嵌入式系统原理与实验

Cortex M3 实验教程（下）

上海交通大学 嵌入式系统原理与实验课程组

2012. 4

目　　录

第七章　I2C 总线和米字管

7.1　I2C 总线

7.1.1　I²C 总线定义

Inter-IC 总线又称 I²C 总线是一种简单的双向一线制串行通信总线。多个符合 I²C 总线标准的器件都可以通过同一条 I²C 总线进行通信，而不需要额外的地址译码器。I²C 总线应用中主要涉及如下几个基本概念：

- 发送器：本次传送中发送数据（不包括地址和命令）到总线的器件。
- 接收器：本次传送中从总线接收数据（不包括地址和命令）的器件。
- 主机：初始化发送、产生时钟信号和终止发送的器件，它可以是发送器或接收器。主机通常是微控制器。
- 从机：被主机寻址的器件，它可以是发送器或接收器。

7.1.2　信号线与连接方式

I2C 总线仅使用两个信号：SDA 和 SCL。 SDA 是双向串行数据线，SCL 是双向串行时钟线。当 SDA 和 SCL 线为高电平时，总线为空闲状态。

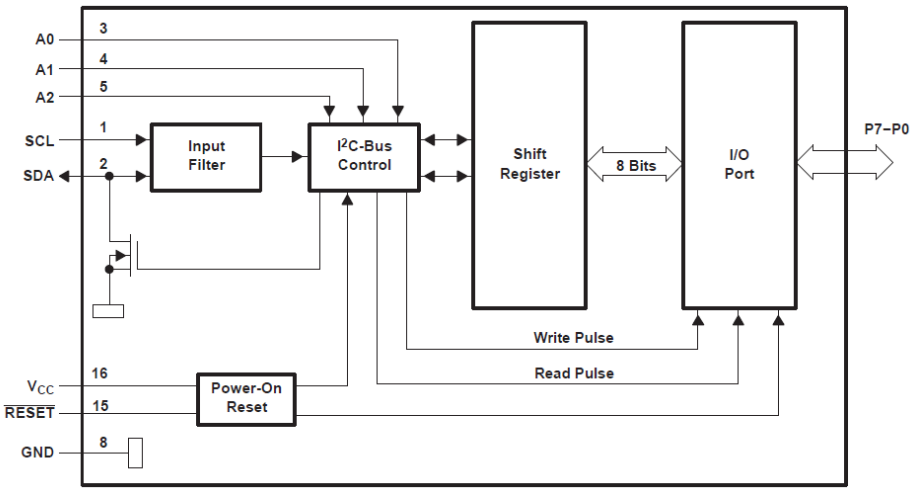


图 7-1　I²C 总线连接形式

7.1.3　数据有效性

在时钟 SCL 的高电平期间，SDA 线上的数据必须保持稳定。SDA 仅可在时钟 SCL 为低电平时改变，如图 7-2 所示。

7.1.4　起始和停止条件

I²C 总线的协议定义了两种状态：起始和停止。当 SCL 为高电平时，在

SDA 线上从高到低的跳变被定义为起始条件；而当 SCL 为高电平时，在 SDA 线上从低到高的跳变则被定义为停止条件。总线在起始条件之后被看作为忙状态。总线在停止条件之后被看作为空闲。如图 7-3 所示。

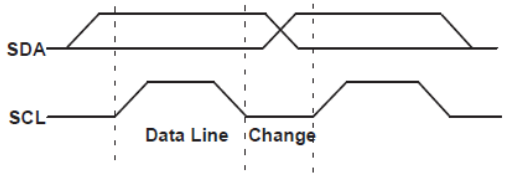


图 7-2　I^2C 总线的数据有效性

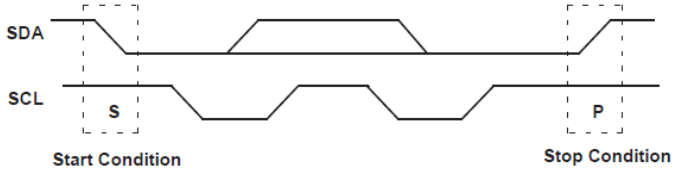


图 7-3　I^2C 总线起始条件和停止条件

SDA 线上的每个字节必须为 8 位长。不限制每次传输的字节数。每个字节后面必须带有一个应答位。数据传输时 MSB 在前。当接收器不能接收另一个完整的字节时，它可以将时钟线 SCL 拉到低电平，以迫使发送器进入等待状态。当接收器释放时钟 SCL 时继续进行数据传输。

7.1.5　带有 7 位地址的数据格式

数据传输的格式如图 7-4 所示。从机地址在起始条件之后发送。该地址为 7 位，后面跟的第 8 位是数据方向位，这个数据方向位决定了下一个操作是接收（高电平）还是发送（低电平），0 表示传输（发送）；1 表示清求数据（接收）。

数据传输始终由主机产生的停止条件来中止。然而，通过产生重复的起始条件和寻址另一个从机（而无需先产生停止条件），主机仍然可以在总线上通信。因此，在这种传输过程中可能会有接收/发送格式的不同组合。

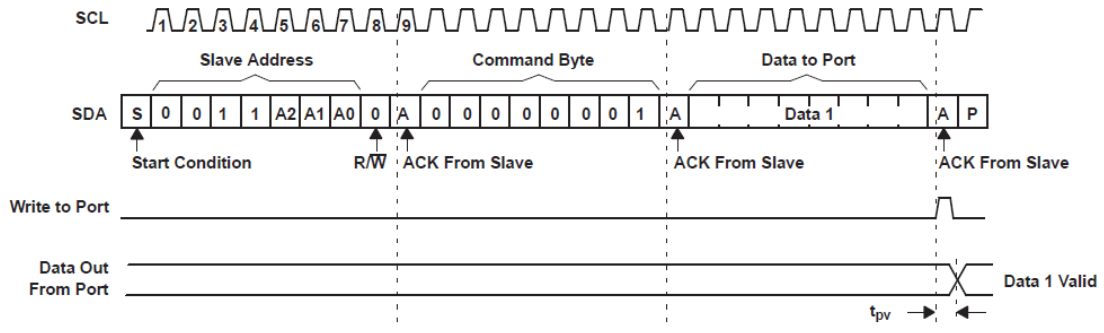


Figure 6. Write to Output Port Register

图 7-4　带 7 位地址的完整数据传输

首字节的前面 7 位组成了从机地址（见图 7-5）。第 8 位决定了消息的方向。首字节的 R/S 位为 0 表示主机将向所选择的从机发送信息。该位为 1 表示主机将接收来自从机的信息。

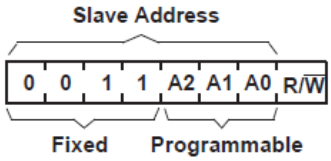


图 7-5 在第一个字节的 R/S 位

带有 I²C 总线的器件除了有从机地址（Slave Address）外，还有数据地址（也称子地址）。从机地址是指该器件在 I²C 总线上被主机寻址的地址，而数据地址是指该器件内部不同部件或存储单元的编址。

数据地址实际上也是像普通数据那样进行传输的，传输格式仍然是与数据相统一的，区分传输的到底是地址还是数据要靠收发双方具体的逻辑约定。数据地址的长度必须由整数个字节组成，可能是单字节，也可能是双字节，还可能是 4 字节，这要看具体器件的规定。

7.2 I2C 功能概述

7.2.1 SCL 时钟速率

I²C 总线时钟速率由以下参数决定：

- CLK_PRD：系统时钟周期；
- SCL_LP：SCL 低电平时间（固定为 6）；
- SCL_HP：SCL 高电平时间（固定为 4）；
- TIMER PRD：位于寄存器 I²CMTPR（I²C Master Timer Period）单的可编程值。

I²C 时钟周期的计算方法如下：

SCL_PERIOD = 2×（1+TIMER_PRD）×（SCL_LP+SCL_HP）×CLK_PRD

例如：CLK_ PRD=50ns（系统时钟为 20MHz），TIMER_ PRD=2，SCL_LP=6，SCL_ HP=4，则 SCL_PERIOD 为 3μs，即 333KHz。

7.2.2 中断控制

I²C 总线能够在观测到以下条件时产生中断：

- 主机传输完成
- 主机传输过程中出现错误
- 从机传输时接收到数据
- 从机传输时收到主机的清求

对 I²C 主机模块和从机模块来说，这是独立的中断信号。但两个模块都能产生多个中断时，仅有单个中断信号被送到中断控制器。

- I²C 主机中断

当传输结束（发送或接收）、或在传输过程中出现错误时，I²C 主机模块产生一个中断。调用 I2CMasterIntEnable()函数（参见表 7-12）可使能 I²C 主

机中断。当符合中断条件时，软件必须通过 I2CMasterErr()函数（参见表 7-11）检查来确认错误不是在最后一次传输中产生。

如果最后一次传输没有被从机应答或如果主机由于与另一个主机竞争时丢失仲裁而被强制放弃总线的所有权，那么会发出一个错误条件。如果没有检测到错误，则应用可继续执行传输。可通过 I2CMasterIntClear()函数（参见表 7-15）来清除中断状态。

如果应用不要求使用中断（即基于轮询的设计方法），那么原始中断状态总是可以通过函数调用 I2CMasterIntStatus(false)来观察到（参见表 7-14）。

- I2C 从机中断

从机模块在它接收到来自 I^2C 主机的请求时产生中断。调用 I2CS1aveIntEnable()函数（参见表 7-22）可使能 I^2C 主机中断。软件通过调用 I2CS1aveStatus()函数（参见表 7-26）来确定模块是否应该写入（发送）数据或读取（接收）数据。通过调用 I2CS1aveIntClear()函数（参见表 7-27）来清除中断。

如果应用不要求使用中断（即基于轮询的设计方法），那么原始中断状态总是可以函数调用 I2CS1aveIntStatus（false）来观察到。

7.2.3 主机命令序列

I^2C 模块在主机模式下有多种收发模式：

- 主机单次发送：S |SLA+W | data | P
- 主机单次接收：S | SLA+R | data | P
- 主机突发发送：S | SLA+W | data | P
- 主机突发接收：S | SLA+R | data | | P
- 主机突发发送后主机接收：S | SLA+W | data | | Sr | SLA+R | | P
- 主机突发接收后主机发送：S | SLA+R | data | | Sr |SLA+W | | P

在传输格式单，S 为起始条件、P 为停止条件、SLA+W 为从机地址加写操作、SLA+R 为从机地址加读操作、data 为传输的有效数据，Sr 为重复起始条件（在物理波形上等同与 S）。在单次模式中每次仅能传输一个字节的有效数据，而在突发模式中一次可以传输多个字节的有效数据。在实际应用当中以“主机突发发送”和“主机突发发送后主机接收”这两种模式最为常见。

7.2.4 从机状态控制

当 I^2C 模块作为总线上的从机时，收发操作仍然是由（另外的）主机控制的。当从机被寻址到时会触发中断，将被要求接收或发送数据。通过调用函数 I2CS1aveStatus()，就能获得主机的操作要求，有以下几种情况：

- 主机已经发送了第 1 个字节：该字节应当被视为数据地址（或数据地址首字节）；
- 主机已经发送了数据：应当及时读取该数据（也可能是数据地址后继字节）；
- 主机要求接收数据：应当根据数据地址找到存储的数据然后回送给主机。

7.3 板上 I2C 资源及特性

TLV320AIC238 芯片、ADXL345 芯片、PCA9557 芯片上均有 I^2C 总线接口。每条总线有一条 SDA 双向串行数据线和一条 SCL 双向串行时钟线。数据传输速率可根据需要选择高速或低速。

I2C 总线具有如下特性：

- 总线仅由 2 根信号线组成

可节省芯片 I/O、节省 PCB 面积、节省线材成本，等等。

- 总线协议简单容易实现

协议的基本部分相当简单，因此在芯片内部以硬件的方法实现 I^2C 部件的逻辑是很容易的。即使 MCU 内部没有硬件的 I^2C 总线接口，也能够方便地利用开漏的 I/O（如果没有，可用准双向 I/O 代替）来模拟实现。

- 支持的器件多

主流半导体公司生产的大量器件带有 I^2C 总线接口，这为应用工程师设计产品时选择合适的 I^2C 器件提供了广阔的空间。在现代微控制器设计当中 I^2C 总线接口已经成为标准的重要片内外设之一。

- 总线上可同时挂接多个器件

同一条 I^2C 总线上可以挂接多个器件，器件之间靠不同的编址来区分的，不需要附加的 I/O 线或地址译码部件。

- 总线可裁减性好

在原有总线连接的基础上可以随时新增或者删除器件。用软件可以很容易实现 I^2C 总线的自检功能，能够及时发现总线上的变动。

- 总线电气兼容性

I^2C 总线规定器件之间以开漏 I/O 相连接，只要选取适当的上拉电阻即能实现不同逻辑电平之间的互联通信，而不需要额外的转换。

- 支持多种通信方式

一主多从、多主机通信及广播模式等。

- 通信速率高并兼顾低速通信

I^2C 总线标准传输速率为 100kbps（每秒 100k 位）。在快速模式下为 400kbps。按照后来修订的版本，位速率可高达 3.4Mbps。

I^2C 总线的通信速率也可以低至几 kbps 以下，用以支持低速器件（比如软件模拟的实现）或者用来延长通信距离。从机也可以在接收和响应一个字节后使 SCL 线保持低电平迫使主机进入等待状态直到从机准备好下一个要传输的字节。

- 有一定的通信距离

I^2C 总线通信距离通常为几米到十几米。通过降低传输速率、屏蔽、中继等办法，通信距离可延长到数十米乃至数百米以上。

7.4 Stellaris 库函数

7.4.1 主机模式收发控制

函数 I2CMasterInitExpClk()用来初始化 I^2C 模块为主机模式，并选择通信

速率为 100kbps 的标准模式还是 400kbps 的快模式，但在实际编程时常常以更方便的宏函数 I²CMasterInit()来代替。详见表 7-1 和表 7-2 的描述。

函数 I2CMasterEnable()和 I2CMasterDisable()用来使能或禁止主机模式下总线的收发。参见表 7-3 和表 7-4 的描述。函数 I2CMasterControl()用来控制 I²C 总线在主模式下收发数据的各种总线动作。在控制总线收发数据之前要调用函数 I2CMasterslaveAddrSet()来设置器件地址和读写控制位，如果是要发送数据，还要调用函数 I2CMasterDataPut()来设置首先发送的数据字节（应当是数据地址）。在总线接收到数据后，要通过函数 I2CMasterDataGet()来及时读取收到的数据。详见表 7-5~7-8 的描述。

函数 I2CMasterBusy()用来查询主机当前的状态是否忙，而 I2CMasterBusBusy()用来确认在多机通信当中是否有其它主机正在占用总线。参见表 7-9 和表 7-10 的描述。

在 I²C 主机通信过程中可能会遇到一些错误情况，如被寻址的器件不存在、发送数据时从机没有应答等等，都可以通过调用函数 I2CMasterErr()来查知。详见表 7-11 的描述。

表 7-1 函数 I2CMasterInitExpClk()

功能	I2C 主机模块初始化（要求提供明确的时钟速率）
原型	void I2CMasterInitExpClk(unsigned long ulBase, unsigned long uII2CClk, tBoolean bFast)
参数	ulBase ： I2C 主机模块的基址 uII2CClk ： 提供给 I2C 模块的时钟速率，即系统时钟频率 bFast ： 取值 false 以 100kbps 标准位速率传输数据，取值 true 以 400kbps 快模式传输数据
返回	无

表 7-2 宏函数 I2CMasterInit()

功能	I2C 主机模块初始化
原型	#define I2CMasterInit(a, b) I2CMasterInitExpClkk(a, SysCtlClockGet( ), b)
参数	参见表 7-1
返回	无

表 7-3 函数 I2CMasterEnable()

功能	使能 I2C 主机模块
原型	void I2CMasterEnable(unsigned long ulBase)
参数	ulBase ： I2C 主机模块的基址
返回	无

表 7-4 函数 I2CMasterDisable()

功能	禁止 I2C 主机模块
原型	void I2CMasterDisable（unsigned long ulBase）
参数	ulBase ： I2C 主机模块的基址
返回	无

表 7-5 函数 I2CMasterSlaveAddrSet()

功能	设置 I2C 主机将要放到总线上的从机地址
原型	void I2CMasterSlaveAddrSet（unsigned long ulBase, unsigned char ucSlaveAddr, tBoolean

	bReceive）
参数	ulBase ： I2C 主机模块的基址 ucSlaveAddr ： 7 位从机地址(这是纯地址，不含读/写控制位) bReceive ： 取值 false 表示主机将要写数据到从机，取值 true 表示主机将要从从机读取数据
返回	无
说明	本函数仅仅是设置将要发送到总线上的从机地址，而并不会真正在总线上产生任何动作

表 7-6 函数 I2CMasterDataPut()

功能	从主机发送一个字节
原型	void I2CMasterDataPut（unsigned long ulBase, unsigned char ucData）
参数	ulBase ： I2C 主机模块的基址 ucData ： 待发送数据
返回	无
说明	本函数实际上并不会真正发送数据到总线上，而是将待发送的数据存放在个数据寄存器里

表 7-7 函数 I2CMasterDataGet()

功能	接收一个已经发送到主机的字节
原型	unsigned long I2CMasterDataGet（unsigned long ulBase）
参数	ulBase ： I2C 主机模块的基址
返回	接收到的字节（自动转换为长整型）

表 7-8 函数 I2CMasterControl()

功能	控制主机模块在总线上的动作
原型	void I2CMasterControl（unsigned long ulBase, unsigned long ulCmd）
参数	ulBase ： I2C 主机模块的基址 u1Cmd ： 向主机发出的命令，取下列值之一 I2C_MASTER_CMD_SINGLE_SEND//单次发送 I2C_MASTER_CMD_SINGLE_RECEIVE//单次接收 I2C_MASTER_CMD_BURST_SEND_START//突发发送起始 I2C_MASTER_CMD_BURST_SEND_CONT//突发发送继续 I2C_MASTER_CMD_BURST_SEND_FINISH//突发发送完成 I2C_MASTER_CMD_BURST_SEND_ERROR_STOP//突发发送遇错误停止 I2C_MASTER_CMD_BURST_RECEIVE_START//突发接收起始 I2C_MASTER_CMD_BURST_RECEIVE_CONT//突发接收继续 I2C_MASTER_CMD BURST_RECEIVE_FINISH//突发接收完成 I2C_MASTER_CMD_BURST_RECEIVE_ERROR_STOP//突发接收遇错误停止
返回	无

表 7-9 函数 I2CMasterBusy()

功能	确认 I2C 主机是否忙
原型	tBoolean I2CMasterBusy（unsigned long ulBase）
参数	ulBase：I2C 主机模块的基址

返回	忙返回 true，不忙返回 false
说明	本函数用来确认 I2C 主机是否正在忙于发送或接收数据

表 7-10 函数 I2CMasterBusBusy()

功能	确认 I2C 总线是否忙
原型	tBoolean I2CMasterBusBusy（unsigned long ulBase）
参数	ulBase ： I2C 主机模块的基址
返回	忙返回 true，不忙返回 false
说明	本函数通常用于多主机通信环境当中，用来确认其他主机是否正在占用总线

表 7-11 函数 I2CMasterErr()

功能	获取 I2C 主机模块的错误状态
原型	unsigned long I2CMasterErr（unsigned long ulBase）
参数	ulBase ： I2C 主机模块的基址
返回	错误状态，是下列值之一： I2C_MASTER_ERR_NONE//没有错误 I2C_MASTER_ERR_ADDR_ ACK//地址应答错误 I2C_MASTER_ERR_DATA_ACK//数据应答错误 I2C_MASTER_ERR_ARB_LOST//丢失仲裁错误

7.4.2 主机模式中断控制

I^2C 总线主机模式的中断控制函数有：中断的使能与禁止控制函数 I2CMasterIntEnable() 和 I2CMasterIntDisable() 、中断状态查询函数 I2CMasterIntS tatus()、中断状态清除函数 I2CMastexlntCleax()。详见表 7-12，表 7-13，表 7-14 和表 7-15 的描述。

表 7-12 函数 I2CMasterIntEnable()

功能	使能 I2C 主机中断
原型	void I2CMasterIntEnable （unsigned long ulBase）
参数	ulBase ： I2C 主机模块的基址
返回	无

表 7-13 函数 I2CMasterIntDisable()

功能	禁止 I2C 主机中断
原型	void I2CMasterlntDisable （unsigned long ulBase）
参数	ulBase ： 主机模块的基址
返回	无

表 7-14 函数 I2CMasterIntStatus()

功能	获取 I2C 主机的中断状态
原型	tBoolean I2CMasterIntStatus（unsigned long ulBase, tBoolean bMasked）
参数	ulBase ： 主机模块的基址 bMasked：取值 false 将获取原始的中断状态，取值 true 将获取屏蔽的中断状态

返回	false-没有中断，true-产生了中断请求

表 7-15 函数 I2CMasterIntClear()

功能	清除 I2C 主机的中断状态
原型	void I2CMasterIntClear（unsigned long ulBase）
参数	ulBase：I2C 主机模块的基址
返回	无

7.4.3 从机模式收发控制

函数 I2CS1aveInit()用来初始化 I^2C 模块为从机模式，并指定从机地址。参见表 7-16 的描述。

函数 I2CS1aveEnable()和 I2CS1aveDisable()用来使能或禁止从机模式下总线的收发。参见表 7-17 和表 7-18 的描述。

函数 I2CSlavestatus()用来获取从机的状态，即 I^2C 模块处于从机模式下，当有（其它的）主机寻址到本从机时要求发送或接收数据的状况。该函数在处理从机收发数据过程中起着至关重要的作用。参见表 7-19 的描述。

函数 I2CS1aveDataGet()用来读取从机已经接收到的数据字节，函数 I2CS1aveDataPut()用来发送从机要传输到（其它的）主机上的数据字节。参见表 7-20 和的表 7-21 描述。

表 7-16 函数 I2CSlaveInit()

功能	初始化 I2C 从机模块
原型	void I2CSlaveInit（unsigned long ulBase, unsigned char ucSlaveAddr）
参数	ulBase：I2C 从机模块的基址，取下列值之一 ucSlaveAddr：7 位从机地址(这是纯地址，MSB 应当为 0)
返回	无

表 7-17 函数 I2CSlaveEnable()

功能	使能 I2C 从机模块
原型	void I2CS1aveEnable（unsigned long ulBase）
参数	ulBase：I2C 从机模块的基址，取下列值之一
返回	无

表 7-18 函数 I2CSlaveDisable()

功能	禁止 I2C 从机模块
原型	void I2CSlaveDisable （unsigned long ulBase）
参数	ulBase：I2C 从机模块的基址，取下列值之一
返回	无

表 7-19 函数 I2CSlaveStatus()

功能	获取 I2C 从机模块的状态
原型	unsigned long I2CSlaveStatus（unsigned long ulBase）
参数	ulBase：I2C 从机模块的基址

返回	主机机请求的动作(如果有的话)，可能是下列值之一： I2C_SLAVE_ACT_NONE//主机没有任何请求动作 I2C_SLAVE_ACT_RREQ_FBR//主机已发送数据到从机，并且收到跟在从机地址后的第 1 个字节 I2C_SLAVE_ACT_RREQ//主机已经发送数据到从机 I2C_SLAVE_ACT_TREQ//主机请求从机发送数据

表 7-20 函数 I2CSlaveDataGet()

功能	获取已经发送到从机模块的数据
原型	unsigned long I2CSlaveDataGet （unsigned long ulBase）
参数	ulBase：I2C 从机模块的基址
返回	获取到的 1 个字节(自动转换为 unsigned long 型)

表 7-21 函数 I2CSlaveDataPut()

功能	从从机模块发送数据
原型	void I2CS1aveDataPut（unsigned long ulBase, unsigned char ucData）
参数	ulBase：I2C 从机模块的基址 ucData：要发送的数据
返回	无
说明	本函数执行的结果是把将要发送的数据存放到个寄存器里，而并不能在总线上立即产生什么动 作，只有在(其它的)主机控制的 SCL 信号的作用下才能把数据一位一位地发送出去。

7.4.4 从机模式中断控制

I^2C 总线从机模式的中断控制函数有：中断的使能与禁止控制函数 I2CS1aveIntEnable() 和 I2CS1aveIntDisable() 、中断状态查询函数 I2CS1aveIntStatus()、中断状态清除函数 I2CSlaveIntClear()。参见表 7-22～表 7-25 的描述。

表 7-22 函数 I2CSlaveIntEnable()

功能	使能 I2C 从机模块的中断
原型	void I2CSlaveIntEnable（unsigned long ulBase）
参数	ulBase：I2C 从机模块的基址
返回	无

表 7-23 函数 I2CSlaveIntDisable()

功能	禁止 I2C 从机模块的中断
原型	void I2CSlaveIntDisable （unsigned long ulBase）
参数	ulBase：I2C 从机模块的基址
返回	无

表 7-24 函数 I2CSlaveIntStatus()

功能	获取 I2C 从机的中断状态
原型	tBoolean I2CS1aveIntStatus（unsigned long ulBase, tBoolean bMasked）
参数	ulBase：I2C 从机模块的基址 bMasked：取值 false 将获取原始的中断状态，取值 true 将获取屏蔽的中断状态
返回	false -没有中断，true-产生了中断请求

表 7-25 函数 I2CSlaveIntClear()

功能	清除 I2C 从机的中断状态
原型	void I2CS1aveIntClear（unsigned long ulBase）
参数	ulBase：I2C 从机模块的基址
返回	无

7.4.5 中断注册与注销

这两个函数用来注册或注销 I^2C 总线在主机（或从机）模式下的中断服务函数，参见表 7-26 和表 7-27 的描述。

表 7-26 函数 I2CIntRegister()

功能	注册一个 I2C 中断服务函数
原型	void I2CIntRegister（unsigned lone ulBase. void （*pfnHandler （void））
参数	ulBase：I2C 从机模块的基址 pfnHandler：函数指针，指向 I2C 主机或从机中断出现时调用的函数
返回	无

表 7-27 函数 I2CIntUnregister()

功能	注销 I2C 中断的服务函数
原型	void I2CIntUnregister（unsigned long ulBase）
参数	ulBase：I2C 从机模块的基址
返回	无

其他详细内容参见文件 LM3SLib_I2C.pdf。

7.5 米字管

米字管是一种半导体发光器件,其基本单元是发光二极管。与传统的 LED 数码管相比，除了能显示 0~9 等数字外，米字管还能显示 A~Z 等 26 个英文字母，因此其应用范围更广。

米字管通常分为共阴极和共阳极两种。

7.5.1 共阴极米字管

共阴极米字管共有 18 个引脚，各段分别接高电平时，相应段会点亮，构成不同的组合，显示不同的数字或英文字母。例如当引脚 1、2、3、4、5、8、9、10、11、12、13 和 14 接低电平，引脚 7、11、15、16、17 和 18 接高电平时，米字管显示数字 0。

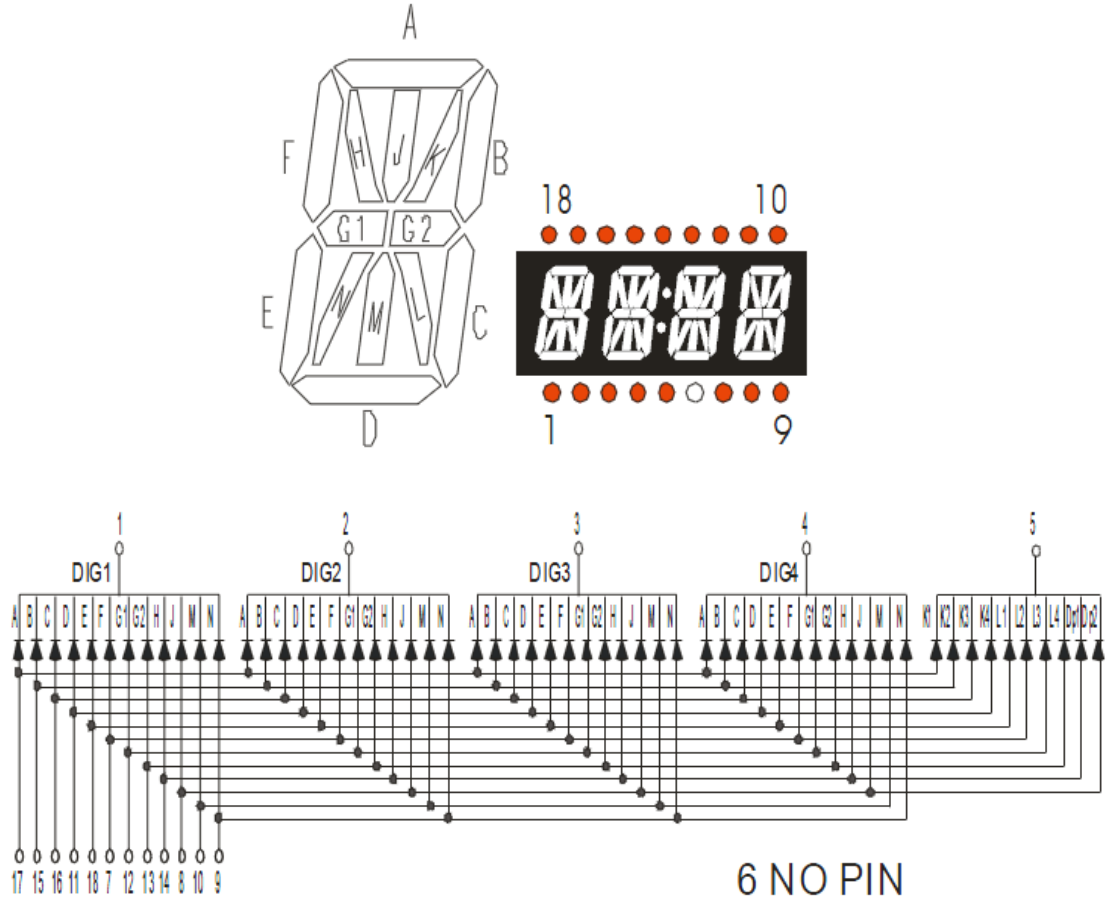


图 7-6　共阴极米字管

7.5.2　共阳极米字管

共阳极米字管共有 18 个引脚，各段分别接低电平时，相应段会点亮，构成不同的组合，显示不同的数字或英文字母。例如当引脚 1、2、3、4、5、8、9、10、11、12、13 和 14 接高电平，引脚 7、11、15、16、17 和 18 接低电平时，米字管显示数字 0。

