的情况编写合适的键盘扫描程序，并定义合适的键值作为菜单响应控制的控制键。

首先，菜单响应控制代码需要定义几个变量，如下：

```
    //add your code here
        unsigned char uiKey=0;
        //uiKey 用于存放扫描的键值状态
        unsigned char Item_Num,Update_Flag,Enter_Flag=0;
        //Item_Num:当前菜单组当中共有几项菜单项,刷新时使用
        //Update_Flag:菜单界面刷新标识
```

```
//Enter_Flag:确定键按下标识~~

unsigned char PageItem_Num;
//PageItem_Num:每页(全显示屏内)可以显示多少个菜单项

unsigned char First_Index=0,Active_Index=0,Temp_Index=0;
//First_Index:当前处在显示屏当中的最前面的菜单项序号
//Active_Index:当前指向的菜单项,即当前活动的菜单项
//Temp_Index:中间变量
unsigned char Exit_flag=1;      //菜单响应循环退出标识
```

上面定义的变量当中，有几个是用于暂存菜单组资源中的一些数据的，比如 Item_Num、PageItem_Num 等，其实在前面介绍的菜单显示控制的代码（Menu_GUI.c）当中已有定义了，这里再定义实际上是为多级菜单的使用而做的；因为在 Menu_GUI.c 当中定义的变量只能供当前选用的菜单组使用，如有多级菜单时，相互之间有层层的调用关系，单靠其中的变量是无法得知全部的菜单组资源的参数的。由此，建议每个菜单界面都应定义上述的变量，以供当前的菜单响应控制使用。

Exit_flag 变量为循环退出标识，将在菜单响应控制的循环当中控制着是否继续下去，该变量被设置为 0 时，表示要退出当前的菜单。当然，如用户的菜单只有一层，而且始终在使用该菜单的话，可以不使用这个变量，直接在菜单响应控制的循环当中用 1 替代它即可。

菜单响应控制代码当中要使用到一些函数，所以在其前面需对一些外部头文件进行包含声明。如下：

```
#include "./Driver/LCD_Driver/LCD_Dis.h"
#include "./Driver/Key_Service/Key.h"
#include "./MzMenu_GUI/Menu_GUI.h"
```

Key.h 是键盘扫描程序的头文件，这里不作说明。

LCD_Dis.h 为通用版的 LCD 驱动程序的用户接口程序头文件，可视情况而选择是否使用它（至少在用户的菜单响应控制代码当中是可以选择的）。

菜单响应控制的参考代码如下：

```
Key_Initial();                                          //键盘扫描初始化(端口)

LCD_Init();                                             //初始化 LCD
Item_Num = GetMLNum(Menu_List01);               //获取要显示的菜单的菜单项个数
PageItem_Num = GetMLiNum_Page((unsigned char **)Menu_List01);    //获取要显示的菜单
                                                //的页数
Initial_Menu((unsigned char**)Menu_List01);
while(Exit_flag)
{
    uiKey = Key_Get();                          //获取键值
    if(uiKey)                                   //如有键按下则响应
    {
        switch(uiKey)                           //判断键值进行分支控制界面
```

```
        {
            case 3:                     //确定键按下
                Enter_Flag=1;           //enter flag set
                break;
            case 2:                     //down      向下键按下
                if(Active_Index<Item_Num)
                {
                    Active_Index++;
                    if(Temp_Index<PageItem_Num-1) Temp_Index++;
                    else if(First_Index<Item_Num) First_Index++;
                    Update_Flag = 1;
                }
                break;
            case 1:                     //up    向上键按下
                if(Active_Index>0)
                {
                    Active_Index--;
                    if(Temp_Index>0) Temp_Index--;
                    else if(First_Index>0) First_Index--;
                    Update_Flag = 1;
                }
                break;
        /*      case 4:                         //返回键按下
                Exit_flag = 0;//
                break;*/
            default:break;
        }
    }
    if(Update_Flag)                         //Update_Flag 为 1 时刷新菜单
    {
        UpDate_Menu(First_Index,Active_Index,(unsigned char**)Menu_List01);
        Update_Flag=0;
    }
    if(Enter_Flag)                          //有确定键按下时，则进入相应的功能函数
    {
        Enter_Flag = 0;
        switch(Active_Index)
        {
            case 0:                             //这里仅定义了可以响应第一项菜单
            //  Show_DemoTast();
            //  Redraw_Menu(First_Index,Active_Index,(unsigned char**)Menu_List01);
            //  Update_Flag = 1;
                break;
```

```
                    case 1: break;                  //如感兴趣可以自己加进去玩玩
                    case 2: break;
                    case 3: break;
                    case 4: break;
                    case 5: break;
                    default:break;
                }
            }
            KeyScan_Service();              //键盘扫描的服务程序,最好在 1KHz 的定时/时基中断调用
        }
```

Key_Initial()是键盘扫描程序初始化函数，用户可根据自己的键盘扫描程序来修改这块的代码，而如果在之前已经进行过键盘扫描的初始化，则此处没有必要再重新初始化；比如在一个二级的菜单响应控制代码当中，它是由上一级的菜单响应控制代码响应操作而进入的，这时就无需再初始化扫键了。