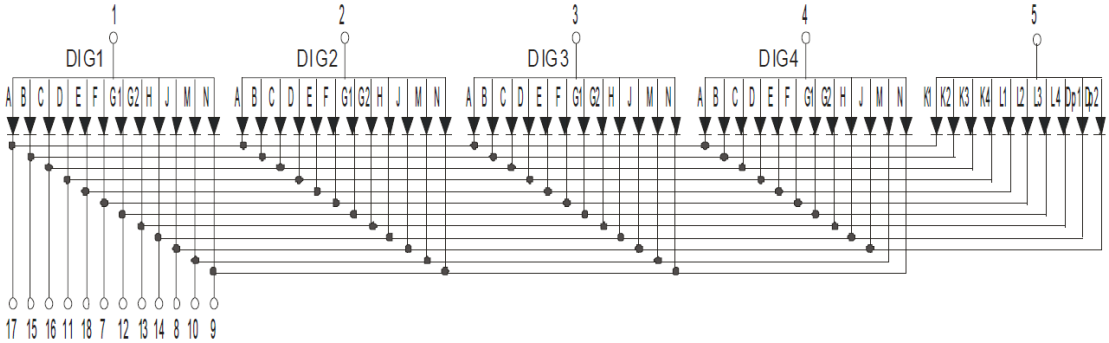


图 7-7　共阳极米字管

7.5.3 板上米字管资源及特性

Cortex-M3 板上有一组 4 位的 LM-S05441AH 米字管，发光颜色为红色。

7.6 本实验使用的资源说明

LED：

LED	对应端口
LED0（黄灯）	PF3
LED1（绿灯）	PF2

低电平有效

五向键：

按键	对应端口	对应操作
SW_1	PB6	右
SW_2	PE5	按下
SW_3	PE4	上
SW_4	PF1	下
SW_5	PB4	左

低电平触发

米字管：S05441A

LED 灯：D5、D6、D7、D13、D14、D15、D16

上位机：

按键	对应操作
S/s	发送数据
R/r	接收数据

■ 实验原理图

见图 7-8。

7.7 实验七 I2C 应用及编程

■ 实验概述：

本实验使用了实验板上的五个LED灯和数码米字管两个模块，通过I^2C总线分别实现五向导航键对其的控制，其中数码米字管还实现电子钟功能。

本次实验重点在于熟悉、理解ARM的I^2C总线的工作原理，掌握与I^2C总线有关的各程序的作用和设置方法。主要涉及以下知识点：

- 了解I2C总线的特点和功能；
- 学会使用I2C自检程序；

- 会使用I2C总线对PCA9557芯片进行操作；
- 学会常用键盘防抖方法。

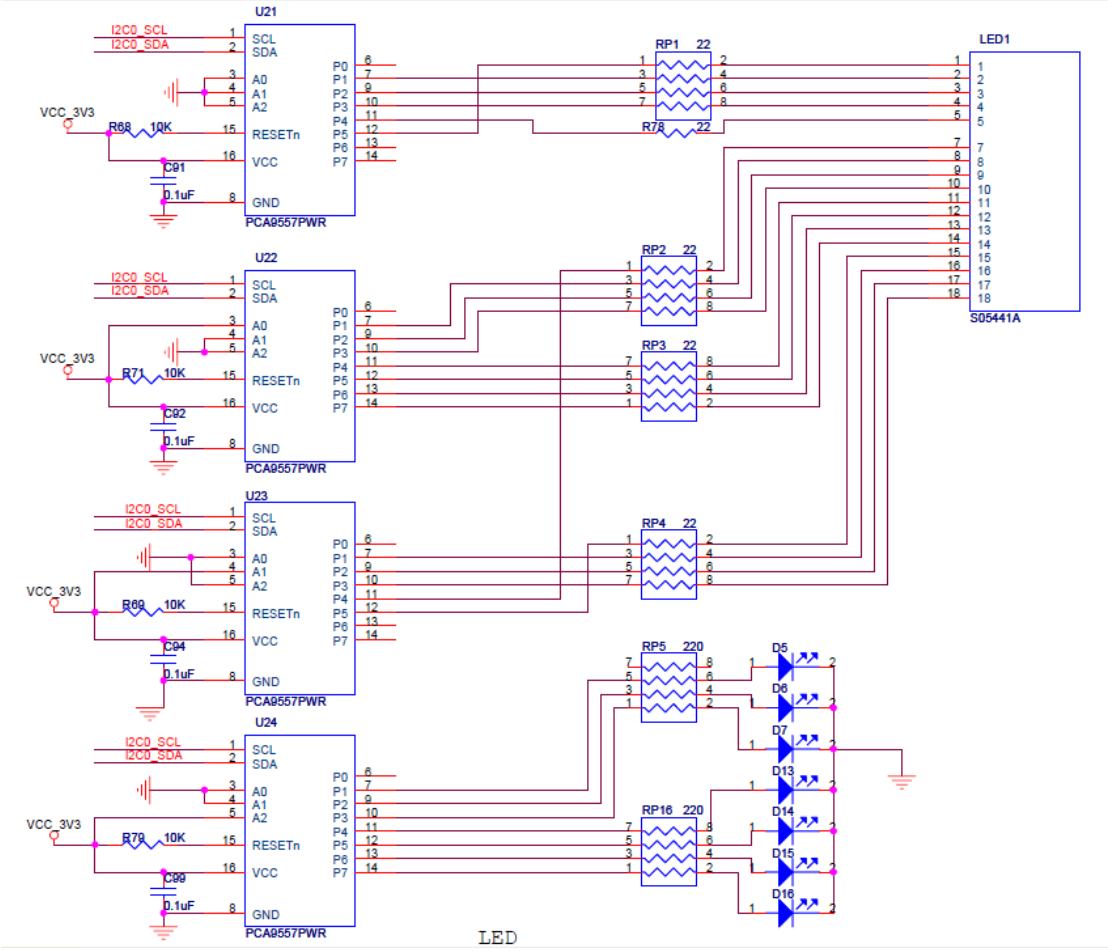
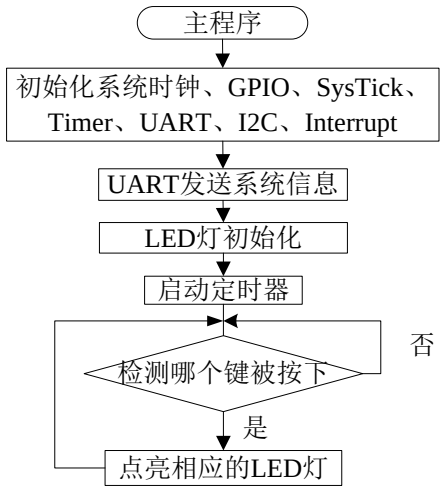


图 7-8　实验板部分原理图

实验 7.1　五向导航键控制 LED 灯实验

本实验流程图如下:



■ Step1：准备

进入工程文件夹 7-1I2C_LED，点击 Experiment.uvproj 打开工程项目

■ Step2：I2C 主机初始化配置-使能模块

I2C 配置文件为 I2CConfigure.h，I2CConfigure.c，打开 I2CConfigure.c，在 I2C0MasterInitial(void) 中使能 I2C0 模块：调用 SysCtlPeripheralEnable(unsigned long ulPeripheral) 使能外设，参数为 SYSCTL_PERIPH_I2C0，该函数在 driverlib/sysctl.h 声明

■ Step3：I2C 主机初始化配置-初始化主机模式，设置传输速率

1、初始化 I^2C 模块为主机模式，并设置传输速率 100kbps: 函数 I2CMasterInitExpClk（unsigned long ulBase, unsigned long uII2CClk, tBoolean bFast）用来初始化 I^2C 模块，并选择通信速率为 100kbps 的标准模式还是 400kbps 的快模式，但在实际编程时常常以更方便的宏函数 I^2CMasterInit（unsigned long ulBase, Boolean bFast）来代替

2、I2CConfigure.c 已经实现了主机速度设置函数 I2CMasterSpeedSet(unsigned long ulBase, unsigned long ulSpeed)，调用该函数设置传输速度为 100kbps

■ Step4：I2C 主机初始化配置-使能 I2C 主机模块

I2CMasterEnable(long ulBase)使能 I2C 主机模块，step2~step4 的参考图如下

```
#include "UARTConfigure.h"
#include "I2CConfigure.h"

unsigned char LEDSerial=0x01;

void I2C0MasterInitial(void)
{
    unsigned char tempCounter;

    /*Step2 To do*/                    //使能 I2C0 模块
    /*Step2 To do*/                    //100kbps 的普通模式
    /*Step3 To do*/                    //任意设置传输速度,此处为默认 100kbps
    /*Step4 To do*/                    //使能 I2C 主机模块

    //配置所有 PCA9557 为输出模式，并关闭所有 LED
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,LED_I2CADDR,PCA9557_
REG_CONFIG,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,LED_I2CADDR,PCA9557_
REG_OUTPUT,0x00);
    //LED_OnChip 启动动画
    for (tempCounter=0;tempCounter<8;tempCounter++)
    {
        LEDSerial<<=1;
```

```
        SysCtlDelay(SysCtlClockGet()/60);

    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,LED_I2CADDR,PCA9557_
REG_OUTPUT,LEDSerial);
    }
}
```

■ Step5：主机突发传输控制-设置从机地址

用函数 I2CMasterslaveAddrSet()来设置器件地址和读写控制位，参数为 I2C 主机模块的基址 ulBase，从机地址 ucSla，数据方向为主机发送至从机。

■ Step6：主机突发传输控制-发送子地址

1、调用函数 I2CMasterDataPut(unsigned long ulBase, unsigned char ucData)，此时数据为数据地址 ucAddr，将其发送给主机

2、开始突发发送，调用 void I2CMasterControl（unsigned long ulBase, unsigned long ulCmd）控制主机模块在总线上的动作，动作为突发发送起始(I2C_MASTER_CMD_BURST_SEND_START)

■ Step7：主机突发传输控制-发送数据

与 step6 类似，此时要传送的数据为 ucData，主机模块在总线上的动作为 I2C_MASTER_CMD_BURST_SEND_FINISH，Step5~step7 的参考图如下

```
unsigned char I2CMasterTransmit_Burst_2Bytes
                                    (unsigned long ulBase,      //发送完成后才会退出
                                     unsigned char ucSla,       //从机地址
                                     unsigned char ucAddr,          //从机子地址
                                     unsigned char ucData)      //传送的 1 个数据位
{
    /*step5*/                   //false 表示主机发送至从机
    /*step6*/                   //设置要发送的子地址
    /*step6*/                   //突发发送起始
    while(I2CMasterBusy(ulBase));
    if (I2CMasterErr(ulBase)!=I2C_MASTER_ERR_NONE)
    {
        UARTStringPut(UART0_BASE,"I2C0 Transmission Fault!!\r\n");
        return 0;
    }

    /*step7*/                   //设置要发送的数据
    /*step7*/                   //突发发送结束
    while(I2CMasterBusy(ulBase));
    if (I2CMasterErr(ulBase)!=I2C_MASTER_ERR_NONE)
    {
```

```
            UARTStringPut(UART0_BASE,"I2C0 Transmission Fault!!\r\n");
            return 0;
        }
        return 1;
}
```

■ Step8：I2C 上拉测试-设置所选端口指定的管脚为输入模式

打开 GPIODriverConfigure.c,在 I2C0PullUpTest(void)中设置 SCL，SDA 两个端口为输入模式。

通过调用 **GPIOPinTypeGPIOInput**(unsigned long ulPort, unsigned char ucPins)来实现，参数 ulPort 为 I2C0_PIN_BASE，ucPins 为 I2C0SCL_PIN | I2C0SDA_PIN

■ Step9：I2C 上拉测试-打开 GPIO 输出弱下拉

打开 SCL，SDA 两个端口输出若下拉，调用函数 GPIOPadConfigSet(unsigned long ulPort, unsigned char ucPins, unsigned long ulStrength, unsigned long ulPadType)

设置驱动强度为 8mA,类型为带弱上拉的推挽，可通过传递参数 GPIO_STRENGTH_8MA，GPIO_PIN_TYPE_STD_WPD

Step8~step9 的参考图如下

```
unsigned char I2C0PullUpTest(void)
{
    //设置所选端口指定的管脚为输入模式
    /*Step8 To do*/
    //打开 GPIO 输出弱下拉
    /*step9 To do */
    SysCtlDelay(SysCtlClockGet()/1500);
    //测试外部上拉是否发挥作用
    if (GPIOPinRead(I2C0_PIN_BASE,
        I2C0SCL_PIN | I2C0SDA_PIN) !=
        (I2C0SCL_PIN | I2C0SDA_PIN))
        return 1;
    GPIOPinTypeI2C(I2C0_PIN_BASE, I2C0SCL_PIN | I2C0SDA_PIN);
    return 0;
}
```

■ Step10：主循环等待显示

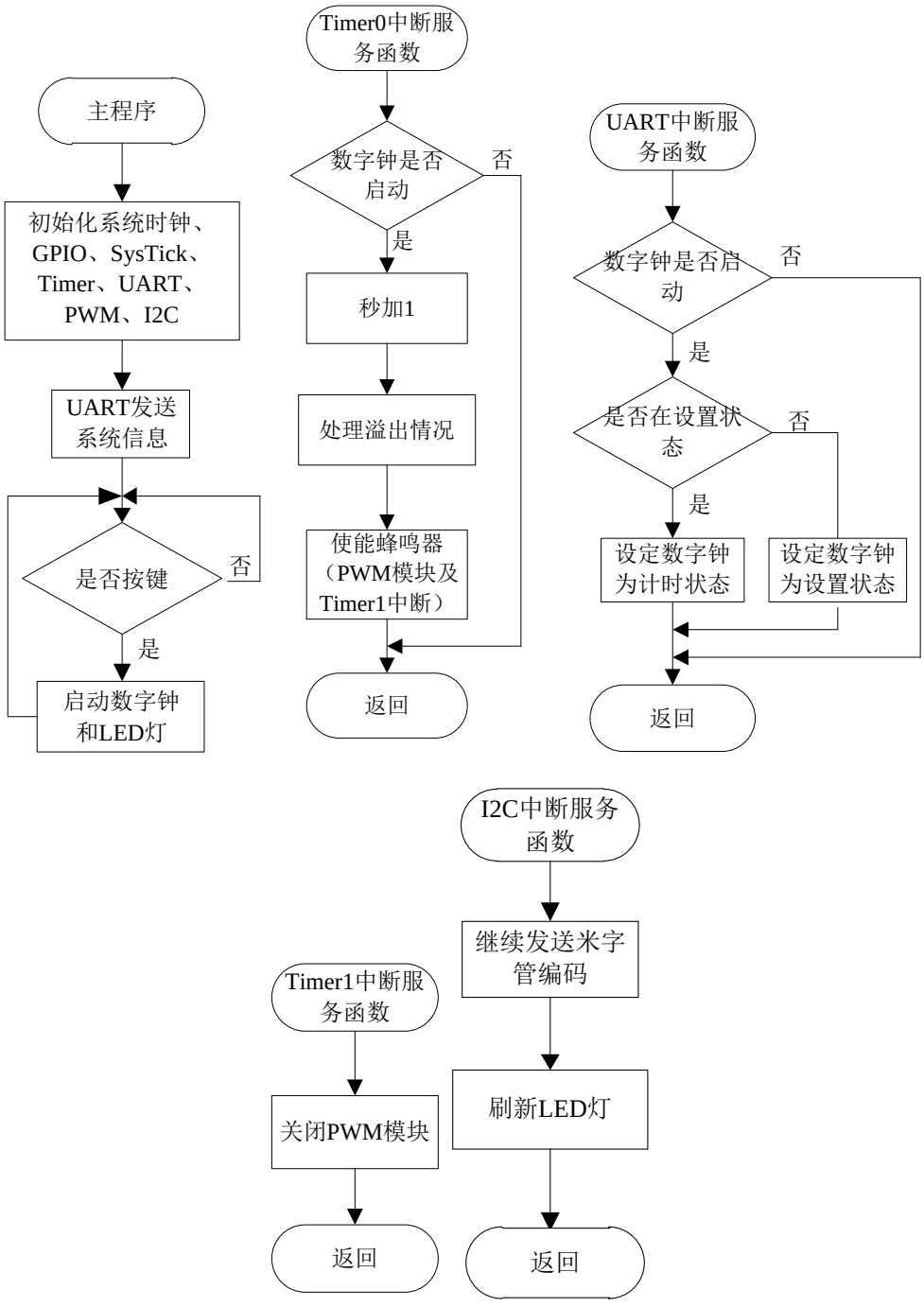
打开 main.c,在 while(1)的循环中输入 KeyNumber 为 0 和不为 0 两种情况下的代码，调用 I2CMasterTransmit_Burst_2Bytes(),传输的数据位分别为 0x01<<KeyNumber 和 0x00，主机基址为 I2C0_MASTER_BASE，从机地址为 LED_I2CADDR，从机子地址为 PCA9557_REG_OUTPUT, step10 的参考图如下

```
int main(void)                          //主函数
{
    ClockInitial();
    GPIOInitial();                                          6
    SysTickInitial();
    TimerInitial();
    UART0Initial();
    SystemState |= I2C0PullUpTest();
    if (!(SystemState & 0x01))    I2C0MasterInitial();
    UARTSystemCheckInformationTransmit(UART0_BASE,SystemState);
    SysTickEnable();

    LEDOn(LED_ALL);
    while (!KeyNumber);
    LEDOff(LED_ALL);
    UARTStringPut(UART0_BASE,"System activated!\r\n");

    IntMasterEnable();
    TimerEnable(TIMER0_BASE,TIMER_A);                       //打开 TIMER0

    while(1)
    {
        if (KeyNumber)
        /*Step10 To do  数据位为 0x01<<KeyNumber*/
        else
        /*Step10 To do 0x00*/
    }
}
```

■ Step 11: 编译和下载

实验 7.1　米字管电子钟实验

■ **本实验的流程图如下:**



■ **Step1：准备**

进入工程文件夹 7-2 I2C_NixieTube，点击 Experiment.uvproj 打开工程项目

■ **Step2：I2C 主机初始化配置-使能模块及初始化设置**

I2C 配置文件为 I2CConfigure.h，I2CConfigure.c，打开 I2CConfigure.c：

1. 在 I2C0MasterInitial(void) 中使能 I2C0 模块：调用 SysCtlPeripheralEnable(unsigned long ulPeripheral) 使能外设，参数为 SYSCTL_PERIPH_I2C0，该函数在 driverlib/sysctl.h 声明

2. I2C 主机模块初始化，并选择通信速率为 100kbps 的标准模式：调用 I2CMasterInit(unsigned long ulBase, Boolean bFast)，其中的参数有全局变量 I2C0_MASTER_BASE

■ Step3：I2C 主机初始化配置-设置传输速率

设置传输速度为 400kbps：调用主机速度设置函数 I2CMasterSpeedSet(unsigned long ulBase, unsigned long ulSpeed)

■ Step4：I2C 主机初始化配置-使能 I2C 主机模块

1. 使能I2C主机：调用I2CMasterEnable(unsigned long ulBase);

2. 使能 I2C 主模块中断：调用 I2CMasterIntEnable(unsigned long ulBase);

Step2~Step4 的参考图如下:

```
void I2C0MasterInitial(void)
{
    unsigned char tempCounter;

    /* step2_1*/                    //使能 I2C0 模块
    /* step2_2*/                    //I2C 主机模块初始化，100kbps 的普通模式
    /* step3 */                         //设置传输速度，此处为 400kbps
    /* step4_1*/                    //使能 I2C 主机
    /* step4_2*/                      //使能 I2C 主模块中断


    //配置所有 PCA9557 为输出模式，并关闭所有 LED
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,LED_I2CADDR,PCA9557_
REG_CONFIG,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEG_I2CADDR,PC
A9557_REG_CONFIG,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEL1_I2CADDR,PC
A9557_REG_CONFIG,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEL2_I2CADDR,PC
A9557_REG_CONFIG,0x00);

    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,LED_I2CADDR,PCA9557_
REG_OUTPUT,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEG_I2CADDR,PC
A9557_REG_OUTPUT,0xff);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEL1_I2CADDR,PC
A9557_REG_OUTPUT,0xff);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEL2_I2CADDR,PC
```

```
A9557_REG_OUTPUT,0xff);

    //LED_OnChip 启动动画
    for (tempCounter=0;tempCounter<8;tempCounter++)
    {
        LEDSerial<<=1;
        SysCtlDelay(SysCtlClockGet()/60);

    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,LED_I2CADDR,PCA9557_
REG_OUTPUT,LEDSerial);
    }
    IntEnable(INT_I2C0);                                    //打开 I2C 中断
}
```

■ Step5：获取 I2C 主机的中断状态

打开I2CISR.c, 读取中断状态并赋值于ulStatus：调用I2CMasterIntStatus(unsigned long ulBase, tBoolean bMasked)；

■ Step6：清除 I2C 主机的中断状态

清除中断状态：调用I2CMasterIntClear(unsigned long ulBase)；

■ Step7：错误情况处理

当符合中断条件时，必须通过I2CMasterErr(unsigned long ulBase)函数进行错误检测，当返回值为I2C_MASTER_ERR_NONE则表示无错误；如果遇到错误情况，便将传送状态I2C0TransmitionState置零;

Step5~Step7 的参考图如下:

```
#include "I2CConfigure.h"
#include "NixieTubeConfigure.h"
#include "I2CISR.h"

unsigned char I2C0TransmitionState=0;

void I2C0_ISR(void)
{
    unsigned long ulStatus;
    /* step5*/              //读取中断状态
    /* step6*/              //清除该中断状态
    /* step7*/              //遇到错误情况处理

    if (ulStatus)
    {
```

```
        I2C0TransmitionState++;
        switch (I2C0TransmitionState)
        {
            case 1:
……
```

■ Step8：米字管选择设置

通过设置变量 NixieTubeSegment 来选择米字管，参考图如下：

```
void NixieTubeCoding(void)
{
    unsigned char NixieTubePointer=0;      //查表指示符

    /* step8*/
    switch (NixieTubeSlice)
    {
        case 0:              //选择左数第 1 根米字管
        case 1:              //选择左数第 2 根米字管
        case 2:              //选择左数第 3 根米字管
        case 3:              //选择左数第 4 根米字管
        case 4:              //选择第 5 根特殊米字管
        default:
    }
    if (NixieTubeSlice<4)
    {
……
    }
```

■ Step9：加入程序的主循环部分

检测五向导航键中任意键被按下后开启计秒模式。若中键被按下则进入计数模式，向上为加计数，向下为减计数。若此时中键再次被按下，则从原中断处继续计秒。

计时终止和开始由 TimerDisable()和 TimerEnable()实现；时钟模式下，时间显示的参数为 Set_Minute 和 Set_Second；设置模式下，时间显示的参数为 Counter_Minute 和 Counter_Second；

```
    while(1)
    {
        if (/* step9_1*/)   //按下按键时
        {
            if (/* step9_2*/)     //时钟模式时
            {
                sprintf(NixieTube,"%04u:",/* step9_3*/);     //显示时间
                /* step9_4*/                                  //进入设置模式
```

```
                    /* step9_5*/                                    //计时终止
                }
                else
                {
                    sprintf(NixieTube,"%04u:",/* step9_6*/);     //显示时间
                    /* step9_7*/                                      //进入时钟模式
                    /* step9_8*/                                    //设置秒
                    /* step9_9*/                                    //开始计时
                }
                LEDOverturn(LED_ALL);
                KeyNumber=0;
……
  }
```

■ Step 10: 编译和下载

第 八 章　PWM 模块

8.1　PWM 简介

脉冲宽度调制（PWM，Pulse-Width Modulation），也简称为脉宽调制，它使用了微处理器的数字输出来实行对模拟电路的控制，这种技术被广泛应用在广泛应用在从测量，通信，控制，变换等各种领域中。

在 PWM 中，电压或电流源是以一种通或断的重复脉冲序列形式被加到模拟负载上去的。通的时候即是直流供电被加到负载上，断的时候即是供电被断开。在给定的任何时刻，满幅值的直流供电要么完全有，要么完全无，只有 0 和 1 两种状态， 所以 PWM 信号是数字的。在脉宽调制(PWM)中，我们使用高分辨率计数器来调制 PWM 输出方波的占空比，从而可以对模拟信号电平进行编码。

PWM 的一个优点是从处理器到被控系统信号都是数字形式的，无需进行数模转换。让信号保持为数字形式可将噪声影响降到最小。噪声只有在强到足以将逻辑 1 改变为逻辑 0 或将逻辑 0 改变为逻辑 1 时，也才能对数字信号产生影响。

对噪声抵抗能力的增强是 PWM 的另外一个优点，而且这也是在某些时候将 PWM 用于通信的主要原因。从模拟信号转向 PWM 可以极大地延长通信距离。在接收端，通过适当的 RC 或 LC 网络可以滤除调制高频方波并将信号还原为模拟形式。

8.2　PWM 基本概念

8.1.1　占空比

占空比(Duty Cycle)在电信领域中有如下含义：在一串理想的脉冲周期序列中（如方波），正脉冲的持续时间与脉冲总周期的比值。占空比是指高电平在一个周期之内所占的时间比率。如果方波所占比例为 20%，即占空比为 0.2，说明正电平所占时间为 0.2 个周期。

PWM 模块通过调整方波的占空比来模拟各种各样的波形，理论上当带宽足够时，PWM 可以模拟出任意波形出来。

8.1.2　死区

死区时间一般是指控制不到的时间域。在变频器里一般是指功率器件输出电压、电流的“0”区，在传动控制里一般是指电机正反向转换电压、电流的过零时间。

在 PWM 中，死区时间是在输出时，为了使 H 桥或半 H 桥的上下管不会因为开关速度问题发生同时导通而设置的一个保护时段。通常也指 PWM 的响应时间。

由于 IGBT（绝缘栅极型功率管）等功率器件都存在一定的结电容，所以

会造成器件导通关断的延迟现象。一般在设计电路时已尽量降低该影响，比如尽量提高控制极驱动电压电流，设置结电容释放回路等。为了使 IGBT 工作可靠，避免由于关断延迟效应造成上下桥臂直通，有必要设置死区时间，也就是上下桥臂同时关断时间。死区时间可有效地避免延迟效应所造成的一个桥臂未完全关断，而另一桥臂又处于导通状态，避免直通炸模块。

死区时间设置的大，PWM 模块工作会更加可靠，但是会导致输出波形失真并且降低输出效率。如果死区时间设置的小，输出波形要好一些，但是会降低可靠性。所以死区时间的设置一般是在保证安全的条件下，越小越好。

8.1.3 PWM 作数模转换

PWM 可以输出周期固定，脉宽可控的脉冲波形，加上低通滤波器后，PWM 可以将数字信号转换成模拟信号。

PWM 含有两种信号，一种叫做载波信号，一种叫做调制信号。载波信号控制的是开关器件的开关频率，调制信号控制的是所需要的输出信号。由于载波信号的频率远大于调制信号，所以如果输出接上低通滤波器之后，滤除频率较高的载波，就剩下了频率较低的调制信号。如果输入调制信号是正弦波即每个周期的脉宽是按照正弦波的规律变化，则经过低通滤波器后即可得到正弦波形。