LCD_Initil 函数的调用也如此，如果是在二级菜单的响应控制代码中的话，是无需进行第二次初始化的。

接下来，调用 Menu_GUI.c 中的函数来初始化变量 Item_Num 和 PageItem_Num 的数值，注意，在调用相关的菜单 GUI 显示控制函数时，都要传递当前选择的菜单组资源到调用的函数当中的。有些编译器对数据的类型定义要求较严，这时在传递参数时需要对参数进行类型声明或强制转换。

每一个菜单组资源在使用时，在该菜单组的响应控制代码当中都要对其进行初始化，也就是调用 Initial_Menu 函数。

从前面的代码中可以看出，这份示例的菜单响应控制代码使用的菜单组资源是 Menu_List01，它的定义在前面已经介绍过了。

while(Exit_flag)定义了菜单响应控制的循环，标志变量 Exit_flag 决定着这个循环是否继续下去。

Key_Get()函数可以获取到当前有效的按键值，当然，用户使用自己定义的键盘扫描程序时，应视自己的代码而定，反正只要能够获取到有效的按键值就可以了。获取的键值存放于 uiKey 变量当中；需要注意的是，在没有键按下时，获取键值的函数应返回 0 或其它的数值以区别有效的键值。

在上面的代码当中，当有效的按键按下时，uiKey 的值为非零，这时会进行一个 Switch 的分支判断，以响应不同的按键。在代码当中，定义了 uiKey 的键值为 1、2、3 时的按键含意，也就是按键的作用；键值 1 为菜单项向上移动，键值 2 为菜单项向下移动，键值 3 为确定键，而在代码当中，有一段屏蔽掉的代码，为退出键的定义的，定义的键值为 4。

当键值为 1 时，会调整当前活动的菜单项，并置标志变量 Update_flag 为 1，让程序在循环当中刷新显示的菜单。

键值为 2 时，也调整当前活动的菜单项，当然是向下调整了，同时置标志变量 Updage_flag 为 1，以控制刷新显示的菜单。

键值为 3 时，为确定键按下，置 Enter_flag 为 1，在后面的代码当中会进行相对应的菜单项响应操作分支。

当键值分支处理代码执行结束后，循环当中会判断当前的显示是否需要刷新，即判断 Update_flag 的值，如果需要刷新，则调用 UpDate_Menu 函数。

随后的代码中，判断 Enter_flag 即确定按键是否按下，如按下，则进入菜单响应控制的分支处理；在 switch 分支中，根据 Active_Index 的值判断当前处于选择状态的菜单项是哪一个，然后进行分支处理，也就是每个 case 分支对应一个菜单项的响应，这里用户可以根据自己的需要进行代码的编写;例如在上面的代码当中就有一段屏蔽的代码,其响应第一项菜单项，调用了一个 Show_DemoTask 函数进入了一个显示界面(具体显示什么无所谓了,仅供参考)，从该函数退出后，需要调用一个 Redraw_Menu 函数，重绘当前的菜单显示，然后置显示刷新标识 Update_flag 为 1，在下一次循环中完成刷新显示。

每次循环里，都调用一个键盘扫描的函数：KeyScan_Service()，其实可以根据用户自己的键盘扫描程序架构而定的，这里仅供参考而已。

4.3.2. 订制一个有二级菜单的工程

先简单介绍一下这个有二级菜单的工程的一些要求，如下：
要求一级菜单有五项菜单项，分别为：

- 绘点
- 绘直线
- 绘矩形
- 字符演示
- 帮助

而字符演示这项的菜单响应有二级的菜单，即要求选择该项菜单项时，按下确定按键时会调出二级菜单，二级菜单要求有以下几项：

- 单个西文字符
- 西文字符串
- 中文字符
- 返回上一级

而在一级菜单的菜单项响应时，绘点这项就会在屏幕上绘制一个点，绘直线就在屏上绘制直线，基本上跟菜单中文字是意思相同的，只有“字符演示”这一项是展开二级菜单的；类似，在二级菜单当中，每一项的响应也是与其名字意义相同。
而设定以上的菜单项中的文字都是 16*16 点阵的汉字,而且事选都取好了字模放置在自定义的汉字库当中，并且都在 Font_Set 函数当中修改好了相关的配置；而定义好的汉字在字库中的序号分别为：
“绘点直线矩形字符演示帮助单个西文串中返回上一级”这些字符的序号从 0~22 依次排列；而字库的类型序号为 3。

接下来就可以进行菜单系统的设计了，首先对于 Menu_GUI 的源码，只需要修改一部分的内容就可以了，也就是菜单资源的定义，即 Menu_Resource.c 中的定义。
修改的代码如下：

```
//定义单条菜单项内容,格式有两种,一种为支持汉字库 LCD 的纯汉字菜单项,一种是西文字符与自定
//义汉字库的混合菜单项的如下：
//一:直接用汉字的字串即可,不同的编译器可能在汉字的 GB 码数据类型上有所不一样
//二:菜单项字符数,第一个字符在字库中的序号,第二个字符,....
```

```
//注：在第二种情况下,为了区分自定义汉字与 ASCII 码，特定将自定义汉字库中的汉字编码前加上
//      128 作为标识
code unsigned char Menu_String01[]={2,0+0x80,1+0x80};
code unsigned char Menu_String02[]={3,0+0x80,2+0x80,3+0x80};
code unsigned char Menu_String03[]={3,0+0x80,4+0x80,5+0x80};
code unsigned char Menu_String04[]={4,6+0x80,7+0x80,8+0x80,9+0x80};
code unsigned char Menu_String05[]={2,10+0x80,11+0x80};
code unsigned char Menu_String06[]={6,12+0x80,13+0x80,14+0x80,15+0x80,6+0x80,7+0x80};
code unsigned char Menu_String07[]={5,14+0x80,15+0x80,6+0x80,7+0x80,16+0x80};
code unsigned char Menu_String08[]={4,17+0x80,15+0x80,6+0x80,7+0x80};
code unsigned char Menu_String09[]={5,18+0x80,19+0x80,20+0x80,21+0x80,22+0x80};
//定义某一组菜单的配置，格式如下：
//{该组菜单的菜单项数目,该组菜单中汉字所选用的字符类型,该组菜单中 ASCII 码所选用的类型,
//该组菜单中每条菜单项所占用的 Y 轴大小,该组菜单中菜单项显示的 X 轴偏移位}
code unsigned char Menu_List01_Config[]={5,3,1,16,10};
code unsigned char Menu_List02_Config[]={4,3,1,16,10};
//定义一组菜单的菜单项，格式如下：
//{该组菜单所对应的配置,第一条菜单项,第二条菜单项.....}
//注：菜单组列表中菜单项的数目要与相应的配置里一至哦！
code unsigned char *Menu_List01[]=//
{(unsigned char *)Menu_List01_Config,(unsigned char *)Menu_String01,
(unsigned char *)Menu_String02,(unsigned char *)Menu_String03,
(unsigned char *)Menu_String04,(unsigned char *)Menu_String05};
code unsigned char *Menu_List02[]=//
{(unsigned char *)Menu_List02_Config,(unsigned char *)Menu_String06,
(unsigned char *)Menu_String07,(unsigned char *)Menu_String08,
(unsigned char *)Menu_String09};
```

然后在 Menu_GUI.h 文件当中，添加定义的菜单组资源数组的声明，如下：

```
extern code UCHAR *Menu_List01[];
extern code UCHAR *Menu_List02[];
```

随后可以编写响应菜单项的功能函数，其实在这里也就是一些显示的演示程序，没有太多实际的意义，仅供参考，用户根据自己设计的需要来自行设计。

一级菜单的各项菜单项响应函数大概如下：

“绘点”菜单项，对应一绘点的演示界面，代码如下：

```
void Show_DotTest(void)
{
        unsigned int Key=0;
        unsigned Exit_flag=1;
        ClrScreen(0);                   //清屏
        SetPaintMode(0,1);              //设置绘图模式及前景色
```

```
        PutPixel(0,0);                  //绘制点
        PutPixel(2,0);
        PutPixel(4,0);
        PutPixel(6,0);
        PutPixel(8,0);
        PutPixel(9,0);
        PutPixel(10,0);
        PutPixel(0,2);
        PutPixel(0,4);
        PutPixel(0,6);
        PutPixel(0,8);
        PutPixel(0,10);

        while(Exit_flag)
        {
            Key = Key_Get();
            if(Key!=0)
                Exit_flag = 0;          //任意键按下则返回
            KeyScan_Service();
        }
    }
```

“绘直线”菜单项对应的函数代码如下：

```
    void Show_LineTest(void)
    {
        unsigned int Key=0;
        unsigned Exit_flag=1;
        ClrScreen(0);                   //清屏
        SetPaintMode(0,1);              //设置绘图模式及前景色
        Line(127,63,0,63);              //绘制一条直线
        Line(10,12,10,42);
        while(Exit_flag)
        {
            Key = Key_Get();
            if(Key!=0)
                Exit_flag = 0;          //任意键按下时,退出返回
            KeyScan_Service();
        }
    }
```

“绘矩形”菜单项对应的函数源代码如下：

```
    void Show_RectanglTest(void)
```

```
{
        unsigned int Key=0;
        unsigned Exit_flag=1;
        ClrScreen(0);                       //清屏
        SetPaintMode(0,1);                  //设置绘图模式及前景色
        Rectangle(12,12,42,42,1);       //矩形填充
        Rectangle(52,12,82,42,0);       //绘制矩形框
        while(Exit_flag)
        {
            Key = Key_Get();
            if(Key!=0)
                Exit_flag = 0;              //任意键按下时,退出返回
            KeyScan_Service();
        }
}
```

“帮助”菜单项的响应函数里将显示一串字符串，源代码如下：

```
void Show_HelpTest(void)
{
        unsigned int Key=0;
        unsigned Exit_flag=1;
        ClrScreen(0);                       //清屏
        SetPaintMode(0,1);                  //设置绘图模式及前景色
        FontSet(1,1);                       //设置字体类形，字符色为 1
        PutString(5,10,"Wellcome to MzDesign!!");       //显示字符串
        PutString(5,50,"www.mzdesign.com.cn");
        Line(4,60,120,60);                  //绘制一条直线
        while(Exit_flag)
        {
            Key = Key_Get();
            if(Key!=0)
                Exit_flag = 0;              //任意键按下时,退出返回
            KeyScan_Service();
        }
}
```