8.3实验板上 PWM 资源及其特性

Stellaris 系列 ARM 提供了 4 个 PWM 发生器模块和一个控制模块。每个 PWM 发生器模块包含 1 个定时器、2 个比较器、1 个 PWM 信号发生器、1 个死区控制器，以及一个中断/ADC 触发选择器。

每个 PWM 发生器模块产生两个 PWM 信号，4 个 PWM 发生器一共生成 8 个信号 PWM 0, PWM 1，PWM 2，PWM 3, PWM 4, PWM 5, PWM 6，PWM7，他们分别又 PWM_GEN0, PWM_GEN1, PWM_GEN2, PWM_GEN3 产生。在这些 PWM 信号输出到管教之前，还有一个 PWM 输出控制器，控制各个 PWM 发生器生成信号的输出。

同时 Stellaris 系列 ARM 单片机的 PWM 模块还具有死区控制功能，可以对 PWM 信号发生器产生的信号进行死区生成，满足用户的需要。

Stellaris 系列单片机 PWM 模块特性:

- 4 个 PWM 发生器，产生 8 路 PWM 信号；
- 多种的 PWM 产生方法；
- 每个 PWM 发生器都带有死区发生器；
- 灵活可控的输出控制模块；
- 安全可靠的错误检测保护功能；
- 丰富的中断机制和 ADC 触发。

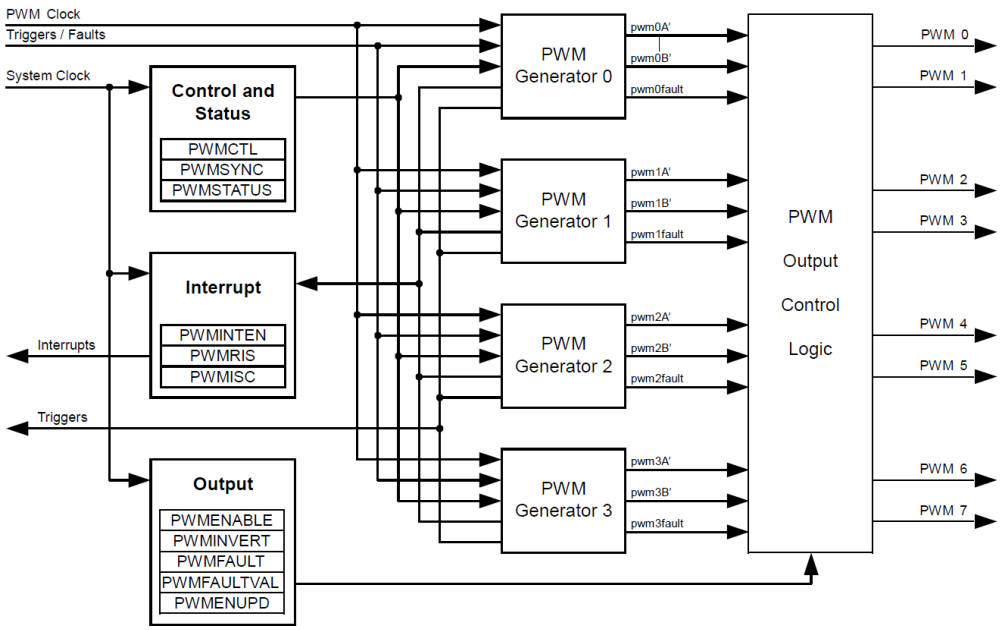


图 8-1　PWM 模块整体结构图

8.3.1　PWM 定时器

PWM 定时器用来触发 PWM 信号发生器产生 PWM 信号。板上的 PWM 定时器有两种工作模式：递减计数模式和先递增后递减计数模式。在递减计数模式中，定时器从设定的数值开始倒数，计数到零时又返回到设定数值并继续递减计数。在先递增后递减计数模式中，定时器从 0 开始往上计数，一直计数到装载值，然后从设定值递减到零，接着再递增到装载值，由此往复。通常，递减计数模式用来产生左对齐或右对齐的 PWM 信号，而先递增后递减计数模式用来产生中心对齐的 PWM 信号。

PWM 定时器输出 3 个信号，这些信号之后将会被 PWM 信号发生器使用。一个是方向信号（在递减计数模式中，该信号始终为低电平，在先递增后递减计数模式中，则是在高低电平之间切换）。另外两个信号为零脉冲和装载脉冲。当计数器计数值为 0 时，零脉冲信号发出一个宽度等于时钟周期的高电平脉冲；当计数器计数值等于设定值时，装载脉冲也发出一个宽度等于时钟周期的高电平脉冲。

8.3.2　PWM 比较器

板上的每个 PWM 发生器包含两个比较器，用于监控计数器的值。当比较器的值与计数器的值相等时，比较器输出宽度为单时钟周期的高电平脉冲。在先递增后递减计数模式中，比较器在递增和递减计数时都要进行比较，因此必须通过计数器的方向信号来限定。这些限定脉冲在生成 PWM 信号的过程中使用。

8.3.3　PWM 信号发生器

PWM 发生器整体构架如图 8-2 所示。

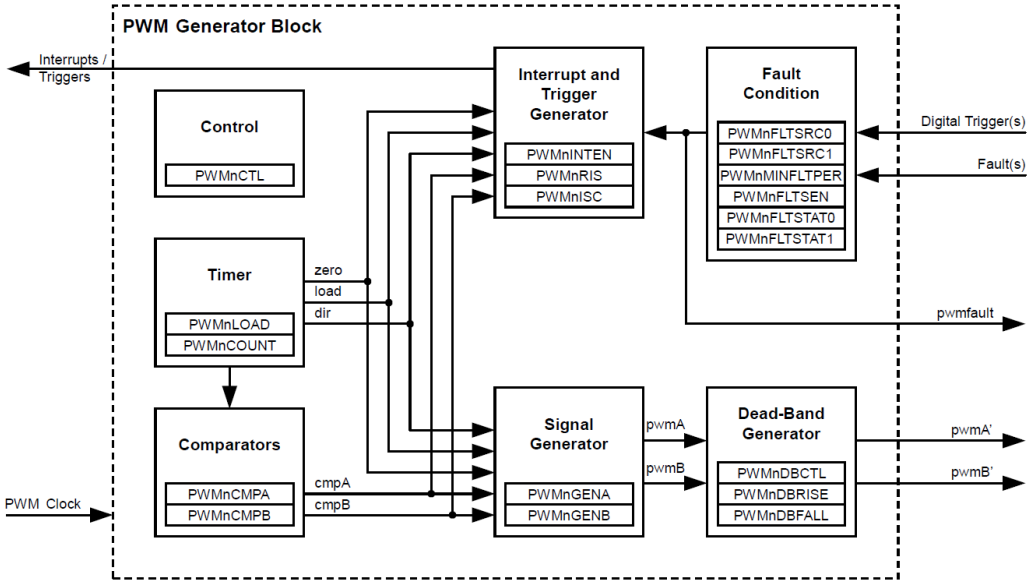


图 8-2 PWM 信号发生器整体结构图

输入 zero, load, dir, cmpA, cmpB 信号后，PWM 信号发生器发生将会产生两个 PWM 信号。根据不同的输入 PWM 信号发生器将会产生不同的占空比的方波。

在递减计数模式中，能够影响 PWM 信号的事件有四个：zero，load，cmpA 递减，cmpB 递减。而在先递增后递减模式中，因为计算方向可以有两个不同的值，所以影响 PWM 信号生成的事件增加到了 6 个：zero，load，cmpA 递增， cmpB 递增，cmpA 递减，cmpB 递减。值得注意的是，当 cmpA 或 cmpB 事件与 zero 或者 load 事件重合时，它们可以被忽略。而如果 cmpA 和 cmpB 时间重合，则第一个信号 pwmA 只根据 cmpA 事件生成，第二个信号 pwmB 只根据 cmpB 事件生成。

这 5 个输入信号事件对 PWM 信号的输出的影响都是可以修改编程的的：信号可以忽略，可以翻转，可以驱动为低电平、可以驱动为高电平。对这些信号时间的响应可以生成一对不同位置和不同占空比的 PWM 信号，这对信号量之后还可以通过死区控制器进行进一步的修改。

8.3.4 PWM 死区发生器

如图 8-2 所示，死区发生器是对 PWM 信号发生器产生的信号的最后一个模块。

从 PWM 发生器产生的两个 PWM 信号将会通过死区发生器。如果此时死区发生器没有被使能，则 PWM 信号只是简单地通过，不会发生改变。但如果使能了死区发生器，那么第二个 PWM 信号将会被丢弃，死区发生器会在第一个 PWM 信号基础上产生新的两个 PWM 信号。输出的第一个 PWM 信号为附加了上升沿延迟的输入信号，延迟时间可以编程修改。第二个输出的 PWM 信号是输入信号的反相信号，并且在输入信号的下降沿和这个新信号的上升沿之间增加了可编程的延迟时间。

死区发生器输出的两个信号 pwmA’和 pwmB’是一对高电平有效的信号。

除了在跳变处的延迟时间外，两个信号其中定有一个信号为高电平。这样这两个信号就可以用来驱动半-H 桥（half-H bridge），又由于它们带有死区，因而还可以避免冲过电流（shoot through current）导致电力电子管受损。

8.4 实验中的 PWM 库函数简介

8.4.1 配置函数

函数 PWMGenConfigure( )用来对指定 PWM 发生器进行基本配置。然后我们使用 PWMGenPeriodSet( ) 和 PWMPulseWidthSet( )来设置 PWM 的占空比，并用 PWMOutputState ( )配置管脚输出。在实际编程中，我们一般使用自己封装的函数 PWMInitial( )来进行总体的配置，简化操作。各配置函数具体说明可见下列表格。

表 8-1 函数 PWMGenConfigure( )

原型	void PWMGenConfigure(unsigned long ulBase, unsigned long ulGen, unsigned long ulConfig)
参数	ulBase is the base address of the PWM module。 ulGen is the PWM generator to configure. Must be one of PWM_GEN_0, PWM_GEN_1, PWM_GEN_2, or PWM_GEN_3. ulConfig is the configuration for the PWM generator. The ulConfig parameter contains the desired configuration. It is the logical OR of the following: PWM_GEN_MODE_DOWN or PWM_GEN_MODE_UP_DOWN to specify the counting mode PWM_GEN_MODE_SYNC or PWM_GEN_MODE_NO_SYNC to specify the counter load and comparator update synchronization mode PWM_GEN_MODE_DBG_RUN or PWM_GEN_MODE_DBG_STOP to specify the debug behavior PWM_GEN_MODE_GEN_NO_SYNC, PWM_GEN_MODE_GEN_SYNC_LOCAL, or PWM_GEN_MODE_GEN_SYNC_GLOBAL to specify the update synchronization mode for generator counting mode changes PWM_GEN_MODE_DB_NO_SYNC, PWM_GEN_MODE_DB_SYNC_LOCAL, or PWM_GEN_MODE_DB_SYNC_GLOBAL to specify the deadband parameter synchronization

	mode PWM_GEN_MODE_FAULT_LATCHED or PWM_GEN_MODE_FAULT_UNLATCHED to specify whether fault conditions are latched or not PWM_GEN_MODE_FAULT_MINPER or PWM_GEN_MODE_FAULT_NO_MINPER to specify whether minimum fault period support is required PWM_GEN_MODE_FAULT_EXT or PWM_GEN_MODE_FAULT_LEGACY to specify whether extended fault source selection support is enabled or not
返回	None.
说明	This function is used to set the mode of operation for a PWM generator. The counting mode, synchronization mode, and debug behavior are all configured. After configuration, the generator is left in the disabled state.
功能	Configuration of the PWM generator.

表 8-2 函数 PWMGenPeriodSet ( )

原型	void PWMGenPeriodSet(unsigned long ulBase, unsigned long ulGen, unsigned long ulPeriod)
参数	ulBase is the base address of the PWM module. ulGen is the PWM generator to be modified. Must be one of PWM_GEN_0, PWM_GEN_1, PWM_GEN_2, or PWM_GEN_3. ulPeriod specifies the period of PWM generator output, measured in clock ticks.
返回	None.
说明	This function sets the period of the specified PWM generator block, where the period of the generator block is defined as the number of PWM clock ticks between pulses on the generator block zero signal.
功能	Set the period of a PWM generator.

表 8-3 函数 PWMPulseWidthSet ( )

原型	void PWMPulseWidthSet(unsigned long ulBase, unsigned long ulPWMOut, unsigned long ulWidth)
参数	ulBase is the base address of the PWM module. ulPWMOut is the PWM output to modify. Must be one of PWM_OUT_0, PWM_OUT_1, PWM_OUT_2, PWM_OUT_3, PWM_OUT_4, PWM_OUT_5, PWM_OUT_6, or PWM_OUT_7. ulWidth specifies the width of the positive portion of the pulse.
返回	None.
说明	This function sets the pulse width for the specified PWM output, where the pulse width is defined as the number of PWM clock ticks.
功能	Sets the pulse width for the specified PWM output.

表 8-4 函数 PWMOutputState ( )

原型	void PWMOutputState(unsigned long ulBase, unsigned long ulPWMOutBits, tBoolean bEnable)
参数	ulBase is the base address of the PWM module. ulPWMOutBits are the PWM outputs to be modified. Must be the logical OR of any of PWM_OUT_0_BIT, PWM_OUT_1_BIT, PWM_OUT_2_BIT, PWM_OUT_3_BIT, PWM_OUT_4_BIT, PWM_OUT_5_BIT, PWM_OUT_6_BIT, or PWM_OUT_7_BIT. bEnable determines if the signal is enabled or disabled.
返回	None.
说明	This function is used to enable or disable the selected PWM outputs. The outputs are selected using the parameter ulPWMOutBits. The parameter bEnable determines the state of the selected outputs. If bEnable is true, then the selected PWM outputs are enabled, or placed in the active state. If bEnable is false, then the selected outputs are disabled, or placed in the inactive state.
功能	Enables or disables PWM outputs.

表 8-5 函数 PWMInitial( )

原型	void PWMInitial(void)
参数	None
返回	None.
说明	This function does some basic configuration on PWM generator. It sets the counting of PWM generator and starts a period of PWM output. It could be modified by user for specified configuration.
功能	Configure and enable PWM generator.

8.4.2 使能和禁止函数

函数 PWMGenEnable( )使能相应的 PWM 发生器。函数 PWMGenDisable( ) 关闭禁用对应的 PWM 发生器。

表 8-6 函数 PWMGenEnable( )

原型	void PWMGenEnable(unsigned long ulBase, unsigned long ulGen)
参数	ulBase is the base address of the PWM module. ulGen is the PWM generator to be enabled. Must be one of PWM_GEN_0, PWM_GEN_1, PWM_GEN_2, or PWM_GEN_3.
返回	None.
说明	This function allows the PWM clock to drive the timer/counter for the

	specified generator block.
功能	Enables the timer/counter for a PWM generator block.

表 8-7 函数 PWMGenDisable( )

原型	void PWMGenDisable(unsigned long ulBase, unsigned long ulGen)
参数	ulBase is the base address of the PWM module. ulGen is the PWM generator to be disabled. Must be one of PWM_GEN_0, PWM_GEN_1, PWM_GEN_2, or PWM_GEN_3.
返回	None.
说明	This function blocks the PWM clock from driving the timer/counter for the specified generator block.
功能	Disables the timer/counter for a PWM generator block.

8.4.3 死区控制函数

函数 PWMDeadBandEnable( )设置并使能了指定 PWM 发生器的死区控制器，而函数 PWMDeadBandDisable( )禁用了指定 PWM 发生器的死区控制器。

表 8-8 函数 PWMDeadBandEnable( )

原型	PWMDeadBandEnable(unsigned long ulBase, unsigned long ulGen, unsigned short usRise, unsigned short usFall)
参数	ulBase is the base address of the PWM module. ulGen is the PWM generator to modify. Must be one of PWM_GEN_0, PWM_GEN_1, PWM_GEN_2, or PWM_GEN_3. usRise specifies the width of delay from the rising edge. usFall specifies the width of delay from the falling edge.
返回	None.
说明	This function sets the dead bands for the specified PWM generator, where the dead bands are defined as the number of PWM clock ticks from the rising or falling edge of the generator’s OutA signal. Note that this function causes the coupling of OutB to OutA.
功能	Enables the PWM dead band output, and sets the dead band delays.

表 8-9 函数 PWMDeadBandDisable( )

原型	void PWMDeadBandDisable(unsigned long ulBase, unsigned long ulGen)
参数	ulBase is the base address of the PWM module. ulGen is the PWM generator to modify. Must be one of PWM_GEN_0, PWM_GEN_1, PWM_GEN_2, or PWM_GEN_3.
返回	None.
说明	This function disables the dead band mode for the specified PWM generator. Doing so decouples the OutA and OutB signals.
功能	Disables the PWM dead band output.

8.5 Buzzer（蜂鸣器）

蜂鸣器（Buzzer）是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中作发声器件。

蜂鸣器发声原理是电流通过电磁线圈，使电磁线圈产生磁场来驱动振动膜发声的，因此需要一定的电流才能驱动它，单片机 IO 引脚输出的电流较小，单片机输出的 TTL 电平基本上驱动不了蜂鸣器，因此需要增加一个电流放大的电路。

PWM 输出口直接驱动是利用 PWM 输出口本身可以输出一定的方波来直接驱动蜂鸣器。在单片机中我们可以设置占空比、周期等参数，通过设置这些参数 PWM 生成蜂鸣器要求的频率方波后，只要打开 PWM 输出，PWM 输出口就能输出该频率的方波，这个时候利用这个波形就可以驱动蜂鸣器了。

8.6本实验使用资源说明

本实验使用了 Buzzer 模块，增加的资源如下所示：

Buzzer	PF3

由于 Buzzer 和 LED0 都使用了 PF3 口，所以我们使用 PWM_GEN2 输出的信号 PWM5，这个信号将会发送到 PF3 口，驱动 Buzzer 发声。

本实验涉及部分电路图：

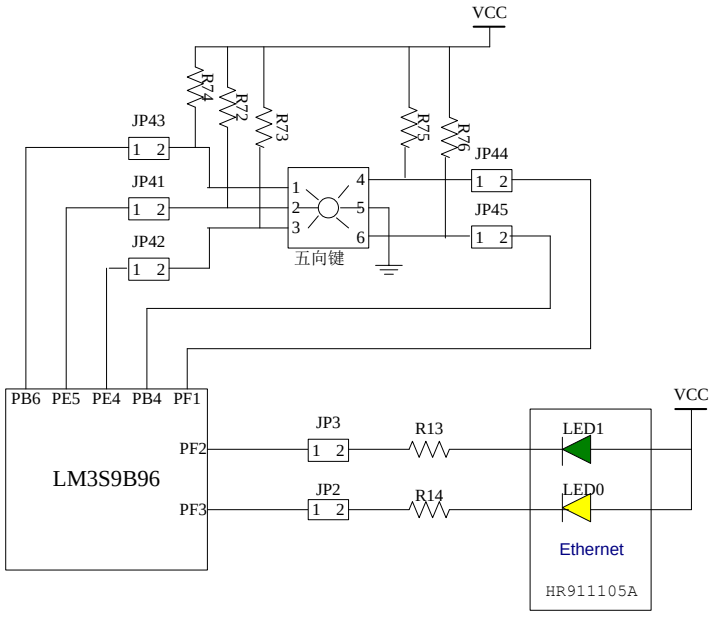


图 8-3

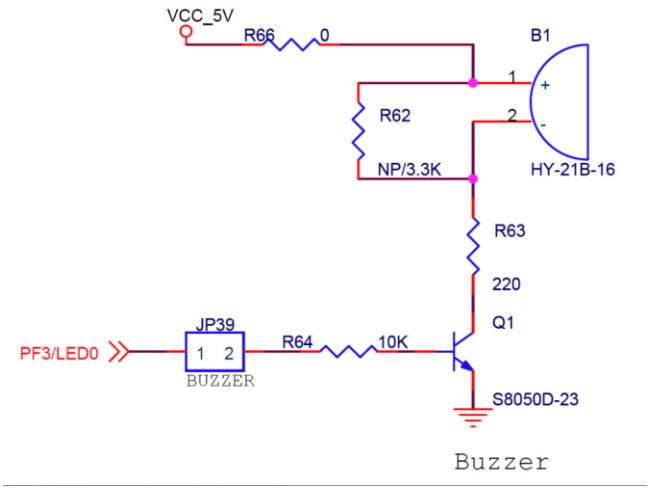


图 8-4 Buzzer

8.7 实验八(1) PWM 呼吸灯实验

■ 实验概述

本次实验目的在于了解 PWM 模块的基本功能，并学会对 PWM 模块的输出周期和占空比进行基本设置，产生对应的数字信号输出，驱动 LED 产生呼吸效果。

■ 实验效果

本实验在上一章 I2C 电子钟的基础上添加了呼吸灯的效果。
板子启动后，网络口绿灯，周期性关闭开启，黄灯则呈现呼吸效果。

■ 实验所用资源与跳帽设置

检查跳帽连接是否正确，跳帽连接如下所示：

表 8-10 跳帽设置

连接跳帽	对应端口	断开跳帽	对应按键
JP3	PF2	JP35	LED1
JP2	PF3	JP39	LED0
JP43	PB6	JP20,JP28	SW_RIGHT
JP41	PE5	JP24	SW2-PRESS
JP42	PE4	JP25	SW3-UP
JP44	PF1	JP30	SW4-DOWN
JP45	PB4	JP48	SW5-LEFT

■ 实验流程图

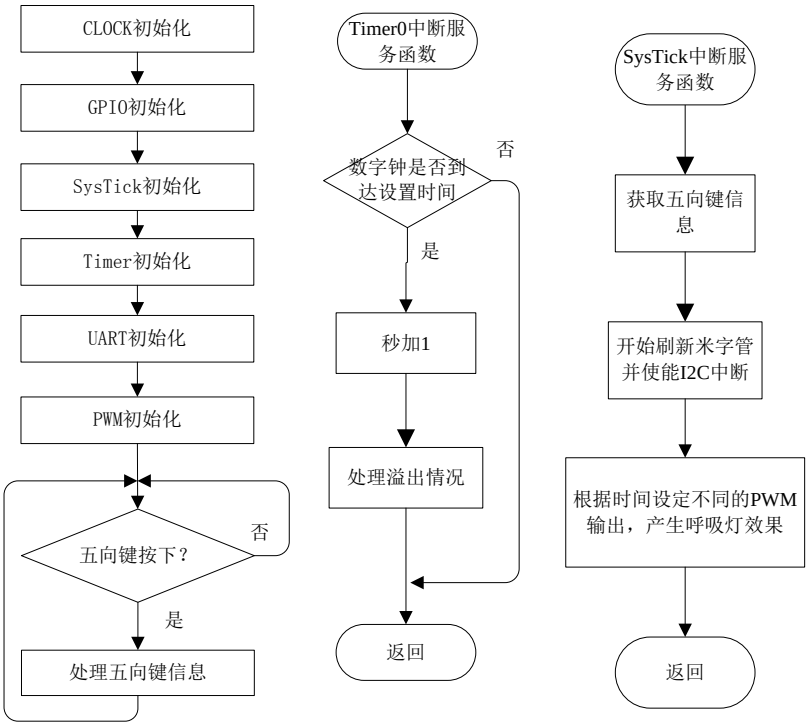


图 8-5 实验程序流程图

■ 实验步骤说明

我们需要在原有工程中对如下文件进行修改，以完成实验。其中 PWMConfigure.c 初始化 PWM 模块，而在 SYSTickISR.c，我们让 PWM 输出占空比随着时间推移发生变化，得到呼吸灯的效果。

■ PWMConfigure.c

建立 PWMConfigure.c 文件，并且加入工程。

PWMConfigure.c 文件负责初始化 PWM，使能 PWM 发生器和输出引脚，并且设置 PWM 输出的周期和占空比。

首先，我们在 PWMConfigure.c 文件中加下列引用：

```
#include "HardwareLibrary.h"
#include "LuminaryDriverLibrary.h"
#include "SysCtlConfigure.h"
#include "PWMConfigure.h"
```

这些引用文件使得我们可以调用芯片使能和设置函数。其中 PWMConfigure.h 中设置了本节实验使用的 PWM 输出的周期。

之后我们定义自己封装的 PWM 初始化函数 PWMInitial( )。

```
void PWMInitial(void)
{
}
```

然后在 PWMInital( )我们依次调用函数 SysCtlPeripheralEnable( )，SysCtlPWMClockSet( )， PWMGenConfigure( )， PWMGenPeriodSet( )，PWMPulseWidthSet( )，PWMOutputState( )，PWMGenEnable( )来初始化 PWM 模块。

其中函数 SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM)使能 PWM 模块， SysCtlPWMClockSet(SYSCTL_PWMDIV_1)设置时钟分频为 1，函数 PWMGenConfigure( )初始化 PWM 信号发生器，因为我们这里使用了 PWM 模块的引脚 5 输出到 LED0，所以我们选择初始化对应的 PWM_GEN_2 信号发生器，并使能输出引脚 5，我们使用 PWMGenConfigure(PWM_BASE,PWM_GEN_2, PWM_GEN_MODE_DOWN |PWM_GEN_MODE_NO_SYNC) 初始化 PWM_GEN_2 。函数 PWMGenPeriodSet()和 PWMPulseWidthSet()设置 PWM 输出周期和占空比，初试的占空比和周期均设为 BreathingLEDPeriod,点亮 LED0。然后我们使用函数 PWMOutputState( ) 使能 PWM 模块的输出引脚 5, PWMOutputState(PWM_BASE,PWM_OUT_5_BIT,true) 。至此，PWM 模块初始化完成，然后我们使能 PWM 信号发生器，点亮 LED0。

具体函数的参数设置可以参照前面的库函数介绍。

■ SysTickISR.c

为了产生呼吸灯效果，我们在 SysTickISR.c 文件中加入下列代码，使得随着时间变化，PWM 模块输出不同占空比的方波到 LED 上，从而实现 LED 呼吸灯的效果。

```
switch (BreathingLEDPWM)
    {
        case 1:
        {
            BreathingLEDFlag=0;
            break;
        }
        case BreathingLEDPeriod/2:
        {
            if (BreathingLEDFlag)
                LEDOn(LED_1);
            else
                LEDOff(LED_1);
            break;
        }
        case BreathingLEDPeriod:
        {
            BreathingLEDFlag=1;
            break;
```

```
            }
            default:   break;
        }
        if (BreathingLEDFlag)
            BreathingLEDPWM=BreathingLEDPWM-1;
        else
            BreathingLEDPWM=BreathingLEDPWM+1;
        PWMPulseWidthSet(PWM_BASE, PWM_OUT_5,
                         //处理非线性
                         //sqrt(BreathingLEDPWM)*sqrt(BreathingLEDPeriod));

BreathingLEDPeriod-BreathingLEDPWM*BreathingLEDPWM/BreathingLEDPeriod+1);
                         //不处理非线性
                         //BreathingLEDPWM);
```

这里我们利用了一个BreathingLEDPWM变量实现了对PWM输出的周期性变化。

每触发一个 SYSTick 中断时，我们可以通过调整 PWM 的占空比来调节 PWM 输出的方波，而每次触发中断我们都会改变 BreathingLEDPWM 的值，在不同的 PWM 输出作用下，LED 就会呈现出呼吸效果。

8.8 实验八(2) PWM BUZZER 实验

■ 实验概述

本实验在上一章 I2C 总线实验的基础上，让电子时钟在运行和设置的过程中利用 PWM 输出和 Buzzer 模块发出不同的蜂鸣声。

本次实验目的在于了解 PWM 模块的基本功能，并学会对 PWM 模块进行基本设置，产生数字信号输出。

■ 实验效果

启动时，将会有短暂蜂鸣，并且初始状态下米字管上会显示“CTM3”字样，LED1 灯亮。

按任意键后，电子钟开始走动，并且蜂鸣器开始发出声音，默认情况下，电子钟走到 59:59 时将会停止。

按下中键，会有提示音出现，之后进入电子钟的设置阶段，或者完成电子钟的设置。若进入设置阶段，定期蜂鸣和 LED 轮闪效果将会消失，若结束设置，阶段，则恢复原来状态。

在设置状态时按下左键或右键，对电子钟的设置将会在分和秒之间切换，切换时，对应位置会闪烁。

在设置电子钟的时间时，上键和下键将对时钟的情况进行具体调整。上键为加时间，下键为减时间。

在设置出新的时间后，电子时钟将会运作至设定时间。

■ 实验所用资源与跳帽设置

检查跳帽连接是否正确，跳帽连接如下所示：

表 8-11 跳帽设置

连接跳帽	对应端口	断开跳帽	对应按键
JP3	PF2	JP35	LED1
JP39	PF3	JP2	BUZZER
JP43	PB6	JP20,JP28	SW_RIGHT
JP41	PE5	JP24	SW2-PRESS
JP42	PE4	JP25	SW3-UP
JP44	PF1	JP30	SW4-DOWN
JP45	PB4	JP48	SW5-LEFT