“字符演示”的菜单项将来打开二级菜单，所以暂时将该菜单项的响应函数放置到后面，先将二级菜单当的各项菜单项对应的功能演示函数介绍。

二级菜单中，“单个西文字符”菜单项的响应函数定义为：Show_CharTest()，而菜单项“西文字符串”的响应函数定义为：Show_StringTest()，“中文字符”菜单项的响应函数定义为：Show_ChTest()。以上几个函数就不作详细的介绍了，而菜单项“返回上一级”的响应时无需调用函数那么复杂，可以在其的分支当中将菜单循环标志变量置为 0 即可实现。下面看一

下在这个二级菜单的响应控制代码，也就是一级菜单中的菜单项“字符演示”所对应的响应函数：

```
void CharTest_Menu(void)
{
        unsigned char uiKey=0;
        unsigned char Item_Num,Update_Flag,Enter_Flag=0;
        unsigned char PageItem_Num;
        unsigned char First_Index=0,Active_Index=0,Temp_Index=0;

        Item_Num = GetMLNum(Menu_List02);              //获取要显示的菜单的菜单项个数
        PageItem_Num = GetMLiNum_Page((unsigned char **)Menu_List02);//获取显示的菜单的页数
        Initial_Menu((unsigned char**)Menu_List02);
        while(1)
        {
            uiKey = Key_Get();                         //获取键值
            if(uiKey)                                  //如有键按下则响应
            {
                switch(uiKey)                          //判断键值进行分支控制界面
                {
                    case 3:                    //确定键按下
                        Enter_Flag=1;              //enter flag set
                        break;
                    case 2:                    //down     向下键按下
                        if(Active_Index<Item_Num)
                        {
                            Active_Index++;
                            if(Temp_Index<PageItem_Num-1) Temp_Index++;
                            else if(First_Index<Item_Num) First_Index++;
                            Update_Flag = 1;
                        }
                        break;
                    case 1:                    //up   向上键按下
                        if(Active_Index>0)
                        {
                            Active_Index--;
                            if(Temp_Index>0) Temp_Index--;
                            else if(First_Index>0) First_Index--;
                            Update_Flag = 1;
                        }
                        break;
                    default:break;
                }
            }
```

```
                if(Update_Flag)                         //Update_Flag 为 1 时刷新菜单
                {
                    UpDate_Menu(First_Index,Active_Index,(unsigned char**)Menu_List02);
                    Update_Flag=0;
                }
                if(Enter_Flag)                          //有确定键按下时，则进入相应的功能函数
                {
                    Enter_Flag = 0;
                    switch(Active_Index)
                    {
                        case 0:                         //"单个西文字符"菜单项
                            Show_CharTest();
                            Redraw_Menu(First_Index,Active_Index,(unsigned char**)Menu_List02);
                            Update_Flag = 1;
                            break;
                        case 1:                         //"西文字符串"菜单项
                            Show_StringTest();
                            Redraw_Menu(First_Index,Active_Index,(unsigned char**)Menu_List02);
                            Update_Flag = 1;
                            break;
                        case 2:                         // "中文字符"菜单项
                            Show_ChTest();
                            Redraw_Menu(First_Index,Active_Index,(unsigned char**)Menu_List02);
                            Update_Flag = 1;
                            break;
                        case 3: Exit_flag = 0;break;    //"返回上一级"菜单项
                        default:break;
                    }
                }
                KeyScan_Service();              //键盘扫描的服务程序,最好在 1KHz 的定时/时基中断调用
            }
    }
```

然后再来看看一级菜单的源代码；在此一级菜单直接在 main 函数中循环响应，仅供参考，如下：

```
    void main(void)
    {
            unsigned char uiKey=0;
            unsigned char Item_Num,Update_Flag,Enter_Flag=0;
            unsigned char PageItem_Num;
            unsigned char First_Index=0,Active_Index=0,Temp_Index=0;

            Key_Initial();                                          //键盘扫描初始化(端口)
```

```
LCD_Init();                                                    //初始化 LCD
Item_Num = GetMLNum(Menu_List01);              //获取要显示的菜单的菜单项个数
PageItem_Num = GetMLiNum_Page((unsigned char **)Menu_List01);//获取显示的菜单的页数
Initial_Menu((unsigned char**)Menu_List01);
while(1)
{
    uiKey = Key_Get();                         //获取键值
    if(uiKey)                                  //如有键按下则响应
    {
        switch(uiKey)                          //判断键值进行分支控制界面
        {
            case 3:                    //确定键按下
                Enter_Flag=1;              //enter flag set
                break;
            case 2:                    //down     向下键按下
                if(Active_Index<Item_Num)
                {
                    Active_Index++;
                    if(Temp_Index<PageItem_Num-1) Temp_Index++;
                    else if(First_Index<Item_Num) First_Index++;
                    Update_Flag = 1;
                }
                break;
            case 1:                    //up   向上键按下
                if(Active_Index>0)
                {
                    Active_Index--;
                    if(Temp_Index>0) Temp_Index--;
                    else if(First_Index>0) First_Index--;
                    Update_Flag = 1;
                }
                break;
            default:break;
        }
    }
    if(Update_Flag)                            //Update_Flag 为 1 时刷新菜单
    {
        UpDate_Menu(First_Index,Active_Index,(unsigned char**)Menu_List01);
        Update_Flag=0;
    }
    if(Enter_Flag)                             //有确定键按下时，则进入相应的功能函数
    {
        Enter_Flag = 0;
```

```
                    switch(Active_Index)
                    {
                        case 0:                                 //"绘点"菜单项
                            Show_DotTest();
                            Redraw_Menu(First_Index,Active_Index,(unsigned char**)Menu_List01);
                            Update_Flag = 1;
                            break;
                        case 1:                                 //"绘直线"菜单项
                            Show_LineTest();
                            Redraw_Menu(First_Index,Active_Index,(unsigned char**)Menu_List01);
                            Update_Flag = 1;
                            break;
                        case 2:                                 //"绘矩形"菜单项
                            Show_ RectanglTest();
                            Redraw_Menu(First_Index,Active_Index,(unsigned char**)Menu_List01);
                            Update_Flag = 1;
                            break;
                        case 3:                                 //进入二级菜单
                            CharTest_Menu();
                            Redraw_Menu(First_Index,Active_Index,(unsigned char**)Menu_List01);
                            Update_Flag = 1;
                            break;
                        case 4:                                 //"帮助"菜单项
                            Show_HelpTest();
                            Redraw_Menu(First_Index,Active_Index,(unsigned char**)Menu_List01);
                            Update_Flag = 1;
                            break;
                        default:break;
                    }
                }
                KeyScan_Service();              //键盘扫描的服务程序,最好在 1KHz 的定时/时基中断调用
            }
        }
```

到这里，基本上就完成了在 Mz_MenuGUI 的基础之上搭建一个具有二级菜单的程序框架；其实介绍它的目的在于分绍一种根据显示界面而定下程序框架的编程方法，并非在于手把手的教读者去设计一个这样的程序，这点希望读者明白，重要的是编程的思想不在于步骤。

5. 移植通用版 LCD 驱动程序到另一颗 MCU

将通用版的 LCD 驱动程序移植到另外的 MCU 上并不复杂，而需要做的工作也很少，在前面介绍驱动程序代码时已经介绍过了，基本上只需要修改驱动当中与 MCU 相关的代码，主要就是 LCD_Driver_User.c 和驱动的端口配置文件 LCD_PortConfig.h 文件。本章将以凌阳公司的 SPCE061A 单片机作为移植的对像，下面一步一步分析。

5.1. 修改驱动中的底层代码

与 MCS51 系列 MCU 不同，凌阳公司的 SPCE061A 属于流行的 SOC 概念的 MCU，特征之一就是将众多以前放置在 MCU 外部的硬件资源集成到 MCU 当中，而且不将部线拉出 MCU 外部，不过这些都与本驱动程序没太多关系，感兴趣的读者可以在网络上查找相关的资料参考，不作过多介绍。

5.1.1. 修改 LCD_PortConfig.h 的端口配置

SPCE061A 是不支持端口的位操作指令的，所以对于端口的操作只能利用对端口的读—改—写方式进行，即读回当前端口的输出状态，然后根据需要改变输出状态的端口来修改读回的值，然后再将改后的值输出至端口。
以下便是修改后的端口配置头文件：

```
//note:如果您使用 C 语言编写 LCD 的底层接口程序的话，这里的定义才会有用
// this file for MCU I/O port or the orther`s hardware config
// for LCD Display
//Define the MCU Register
#define     P_IOA_Data          (volatile unsigned int *)0x7000
#define     P_IOA_Buffer        (volatile unsigned int *)0x7001
#define     P_IOA_Dir           (volatile unsigned int *)0x7002
#define     P_IOA_Attrib        (volatile unsigned int *)0x7003
#define     P_IOA_Latch         (volatile unsigned int *)0x7004
#define     P_IOB_Data          (volatile unsigned int *)0x7005
#define     P_IOB_Buffer        (volatile unsigned int *)0x7006
#define     P_IOB_Dir           (volatile unsigned int *)0x7007
#define     P_IOB_Attrib        (volatile unsigned int *)0x7008

// Define for the port use by LCD Driver
#define LCD_EP                  0x1000
#define LCD_RW                  0x0800
#define LCD_A0                  0x0400
#define LCD_RE                  0x0200
#define LCD_CS                  0x0100
```

```
#define LCD_CMD_Dir             P_IOA_Dir
#define LCD_CMD_Attrib          P_IOA_Attrib
#define LCD_CMD_Buffer          P_IOA_Buffer

#define LCD_Data_BUS_Out        P_IOA_Buffer
#define LCD_Data_BUS_In         P_IOA_Data
#define LCD_Data_BUS_Dir        P_IOA_Dir
#define LCD_Data_BUS_Attrib     P_IOA_Attrib

#define LCD_Data_BUS_Byte       0//1
```