■ 实验流程图

见图 8-6。

■ 实验步骤说明

我们在 PWMConfigure.c 文件中自定义一个 PWM 模块的初始化函数，并封装一个产生蜂鸣声的函数。一开始，我们初始化 PWM 模块，然后在电子钟走动和设置的过程中，我们调用蜂鸣函数发出不同蜂鸣声。

在 PWMConfigure.h 中我们已经预先定义了 START_SOUND_PERIOD，ALARM_SOUND_PERIOD 和 SET_SOUND_PERIOD，三种不同的 PWM 输出占空比，对应电子钟走动和设置时发出的不同蜂鸣声。

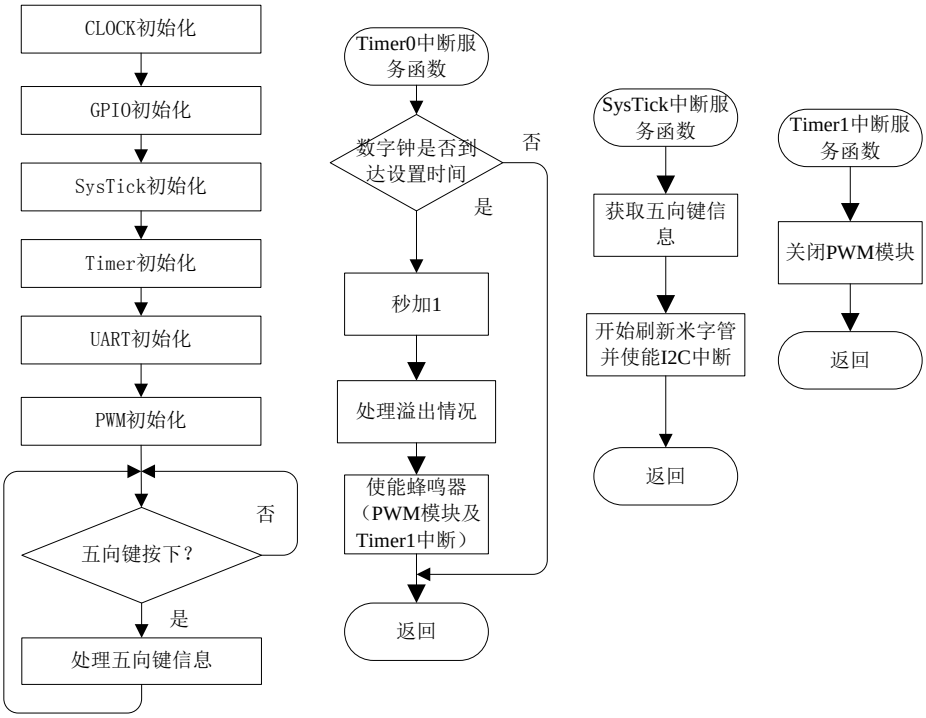


图 8-6 实验程序流程图

■ PWMConfigure.c

和第一个实验一样，我们建立文件 PWMConfigure.c 并将其加入工程，然后在 PWMConfigure.c 文件中加下列引用：

```
#include "HardwareLibrary.h"
#include "LuminaryDriverLibrary.h"
#include "SysCtlConfigure.h"
#include "PWMConfigure.h"
```

之后定义自己封装的 PWM 初始化函数 PWMInitial( )。

```
void PWMInitial(void)
{
}
```

在 PWMInital( )中，我们依次调用函数 SysCtlPeripheralEnable( )，SysCtlPWMClockSet( )， PWMGenConfigure( )， PWMGenPeriodSet( )，PWMPulseWidthSet( )，PWMOutputState( )初始化 PWM 模块，然后我们使能 PWM 发生器一段时间，生成一段启动蜂鸣。

其中函数 SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM)使能 PWM 模块，SysCtlPWMClockSet(SYSCTL_PWMDIV_1)设置时钟分频为 1，函数 PWMGenConfigure( )初始化 PWM 信号发生器，因为我们这里使用了 PWM 模块的引脚 5 输出到 BUZZER，所以我们选择初始化对应的 PWM_GEN_2 信号发生器，并使能输出引脚 5，我们使用

PWMGenConfigure(PWM_BASE,PWM_GEN_2, PWM_GEN_MODE_DOWN |PWM_GEN_MODE_NO_SYNC) 初始化 PWM_GEN_2 。函数 PWMGenPeriodSet()和 PWMPulseWidthSet()设置 PWM 输出周期和占空比，然后我们使用函数 PWMOutputState( )使能 PWM 模块的输出引脚 5, PWMOutputState(PWM_BASE,PWM_OUT_5_BIT,true) 。至此，PWM 模块初始化完成，然后我们使能 PWM 信号发生器，产生一段启动音，代码如下所示，这里我们使用了一个 Delay 函数来控制启动音输出的时间。

```
PWMGenEnable(PWM_BASE,PWM_GEN_2);
SysCtlDelay(TheSysClock/15);
PWMGenDisable(PWM_BASE,PWM_GEN_2);
```

接着我们定义自己蜂鸣函数 void BuzzerBeep(unsigned long ulPeriod,unsigned long ulTime)。这个函数接受参数 ulPeriod 和 ulTime，它们分别设置了 PWM 输出的占空比和输出时间，我们利用一个 Timer 来定时禁能 PWM 发生器，从而控制 PWM 的输出时间。利用 Timer 控制 PWM 输出时间的代码如下所示。

```
TimerLoadSet(TIMER1_BASE,TIMER_A,ulTime);
PWMGenEnable(PWM_BASE,PWM_GEN_2);
TimerEnable(TIMER1_BASE,TIMER_A);
```

这里调用了 Timer1，当定时结束时，PWM 信号发生器将会被禁能。

■ TimerISR.c

在 TimerISR.c 的 Timer0A_ISR(void)，我们定期更新电子钟计数值，在这里我们在更新计数值的同时，调用 PWMConfigure.c 中的 BuzzerBeep()函数，使电子钟运行时发出蜂鸣声。

而在 Timer1A_ISR(void)中我们禁能了 PWM 信号发生器。

■ main.c

main 函数中，我们在的初始化中调用 PWMInitial()初始化 PWM 模块。并在响应电子钟设置的操作中加入函数 BuzzerBeep( )的调用，实现设置时的蜂鸣声。

第 9 章　LCD 字符驱动

在日常生活中，我们对液晶显示屏(LCD, Liquid Crystal Display)并不陌生，它广泛应用与计算器、电子表等电子产品中，主要是用来显示数字、专用符号和图形等。其具有显示质量高，体积小、重量轻，功耗低等一些优点。

LCD 是利用液晶的物理特性，通过调整施加在 LCD 电极上电位信号的相位、峰值、频率等，建立驱动电场，对其显示区域进行控制，从而显示出图形。LCD 的分类方法很多，例如，根据不同的显示方式可以将其分为字段型、点阵型、字符型等；根据显示的颜色可以分为黑白显示屏、多灰度显示屏、彩色显示屏等；如果根据驱动方式来分，又可以分为静态驱动、单纯矩阵驱动和主动矩阵驱动。

9.1　1602 字符型 LCD

字符型 LCD 是专门用于显示数字、字母、图形符号及少量自定义符号的液晶显示屏，这类显示屏把 LCD 控制器、点阵驱动器、字符存储器、显示体及少量的阻容元件等集成为一个液晶显示模块。鉴于字符型液晶显示模块目前在国际上已经规范化，其电特性及接口特性是统一的，因此，只要设计出一种型号的接口电路，在指令上稍加修改即可使用各种规格的字符型液晶显示模块。下面以 1602 字符型 LCD 为例来介绍一下字符型 LCD。

1602 字符型 LCD 模块的应用非常广泛，尽管各厂家的对其命名不尽相同，但实际上这些产品都是同样规格的 1602 模块或兼容模块。1602 字符型 LCD 模块最初采用的 LCD 控制器采用的是 HD44780，在各厂家生产的 1602 模块当中，基本上也都采用了与之兼容的控制 IC，所以从特性上基本上是一样的。当然，很多厂商提供了不同的字符颜色、背光色之类的显示模块。

通常见到的 1602 字符型 LCD 的规格基本如下：

表 9-1 1602 LCD 基本规格

显示容量:	16×2 个字符
芯片工作电压:	4.5~5.5V
工作电流:	2.0mA(5.0V)
模块最佳工作电压:	5.0V
字符尺寸:	2.95×4.35(W×H)mm

9.1.1　1602 字符型 LCD 结构

1） 1602 字符型 LCD 的外部结构

下图为一般的 1602 LCD 模块的结构尺寸示意图。

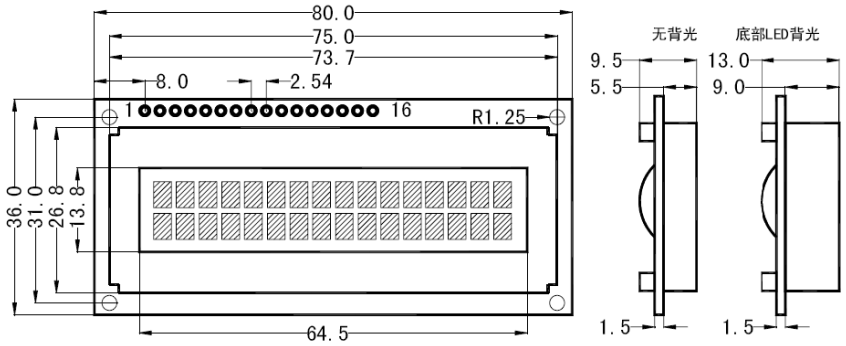


图 9-1　1602 LCD 模块结构尺寸图

模块的引脚说明如下表：

表 9-2　1602 LCD 引脚说明

编号	符号	引脚说明	编号	符号	引脚说明
1	VSS	电源地	9	D2	Data　I/O
2	VDD	电源正极	10	D3	Data　I/O
3	VL	液晶显示偏压信号	11	D4	Data　I/O
4	RS	数据/命令选择端(H/L)	12	D5	Data　I/O
5	R/W	读/写选择端(H/L)	13	D6	Data　I/O
6	E	使能信号	14	D7	Data　I/O
7	D0	Data　I/O	15	BLA	背光电源正极
8	D1	Data　I/O	16	BLK	背光电源负极

当然，有的模块是不带背光的，这时候管脚 15 和 16 就是没有意义的。

2） 1602 字符型 LCD 内部模块

1602 内部结构由显示数据寄存器(Data Display RAM, DDRAM)、字符发生器 ROM(Character Generator ROM, CGROM)、字符发生器 RAM(Character Generator RAM, CGRAM)、指令寄存器((InstructionRegister,IR)、数据寄存器(DataRegister, DR)、忙标志(BusyFlag, BF)、地址计数器(AddressCounter, AC)及时序发生等电路组成。

a) DDRAM 用以存放要显示的数据，只要将标准的 ASCII 码放入 DDRAM，内部控制线路就会自动将数据传送到显示器上，并显示出该 ASCII 码对应的字符。

b) CGROM 可由 8 为字符码生成 160 种 5×7 点阵字符和 32 种 5×10 点阵字符。

c) CGRAM 是提供给用户自定义特殊字符用的，它的容量仅为 64 字节，编址为 00～3FH。作为字符字模使用的仅是一个字节中的低 5 位，每个字节的高 3 位留给用户作为数据存储器使用。如果用户自定义字符由 5×7 点阵构成，可定义 8 个字符。

d) IR 负责存储 MCU 要写给 LCD 的指令码。当 RS 及 R/W 引脚信号为 0 且 E 引脚信号由 1 变为 0 时，D0～D7 引脚上的数据便会存入到 IR 寄存器中。

e) DR 负责存储单片机要写到 CGRAM 或 DDRAM 的数据。因此，可将 DR 视为一个数据缓冲区，当 RS 及 R/W 引脚信号为 1 且 E 引脚信号由 l 变为 0 时，读取数据。当 RS 引脚信号为 1，R/W 引脚信号为 0 且 E 引脚信号由 1 变为 0 时，存入数据。

f) BF 为忙碌信号。当 BF 为 1 时，不接收单片机送来的数据或指令；当 BF 为 0 时，接收外部数据或指令。所以，在写数据或指令到 LCD 之前，必须查看 BF 是否为 0。

g) AC 负责计数写入/读出 CGRAM 或 DDRAM 的数据地址，AC 依照 MCU 对 LCD 的设置值而自动修改它本身的内容。

9.1.2 1602 LCD 的控制指令

1602 LCD 内部的控制器共有 11 条控制指令，如表 9.3 所示：

9.1.3 1602 LCD 的基本操作和驱动

1602 LCD 的基本的操作分为以下四种：

1) 状态字读操作：输入 RS=0、R/W=1、EP=1；输出：DB0~DB7 读出为状态字；

2) 数据读出操作：输入 RS=1、R/W=1、EP=1；输出：DB0~DB7 读出为数据；

3) 指令写入操作：输入 RS=0、R/W=0、EP=上升沿；输出：无；

4) 数据写入操作：输入 RS=1、R/W=0、EP=上升沿；输出：无。

由于其是一个慢显示器件，所以在正常工作执行每一条指令之前都一定要确认模块的 BF 标志是否为低电平，否则指令失效。

1) 读操作

读操作的时序图如下：

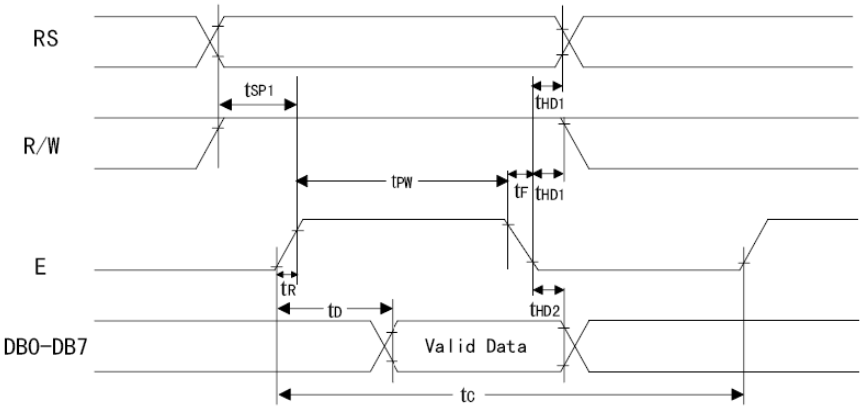


图 9-2 1602 LCD 读操作时序

表 9-3　LCD 内部控制指令

序号	指令	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	说明
1	清屏	0	0	0	0	0	0	0	0	0	1	DD RAM 的内容全部被清除，光标回原位(屏幕的左上角)，AC=0
2	归位	0	0	0	0	0	0	0	0	1	*	光标和光标所在位的字符回原位，但 DDRAM 单元内容不变，AC=0
3	输入方式设置	0	0	0	0	0	0	0	1	I/D	S	A=1:数据读写操作后，AC 自动增 1；A=0:数据读写操作后，AC 自动减 1。 S=1:当数据写入 DDRAM 显示将全部左移(A=1)或全部右移(A=0)； S=0：显示不移动，光标左移(A=1)或右移(A=0)。
4	显示开/关控制	0	0	0	0	0	0	1	D	C	B	D：显示控制，D=1，开显示(Display ON)；D=0，关显示(Display OFF)。 C：光标控制，C=1，开光标显示；C=0，关光标显示。 B：闪烁控制，B=1，光标所指的字符同光标一起交变闪烁；B=0，不闪烁。
5	光标或字符移位	0	0	0	0	0	1	S/C	R/L	*	*	该指令使光标或画面在没有对 DDRAM 进行读写操作时被左移或右移，不影响 DDRAM。S/C=0、R/L=0，光标左移一个字符位，AC 自动减 1；S/C=0、R/L=1，光标右移一个字符位，AC 自动加 1；S/C=1、R/L=0，光标和画面一起左移一个字符位；S/C=1、R/L=1，光标和画面一起右移一个字符位。
6	功能设置	0	0	0	0	1	DL	N	F	*	*	DL=1，8 位数据总线 DB7～DB0；DL=0，4 位数据总线 DB7～DB4， N：设置显示行数，N=1，2 行显示；N=0，1 行显示。 F:设置点阵模式，F=0，5×7 点阵；F=1，5×10 点阵。
7	CGRAM 地址设置	0	0	0	1	CGRAM 地址						地址码 A5～A0 被送入 AC 中，在此后，可以将用户自定义的显示字符数据写入 CG RAM 或从 CG RAM 中读出
8	DDRAM 地址设置	0	0	1	DDRAM 地址							若是一行显示，地址码 A6～A0＝00～4FH 有效；若是二行显示，首行址码 A6～A0＝00～27H 有效，次行址码 A6～A0＝40～67H 有效
9	读 BF 标志或 AC 值	0	1	BF	AC 地址							BF 为内部操作忙标志，BF=1，忙；BF=0，不忙。AC6～AC0 为地址计数器 AC 的值。当 BF=0 时，送到 DB6～DB0 的数据(AC6～AC0)有效
10	写入 CGRAM 或 DDRAM	1	0	Data								该指令根据最近设置的地址，将数据写入 DD AM 或 CG RAM 中
11	读 CGRAM 或 DDRAM 数据	1	1	Data								该指令根据最近设置的地址，从 DDRAM 或 CG RAM 读数据到总线 D7～D0

注：*号表示该位数据可为 0 或 1

相应代码如下

```
//读状态
unsigned char ReadStatusLCD(void)
{
 LCD_Data=0xFF; //置高用于读取数据
 LCD_RS=0;//RS=0 时读取数据
 LCD_RW=1; //RW=1 读取
 LCD_E=0;//若晶振速度太高可以在这后加小的延时
 LCD_E=0;//延时管脚 E 下降沿数据写入
 LCD_E=1;//管脚 E 恢复高电平
 while(LCD_Data & 0x80); //检测忙信号
 return(LCD_Data);
}
```

2) 写操作

写操作的时序图如下：

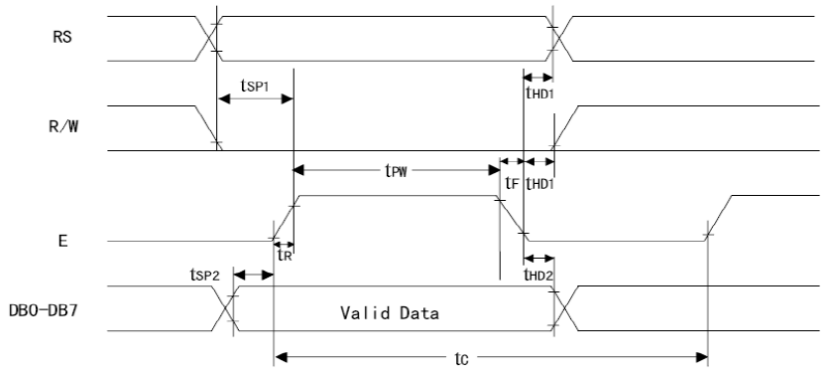


图 9-3 1602 LCD 写操作时序

写入显示数据的代码如下：

```
// 写数据：
void WriteDataLCD(unsigned char WDLCD)
{
 ReadStatusLCD(); //检测忙
 LCD_Data=WDLCD;
 LCD_RS=1;  //RS=1 时写入数据
 LCD_RW=0;  //RW=0 时写入
 LCD_E=0; //若晶振速度太高可以在这后加小的延时
 LCD_E=0; //延时管脚 E 下降沿数据写入
 LCD_E=1;  //管脚 E 恢复高电平
}
```

写指令的代码如下：

```
// 写指令：
void WriteCommandLCD(unsigned char WCLCD,BuysC)
```

```
//BuysC 为 0 时忽略忙检测
{
 if(BuysC) ReadStatusLCD(); //根据需要检测忙
 LCD_Data=WCLCD;
 LCD_RS=0; //RS=0 时写入命令
 LCD_RW=0;
 LCD_E=0;
 LCD_E=0;
 LCD_E=1;
}
```

3) 字符显示

显示字符时要先输入显示字符的地址，即告诉模块在哪里显示字符。1602 LCD 的控制器内置有 80 个 byte 的显存，而 1602 LCD 只有两行×16 个字符的显示区域，所以显存中有些地址是无法对应上 LCD 屏的，下图为显存地址对应图：

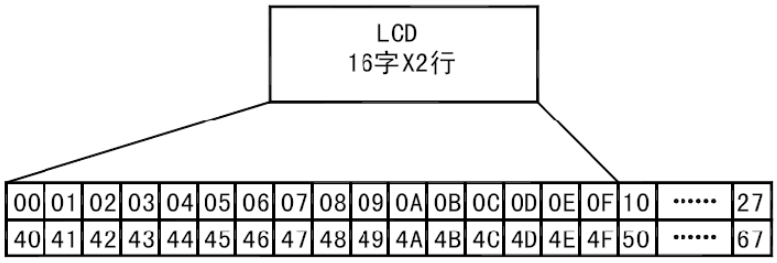


图 9-4 1602 LCD 内部显示地址

例如第二行第一个字符的地址是 40H，那么是否直接写入 40H 就可以将光标定位在第二行第一个字符的位置呢？这样不行，因为写入显示地址时要求最高位 D7 恒定为高电平所以实际写入的数据应该是 01000000B（40H）+10000000B（80H）＝11000000B（C0H）。

1602 液晶模块内部的字符发生存储器（CGROM）已经存储了 160 个不同的点阵字符图形，如下表。

这些字符有：阿拉伯数字、英文字母的大小写、常用的符号、和日文假名等，每一个字符都有一个固定的代码，比如大写的英文字母“A”的代码 01000001B（41H），显示时模块把地址 41H 中的点阵字符图形显示出来，我们就能看到字母“A”。另外，字符库中 0x00~0x0F 未定义,留给使用者的自定义字符使用。但只能使用 0x00~0x07 或者 0x08~0x0F 之一。

4) 1602 LCD 初始化

在液晶模块显示前，必须先进行初始化。初始化时需要向液晶屏模块写入一些控制命令，命令可以参见上面的控制指令系统。下图是 1602 LCD 初始化（或复位）的一般过程。

表 9-4 字符代码与图形对应图

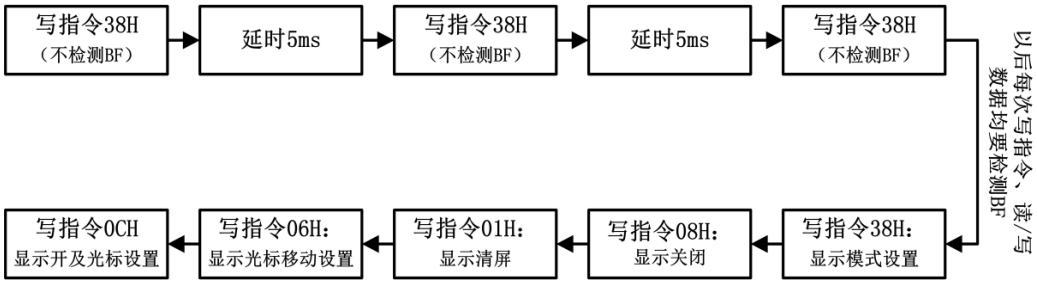


图 9-5　1602 LCD 初始（复位）的一般过程

初始化代码如下：

```
//   LCD 初始化：
void LCD_Init(void)
{
  unsigned char pos;
  LCD_Data = 0;
  WriteCommandLCD(0x38,0); //三次显示模式设置，不检测忙信号
  Ldelay(150);
  WriteCommandLCD(0x38,0);
  Ldelay(150);
  WriteCommandLCD(0x38,0);
  Ldelay(150);
  WriteCommandLCD(0x38,1); //显示模式设置,开始要求每次检测忙信号
  WriteCommandLCD(0x08,1); //关闭显示
```

```
  WriteCommandLCD(0x01,1); //显示清屏
  WriteCommandLCD(0x06,1); //  显示光标移动设置
  WriteCommandLCD(0x0C,1); //  显示开及光标设置
  Ldelay(2000);
}
```

9.2 Kitronix 液晶屏及其控制

9.2.1 Kitronix 液晶屏

在 S700 教学实验板上提供了一块 320×240 像素分辨率的 TFT 液晶图形显示屏。该液晶屏为 Kitronix 公司的 K350QVG-V1-F 显示屏，具有如下特性：

- 3.5 寸、262K 色 TFT 液晶显示模块
- 4 线电阻触摸屏
- 320×RGB×240 分辨率
- 有较宽的工作温度（-20℃～+70℃）
- 白色 LED 背光灯
- 使用 SSD2119 控制器
- 8 位或 16 位的 8080 系统接口，串行或并行 RGB 接口
- 有内部升压模块
- 逻辑电压：3.3V（典型值）
- 外形尺寸：76.90（长）×63.90（宽）×4.40（高）mm
- 面积：72.88（长）×55.36（宽）mm
- 观察方向：12 点

9.2.2 SSD2119 控制器

■ SSD2119 简介

SSD2119 是将电源模块、门驱动、源极驱动集成在芯片中的一款 TFT LCD 驱动芯片。在 320×240 分辨率下，它可以驱动最多 262K 色的液晶屏。

SSD2119 芯片同时集成了控制功能，具有 172800（320×240×18/8）字节的图形显示数据内存（GDDRAM），可以和普通单片机通过 8/9/16/18bits 6800 系列/8080 系列的并行或串行接口通信。

SSD2119 内嵌 DC-DC 转换和电源发生器，可以用最少的外部器件来提供所有必需的驱动电压。它可工作在 1.4V 以下，有几种不同的节能模式，适用于各种便携式电池驱动产品。为了显示动画图像，SSD2119 集成了辅助的 18/6bits 视频接口（VSYNC、 HSYNC、 DOTCLK、DEN），还可以通过软件指令调整集成的 Gamma 控制电路来提供灵活和优质的显示。

图 9-6 是 SSD2119 的原理框图。其中，系统接口（System Interface）由 3 个功能模块组成，主要用来驱动 6800 系列并行接口、8080 系列高速并行接口、3 线串行外设接口或者 4 线串行外设接口，不同接口的选择通过 PS3，PS2，PS1，PS0 控制。RGB 接口单元由 D[17:0]、 HSYNC、VSYNC、 DOTCLK

和 DEN 信号组成。当选择 RGB 接口时，显示操作将和外部控制信号 HSYNC、VSYNC、DOTCLK 同步。当 DEN 使能写操作后，控制信号控制数据的写入。

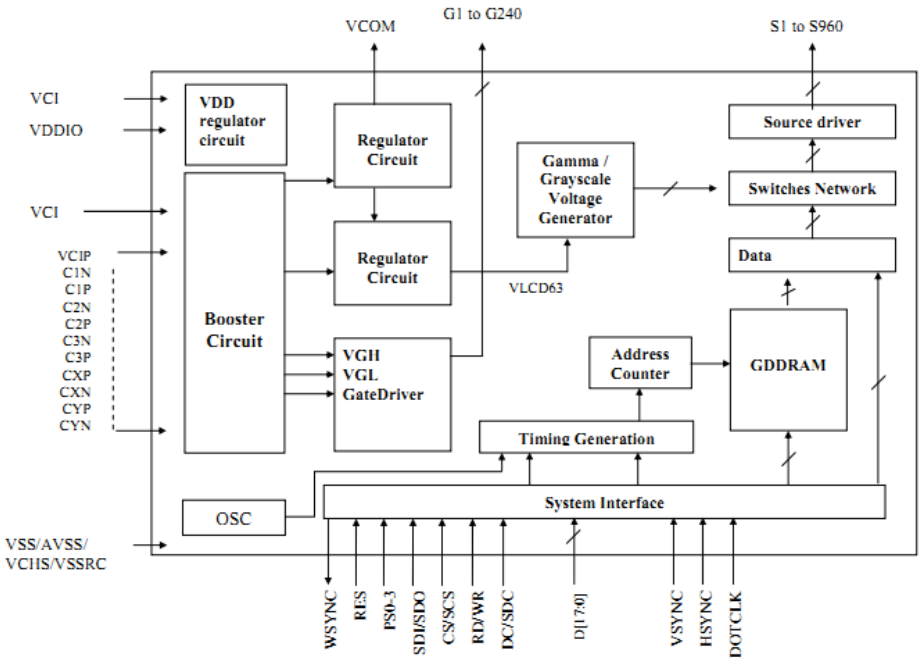


图 9-6　SSD2119 控制芯片原理图

振荡器（OSC）：为片上低功耗晶振，用来给 DC-DC 电压转换模块提供时钟。

时钟发生器（Timing Generator）：产生内部电路的时钟信号。

地址计数器（Address Counter）：用来给 GDDRAM 分配地址。当设置地址指令写入 IR 后，地址信息从 IR 传送到 AC。当向 GRAM 写入后，地址控制器自动加 1(或者减 1)，读取数据后，地址控制器不进行更新。

图形显示数据内存(GDDRAM)：是比特映射静态 RAM，大小 320 RGB×240×18/18。可以通过软件指令控制图形显示区域和效果。

伽马/灰度电压发生器（Gamma/Grayscale Voltage Generator）：用来产生与灰度图对应的 LCD 驱动电路。

助推器和稳压电路（Booster and Regulator Circuit）：产生 VGH、VGI、VCOM 和 VLCD0~63 电压。

数据锁存（Data Latches）：锁存要现实的数据。

液晶驱动电路（Liquid Crystal Driver Circuit）：用于驱动液晶屏显示图形。

■ SSD2119 的控制

同 1602 类似，为完成显示，首先必须为 LCD 屏幕上个像素点编址。表 9.5 是 SSD2119 内 GDDRAM 对应 LCD 上像素点的地址表。

因屏幕分辨率为 320×240，则水平地址从 0000H 到 013FH，垂直地址从 0000H 到 00EFH。可以看出，若地址为 00XXH，0YYYH，则 XX 应表示水平地址，YYY 应表示垂直地址。另外，表 9-5 中的 S0-S959 用来表示每个像素点的 RGB 分量。

表 9-5　LCDGDDRAM 地址表

	RL=1	S0	S1	S2	S3	S4	S5	S6	S7	S8	…	S954	S955	S956	S957	S958	S959	
	RL=0	S959	S958	S957	S956	S955	S954	S953	S952	S951	…	S5	S4	S3	S2	S1	S0	
	BGR=0	R	G	B	R	G	B	R	G	B	…	R	G	B	R	G	B	Vertical address
	BGR=1	B	G	R	B	G	R	B	G	R	…	B	G	R	B	G	R	
TB=1	TB=0																	
G0	G239	0000H,0000H			0000H, 0001H			0000H, 0010H			…	0000H, 013EH			0000H, 013FH			0
G1	G238	0001H,0000H			0001H, 0001H			0001H, 0010H			…	0001H, 013EH			0001H, 013FH			1
G2	G237	0010H,0000H			0010H, 0001H			0010H, 0010H			…	0010H, 013EH			0010H, 013FH			2
G3	G236	0011H,0000H			0011H, 0001H			0011H, 0010H			…	0011H, 013EH			0011H, 013FH			3
G4	G235	0100H,0000H			0100H, 0001H			0100H, 0010H			…	0100H, 013EH			0100H, 013FH			4
. . .	. . .	. . .			. . .			. . .			. . .	. . .			. . .			. . .
G236	G3	013CH, 0000H			013CH, 0001H			013CH, 0010H			…	00ECH, 013EH			00ECH, 013FH			236
G237	G2	013DH, 0000H			013DH, 0001H			013DH, 0010H			…	00EDH, 013EH			00EDH, 013FH			237
G238	G1	013EH, 0000H			013EH, 0001H			013EH, 0010H			…	00EEH, 013EH			00EEH, 013FH			238
G239	G0	013FH, 0000H			013FH, 0001H			013FH, 0010H			…	00EFH, 013EH			00EFH, 013FH			239
Horizontal address		0			1			2			…	318			319			

知道了各像素点的地址后，还必须要结合一系列的控制指令才能正确的在屏幕上显示内容，SSD2119 涉及的控制指令比 1602 复杂得多，各项指令的详细说明可参看 SSD2119 的数据手册，下面只简单提一下一些常用指令。

a)　晶振开关指令(R00H)(POR=0000H)：

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	OSCEN
POR		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

当 OSCEN=1 时打开晶振，当 OSCEN=0 时关闭晶振。

b)　驱动输出控制(R01H)(POR=3AEFH)：

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	0	RL	REV	GD	BGR	SM	TB	0	MUX7	MUX6	MUX5	MUX4	MUX3	MUX2	MUX1	MUX0
POR		0	0	1	1	1	0	1	0	1	1	1	0	1	1	1	1

REV：显示灰度级反转控制，当 REV=1 时 LCD 显示的图像灰度级反转。

GD：输出选择控制。

GD		Left Side	Right Side
0	Normal	G1, 3, 5,…, 239	G240, 218, … , 4, 2
1	Flip	G2, 4, 6,…, 240	G239, 317, … , 3, 1

BGR：控制 GDDRAM 中 BGR 写入方式，见表 9.5。

SM：扫描方式控制，当 GD=0，若 SM=0，从 G1 依次扫到 G240；
若 SM=1，则按 G1，G3，…，G239，G2，…，G240 扫描。

RL：源驱动器的输出转变方向控制，见表 9.5。

MUX[7:0]：指定 LCD 的某一行，其值不能超过 240。

TB：TB = 1，从 G1 扫描至 G240；TB=0 从 G240 扫描至 G1。

c)　显示控制指令(R07H)(POR=0000H)：

R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
W	1	0	0	0	PT1	PT0	VLE2	VLE1	SPT	0	0	GON	DTE	CM	0	D1	D0
POR		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

VLE [2:1]：当 VLE1=1 或 VLE2=1 时，用 R41 寄存器中的数据 VL17-0 在第一个屏幕中显示垂直的滚动条。当 VLE1=1，VLE2=1 时，用数据 VL1[8:0]和 VL2[8:0]分别在第一个和第二个屏幕中显示垂直滚动条。

D [1:0]：当 D1=1 时显示打开，当 D1=0 时显示关闭。显示关闭时，显示的数据仍然保留在 GDDRAM 中，通过设置 D1=1 可以立刻让保存在 GDDRAM 中的数据显示出来。当 D[1:0]="01"时，显示屏幕关闭但内部显示操作仍然进行着。当 D[1:0]="00"时，内部显示操作和显示屏幕都停止。GON，DTE，D1，D0 详细控制指令表如下：

GON	DTE	D1	D0	Internal Display Operation	Source output	Gate output
0	0	0	0	Halt	GND	V_{GH}
0	0	0	1	Operation	GND	V_{GH}
1	0	0	1	Operation	GND	V_{GOFFL}
1	0	1	1	Operation	Grayscale level output	V_{GOFFL}
1	1	1	1	Operation	Grayscale level output	Selected gate line: V_{GH} Non-selected gate line: V_{GOFFL}

d) 写数据到 GRAM(R22H)：

R/W	DC	D[17:0]
W	1	WD[17:0] mapping depends on the interface setting

WD [17:0]：将 GDDRAM 中的所有数据转换成 18bit，写数据进入 GDDRAM。数据转换成 18bit 的格式依赖于使用的接口。根据 GDDRAM 中的数据 SSD2119 选择灰度级。数据写入 GDDRAM 后，根据 AM 和 ID 比特位，地址自动更新。

e) 从 GRAM 读数据(R22H)：

R/W	DC	D[17:0]
R	1	RD[17:0] mapping depends on the interface setting

RD[17:0]：从 GDDRAM 中读取 18 比特数据。

f) RAM 地址设置 (R4Eh-R4Fh)：

Reg#	R/W	DC	IB15	IB14	IB13	IB12	IB11	IB10	IB9	IB8	IB7	IB6	IB5	IB4	IB3	IB2	IB1	IB0
R4Eh	W	1	0	0	0	0	0	0	0	XAD8	XAD7	XAD6	XAD5	XAD4	XAD 3	XAD 2	XAD 1	XAD 0
	POR		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R4Fh	W	1	0	0	0	0	0	0	0	0	YAD7	YAD6	YAD5	YAD4	YAD3	YAD2	YAD1	YAD0
	POR		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

YAD [7:0]：在地址计数器 (AC)中初始化 GDDRAM 的 Y 地址

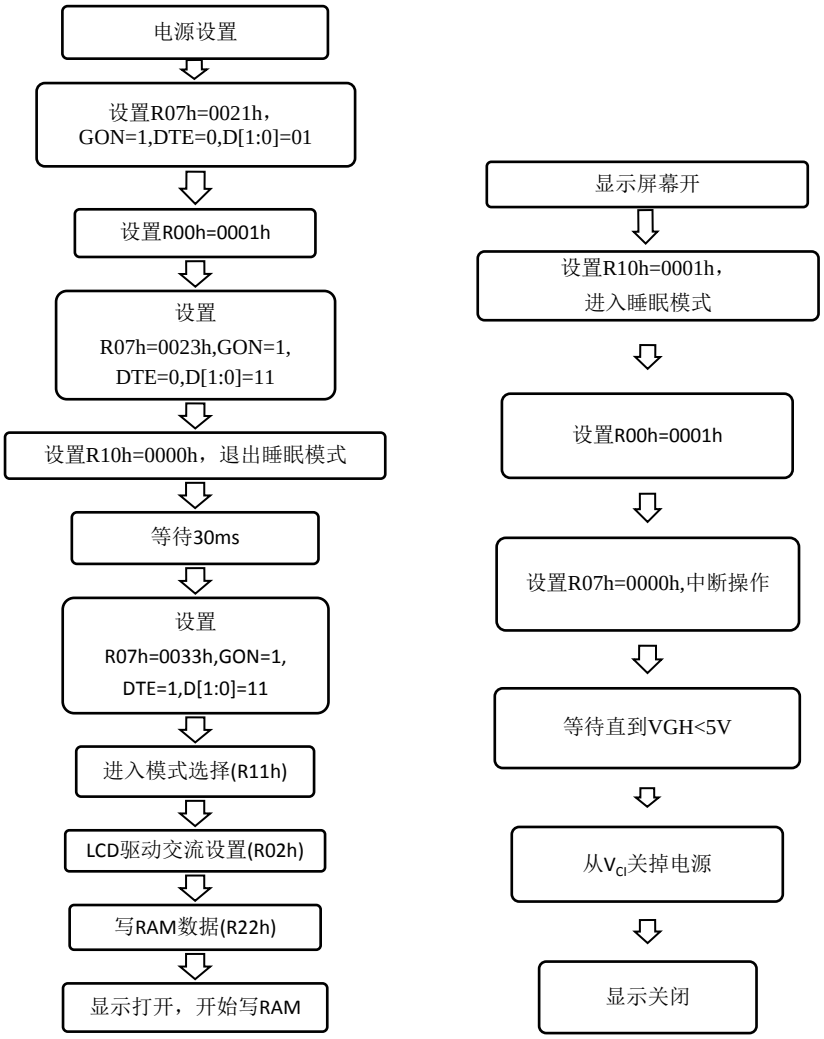
XAD [8:0]：在地址计数器 (AC)中初始化 GDDRAM 的 X 地址

根据 AM，I/D 位的设置，写数据进入 GDDRAM 后，地址计数器会自动更新，因此数据可以连续写入 GDDRAM 中。当数据从 GDDRAM 中读取时地址计数器不会自动更新。在待机模式(standby mode)下不能进行 GDDRAM 地址设置。

■ 工作模式设置流程

SSD2119 芯片可通过一系列的指令设置 LCD 工作在不同的模式下。图 9-7 所示的流程分别实现了 LCD 打开显示和关闭显示的设置。

另外，SSD2119 还可控制 LCD 屏幕在睡眠、深度睡眠等待机模式以减少能耗，具体的设置流程也可以参考其数据手册。



（a）屏幕显示打开流程图　　(b) 屏幕关闭流程图

图 9-7　屏幕工作模式设置流程

9.3 控制原理图

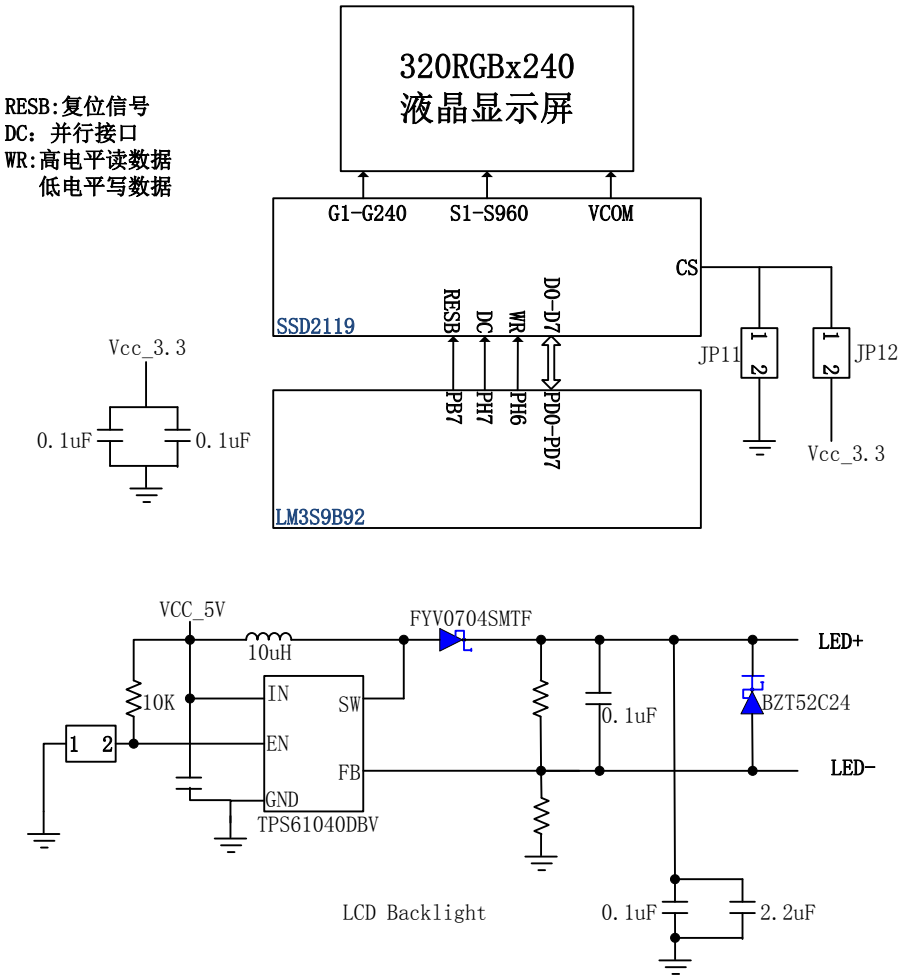


图 9-8　LCD 显示原理图

9.4 本实验使用的资源说明

LED：低电平有效

名称	对应端口
LED1(绿灯)	PF2

五向键：低电平触发

名称	对应端口	对应操作
SW_1	PB6	右
SW_2	PE5	按下
SW_3	PE4	上
SW_4	PF1	下
SW_5	PB4	左

BUZZER：输出给蜂鸣器的 PWM 信号

名称	对应端口
BUZZER	PF3

LCD：

名称	对应端口	功能说明
LCD0	PD0	LCD 数据线 D0
LCD1	PD1	LCD 数据线 D1
LCD4	PD4	LCD 数据线 D4
LCD5	PD5	LCD 数据线 D5
L_RSTN	PB7	LCD 复位信号
L_EN		LCD 开关，连接表示拉低使能 LCD
BLEN		LCD 背光开关，断开表示使能

9.5 Stellaris 库函数

9.5.1 图形库概述

在 LCD 显示时，如果直接使用底层的指令是极其不方便的，德州仪器(TI)为具备图形显示功能的 Stellaris 控制板提供了一套 Stellaris 图形库，该库提供了用于创建用户图形界面的基本图形和部件集，能够极大的方便 LCD 显示。该图形库由三层构成。

1. 第一层，显示驱动层（The display driver layer）。提供了特定显示屏到图形库的接口，负责同显示屏控制器通信、处理比较底层的指令等，包含图形库用于在屏幕上绘图的基本例程和操作，如初始化显示屏等。
2. 第二层，基本图形层（The graphics primitives layer）。提供了一套低层次的绘图操作，这些操作包括画线，圆，文字，位图图像。
3. 第三层，部件层（The widget layer）。该层将一个或多个基本图形封装为部件，应用这些封装好的部件，可以在显示屏上绘制自定义的用户界面。该层还为用户界面提供一定的交互功能。

可以看出，较高层是在前一层的基础上进行扩展，以提供更多的复杂功能。实际上，这三层提供的功能和接口是互不冲突的，即我们可以根据应用的需求和系统的实际情况来灵活的选择利用某一层来实现屏显。

在本次实验中，我们主要用到了第二层提供的某些功能，接下来将简要介绍涉及到的相关内容，第二层未涉及到的部分以及第一层和第三层的内容可以参看 TI 提供的技术文档《Stellaris® Graphics Library USER’S GUIDE》。

9.5.2 基本图形层

为方便绘图操作，基本图形层定义了几个特殊的数据类型来存储绘图所需的相关参数，它们分别是 tContext、tDisplay、tFont 和 tRectangle，本实验中主要使用了 tContext 和 tRectangle 类型。

表 9.6　tContext 类型

定义	typedef struct { long lSize; const tDisplay *pDisplay; tRectangle sClipRegion; unsigned long ulForeground; unsigned long ulBackground; const tFont *pFont; } tContext
成员	**lSize** 该类型结构的大小 **pDisplay** 指向要显示的屏幕 **sClipRegion** 屏幕显示时的裁剪区域(The clipping region) **ulForeground** 屏幕显示时使用的文字或图形的颜色 **ulBackground** 背景色 **pFont** 显示文字的字体
说明	该结构定义了一个用于在屏幕上显示的显示环境。在任何时间，可以同时定义多个显示环境。

表 9.7　tRectangle 类型

定义	typedef struct { short sXMin; short sYMin; short sXMax; short sYMax; } tRectangle
成员	**sXMin** 矩形X坐标（水平）的最小值 **sYMin** 矩形Y坐标（垂直）的最小值 **sXMax** 矩形X坐标（水平）的最大值 **sYMax** 矩形Y坐标（垂直）的最大值
说明	该结构定义了一个矩形的边界，边界内的部分就是矩形的范围。

在显示图形前，必须指定显示的环境（Drawing Context），即：显示操作的对象（指定某屏幕）、颜色、字体以及显示的具体区域等。该环境的初始化使用函数 GrContextInit()完成，其中的颜色可由 GrContextBackgroundSet() 或 GrContextForegroundSet()进行设置，字体则可以由 GrContextFontSet()来设置。

表 9.8　GrContextInit()函数

原型	void GrContextInit(tContext *pContext, const tDisplay *pDisplay)
参数	**pContext** 指向要初始化的显示环境 **pDisplay** 指向描述显示驱动的 tDisplayInfo 结构
返回	无
说明	该函数用于初始化显示环境，这里提供的显示驱动将用于接下来的所有显示操作，默认的裁剪区域也会被设置为显示屏幕的大小。
功能	初始化了一个显示环境

表 9.9　GrContextBackgroundSet()定义

原型	#define GrContextBackgroundSet(pContext, ulValue)
参数	**pContext** 指向要设定的显示环境 **ulValue** 为 24 位的 RGB 颜色
返回	无
说明	该函数指定了某显示环境显示时所使用的背景颜色。
功能	设定背景色

表 9.10　GrContextForegroundSet()定义

原型	#define GrContextForegroundSet(pContext, ulValue)
参数	**pContext** 指向要设定的显示环境 **ulValue** 为 24 位的 RGB 颜色
返回	无
说明	该函数指定了某显示环境进行显示绘图时所使用的颜色。
功能	设定显示使用的颜色

表 9.11　GrContextFontSet()定义

原型	#define GrContextFontSet(pContext, pFnt)
参数	**pContext**　指向要设定的显示环境 **pFnt**　指向显示所要使用的字体
返回	无
说明	该函数用于设定显示字符时所使用的字体。
功能	设定字体

在初始化绘图环境之后，就可以利用相关函数绘制所需的图形或文字。以下是实验中会涉及到的一些函数。

表 9.12　GrStringDrawCentered()函数

原型	#define GrStringDrawCentered(pContext, pcString, lLength, lX, lY, bOpaque)
参数	**pContext** 指向要使用的显示环境 **pcString** 指向要显示的字符串 **lLength** 字符串中应该显示在屏幕上的字符数 **lX** 字符串中心在屏幕X轴上的坐标值 **lY** 字符串中心在屏幕Y轴上的坐标值 **bOpaque** 为true时每个字符的背景都要显示；为false时则不显示（保留背景原样）
返回	无
说明	该函数以提供的坐标为中心在屏幕上显示字符。参数lLength使得可以在停止点不插入NULL字符的情况下，只显示字符串中一部分字符；特别的当该参数的值设为-1时，将显示整个字符串。
功能	显示字符

表 9.13　GrContextDpyHeightGet()函数

原型	#define GrContextDpyHeightGet(pContext)
参数	**pContext** 指向要查询的显示环境
返回	返回以像素为单位高度
说明	该函数用于查询所使用的显示环境的显示高度。
功能	得到显示环境所使用的显示高度

与此类似的，GrContextDpyWidthGet()则是获取宽度。

表 9.14　GrStringHeightGet()函数

原型	#define GrStringHeightGet(pContext)
参数	**pContext** 指向要查询的显示环境
返回	返回以像素为单位的字符串的高度
说明	该函数可以确定字符串的高度，该高度是指字符串顶部和底部之间（包括任何上伸和下伸）的偏移量。
功能	获取字符串的高度

表 9.15　GrRectDraw()函数

原型	void GrRectDraw(const tContext *pContext, const tRectangle *pRect)
参数	**pContext** 指向使用的显示环境 **pRect** 指向要绘制的矩形的参数
返回	无
说明	该函数用于显示矩形，矩形区域由lXMin 、 lXMax、 lYMin和lYMax指定
功能	在屏幕上绘制矩形

表 9.16　GrRectFill()函数

原型	void GrRectFill(const tContext *pContext, const tRectangle *pRect)
参数	**pContext** 指向使用的显示环境 **pRect** 指向要绘制的矩形的参数
返回	无
说明	该函数用于显示一个填充的矩形，矩形区域由lXMin 、 lXMax、 lYMin和lYMax指定。
功能	在屏幕上绘制一个填充的矩形

9.6　实验九　LCD 字符驱动

■ 实验概述

本实验在实验八的基础上增添了 LCD 显示，可以通过 LCD 显示数字钟的运行状态。

该实验旨在了解 LCD 显示的基本原理，通过实验掌握 Stellairs 图形库的基本内容，并能够利用该库函数实现简单的 LCD 显示。主要涉及以下知识点：

- LCD模块的初始化配置

- Stellairs图形库及其函数使用
- LCD显示控制

■ 实验效果：

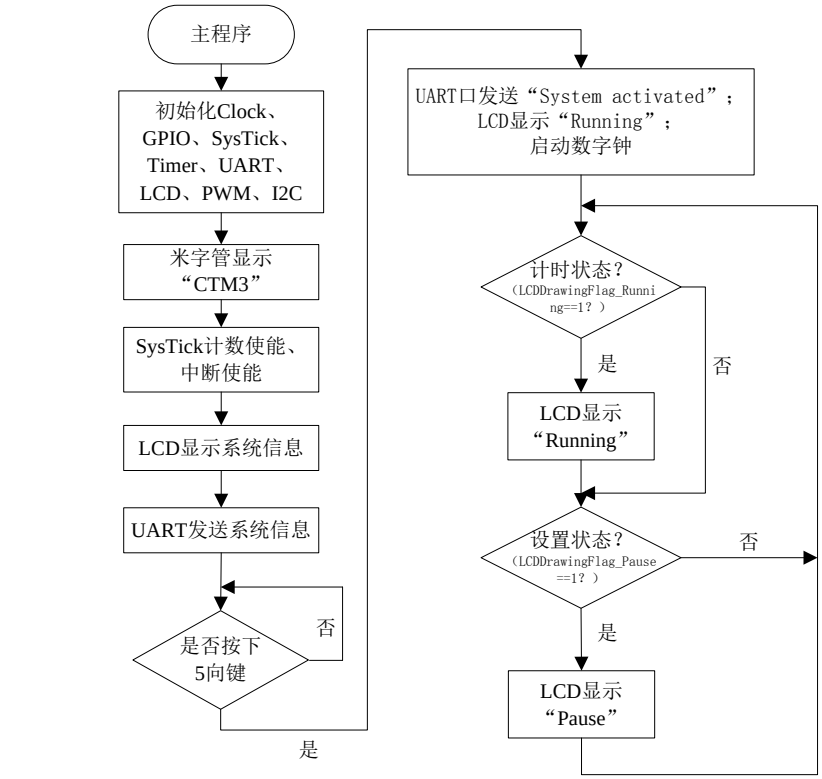
下载完程序后，打开串口调试工具 SSCOM，按下复位键，之后实验效果如下：

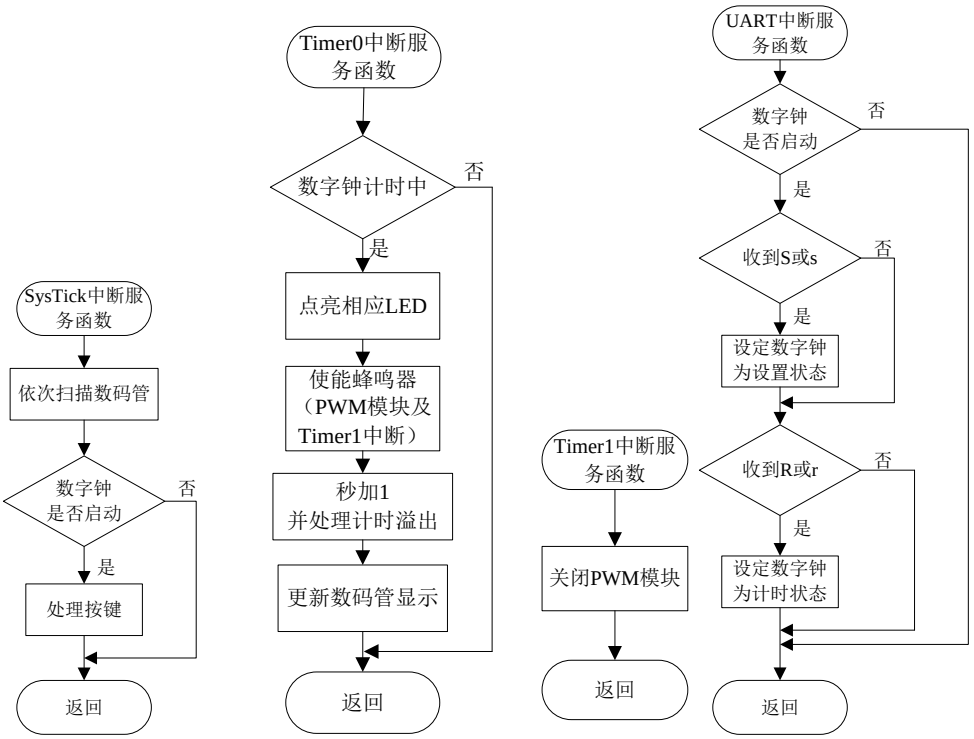
1. 系统初始化时，米字管显示 CMT3，LCD 显示系统信息。
2. 当按下 Press 键时，数字钟启动为计时模式，米字管显示计时值，LCD 显示“Running”。
3. 再次按下 Press 键，计入设置模式，米字管显示目标值，LCD 显示“Pause”，可通过 Up，Down，Right，Left 按键调整计时目标值。
4. 再按下 Press 键，重新回到计时模式。若计时到达目标值则停止计时。

■ 跳帽设置：

连接跳帽	对应端口	断开跳帽	对应按键
JP3	PF2	JP35	LED1
JP41	PE5	JP24	SW2-PRESS
JP42	PE4	JP25	SW3-UP
JP43	PB6	JP20,JP28	SW1-RIGHT
JP44	PF1	JP30	SW4-DOWN
JP45	PB4	JP48	SW5-LEFT
JP39	PF3	JP2	BUZZER
JP38	PB7		LCD_RST\
JP11		JP12	LCD_CS\(下拉)
JP6	PD0	JP29,JP36	LCD_D0
JP7	PD1	JP27,JP37	LCD_D1
JP8	PD4	JP26	LCD_D4
JP9	PD5	JP31	LCD_D5
		JP23	LCD_Backlight

■ 实验流程图


主程序
初始化Clock、GPIO、SysTick、Timer、UART、LCD、PWM、I2C
米字管显示“CTM3”
SysTick计数使能、中断使能
LCD显示系统信息
UART发送系统信息
是否按下5向键
否
是
UART口发送“System activated”；
LCD显示“Running”；
启动数字钟
计时状态？
(LCDDrawingFlag_Running==1？)
是
否
LCD显示“Running”
设置状态？
(LCDDrawingFlag_Pause==1？)
否
是
LCD显示“Pause”


SysTick中断服务函数
依次扫描数码管
数字钟是否启动
否
是
处理按键
返回
Timer0中断服务函数
数字钟计时中
否
是
点亮相应LED
使能蜂鸣器（PWM模块及Timer1中断）
秒加1并处理计时溢出
更新数码管显示
返回
Timer1中断服务函数
关闭PWM模块
返回
UART中断服务函数
数字钟是否启动
否
是
收到S或s
否
是
设定数字钟为设置状态
收到R或r
否
是
设定数字钟为计时状态
返回