在代码当中，对 SPCE061A 的端口的寄存器进行了定义，其实是与凌阳公司提供的头文件里是一样的，只不过这里将其单独提出来，免得再包含头文件了。

接着定义了一些常量，这些定义会在程序当中使用，在操作 LCD 的控制端口时使用；可以从定义中看出 LCD 的控制端口到底接在 MCU 的哪个端口上；而 LCD_CMD_Dir、LCD_CMD_Attrib 以及 LCD_CMD_Buffer 对接在 LCD 上的控制端口进行了重定义。
以上可以看出，LCD 的控制端口：CS 片选、Reset 复位端口、A0 命令数据选择端口、RW 读写控制位和 EP 端口分别接在了 SPCE061A 的 IOA8、9、10、11、12 之上。

然后又重定义了 LCD_Data_BUS_Out 等四个用于 LCD 数据端口的寄存器；最后还定义了一个宏：LCD_Data_BUS_Byte，该定义的值为 0 或者 1，分别表示 LCD 的 8 个数据端口接在 SPCE061A 的端口中的低八位或者是高八位（SPCE061A 的端口每一组为 16 个，与其 CPU 的字长是一样的对应）；代码中的定义表示，LCD 的 8 个数据端口接在 SPCE061A 的 IOA0~7 端口之上。

5.1.2. 修改底层驱动功能函数

LCD_Driver_User.c 当中定义有 LCD 的底层驱动功能函数，主要是写数据、读数据和写寄存器函数，当然也有一些函数可能被修改，下面来看看修改的代码：
LCD_DataWrite 函数：

```
//=============================================================================
// 函数: void LCD_DataWrite(unsigned int Data)
// 描述: 写一个字节的显示数据至 LCD 中的显示缓冲 RAM 当中
// 参数: Data 写入的数据
// 返回: 无
// 版本:
//      2007/01/09          First version
//=============================================================================
void LCD_DataWrite(unsigned int Data)
{
        unsigned int uiTemp=0;
        uiTemp = *LCD_CMD_Buffer;
```

```
        uiTemp &= ~(LCD_EP+LCD_CS+LCD_RW);    //EP CS RW to Low
        uiTemp |= LCD_A0;                     //AO Hight
        *LCD_CMD_Buffer = uiTemp;
#if LCD_Data_BUS_Byte==1
        *LCD_Data_BUS_Out = (*LCD_Data_BUS_Out&0x00ff)|(Data<<8);
#else
        *LCD_Data_BUS_Out = (*LCD_Data_BUS_Out&0xff00)|(Data&0x00ff);
#endif
        uiTemp = *LCD_CMD_Buffer;
        uiTemp |= LCD_EP;                     //EP to Hight
        *LCD_CMD_Buffer   = uiTemp;
        uiTemp |= LCD_CS;                     //CS to Hight
        *LCD_CMD_Buffer = uiTemp;
}
```

其实还是照着时序来写程序，只是换了种端口的操作方式罢了；先从 LCD_CMD_Buffer（重定义为 P_IOA_Buffer 寄存器）当中读出当前 MCU 端口的输出状态，然后以读回的数据进行与、或操作，这样就可以置相对应的控制端口的位所想要的输出状态了，最后再将修改后的数据送至端口输出。

宏“#if LCD_Data_BUS_Byte==1”以及其以下的两个条件宏定义，都是在 LCD_PortConfig.h 当中定义的那个 LCD_Data_BUS_Byte 的值来进行选择性的编译；代码中可看出，当 LCD_Data_BUS_Byte 定义为 1 时，将编译代码“ *LCD_Data_BUS_Out = (*LCD_Data_BUS_Out&0x00ff)|(Data<<8);”，可知 LCD 的数据端口是定义在高八位的；为 0 时则编译另外一条代码。