■ Step1：准备

将工程文件夹“LCD_BasicDisplay”放到相应目录下，打开工程。注意检查 Libraries 文件夹下的库“driverlib-cm3.lib”和“grlib-cm3.lib”是否添加完整，其中“grlib-cm3.lib”是用于 LCD 显示的图形库。

■ Step2: 本实验用到的库函数

打开 main.c，首先可以看到第 1 至 16 行为包含的头文件信息。

```
#include <stdio.h>
#include "HardwareLibrary.h"
#include "LuminaryDriverLibrary.h"
#include "grlib/grlib.h"                    // 图形库

#include "GPIODriverConfigure.h"
#include "SysCtlConfigure.h"
#include "SysTickConfigure.h"
#include "TimerConfigure.h"
#include "WatchDogConfigure.h"
#include "UARTConfigure.h"
#include "I2CConfigure.h"
#include "NixieTubeConfigure.h"
#include "PWMConfigure.h"
#include "LCDConfigure.h"                   // LCD 驱动库
#include "LCDDisplay.h"                     // 系统信息显示
```

本次实验的主要是学习 LCD 显示，故需要首先关注 grlib.h，LCDConfigure.h 和 LCDDisplay.h 相关的文件。

1) 对于 grlib.h，在之前介绍 Stellaris 库函数的时候，已经列出了本次实验将要用到的函数，这里不再赘述。

2) 对于 LCDConfigure.h，打开相应的 LCDConfigure.c 文件，可以看到该自定义库主要是封装了以下一些函数：

```
LCDWriteData();
LCDWriteCommand();
LCDInitial();
Kitronix320x240x16_SSD2119PixelDraw();
Kitronix320x240x16_SSD2119PixelDrawMultiple();
Kitronix320x240x16_SSD2119LineDrawH();
Kitronix320x240x16_SSD2119LineDrawV();
Kitronix320x240x16_SSD2119RectFill();
Kitronix320x240x16_SSD2119ColorTranslate();
Kitronix320x240x16_SSD2119Flush();
```

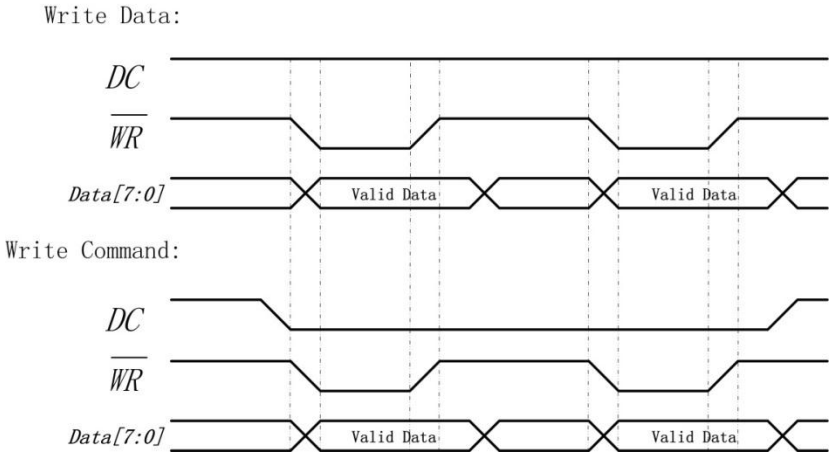
这些函数都分别对应了一个较底层的操作，相当于提供了第一层显示驱动层的功能。

从函数名可以知道这些函数的功能，例如 LCDWriteData()就是将数据写入到 SSD2119，而 LCDWriteCommand()则是将命令写入 SSD2119。在给出的

工程里 LCDWriteData()的内容不完整，需要参考 LCDWriteCommand()将其补全。

```
static void
LCDWriteData(unsigned short usData)
{
    [请添加代码]              //将待写入数据的高8位送到总线（通过移位获取高位）
    [请添加代码]          //拉低写使能信号 WR
    [请添加代码]          //为满足写入操作的时序要求，需要将 WR 重复拉低两次

    [请添加代码]              // 拉高写使能信号 WR

    [请添加代码]          // 将待写入数据的低 8 位送到总线
    [请添加代码]          //拉低写使能信号 WR
    [请添加代码]          //为满足写入操作的时序要求，需要将 WR 重复拉低两次

    [请添加代码]          // 拉高写使能信号 WR
}

static void
LCDWriteCommand(unsigned char ucData)
{
    SET_LCD_DATA(0);            //将待写入命令的高位送到总线
                                //这里通常是 0，因为命令指令都没有超过 8 位

 HWREG(LCD_DC_BASE + GPIO_O_DATA + (LCD_DC_PIN << 2)) = 0; // 拉低 DC

 HWREG(LCD_WR_BASE + GPIO_O_DATA + (LCD_WR_PIN << 2)) = 0;//拉低 WR
 HWREG(LCD_WR_BASE + GPIO_O_DATA + (LCD_WR_PIN << 2)) = 0;//重复

 HWREG(LCD_WR_BASE+GPIO_O_DATA+(LCD_WR_PIN<<2)) = LCD_WR_PIN;
    // 拉高 WR

 SET_LCD_DATA(ucData); //将待写入命令的低 8 位送到总线

 HWREG(LCD_WR_BASE + GPIO_O_DATA + (LCD_WR_PIN << 2)) = 0;//拉低 WR
 HWREG(LCD_WR_BASE + GPIO_O_DATA + (LCD_WR_PIN << 2)) = 0; //重复

 HWREG(LCD_WR_BASE+GPIO_O_DATA+(LCD_WR_PIN<<2)) = LCD_WR_PIN;
    //拉高 WR

 HWREG(LCD_DC_BASE+GPIO_O_DATA+ (LCD_DC_PIN << 2)) = LCD_DC_PIN;
   //拉高 DC
}
```

为方便理解，下图给出了给 SSD2119 写入数据和命令的时序图（忽略了 CS 信号）。



从该图可以看出，在写数据的时候，首先要准备好待写入的数据，然后将$\overline{WR}$信号拉低以写入数据，写完后再将$\overline{WR}$拉高，这样就完成了数据的写入操作。注意，在写数据的时候，没有必要对 DC 进行变动，一直保持其为高就可以了。

3) 对于 LCDDisplay.h，相应的 LCDDisplay.c 文件中包含了用于在 LCD 屏幕上显示系统的自检信息的 LCDSystemCheckInformationDisplay()函数。

在该函数中，首先初始化了 LCD 屏幕，该屏幕已在 LCDConfigure.c 中指定。

```
GrContextInit(&tempContext, &g_sKitronix320x240x16_SSD2119);
```

接着，在屏幕顶部绘制了一个填充的矩形，其填充色为 DarkBlue。然后，在该填充区域绘制了一个 Gold 色边框的矩形。

```
GrContextBackgroundSet(&tempContext,ClrBlack);
GrContextForegroundSet(&tempContext,ClrDarkBlue);
GrRectFill(&tempContext,&tempRect);

GrContextForegroundSet(&tempContext,ClrGold);
GrRectDraw(&tempContext,&tempRect);
```

然后，在矩形框内显示文字，完成该功能主要有三步：设定颜色，设定字体，最后在指定位置绘制文字。

```
GrContextForegroundSet(&tempContext, ClrGold);
GrContextFontSet(&tempContext, &g_sFontCmss18i);
GrStringDrawCentered(&tempContext,"ARM Cortex-M3 Processing System",-1,
                     GrContextDpyWidthGet(&tempContext) / 2,
                     GrStringHeightGet(&tempContext)-3,0);
```

接下来显示文字的操作也与之类似，建议同学们仔细对照代码和显示在LCD 上内容，弄明白每个语句的作用。

上面提到的三个库主要是关于 LCD 的配置和使用，接下来将简要说明一下 LCD 屏的使能和时钟设置。

LCD 的使能是在“GPIODriverConfigure.c”文件中的 GPIOIntial()函数内进行，相关代码在第 12、15 和 45 至 74 行：

```
//GPIO 模块使能
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD); //UART1-RS232 | LCD_DATA
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH); //LCD_WR | LCD_DC
//引脚功能配置
#ifdef     LCD_KITRONIX320X240X16_SSD2119
HWREG(LCD_RST_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY_DD;    //Set
LCD RST
HWREG(LCD_RST_BASE + GPIO_O_CR) = 0x80;
GPIODirModeSet(LCD_RST_BASE, LCD_RST_PIN, GPIO_DIR_MODE_OUT);

GPIOPadConfigSet(LCD_RST_BASE,LCD_RST_PIN,GPIO_STRENGTH_8MA,GPIO
_PIN_TYPE_STD);
HWREG(LCD_RST_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY_DD;
HWREG(LCD_RST_BASE + GPIO_O_CR) = 0x00;

GPIODirModeSet(LCD_DATA_BASE,LCD_DATA_PINS,   GPIO_DIR_MODE_OUT);
//Set LCD Data
GPIOPadConfigSet(LCD_DATA_BASE,LCD_DATA_PINS,
PIO_STRENGTH_8MA,GPIO_PIN_TYPE_STD);

GPIODirModeSet(LCD_DC_BASE,    LCD_DC_PIN, GPIO_DIR_MODE_OUT);
     //Set LCD DC
GPIOPadConfigSet(LCD_DC_BASE,     LCD_DC_PIN,     GPIO_STRENGTH_8MA,
GPIO_PIN_TYPE_STD);

GPIODirModeSet(LCD_RD_BASE,    LCD_RD_PIN, GPIO_DIR_MODE_OUT);
     //Set LCD RD
GPIOPadConfigSet(LCD_RD_BASE,     LCD_RD_PIN,     GPIO_STRENGTH_8MA,
GPIO_PIN_TYPE_STD);

GPIODirModeSet(LCD_WR_BASE,    LCD_WR_PIN, GPIO_DIR_MODE_OUT);
     //Set LCD WR
GPIOPadConfigSet(LCD_WR_BASE,LCD_WR_PIN,GPIO_STRENGTH_8MA,
GPIO_PIN_TYPE_STD);

GPIOPinWrite(LCD_DATA_BASE, LCD_DATA_PINS, 0x00);                    //Set
```

```
    LCD-Pins initial value
    GPIOPinWrite(LCD_DC_BASE, LCD_DC_PIN, 0x00);
    GPIOPinWrite(LCD_RD_BASE, LCD_RD_PIN, LCD_RD_PIN);
    GPIOPinWrite(LCD_WR_BASE, LCD_WR_PIN, LCD_WR_PIN);
    GPIOPinWrite(LCD_RST_BASE, LCD_RST_PIN, 0x00);
    SysCtlDelay(TheSysClock/1000);
    GPIOPinWrite(LCD_RST_BASE, LCD_RST_PIN, LCD_RST_PIN);
    #endif
```

注意，其中的 LCD_RST_BASE、LCD_RST_BIN 等都是 LCD 相关引脚的别名，这些别名的具体定义在“gpiodriverconfigure.h”中。

系统时钟的配置在“SysCtlConfigure.c”中进行的。本次实验的配置代码为：

```
//  系统时钟初始化
void ClockInitial(void)
{
    SysCtlLDOSet(SYSCTL_LDO_2_75V);          //  配置 PLL 前须将 LDO 设为
2.75V
    SysCtlClockSet(SYSCTL_USE_PLL |          //  系统时钟设置，采用 PLL
                   SYSCTL_OSC_MAIN |         //  主振荡器
                   SCTL_XTAL_16MHZ |         //  外接 16MHz 晶振
                     SYSCTL_SYSDIV_4);       //  4 分频，分频结果为 50MHz

      TheSysClock = SysCtlClockGet();        //  获取当前的系统时钟频率
}
```

LCD 成像时需要不停的刷新屏幕，如果扫描刷新的速率过慢，就会影响成像质量。为了达到理想的显示效果，就需要系统提供较高的时钟频率。Stellaris 系列的 ARM 支持多种系统时钟来源，如外接晶振、内部振荡器、内部 PLL（锁相环）等等。前面几个实验都可以采用了外部 16MHz 晶振的配置方式，而在本实验中，为了使用更高的系统时钟，需要采用另外一种使用内部 PLL 的配置方法。

芯片内部的 PLL 单元能够将输入的较低频率的时钟信号锁定到 200MHz 输出。但是，由于处理器内核最高只能工作在 50MHz，所以必须要进行 4 以上的分频。另外，在启用 PLL 之前必须要把 LDO 输出电压设置在最高的 2.75V。这是因为 PLL 单元消耗的功率较大，再加上芯片的其他功耗，如果 LDO 电压不够高就容易造成死机。

根据以上配置，获得的系统时钟频率是 50MHz，满足 LCD 扫描刷新的需求。若有兴趣观察 LCD 屏幕的扫描动作，可以将系统时钟频率降低，例如进行 10 或者 10 以上的分频，就可以看到比较明显的扫描动作。

■ Step3:中断服务程序

本实验中总共使用了四个中断服务程序。

第一个 SysTick 5ms 定时中断服务，主要处理米字管扫描和 5 向键按键动作处理，带有按键防抖功能。每当进入该中断就依次扫描米字管，并判断按键情况，根据按键情况更改系统状态。

第二个中断为 1S 定时 Timer0A 中断。如果没有达到预设的计时目标，当该中断触发时，会将计时秒数加 1，点亮相应的 LED 管，并产生 PWM 波，控制蜂鸣器发声。

第三个为 Timer1A 定时中断，用于禁止 PWM 发生器的定时计数器，使蜂鸣器停止鸣叫。

第四个中断为 UART0 中断，每当 UART 口上接收到数据后触发该中断，如果收到“S”或“s”则暂停计时，并更改 LCDDrawingFlag 标识为 Pause，接收到“R”或“r”则恢复计时，更改 LCDDrawingFlag 标识为 Running。

这几个中断的具体代码可以在 Interrupt Service Routine 文件夹下查看。值得注意的时，所有用到的中断服务程序都必须在启动文件 Startup.s 的中断表中注册。

■ Step4: main 主函数

在准备好其他库函数和配置之后，main 函数演示了 LCD 显示实验。

main 主函数首先要进行系统的初始化：

```
ClockInitial();                    //时钟
GPIOInitial();                     //GPIO
SysTickInitial();                  //系统节拍
TimerInitial();                    //定时器
UART0Initial();                    //UART
GrContextInit(&sContext, &g_sKitronix320x240x16_SSD2119); //LCD Context
LCDInitial();                      //LCD
PWMInitial();                      //PWM

SystemState |= I2C0PullUpTest();
if (!(SystemState & 0x01))
    I2C0MasterInitial();
sprintf(NixieTube,"CTM3 ");   // 米字管显示
SysTickEnable();                   // 系统时钟节拍计数使能
IntMasterEnable();                 // 中断使能
LCDSystemCheckInformationDisplay(SystemState);   //LCD 显示自检信息
UARTSystemCheckInformationTransmit(UART0_BASE,SystemState);//发到 UART 口
LEDOn(LED_1);
```

然后,为控制代码：

```
while (KeyNumber==0);      KeyNumber=0;          // 等待，直到有按键按下
LEDOff(LED_1);
UARTStringPut(UART0_BASE,"System activated!\r\n");
```

```
    [请添加代码]                                    //接下来要使用的 LCD 部分清屏
    [请添加代码]                                    //即可将 sRect 区域用某颜色填充
    // 系统激活，屏幕显示“Running”
    [请添加代码]                                    //设定显示文字的前景颜色
    [请添加代码]                                    //设定字体
    [请添加代码]                                    //在屏幕中部显示“Running”

    [请添加代码]                                    //在米字管上显示计时
    LEDOn(LED_1);

    TimerEnable(TIMER0_BASE,TIMER_A);          // 1s 定时中断使能
    BuzzerBeep(ALARM_SOUND_PERIOD,TheSysClock/5); //蜂鸣器响
    SystemActivatedFlag=true;
```

该段代码实现以下功能：等待按键，当有按键时，激活系统。随即，发送“System activated !”到 UART 口，在 LCD 屏幕上显示“Running”字样，米字管示数修改为计时数，打开定时器，然后更改系统激活标识 SystemActivatedFlag 为 true。

其中，LCD 和米字管显示部分需要同学们根据注释添加相应代码，LCD 文字可用的字体和颜色可以参考图形库手册《Stellaris® Graphics Library USER’S GUIDE》的第 14 章“Predefined Color Reference”和第 15 章“Font Reference”。

然后，程序进入主循环，LCD 显示运行状态并等待中断。

```
while(1)
    {
    if (LCDDrawingFlag_Running)
        {
            [请添加代码]                    //在屏幕中部显示“Running”
                                          // 注意，如果字体或颜色继续使用之前的设定
                                            //这里就可以省去对应的设置语句
            LCDDrawingFlag_Running=0;
        }
        if (LCDDrawingFlag_Pause)
        {
            [请添加代码]                    //在屏幕中部显示“Pause”
            LCDDrawingFlag_Pause=0;
        }
    }
```

■ Step 5: 编译和下载

在补充完整代码，并设置好编译选项之后，就可以对程序进行编译，值得注意的是，由于本次实验使用了大量的中断程序，涉及到的寄存器数量也比

较多，所以在中断保护时有可能会遇到了堆栈溢出的问题，为避免此问题，可以在“Startup.c”中适当增大堆栈的大小。

```
; <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
Stack    EQU        0x00000400
```

编译完成后可以下载编译好的文件到实验板上进行验证调试，下载前注意检查板上的跳线连接情况。

9.7 思考与练习

9.1 对函数 GrStringDrawCentered(pContext, pcString, lLength, lX, lY , bOpaque)：

a) 修改位置参数 lX, lY 的值，在屏幕的不同区域显示系统状态；

b) 修改第三个参数 lLength 的值，观察对显示的影响；

c) 修改第五个参数 bOpaque 的值，观察对显示的影响。

9.2 修改 main.c 函数，使得当系统状态为“Running”的时候，原来 LCD 上显示“ARM Cortex-M3 Processing System”的地方显示为你的学号；当系统状态为“Pause”时，该地方显示你的姓名的拼音。

9.3 使用函数

void GrLineDraw(const tContext *pContext, long lX1, long lY1, long lX2, long lY2)；

void GrLineDrawV(const tContext *pContext, long lX, long lY1, long lY2)；

void GrCircleDraw(const tContext *pContext, long lX, long lY, long lRadius)；

绘制简单的组合图像。

第十章：ADC 模块

10.1 ADC 总体特性

ADC是一种能够将连续的模拟电压信号转换为离散的数字量的外设。Stellaris系列ARM有两个10位转换分辨率的ADC模块，支持16个输入通道，内置一个内部温度传感器。ADC模块含有四个可编程的序列发生器，可对多个模拟输入源进行采样，无需控制器干涉。每个采样序列均可灵活配置输入源、触发事件、中断的产生和序列优先级。

Stellaris系列ARM的ADC模块提供系列特性：

- 16个模拟输入通道
- 单端和差分输入配置
- 内部温度传感器
- 高达1Msps（每秒采样一百万次）的采样率
- 4个可编程的采样转换序列，入口长度1到8，每个序列均带有相应的转换结果FIFO
- 灵活的触发控制：处理器（软件）、定时器、模拟比较器、PWM、GPIO
- 硬件可对多达64个采样值进行平均计算（牺牲速度换取精度）
- 转换器采用内部的3V参考电压 ，也可使用片外参考电平
- 分开的模拟电源和模拟地，跟数字电源和数字地相互独立

10.2 AD 转换方式简介

常见的AD转换方式包括积分型、逐次比较型、并行比较型/穿行比较型等。积分型AD工作原理是将输入电压转换成时间(脉冲宽度信号)或频率(脉冲频率),然后由定时器/计数器获得数字值.其优点是用简单电路就能获得高分辨率,但缺点是由于转换精度依赖于积分时间,因此转换速率极低.初期的单片AD转换器大多采用积分型,现在逐次比较型已逐步成为主流。

逐次比较型AD由一个比较器和DA转换器通过逐次比较逻辑构成,从MSB开始,顺序地对每一位将输入电压与内置DA转换器输出进行比较,经n次比较而输出数字值.其电路规模属于中等.其优点是速度较高、功耗低,在低分辨率(<12位)时价格便宜,但高精度(>12位)时价格很高。

并行比较型AD采用多个比较器，仅作一次比较而实行转换，又称Flash(快速)型。由于转换速率极高,n位的转换需要2n-1个比较器，因此电路规模也极大，价格也高，只适用于视频AD转换器等速度特别高的领域。串并行比较型AD结构上介于并行型和逐次比较型之间，用两次比较实行转换，所以称为Half flash(半快速)型。

10.3 ADC 结构图

10.3.1 双 ADC 模块的连接框图

Stellaris® LM3S9B96 微控制器内置两个相同的模-数转换器（ADC）模块ADC0 和ADC1，共享16 个模拟输入通道。两个ADC 模块相互独立工作，可同时执行不同的采样序列、随时对任一模拟输入通道进行采样、产生各自不同的中断和触发事件

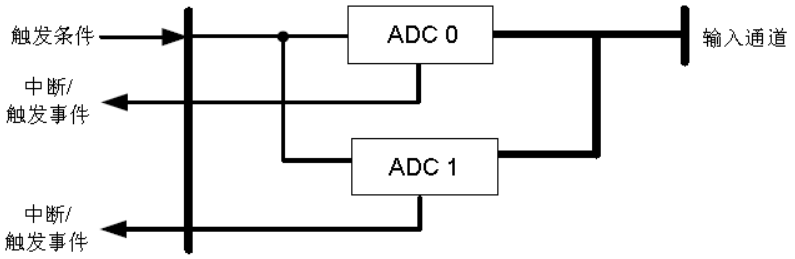


图10-1　　双ADC 模块的连接框图

10.3.2 ADC 模块框图

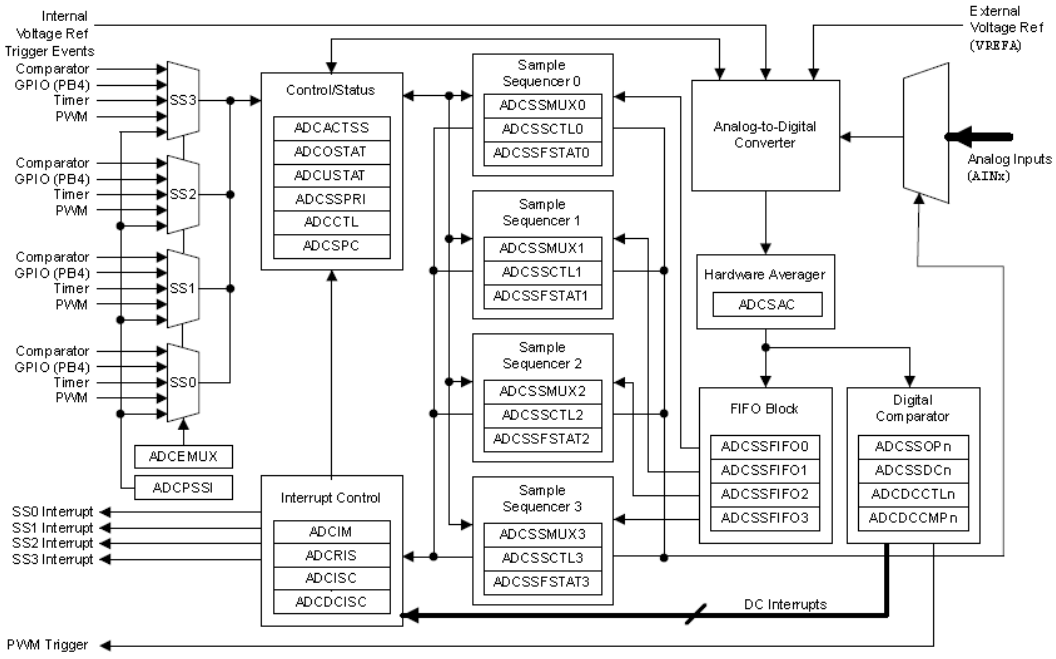


图 10-2　ADC 模块框图

10.4 ADC 功能描述

传统的ADC采用单次采样或双采样获取采样数据，Stellaris系列ARM的ADC通过可编程的采样序列获取采样数据。每个采样序列由一系列程序设定

的连续采样动作组成，实现自动从多个输入源中收集数据，无需控制器对其进行重新配置或干预。每个采样动作都可以编程，如输入源和输入模式（差分输入还是单端输入），采样结束时是否产生中断，是否是队列中最后一个采样动作的标识符等参数进行配置

10.4.1 采样序列发生器

采样序列发生器（Sample Sequencer，简写为SS）处理采样控制和数据采集。4个序列发生器不同的地方在于可以捕获的采样数目和FIFO深度不同。表1给出了每个序列发生器可捕获的最大采样数及相应的FIFO深度。FIFO的每个单元均为32位宽的字，低10位包含的是采样结果。

表10-1　ADC序列发生器的菜样数和FIFO深度

序列发生器	采样数	FIFO深度
SS0	8	8
SS1	4	4
SS2	4	4
SS3	1	1

在一个指定的采样序列中，可以对每个采样动作设置对应的输入管脚，是否使用温度传感器、是否中断使能、是否序列终止符和是否采用差分输入模式。

当配置一个采样序列时，可以在采样序列的任意采样动作后产生中断。同样地，也可以在采样序列中的任何位置结束采样。例如，如果使用序列发生器0，可以在第5个采样后结束并产生中断，也可以在第3个采样后产生中断。

一个采样序列执行完后，可以利用函数ADCSequenceDataGet( )从ADC采样序列FIFO里读取结果。上溢和下溢可以通过函数ADCSequenceOverflow( )和ADCSequenceUnderflow( )进行控制。

10.4.2 模块控制

控制逻辑单元负责除了采样序列发生器外的中断产生、序列优先级和触发进行控制。 大多数的ADC控制逻辑都是14～18MHz的ADC时钟速率下运行。当选择了系统XTAL时，内部的ADC分频器通过硬件自动配置，尽量工作在16MHz操作频率。

10.4.3 中断

采样序列发生器虽然会对引起中断的事件进行检测，但它们不控制中断是否真正被发送到中断控制器。ADC模块的中断信号由相应的状态位来控制。ADC中断状态分为原始的中断状态和屏蔽的中断状态，这可以通过函数ADCIntStatus ( )来查知。函数ADCIntClear( )可以清除中断状态。

10.4.4 优先级设置

当同时出现采样事件（触发）时，可设置优先级，安排它们的处理顺序。优先级值的有效范围是0到3，0优先级最高，3优先级最低。软件必须确保所有激活采样序列发生器单元的优先级是唯一的，否则将导致转换结果数据不连续

10.4.5 采样事件

采样序列发生器的触发事件源可包括如处理器（软件）、定时器、模拟比较器、PWM、GPIO触发。软件可通过函数ADCProcessorTrigger( )来启动采样。 在配置持续采样触发条件时务必慎重，如果一个序列的优先级太高，那么可能会忽略其它低优先级序列。

10.4.6 硬件采样平均电路

硬件平均电路可产生更高精度的结果，其最高可以将64个采样值进行平均，将平均结果作为单次采样结果写入序列发生器FIFO的一个单元。缺点是吞吐量变小，如果对16个采样值进行平均，吞吐量为1/16。硬件平均电路默认是关闭的，关闭时转换器的所有数据直接传送到序列发生器FIFO中。进行平均计算的硬件由相关的硬件寄存器进行控制。每个ADC中只有一个平均电路，所有输入通道（不管是单端输入还是差分输入）都是执行相同的求取平均值操作。

10.4.7 模数转换器

模数转换器采用逐次逼近产生10位A/D转换。通过特殊的平衡输入通道，输入的失真可以降到最低。转换器必须工作在16MHz左右，如果时钟偏差太多，则会给转换结果带来很大误差。

10.4.8 差分采样

除了传统的单端采样外，ADC模块还支持对两个模拟输入通道进行采样的差分采样。

差分信号对0的含义是采样模拟输入端0和1，差分信号对1采样模拟输入2和3，依此类推。差分信号对对应的模拟输入端是固定的，比如模拟输入端0跟3不能组成差分信号对。

表 10-2 差分采样信号对

差分信号对	模拟输入端
0	0 和 1
1	2 和 3
2	4 和 5
3	6 和 7
4	8 和 9

5	10 和 11
6	12 和 13
7	14 和 15

在差分模式下采样电压是奇数和偶数通道的差值，即：

ΔV ＝ VIN_ENEN － VIN_ODD

其中ΔV是差分电压，VIN_EVEN是偶数通道，VIN_ODD是奇数通道。因此：

如果ΔV = 0，则转换结果 = 0x1FF

如果ΔV > 0，则转换结果 > 0x1FF（范围在0x1FF～0x3FF）

如果ΔV < 0，则转换结果 < 0x1FF（范围在0～0x1FF）

差分对指定了模拟输入的极性：偶数编号的输入总是正，奇数编号的输入总是负。为得到恰当的有效转换结果，负输入必须在正输入的±1.5V范围内。此外如果模拟输入高于3V或低于0V，输入电压被钳位在3V或0V。 举个列子，若负端电压为0.75V，则当差分电压低于-0.75V 时正端输入电压将会低于0V 而饱和。若负端电压为2.25V，则当差分电压高于0.75V 时正端输入电压将会高于3.0V 而饱和。

10.4.9 内部温度传感器

温度传感器的主要作用是当芯片温度过高或过低时向系统提示，保障芯片稳定工作。内部温度传感器提供模拟温度读数以及参考电压。输出终端SENSO的电压通过以下等式计算得到： SENSO ＝ 2.7 －（T＋55） / 75

也可以从温度传感器的ADC 结果通过函数转换得到温度读数。下式即可根据ADC 读数计算出温度（单位为℃）：温度 = 147.5 – ((225 × ADC) / 1023)

10.4.10 数字比较器

数字比较器用于对外部信号采样并监控数值的变动，数字比较器内置16路，ADC转换结果可直接发送给数字比较器，与用户编程的门限进行比较，如果超出了门限，则产生一个处理器中断及/或向PWM 模块发送一个触发事件。

10.4.11 内部电压和外部电压

ADC 基准电压可选用内部电压也可以采用外部参考电压。ADC 模块可将内部带隙电路产生的3.0V 电压作为参考电压。当采用外部参考电压时务必慎重，参考电压源的质量必须符合要求，地电平将始终作为最小转换值的参考电平。

表10-3 参考电压

参数符号	参数名称	最小值	标称值	最大值	单位
V_{REFA}	外部参考电压	2.4	-	3.06	V
V_{REFI}	内部参考电压	-	3.0	-	V

10.5 ADC 控制库函数

表10-4 函数ADCSequenceEnable( )

原型	void ADCSequenceEnable(unsigned long ulBase, unsigned long ulSequenceNum)
参数	ulBase：ADC模块的基址，取值ADC_BASE ulSequenceNum：ADC采样序列的编号，取值0、1、2、3
返回	无
功能	使能一个ADC采样序列

表10-5 函数ADCSequenceConfigure

原型	void ADCSequenceConfigure(unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulTrigger, unsigned long ulPriority)
参数	ulBase：ADC模块的基址，取值ADC_BASE ulSequenceNum：ADC采样序列的编号，取值0、1、2、3 ulTrigger：启动采样序列的触发源，取下列值之一： ADC_TRIGGER_PROCESSOR // 处理器事件 ADC_TRIGGER_COMP0 // 模拟比较器0事件 ADC_TRIGGER_COMP1 // 模拟比较器1事件 ADC_TRIGGER_COMP2 // 模拟比较器2事件 ADC_TRIGGER_EXTERNAL // 外部事件（PB4中断） ADC_TRIGGER_TIMER // 定时器事件 ADC_TRIGGER_PWM0 // PWM0事件 ADC_TRIGGER_PWM1 // PWM1事件 ADC_TRIGGER_PWM2 // PWM2事件 ADC_TRIGGER_ALWAYS // 触发一直有效（用于连续采样） ulPriority：相对于其他采样序列的优先级，取值0、1、2、3（优先级依次从高到低）
返回	无
功能	配置ADC采样序列的触发事件和优先级

表10-6 函数ADCSequenceStepConfigure( )

原型	void ADCSequenceStepConfigure(unsigned long ulBase, unsigned long ulSequenceNum, unsigned long ulStep, unsigned long ulConfig)
参数	ulBase：ADC模块的基址，取值ADC_BASE ulSequenceNum：ADC采样序列的编号，取值0、1、2、3 ulStep：步值，决定触发产生时ADC捕获序列的次序，对于不同的采样序列取值也不相同： 采样序列编号 | 步值范围

	0 0--7 1 0--3 2 0--3 3 0 ulConfig：步进的配置，取下列值之间的“或运算”组合形式： ● ADC控制 ADC_CTL_TS // 温度传感器选择 ADC_CTL_IE // 中断使能 ADC_CTL_END // 队列结束选择 ADC_CTL_D // 差分选择 ● ADC通道 ADC_CTL_CH0 // 输入通道0（对应ADC0输入） ADC_CTL_CH1 // 输入通道1（对应ADC1输入） ADC_CTL_CH2 // 输入通道2（对应ADC2输入） ADC_CTL_CH3 // 输入通道3（对应ADC3输入） ADC_CTL_CH4 // 输入通道4（对应ADC4输入） ADC_CTL_CH5 // 输入通道5（对应ADC5输入） ADC_CTL_CH6 // 输入通道6（对应ADC6输入） ADC_CTL_CH7 // 输入通道7（对应ADC7输入） 以此类推到CH15 注意：ADC通道每次（即每步）最多只能选择1个，如果要选取多通道则要多次调用本函数分别进行配置；如果已经选择了内置的温度传感器（ADC_CTL_TS）则不能再选择ADC通道；如果已选择了差分采样模式（ADC_CTL_D），则ADC通道只能选取下列值之一： ADC_CTL_CH0 // 差分输入通道0（对应ADC0和ADC1输入的组合） ADC_CTL_CH1 // 差分输入通道1（对应ADC2和ADC3输入的组合） ADC_CTL_CH2 // 差分输入通道2（对应ADC4和ADC5输入的组合） ADC_CTL_CH3 // 差分输入通道3（对应ADC6和ADC7输入的组合） 以此类推到CH7
返回	无
功能	配置ADC采样序列发生器的步进

表10-7 函数ADCHardwareOversampleConfigure( )

原型	void ADCHardwareOversampleConfigure(unsigned long ulBase, unsigned long ulFactor)
参数	ulBase：ADC模块的基址，取值ADC_BASE ulFactor：采样平均数，取值2、4、8、16、32、64，如果取值0则禁止硬件过采样
返回	无
功能	配置ADC硬件过采样的因数

表10-8 函数ADCSequenceDataGet( )

原型	void long ADCSequenceDataGet(unsigned long ulBase,

	unsigned long ulSequenceNum, unsigned long *pulBuffer)
参数	ulBase：ADC模块的基址，取值ADC_BASE ulSequenceNum：ADC采样序列的编号，取值0、1、2、3 pulBuffer：无符号长整型指针，指向保存数据的缓冲区
返回	复制到缓冲区的采样数
功能	从ADC采样序列里获取捕获到的数据

表10-9　函数ADCProcessorTrigger( )

原型	void ADCProcessorTrigger(unsigned long ulBase, unsigned long ulSequenceNum)
参数	ulBase：ADC模块的基址，取值ADC_BASE ulSequenceNum：ADC采样序列的编号，取值0、1、2、3
返回	无
功能	引起一次处理器触发ADC采样

对于ADC其他功能的详细介绍，参见《LM3S9B96库函数介绍》文件夹中的《LM3SLib_ADC.pdf》。

10.6　本实验使用的资源说明

本实验使用指轮和外部电压，故增加的资源如下表所示

Thumbwheel	PB4
3.0V Reference	PB6

10.7　本实验板原理图

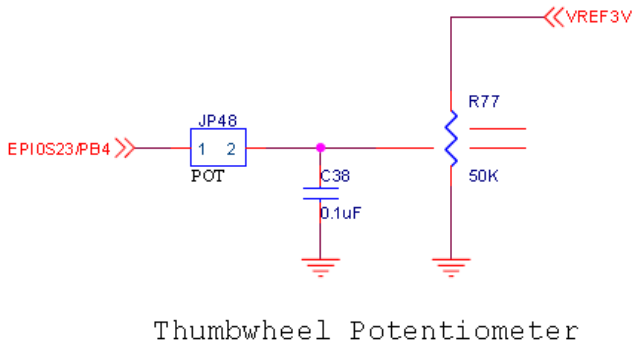


图 10-3　指轮实验板原理图

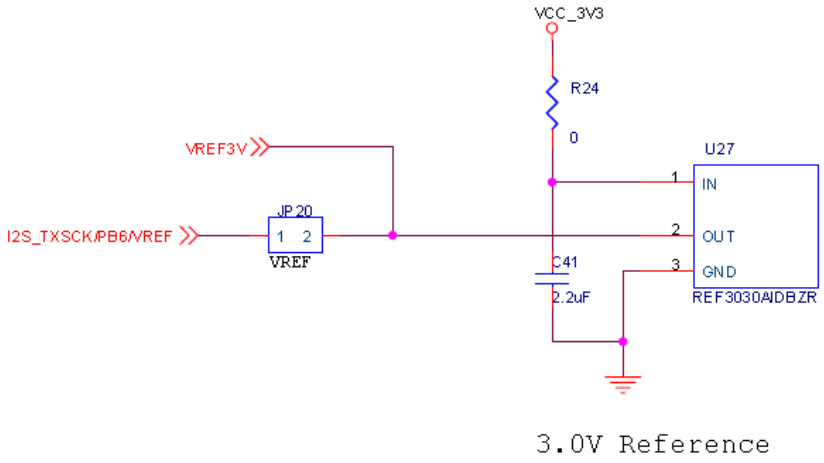


图 10-4　外部电压原理图

10.8　跳帽连接

连接如表10-10所示：

表10-10　跳帽连接

连接跳帽	对应端口	断开跳帽	对应按键
JP3	PF2	JP35	LED1
JP41	PE5	JP24	SW2-PRESS
JP42	PE4	JP25	SW3-UP
JP44	PF1	JP30	SW4-DOWN
JP39	PF3	JP2	BUZZER
JP38	PB7		LCD_RST\
JP11		JP12	LCD_CS\(下拉)
JP6	PD0	JP29,JP36	LCD_D0
JP7	PD1	JP27,JP37	LCD_D1
JP8	PD4	JP26	LCD_D4
JP9	PD5	JP31	LCD_D5
		JP23	LCD_Backlight
JP20	PB6	JP28,JP43	THUMBWHEEL_VREF
JP48	PB4	JP45	THUMBWHEEL_POT
打开 SSCOM			

10.9　实验十(1)　ADC_Thumbwheel

■ 实验概述

本实验通过控制可设置目标时间的电子钟，带状态显示屏，调节时使用

指轮。

实验效果：（1）LCD显示初始信息（2）有键按下则LCD显示信息，电子钟开始计数，蜂鸣器响及LED轮流亮（3）调节指轮时LCD显示相应电压值和相应范围（3）当按下PRESS时处于暂停状态；按下UP键时根据范围值确定电子钟加1秒，加10秒，加1分钟，加10分钟的显示；按下DOWN键时根据范围值确定电子钟减1秒，减10秒，减1分钟，减10分钟的显示。

本次实验学习目的:了解I2C总线的特点和功能；学会使用I2C自检程序；会使用I2C总线对PCA9557芯片进行操作；学会常用键盘防抖方法。

■ 实验流程图

实验流程图及在上个实验室基础上增加的ADC中断服务函数如图5所示

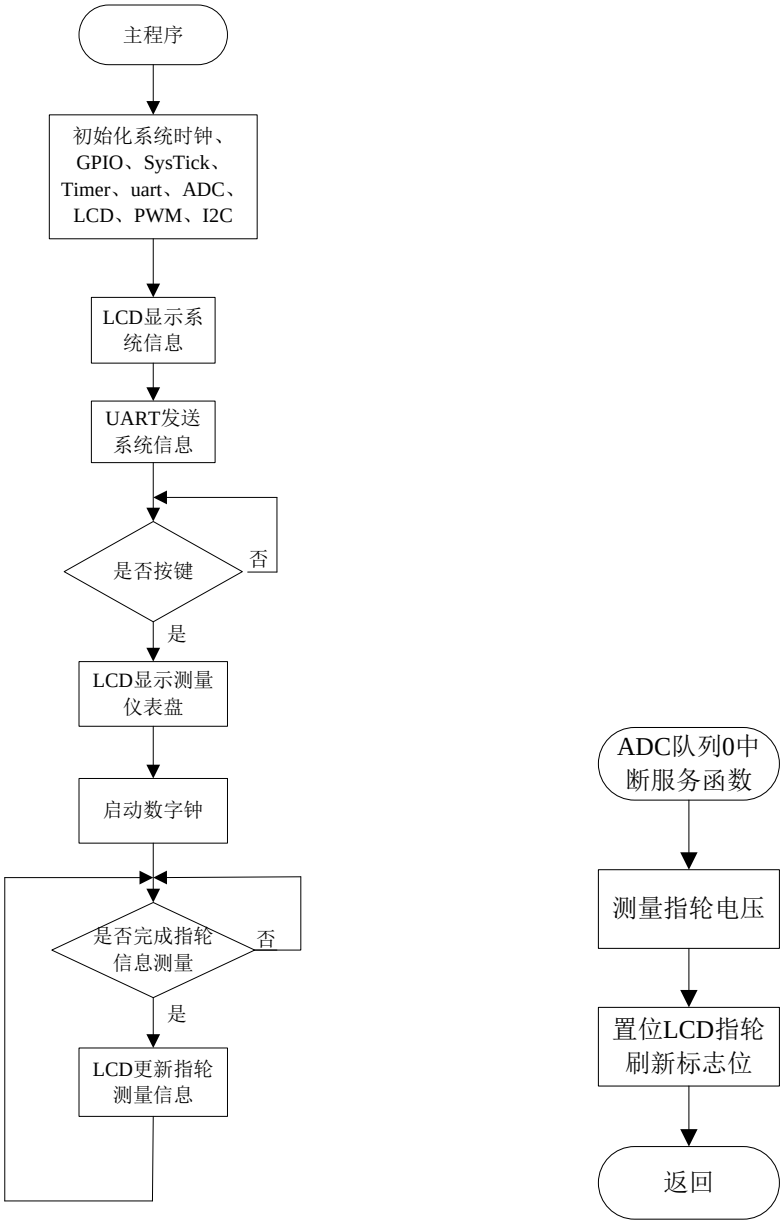


图 10-5 流程图

■ 实验步骤

本实验要求完成的代码有三个部分：

■ Step1：ADCConfigure.c 函数封装

建立一个空白文档以键入ADC驱动程序，文档保存至StellarisWareforC1/codes/LM3S9B96_Experiment/ADC-Thumbwheel/,命名为ADCConfigure.c，然后将该文件加入工程的Source Group中。

```
#include "HardwareLibrary.h"
#include "LuminaryDriverLibrary.h"

#include "ADCConfigure.h"
void ADCInitial(void)
{   [请添加代码]  ;                    //用 SysCtlPeripheralEnable 函数使能 ADC0 模块
    [请添加代码]  ;                    //设置 ADC 采样速率为 500KSPS
    [请添加代码]  ;                    //使能采样序列 0
    HWREG(ADC0_BASE+0x38)=0x01;    //使用外置基准电压
    [请添加代码]  ;                     /* 用 ADCSequenceConfigure 函数配置采样序列：
                                ADC 基址，采样序列编号，处理器触发事件，采样优先
                                级 0 */
    [请添加代码]  ;                      /* 用 ADCSequenceStepConfigure 函数进行采样
                                步进设置：ADC 基址，采样序列编号,步值 0，通道设置
                                为通道 10、队列结束选择、中断使能 */
    ADCHardwareOversampleConfigure(ADC0_BASE,64);
    ADCIntEnable(ADC0_BASE, 0);          //使能 ADC 中断
    IntEnable(INT_ADC0);                 //使能 ADC 采样序列中断
```

■ Step2：在 SysTickISR.c 最后添加代码

```
        ADCTriggerCounter++;
        if (ADCTriggerCounter==50)
        {
            ADCTriggerCounter=0;
            [请添加代码];                 //引起一次处理器触发 ADC 采样
```

■ Step3：编写 ADCISR.c 中断服务函数

```
#include "HardwareLibrary.h"
#include "LuminaryDriverLibrary.h"

#include "ADCISR.h"                                    //包含头文件
```

```
unsigned long MeasureResult_Thumbwheel=0;
void ADC_Sequence_0_ISR(void)
{
    unsigned long ulStatus;
    ulStatus=ADCIntStatus(ADC0_BASE,0,true);
    ADCIntClear(ADC0_BASE,0);
    if (ulStatus)
    {
        [请添加代码];                    //把采样序列获得的数据读入
MeasureResult_Thumbwheel
        LCDDrawingFlag_Thumbwheel= true;
    }
}
```

在本实验中，有必要说明的几点如下，请参见给出的程序加以理解。

■ Step1：使能 Thumbwheel

在 GPIODriverConfigure.c 中的如下代码的作用是使能 Thumbwheel

```
GPIOPinTypeADC(THUMBWHEEL_BASE,THUMBWHEEL_PIN);      //Set
ADC_Thumbwheel
GPIOPinTypeADC(THUMBWHEEL_VREF_BASE,THUMBWHEEL_VREF_PIN);
```

■ Step2：在 GPIODriverConfigure.h 里添加定义

```
#define THUMBWHEEL_BASE          GPIO_PORTB_BASE
#define THUMBWHEEL_PIN           GPIO_PIN_4
#define THUMBWHEEL_VREF_BASE     GPIO_PORTB_BASE
#define THUMBWHEEL_VREF_PIN      GPIO_PIN_6
```

■ Step3：由于按键冲突，left 和 right 不能使用，要注释掉

在 GPIODriverConfigure.c 中两处注释掉 KEYLEFT,KEYRIGHT

```
//GPIOPinTypeGPIOInput(KEYLEFT_BASE,   KEYLEFT_PIN);
//GPIOPinTypeGPIOInput(KEYRIGHT_BASE, KEYRIGHT_PIN);
```

```
unsigned char GetKeyNumber(void)
{
    if          (!HWREG(KEY_PRESS_BASE+GPIO_O_DATA+(KEY_PRESS_PIN<<2)))
    return KEY_PRESS;
    //if (!HWREG(KEY_LEFT_BASE+GPIO_O_DATA+(KEY_LEFT_PIN<<2))) return
KEY_LEFT;
    //if          (!HWREG(KEY_RIGHT_BASE+GPIO_O_DATA+(KEY_RIGHT_PIN<<2)))
    return KEY_RIGHT;
```

■ Step4：注册中断服务函数

首先，打开Startup.s，可以看到添加了ADC_Sequence_0_ISR的EXTERN，代码如下：

```
;*******************************************************************************
;
; Place code into the reset code section.
;
;*******************************************************************************
        AREA    RESET, CODE, READONLY
        THUMB

        EXTERN  SysTick_ISR
        EXTERN  I2C0_ISR
        EXTERN  Timer0A_ISR
        EXTERN  Timer1A_ISR
        EXTERN  UART0_ISR
        EXTERN  ADC_Sequence_0_ISR
```

然后，看到将使用到的中断对应的中断向量名称改为定义中断服务函数时的名称：

```
;*******************************************************************************
;
; The vector table.
;
;*******************************************************************************
        EXPORT  __Vectors
__Vectors
        …
        …
        DCD     ADC_Sequence_0_ISR              ; ADC Sequence 0
```

10.10 实验十(2) ADC_Temperature

■ 实验概述

在实验(一)的基础上增加芯片温度测量并在LCD屏显示芯片温度信息。实验效果为:在实验（一）效果上增加滑动指轮时LCD显示芯片温度信息

■ 实验流程图

在实验（一）的基础上，主流程图增加的测温部分，及芯片温度中断服务函数流程图如下图所示。

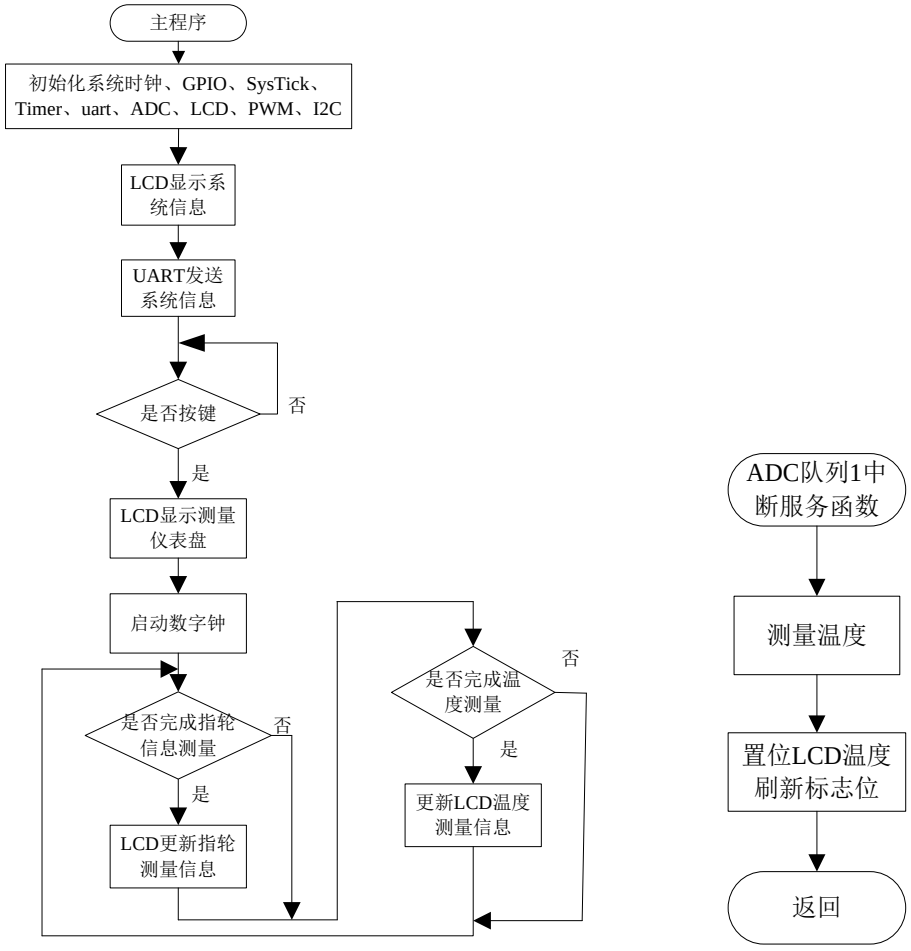


图 10-6 流程图

■ 实验步骤

本实验需要编写的部分如下：

■ Step1：在 ADCConfigure.c 中添加采样序列 1 的配置代码

```
[请添加代码];              //使能采样序列
```

```
[请添加代码];          /*配置采样序列：ADC 基址，采样序列编号，处理器触发事件，采样优先级 0 */
[请添加代码];          /* 用 ADCSequenceStepConfigure 函数进行采样步进设置：ADC 基址，采样序列编号,步值 0，通道设置为通道 10、队列结束选择、中断使能 */
    [请添加代码];          //使能 ADC 中断
    [请添加代码];          //使能 ADC 采样序列中断
```

■ Step2：SysTick 中断服务函数

```
if (ADCTriggerCounter==200)
        {
            ADCTriggerCounter=0;
            [请添加代码];                    //引起一次处理器触发 ADC 序列 1 采样
        }
        else
        {
            if (ADCTriggerCounter%25==0)
            [请添加代码];                    //引起一次处理器触发 ADC 序列 0 采样
        }
```

■ Step3：增加 ADC 采样序列 1 的温度测量中断服务函数

```
void ADC_Sequence_1_ISR(void)
{
    unsigned long ulStatus;
    ulStatus=ADCIntStatus(ADC0_BASE,1,true);
    ADCIntClear(ADC0_BASE,1);
    if (ulStatus)
    {
        [请添加代码];  //把采样序列获得的数据读入 MeasureResult_Temperature
        LCDDrawingFlag_Temperature=1;
    }
}
```

在上一个实验基础上，本实验的几点说明：

■ Step1：主程序 84 行，在系统激活后 LCD 屏显示部分，增加代码实现温度显示

```
    GrStringDraw(&sContext,"ChipTemperature:",-1,15,LCD_TEMPERATURE_LINE,0);
    GrStringDraw(&sContext, "\'C",-1,280,LCD_TEMPERATURE_LINE,0);
```

■ Step2：主程序 158 行，在判断是否完成指轮信息测量后增加代

码，用以判断是否完成芯片温度信息测量，更新 LCD 屏信息

```
if (LCDDrawingFlag_Temperature)
    {
        sprintf(tMessage,"%2.2f",((float)(151040-225*TemperatureResult)/1024));
        GrContextForegroundSet(&sContext, ClrRed);
        GrContextFontSet(&sContext, &g_sFontCm18b);
        GrStringDraw(&sContext, tMessage,-1,220,LCD_TEMPERATURE_LINE,1);
        LCDDrawingFlag_Temperature=false;
        }
```

■ Step3：注册中断服务函数

首先，打开Startup.s，将ADC_Sequence_1_ISR中断名称EXTERN，代码如下：

```
;*******************************************************************************
***
;
; Place code into the reset code section.
;
;*******************************************************************************
***
        AREA    RESET, CODE, READONLY
        THUMB
        …
         …
         EXTERN  ADC_Sequence_1_ISR
```

然后，使用到的中断对应的中断向量名称改为定义中断服务函数时的名称：

```
            DCD     ADC_Sequence_1_ISR          ; ADC Sequence 1
```

第十一章　加速度传感器

11.1　加速度传感器简介

加速度传感器是一种能够测量加速力的设备。加速力就是当物体在加速过程中作用在物体上的力，就好比地球引力。加速度计有两种：一种就是线加速度计,另一种是角加速度计。

加速度是表征物体在空间运动本质的一个基本物理量。因此，可以通过测量加速度来测量物体的运动状态。例如，惯性导航系统就是通过飞行器的加速度来测量它的加速度、速度(地速)、位置、已飞过的距离以及相对于预定到达点的方向等。通常还通过测量加速度来判断运动机械系统所承受的加速度负荷的大小，以便正确设计其机械强度和按照设计指标正确控制其运动加速度，以免机件损坏。对于加速度，常用绝对法测量，即把惯性型测量装置安装在运动体上进行测量。

加速度传感器的基本结构通常是质量－弹簧－阻尼二阶惯性系统。由质量块 m、弹簧 k 和阻尼器 C 所组成的惯性型二阶测量系统。质量块通过弹簧和阻尼器与传感器基座相连接。传感器基座与被测运动体相固连，因而随运动体一起相对于运动体之外惯性空间的某一参考点作相对运动。

由于质量块不与传感器基座相固连,因而在惯性作用下将与基座之间产生相对位移。质量块感受加速度并产生与加速度成比例的惯性力，从而使弹簧产生与质量块相对位移相等的伸缩变形，弹簧变形又产生与变形量成此例的反作用力。当惯性力与弹簧反作用力相平衡时，质量块相对于基座的位移与加速度成正比例，故可通过该位移或惯性力来测量加速度。

加速度器有很多类型：位移式加速度传感器，应变式加速度传感器，由陀螺仪（角速度传感器）的改进的角加速度计等。按照原理可以分为变磁阻式、变电容式、霍尔式等等。

11.2　加速度传感器资源及特性

我们的实验板中采用三轴数字加速度计 ADXL345，ADXL345 是 ADI 公司于 2008 年推出的采用 MEMS 技术具有 SPI 和 I2C 数字输出功能的三轴加速度计，具有小巧轻薄、超低功耗、可变量程、高分辨率等特点：

它只有 3 mm×5 mm×1 mm 的外形尺寸，面大小相当于小拇指指甲盖的 1/3；在典型电压 VS=2.5 V 时功耗电流约为 25～130 μA，比先期采用模拟输出的产品 ADXL330 功耗典型值低了约 70～175 μA；最大量程可达 ±16 g，另可选择 ±2、±4、±8 g 量程，可采用固定的 4 mg/LSB 分辨率模式，该分辨率可测得 0.25 °的倾角变化。

ADXL345 提供一些特殊的运动侦测功能，可侦测出物体是否处于运动状态，并能敏感出某一轴向加速度是否超过了用户自定义门限，可侦测物体是否正在跌落。此外，还集成了一个 32 级 FIFO 缓存器，用来缓存数据以减轻处理器的负担。ADXL345 可在倾斜敏感应用中测量静态重力加速度，也可在

运动甚至振动环境中测量动态加速度，非常适合于移动设备应用，可望在手机、游戏和定位设备、微小型导航设备、硬盘保护、运动健身器材、数码照相机等产品中得到广泛应用。

ADXL345 丰富的功能是通过使用寄存器来实现的。这些丰富的寄存器，用以选择数据格式、FIFO 工作模式、数字通信模式、节电模式、中断使能以及修正各轴偏差等等。