接下来的两个基本的接口函数也比较类似，如下：

```
//=============================================================================
// 函数: unsigned int LCD_DataRead(void)
// 描述: 从 LCD 中的显示缓冲 RAM 当中读一个字节的显示数据
// 参数: 无
// 返回: 读出的数据，
// 版本:
//        2007/01/09        First version
// 注意:
//=============================================================================
unsigned int LCD_DataRead(void)
{
        unsigned int uiTemp=0;
        unsigned int Bus_Dir=0;
        unsigned int Read_Dat=0;
        Bus_Dir = *LCD_Data_BUS_Dir;                //设置数据口为输入
#if LCD_Data_BUS_Byte==1
        *LCD_Data_BUS_Dir = Bus_Dir&0x00ff;
```

```
#else
        *LCD_Data_BUS_Dir = Bus_Dir&0xff00;
#endif
        uiTemp = *LCD_CMD_Buffer;
        uiTemp &= ~(LCD_EP+LCD_CS);             //EP CS    to Low
        uiTemp |= (LCD_A0+LCD_RW);              //AO RW Hight
        *LCD_CMD_Buffer = uiTemp;

        uiTemp |= LCD_EP;                       //EP to Hight
        *LCD_CMD_Buffer   = uiTemp;
        uiTemp &= (~LCD_EP);
        *LCD_CMD_Buffer   = uiTemp;
        Read_Dat = *LCD_Data_BUS_In;
        uiTemp |= LCD_CS;                       //CS to Hight
        *LCD_CMD_Buffer = uiTemp;
        *LCD_Data_BUS_Dir = Bus_Dir;
#if LCD_Data_BUS_Byte==1
        Read_Dat = Read_Dat>>8;
#else
        Read_Dat = Read_Dat&0x00ff;
#endif
        return Read_Dat;
}
//=============================================================================
// 函数: void LCD_RegWrite(unsigned int Command)
// 描述: 写一个字节的数据至 LCD 中的控制寄存器当中
// 参数: Command        写入的数据，低八位有效（byte）
// 返回: 无
// 版本:
//       2007/01/09        First version
//=============================================================================
void LCD_RegWrite(unsigned int Command)
{
        unsigned int uiTemp=0;
        uiTemp = *LCD_CMD_Buffer;
        uiTemp &= ~(LCD_EP+LCD_CS+LCD_RW+LCD_A0);    //EP CS RW A0 to Low
        *LCD_CMD_Buffer = uiTemp;
#if LCD_Data_BUS_Byte==1
        *LCD_Data_BUS_Out = (*LCD_Data_BUS_Out&0x00ff)|(Command<<8);
#else
        *LCD_Data_BUS_Out = (*LCD_Data_BUS_Out&0xff00)|(Command&0x00ff);
#endif
```

```
        uiTemp = *LCD_CMD_Buffer;
        uiTemp |= LCD_EP;                       //EP to Hight
        *LCD_CMD_Buffer   = uiTemp;
        uiTemp |= LCD_CS;                       //CS to Hight
        *LCD_CMD_Buffer = uiTemp;
}
```

在上面的代码中，有一点要注意一下，SPCE061A 单片机的最小的数据存储单元是 16 位长的“word”，而在 C 语言程序当中如果定义有 char 类型的数据，实际上也是占用 16 位长的，在 SPCE061A 的 C 语言里，char 与 int 没有什么区别。

另外，SPCE061A 的端口必需要使用之前对其进行初始化配置，也就是要定义好它的端口是输入还是输出之类的；为此，可以在 LCD 的初始化函数当中对这些端口进行初始化设置，下面定义了一个用于端口初始化的函数如下：

```
//=============================================================================
// 函数: void LCD_PortInit(void)
// 描述: 与 LCD 连接的端口初始化代码
// 参数: 无
// 返回: 无
// 版本:
//      2007/01/09      First version
// 注意:
//=============================================================================
void TimeDelay(int Time);
void LCD_PortInit(void)
{
#if LCD_Data_BUS_Byte==1
        *LCD_Data_BUS_Dir = *LCD_Data_BUS_Dir|0xff00;
        *LCD_Data_BUS_Attrib = *LCD_Data_BUS_Attrib|0xff00;
        *LCD_Data_BUS_Out = *LCD_Data_BUS_Out|0xff00;
#else
        *LCD_Data_BUS_Dir = *LCD_Data_BUS_Dir|0x00ff;
        *LCD_Data_BUS_Attrib = *LCD_Data_BUS_Attrib|0x00ff;
        *LCD_Data_BUS_Out = *LCD_Data_BUS_Out|0x00ff;
#endif
        *LCD_CMD_Dir |= (LCD_EP+LCD_A0+LCD_RW+LCD_CS+LCD_RE);
        *LCD_CMD_Attrib |= (LCD_EP+LCD_A0+LCD_RW+LCD_CS+LCD_RE);
        *LCD_CMD_Buffer |= (LCD_EP+LCD_A0+LCD_RW+LCD_CS+LCD_RE);
        *LCD_CMD_Buffer &= (~LCD_RE);
        TimeDelay(200);
        *LCD_CMD_Buffer |= LCD_RE;
        TimeDelay(50);
}
```

初始化代码中将 SPCE061A 的 IOA 低八位定义为输出口，并初始其输出都为 1；而控制端口 IOA8、9、10、11、12 都定义为输出口，并初始化其输出为高；最后在接 LCD 复位端口的 LCD_RE 端口上输出一个低电平的脉冲，使 LCD 模块进行复位。

而在 LCD_Init 函数上也稍稍作一下修改，也就是调用一下刚才介绍的端口初始化函数而已：

```
void LCD_Init(void)
{
        //LCD 驱动所使用到的端口的初始化（如果有必要的话）
        LCD_PortInit();

        LCD_RegWrite(M_LCD_ON);                 //LCD On
        LCD_RegWrite(M_LCD_POWER_ALL);          //设置上电控制模式

        LCD_RegWrite(M_LCD_ELE_VOL);            //电量设置模式（显示亮度）
        LCD_RegWrite(0x1f);                     //指令数据 0x0000~0x003f

        LCD_RegWrite(M_LCD_VDD_SET);            //V5 内部电压调节电阻设置
        LCD_RegWrite(M_LCD_VDD);
    ……
```

5.2. 与编译器相关的修改

SPCE061A 使用凌阳提供的 unSP IDE 编译环境，在常量定义方面与 Keil C51 的常量定义有些不一样，需要对代码中的一些涉及常量定义的地方修改一下，也就是将“code unsigned char”前面的“code”改成“const”即可。如在 LCD_ASCII.c 中西文字库的定义改成：

```
const unsigned char Asii0610[] =
{
//--  MS Gothic8;  此字体下对应的点阵为：宽 x 高=6x10    --
//--  宽度不是 8 的倍数，现调整为：宽度 x 高度=8x10   --
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,……
```

至此，基本上代码的移植算是完成了。

不过，有一点还是建议去改的，SPCE061A 在定义 char 型数据时，实际占用的是 16 位长的“word”，而在 LCD 驱动当中的字库都定义为 8 位长度的单元，相当浪费存储空间。

所以，要借助一些工具修改一下驱动中定义的常量，将两个 8 位长度的数据装在一个 16 位的“word”当中。（可以利用 Ultra Edit 工具的纵向选择功能，对字库中的数据进行修改）

修改后的字库如下（片断）：

```
const unsigned char Asii0610[] =
{
/*--  MS Gothic8;  此字体下对应的点阵为：宽 x 高=6x10    --*/
/*--  宽度不是 8 的倍数，现调整为：宽度 x 高度=8x10   --*/
```

```
0x0000,0x0000,0x0000,0x0000,0x0000,0x0040,0x4040,0x4040,0x0060,0x0000,
0xA0A0,0xA000,0x0000,0x0000,0x0000,0x0028,0x28FC,0x50FC,0x5050,0x0000,
0x2070,0xA8A0,0x7028,0xA870,0x2000,0x00C8,0xD0D0,0x2058,0x5898,0x0000,
0x0020,0x5020,0x6098,0x9068,0x0000,……
```

这样修改之后，实际上字库当中每个字符的字模数据所占用的存储单元数量已经发生了变化，所以也需要对 LCD_Dis.c 当中的 FonSet 函数进行修改，如下：

```
void FontSet(unsigned char Font_NUM,unsigned char Color)
{
        switch(Font_NUM)
        {
            case 1: Font_Wrod = 5;//10;     //ASII 字符 B
                    X_Witch = 6;
                    Y_Witch = 10;
                    Char_Color = Color;
                    Char_TAB = (unsigned char *)(Asii0610 - (32*5));//10));
            break;
            case 2: Font_Wrod = 24;//48;    //汉字 A
                    X_Witch = 17;
                    Y_Witch = 16;
                    Char_Color = Color;
                    Char_TAB = (unsigned char *)GB1716;
            break;
            default: break;
        }
}
```

然后，在 PutChar 函数当中作一些修改，以便于程序在 16 位长度的数据当中提取出 8 位长度的 byte 数据，如下：

```
void PutChar(unsigned char x,unsigned char y,char a)
{
        unsigned char i,j;              //数据暂存
        unsigned char *p_data;
        unsigned char Temp;
        unsigned char Index = 0;
        p_data = Char_TAB + a*Font_Wrod; //要写字符的首地址
        j = 0;
        while((j ++) < Y_Witch)
        {
            if(y > Dis_Y_MAX) break;
            i = 0;
            while(i < X_Witch)
```

```
                {
                    if((i&0x07)==0)
                    {
                        Temp = *(p_data + (Index>>1));
                        if((Index&0x01)==0)Temp = Temp>>8;
        //              Temp = *(p_data+Index);
                        Index++;
                    }
                    if((Temp & 0x80) > 0) Write_Dot_LCD/*Writ_Dot*/(x+i,y,Char_Color);
                    Temp = Temp << 1;
                    if((x+i) >= Dis_X_MAX)
                    {
                        Index += (X_Witch-i)>>3;
                        break;
                    }
                    i++;
                }
                y ++;
            }
    }
```

6. 移植通用版 LCD 驱动程序到另一块 LCD

将驱动程序移植至另外一块 LCD 其实也很简单，一般只需要修改几个函数，都是在 LCD_Driver_User.c 文件当中，原理上，也就是按照新 LCD 的接口时序要求，重写 LCD_DataWrite、LCD_RegWrite 和 LCD_DataRead 函数；然后根据新的 LCD 的内部控制寄存器的特点，重写相关的函数，如 LCD_Init、Write_Dot_LCD、LCD_Fill 函数等。
当然，LCD 的配置文件 LCD_Config.h 也是需要作一定的改动的，如果必要的话。

除此之外，如果新的 LCD 在控制方式上有较大不同的话，也可以在底层适当增加一些函数，以便能够更快速的操作 LCD，原则上是尽可能利用 LCD 的特性。

在此，就不再详细介绍别的 LCD 应用本书所述的通用版驱动程序的情况了，希望读者理解的是程序的设计思想和架构，有兴趣的朋友可以参考网站上提供的其它代码，有疑问时可以随时联系，谢谢！

当然，本书至此，已经与原来规划的目录有所出入了，其实是在写的过程当中作了一些调整，可能还会再写一本，将 LCD 的驱动进行进阶的分析，界时将 LCD 提升至彩色，并将原来规划的目录的内容移置下一本书当中，如图像显示的内容等。

谢谢观赏

写完了，挺累的~~~~~

2007 年 10 月 31 日

第一版

Edit by 小丑~~~

xinqiang@mzdesign.com.cn

B.Regards~