■ ADXL345 的内部功能框图

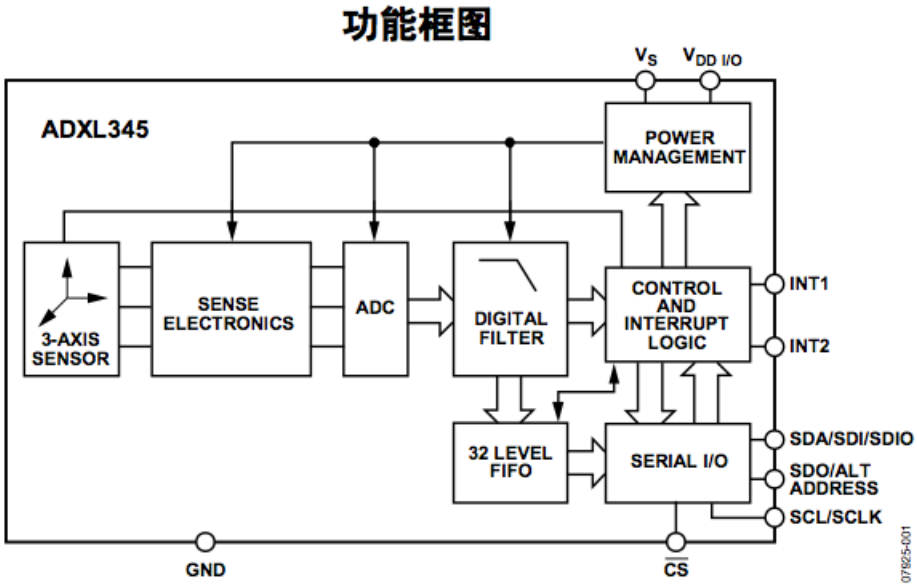


图 11-1　ADXL345 的内部功能框图

■ ADXL345 的工作原理

ADXL345是一款完整的3轴加速度测量系统,可选择的测量范围有 ±2 g, ±4 g, ±8g或 ±16g。既能测量运动或冲击导致的动态加速度,也能测量静止加速度,例如重力加速度,使得器件可作为倾斜传感器使用。该传感器为多晶硅表面微加工结构,置于晶圆顶部。由于应用加速度,多晶硅弹簧悬挂于晶圆表面的结构之上,提供力量阻力。差分电容由独立固定板和活动质量连接板组成,能对结构偏 转进行测量。加速度使惯性质量偏转、差分电容失衡,从而传感器输出的幅度与加速度成正比。相敏解调用于确定加速度的幅度和极性。

■ ADXL345 的寄存器

表 11-1　寄存器映射表

寄存器地址	名称	类型	复位值	描述
0x00	DEVID	*R*	11100101	器件 ID
0x01-0x1C	保留			保留，不操作
0x1D	THRESH_TAP	$R/\overline{W}$	00000000	敲击阈值

0x1E	OFSX	$R/\overline{W}$	00000000	X 轴偏移
0x1F	OFSY	$R/\overline{W}$	00000000	Y 轴偏移
0x20	OFSZ	$R/\overline{W}$	00000000	Z 轴偏移
0x21	DUR	$R/\overline{W}$	00000000	敲击持续时间
0x22	Latent	$R/\overline{W}$	00000000	敲击延迟
0x23	Window	$R/\overline{W}$	00000000	敲击窗口
0x24	THRESH_ACT	$R/\overline{W}$	00000000	活动阈值
0x25	THRESH_INACT	$R/\overline{W}$	00000000	静止时间
0x26	TIME_INACT	$R/\overline{W}$	00000000	静止时间
0x27	ACT_INACT_CTL	$R/\overline{W}$	00000000	轴时能控制活动和静止检测
0x28	THRESH_FF	$R/\overline{W}$	00000000	自由落体阈值
0x29	TIME_FF	$R/\overline{W}$	00000000	自由落体时间
0x2A	TAP_AXES	$R/\overline{W}$	00000000	单机/双击轴控制
0x2B	ACT_TAP_STATUS	*R*	00000000	单机/双击源
0x2C	BW_RATE	$R/\overline{W}$	000001010	数据速率及功率模式控制
0x2D	POWER_CTL	$R/\overline{W}$	00000000	省电特性控制
0x2E	INT_ENABLE	$R/\overline{W}$	00000000	中断使能控制
0x2F	INT_MAP	$R/\overline{W}$	00000000	中断映射控制
0x30	INT_SOURCE	*R*	00000010	中断源
0x31	DATA_FORMAT	$R/\overline{W}$	00000000	数据格式控制
0x32	DATAX0	*R*	00000000	X 轴数据 0
0x33	DATAX1	*R*	0000000	X 轴数据 1
0x34	DATAY0	*R*	0000000	Y 轴数据 0
0x35	DATAY1	*R*	00000000	Y 轴数据 1
0x36	DATAZ0	*R*	00000000	Z 轴数据 0

0x37	DATAZ1	R	00000000	Z 轴数据 1
0x38	FIFO_CTL	$R/\overline{W}$	00000000	FIFO 控制
0x39	FIFO_STATUS	$R/\overline{W}$	00000000	FIFO 状态

■ 常用的寄存器：

1）POWER_CTL，用来设定供电模式，与 BW_RATE 配合，可设定数据率，默认值为 100 Hz。ADXL345 正常供电情况下，能根据输出数据率大小自动调节其功耗。如果要进一步降低功耗， 将 BW_RATE 寄存器中的 LOW_POWER 位置位，进入低功耗模式。Link Bit ：1，连接；0，非连接。AUTO_SLEEP Bit ：0，非自动 sleep 模式；1 自动 sleep 模式。Measure Bit ：0，独立模式；1，测量模式。Sleep Bit 0，正常模式；1，sleep 模式。本实验中设置为 0x08。

表 11-2　寄存器 0x2D 位定义

D7	D6	D5	D4	D3	D2	D1	D0
0	0	Link	AUTO_SLEEP	Measure	Sleep	Wakeup	

表 11-3　Wakeup Bits 在 Sleep 模式下的读取频率设置

设置		频率(Hz)
D1	D0	
0	0	8
0	1	4
1	0	2
1	1	1

2）DATA_FORMAT， 该寄存器的设置影响着 DATAX0DATAX1、DATAY0、DATAY1、DATAZ0、DATAZ1 数据寄存器中的数据格式。DATA_FORMAT 该 8 位寄存器可控制 6 项设置，通过设置 SPI 位可设定 SPI 是采用 3 线还是 4 线接口模式，FULL_RES 位与 RANGE 位，用于设定加速度量程和对应的分辨率模式，SELF_TEST 位用于自检。INT_INVERT 为中断模式设置 0，相对高电平中断；1 相对低电平中断。本实验中设置为 0x0b。

表 11-4　寄存器 0x31 的位定义

D7	D6	D5	D4	D3	D2	D1	D0
SELF_TEST	SPI	INT_INVERT	0	FULL_RES	Justify	Range	

表 11-5　Range 位设置

设置		频率(Hz)
D1	D0	

0	0	±2g
0	1	±4g
1	0	±8g
1	1	±16g

3）FIFO_CTL，设置缓存器具体的工作模式，比如 Bypass、FIFO、Stream、Trigger 模式，各种模式区别如下：在 Bypass 模式中，FIFO 缓存器是退化的，仅 FIFO［0］存储一次采样结果，无论是否被读取，新数据到来时将旧数据覆盖；在 FIFO 模式中，FIFO 缓存器不停地收集数据直到缓存器满，此时如果没有及时读数据，新到样本数据将被丢弃，而当 FIFO 被读取后，它将继续收集新到数据；在 Stream 模式中， FIFO 缓存器不停地收集数据，当缓存器满，自动丢弃 FIFO［0］，其他样本值向前移位填充，最新数据填入 FIFO［31］；在 Trigger 模式中，FIFO 开始工作与 Stream 模式类似，收集样本值直到 FIFO 缓存器满，然后丢弃最旧的数据，一旦触发事件发生（由 FIFO_CTL 寄存器 TRIG_SOURCE 位所定义），FIFO 将保留最后 n 采样值（其中 n 在 FIFO_CTL 寄存器中指定），然后像 FIFO 模式一样运行，即 FIFO 不满时，继续收集新的样本值。

4）INT_MAP、INT_ENABLE，ADXL345 为事件驱动提供两个中断输出引脚：INT1、INT2。所有的中断功能，例如 DATA_READY、FREE_FALL、OVERRUN 等等， 均可同时使用，唯一的限制是有一些功能可能会共享中断引脚。

5）OFSX、OFSY、OFSZ，用来存放标定的 X、Y、Z 轴的偏移量，初始化传感器时使用。

（详细请参考资料：ADXL345 datasheet）

11.3 本实验使用的资源说明

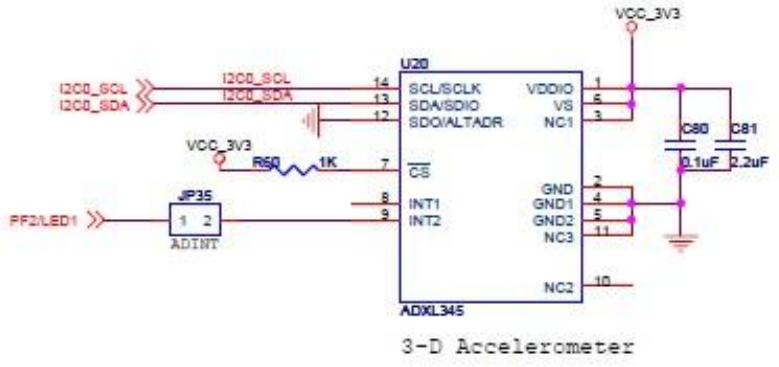


图 11-3 加速度传感器部分试验电路原理图

接口：

按键	对应端口	功能
I2C0_SDA	PB3	I2C 协议 SDA
I2C0_SCL	PE2	I2C 协议 SCL

PF2/LED1	PF2.	中断输出

(I2C 通信协议见章节 7)

图 11-4 为 ADXL345 芯片的方向图，标示在开发板上的坐标系：

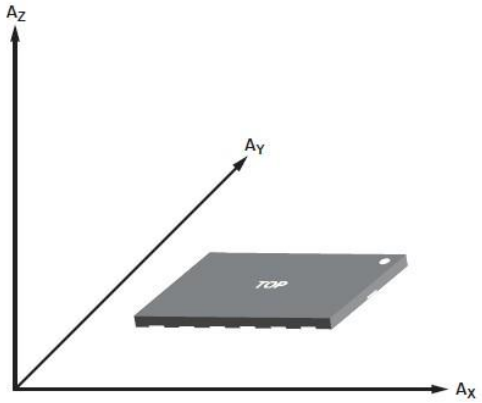


图 11-4　加速度芯片方向图

11.4 实验十一　Acceleration Transducer

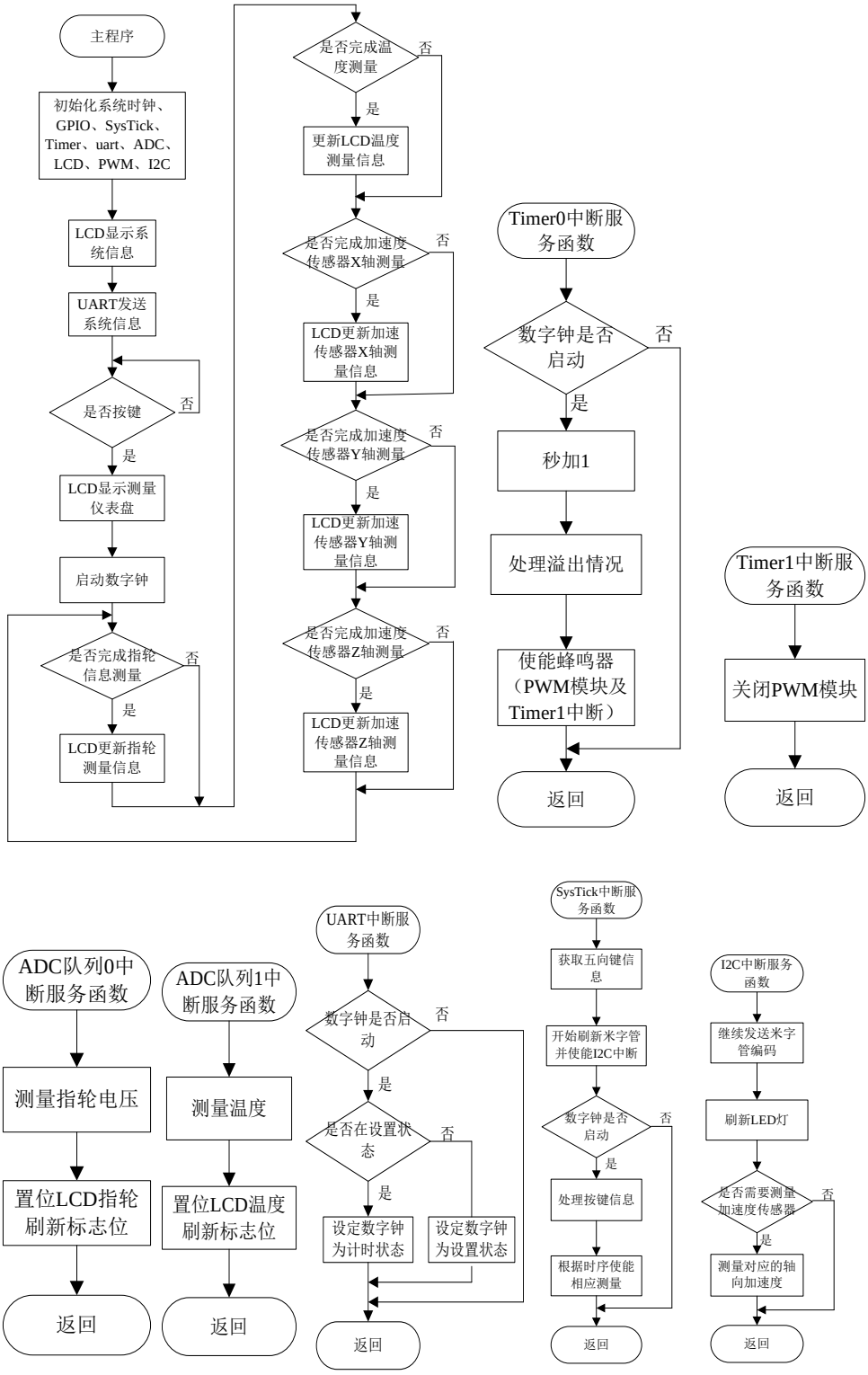
■ 实验目的：

了解 I2C 总线的特点和功能；
学会使用 I2C 自检程序；
会使用 I2C 总线对 PCA9557 芯片进行操作；
学会常用键盘防抖方法。
主要了解加速度传感器的原理,使用 I2C 总线对 ADXL345 芯片进行操作。

■ 实验效果：

具有第十章的全部功能，可设置目标时间的电子钟;带状态显示屏;使用指轮调节 A/D;可显示工作温度;另外在 LCD 上显示三方向轴的加速度。

■ 实验流程图:



11.5 加速度传感器实验软件流程图

■ Step1：实验准备

准备工作：

在上章节实验的基础上

调整跳冒：

连接		断开	
JP41	PE5	JP24	SW2-PRESS
JP42	PE4	JP25	SW3-UP
JP44	PF1	JP30	SW4-DOWN
JP39	PF3	JP2	BUZZER
JP38	PB7		LCD_RST\
JP11		JP12	LCD_CS\(下拉)
JP6	PD0	JP29,JP36	LCD_D0
JP7	PD1	JP27,JP37	LCD_D1
JP8	PD4	JP26	LCD_D4
JP9	PD5	JP31	LCD_D5
		JP23	LCD_Backlight
JP20	PB6	JP28,JP43	THUMBWHEEL_VREF
JP48	PB4	JP45	THUMBWHEEL_POT
JP35	PF2	JP3	Accelerometer Interrupt
打开 SSCOM。			

■ Step2：创建新工程

本章节实验在上章节的基础上完成，复制上章节实验工程中所有的文件。在封装好的各个功能内修改以后步骤的内容，完成实验。

■ Step3：修改 GPIODRIVERCONFIGURE.H

在 GPIODRIVERCONFIGURE.H 中增加

```
#define ACCELEROMETER_INT_BASE     GPIO_PORTF_BASE
#define ACCELEROMETER_INT_PIN      GPIO_PIN_2
```

■ Step4：修改 I2CCONFIGURE.H

在 I2CCONFIGURE.H 中定义 ACCELEROMETER 的基址和其寄存器地址。

```
#define ACCELEROMETER_I2CADDR 0x53

#define ADXL345_REG_DEVICE_ID                0x00
#define ADXL345_REG_THRESH_TAP                0x1d
#define ADXL345_REG_OFFSET_X                 0x1e
#define ADXL345_REG_OFFSET_Y                 0x1f
#define ADXL345_REG_OFFSET_Z                 0x20
#define ADXL345_REG_TAP_DURATION          0x21
#define ADXL345_REG_TAP_LATENCY               0x22
#define ADXL345_REG_TAP_WINDOW                0x23
#define ADXL345_REG_ACT_THRESH                0x24
```

```
#define ADXL345_REG_INACT_THRESH          0x25
#define ADXL345_REG_INACT_TIME            0x26
#define ADXL345_REG_ACT_INACT_CTL         0x27
#define ADXL345_REG_FREEFALL_THRESHOLD 0x28
#define ADXL345_REG_FREEFALL_TIME         0x29
#define ADXL345_REG_TAP_AXIS_CTL          0x2a
#define ADXL345_REG_ACT_TAP_STATUS        0x2b
#define ADXL345_REG_DATARATE_CTL          0x2c
#define ADXL345_REG_POWER_CTL             0x2d
#define ADXL345_REG_INT_ENABLE            0x2e
#define ADXL345_REG_INT_MAP               0x2f
#define ADXL345_REG_INT_SOURSE            0x30
#define ADXL345_REG_DATA_FORMAT           0x31
#define ADXL345_REG_DATA_X0               0x32
#define ADXL345_REG_DATA_X1               0x33
#define ADXL345_REG_DATA_Y0               0x34
#define ADXL345_REG_DATA_Y1               0x35
#define ADXL345_REG_DATA_Z0               0x36
#define ADXL345_REG_DATA_Z1               0x37
#define ADXL345_REG_FIFO_CTL              0x38
#define ADXL345_REG_FIFO_STATUS            0x39
```

■ Step5：修改 GPIODriverConfigure.c

在 GPIODriverConfigure.c 中的 void GPIOInitial(void)中添加以下加粗字体代码，完成对加速度传感器的端口使能及引脚功能配置。

```
void GPIOInitial(void)
{
        //GPIO 模块使能
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA); //UART0
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);      //KEY RIGHT | KEY LEFT
| I2C0 | LCD_RST | LCD_RD | ADC_THUMBWHEEL
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD); //UART1-RS232 | LCD_DATA
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);  //KEY PRESS | KEY UP
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);  //KEY DOWN  | LED1 |
LED0 | PWM_BUZZER | ACCELEROMETER
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOH); //LCD_WR | LCD_DC
    .....
    .....
    GPIOPinTypeADC(THUMBWHEEL_BASE,THUMBWHEEL_PIN);
//Set ADC_Thumbwheel

    GPIOPinTypeADC(THUMBWHEEL_VREF_BASE,THUMBWHEEL_VREF_PIN);
    GPIOPinTypeGPIOInput(ACCELEROMETER_INT_BASE,
```

```
ACCELEROMETER_INT_PIN);                    //Set Accelerometer Interrupt Pin
```

■ Step6：修改 SysTick_ISR()中断函数

在 SysTick_ISR(void)中断函数中修改以下代码，完成定时使能加速度传感器的设置和读取数据。

```
void SysTick_ISR(void)    //SysTick 中断函数，此处 5ms 定时，作时间片用
{
……
……
        MeasureTriggerCounter++;
         switch (MeasureTriggerCounter)
         {
              case 16:   AccelerationEnable=1;break;
              case 33:   ADCProcessorTrigger(ADC0_BASE,0);break;
              case 50:   AccelerationEnable=2;break;
              case 66:   ADCProcessorTrigger(ADC0_BASE,0);break;
              case 82:   AccelerationEnable=3;break;
              case 100:
              {
                   MeasureTriggerCounter=0;
                   ADCProcessorTrigger(ADC0_BASE,1);
                   break;
              }
              default:   break;
         }
……..
……..
}
```

■ Step7：修改 I2CConfigure.c

添加加速度传感器的变量定义:

```
short int volatile AccelerationDirX=0;
short int volatile AccelerationDirY=0;
short int volatile AccelerationDirZ=0;
unsigned char volatile AccelerationEnable=0;
volatile unsigned char LCDDrawingFlag_AccelerationDirX=0;
volatile unsigned char LCDDrawingFlag_AccelerationDirY=0;
volatile unsigned char LCDDrawingFlag_AccelerationDirZ=0;
```

在函数 void I2C0MasterInitial(void)中做一下修改

在 I2C 设备的初始化中添加对 ADXL345 加速度传感器的初始化，完成对 ADXL345 的电源及数据格式设置。添加黑体字代码。

```
void I2C0MasterInitial(void)
{
……...
……
    //配置所有 PCA9557 为输出模式，并关闭所有 LED
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,LED_I2CADDR,PCA9557_
REG_CONFIGURE,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEG_I2CADDR,PC
A9557_REG_CONFIGURE,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEL1_I2CADDR,PC
A9557_REG_CONFIGURE,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEL2_I2CADDR,PC
A9557_REG_CONFIGURE,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,ACCELEROMETER_I
2CADDR,ADXL345_REG_POWER_CTL,0x08);    //set to   low power mode
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,ACCELEROMETER_I
2CADDR,ADXL345_REG_DATA_FORMAT,0x0b);

    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,LED_I2CADDR,PCA9557_
REG_OUTPUT,0x00);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEG_I2CADDR,PC
A9557_REG_OUTPUT,0xff);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEL1_I2CADDR,PC
A9557_REG_OUTPUT,0xff);
    I2CMasterTransmit_Burst_2Bytes(I2C0_MASTER_BASE,TUBE_SEL2_I2CADDR,PC
A9557_REG_OUTPUT,0xff);
……..
……..
}
```

在 I2C0_ISR(void)中断函数中修改以下代码，其中 case10 完成写入 XYZ 方向数据 0 的操作地址；case11 读入 case10 中的地址；case12 完成写入 XYZ 方向数据 1 的操作地址；case13 读入 case12 中的地址；case13 得到加速度传感器 XYZ 方向的加速度数据，使能 LCD 显示信号。

修改的代码如下：

```
void I2C0_ISR(void)   // I2C0 中断服务
{
……..
……..
case 10:
{
```

```
    switch (AccelerationEnable)
    {
        case 0:    I2C0TransmitionState=14;break;
        case 1:
        {

    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE,ACCELEROMETER_I2CADDR,false)
;  // //主机写操作
        I2CMasterDataPut(I2C0_MASTER_BASE,ADXL345_REG_DATA_X0);//写一个
X0 轴数据 0    地址为 0x32

    I2CMasterControl(I2C0_MASTER_BASE,I2C_MASTER_CMD_SINGLE_SEND);    //
启动命令信号发送
        break;
        }
        case 2:
        {

    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE,ACCELEROMETER_I2CADDR,false)
; / //主机写操作
        I2CMasterDataPut(I2C0_MASTER_BASE,ADXL345_REG_DATA_Y0);//写一个
Y0 轴数据 0    地址为 0x34

    I2CMasterControl(I2C0_MASTER_BASE,I2C_MASTER_CMD_SINGLE_SEND);    //
启动命令信号发送
        break;
        }
        case 3:
        {

    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE,ACCELEROMETER_I2CADDR,false)
; //主机写操作
        I2CMasterDataPut(I2C0_MASTER_BASE,ADXL345_REG_DATA_Z0);  //写一个
Z 轴数据 0    地址为 0x36

    I2CMasterControl(I2C0_MASTER_BASE,I2C_MASTER_CMD_SINGLE_SEND);    //
启动命令信号发送
        break;
        }
        default:   break;
    }
    break;
}
case 11:
```

```
{
    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE,ACCELEROMETER_I2CADDR,true);
//主机读操作  命令
    I2CMasterControl(I2C0_MASTER_BASE,I2C_MASTER_CMD_SINGLE_SEND);  //
    启动命令信号发送
    break;
}
case 12:
{
    switch (AccelerationEnable)
    {
        case 1:
        {
        AccelerationDirX=I2CMasterDataGet(I2C_MASTER_BASE);

    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE,ACCELEROMETER_I2CADDR,false)
;   //主机写操作
        I2CMasterDataPut(I2C0_MASTER_BASE,ADXL345_REG_DATA_X1);//写一个
X1 轴数据 1   地址为 0x33

    I2CMasterControl(I2C0_MASTER_BASE,I2C_MASTER_CMD_SINGLE_SEND);
    // 启动命令信号发送
        break;
      }
      case 2:
       {
        AccelerationDirY=I2CMasterDataGet(I2C_MASTER_BASE);

    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE,ACCELEROMETER_I2CADDR,false)
;   //主机写操作
        I2CMasterDataPut(I2C0_MASTER_BASE,ADXL345_REG_DATA_Y1);//写一个
Y1 轴数据 1   地址为 0x35

    I2CMasterControl(I2C0_MASTER_BASE,I2C_MASTER_CMD_SINGLE_SEND);
    // 启动命令信号发送
        break;
        }
        case 3:
        {
        AccelerationDirZ=I2CMasterDataGet(I2C_MASTER_BASE);

    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE,ACCELEROMETER_I2CADDR,false)
;    //主机写操作
        I2CMasterDataPut(I2C0_MASTER_BASE,ADXL345_REG_DATA_Z1);//写一个
```

```
Z1 轴数据 1    地址为 0x37

    I2CMasterControl(I2C0_MASTER_BASE,I2C_MASTER_CMD_SINGLE_SEND);
    // 启动命令信号发送
        break;
        }
        default:  break;
    }
    break;
}
case 13:
{
    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE,ACCELEROMETER_I2CADDR,true);
//主机读操作  命令
    I2CMasterControl(I2C0_MASTER_BASE,I2C_MASTER_CMD_SINGLE_SEND);
    //   启动命令信号发送
    break;
}
case 14:
{
    switch (AccelerationEnable)
    {
    case 1:
    {
    AccelerationDirX += I2CMasterDataGet(I2C_MASTER_BASE)*256;   //   得   到
AccelerationDirX 的数据          LCDDrawingFlag_AccelerationDirX=1;
    break;
    }
    case 2:
    {
    AccelerationDirY  +=  I2CMasterDataGet(I2C_MASTER_BASE)*256;       // 得 到
AccelerationDirY 的数据
    LCDDrawingFlag_AccelerationDirY=1;
    break;
    }
    case 3:
    {
        AccelerationDirZ  +=  I2CMasterDataGet(I2C_MASTER_BASE)*256;   // 得 到
AccelerationDirY 的数据
        LCDDrawingFlag_AccelerationDirZ=1;
        break;
    }
    default:  break;
    }
```

```
        AccelerationEnable=0;
        break;
}
……
……
}       //I2C0 中断服务结束
```

■ Step8： main 主函数

主函数用于完成实验软件流程图中的主要功能。

其代码如下：

```
int main(void)                          //主函数
{
    ClockInitial();
    GPIOInitial();
    SysTickInitial();
    TimerInitial();
    UART0Initial();
    ADCInitial();
    LCDInitial();
    GrContextInit(&sContext, &g_sKitronix320x240x16_SSD2119);

    PWMInitial();
    SystemState |= I2C0PullUpTest();
    if (!(SystemState & 0x01))
        I2C0MasterInitial();
    NixieTubeSet_UnsignedChar('D','O','V','E');
    SysTickEnable();
    IntMasterEnable();
    LCDSystemCheckInformationDisplay(&sContext,SystemState);
    UARTSystemCheckInformationTransmit(UART0_BASE,SystemState);

    while (KeyNumber==0);KeyNumber=0;
    UARTStringPut(UART0_BASE,"System activated!\r\n");

    GrContextForegroundSet(&sContext, ClrBlack);
    GrRectFill(&sContext,&sRect);
    GrContextForegroundSet(&sContext,ClrGreenYellow);
    GrContextFontSet(&sContext,&g_sFontCm40b);
    GrStringDrawCentered(&sContext,"Running",-1,
            GrContextDpyWidthGet(&sContext)/2,LCD_CLOCK_STATE_LINE,1);

    GrContextForegroundSet(&sContext, ClrSnow);
    GrContextFontSet(&sContext, &g_sFontCm18b);
```

```
    GrStringDraw(&sContext, "Thumbwheel
Voltage:",-1,15,LCD_THUMBWHEEL_LINE,0);
    GrStringDraw(&sContext, "mV",-1,280,LCD_THUMBWHEEL_LINE,0);
    GrStringDraw(&sContext, "Adjustment Scale:",-1,15,LCD_SETBIT_LINE,0);
    GrStringDraw(&sContext, "Chip Temperature:",-1,15,LCD_TEMPERATURE_LINE,0);
    GrStringDraw(&sContext, "\'C",-1,280,LCD_TEMPERATURE_LINE,0);
    GrStringDraw(&sContext, "Acceleration
Dir-X:",-1,15,LCD_ACCELEROMETER_X_LINE,0);
    GrStringDraw(&sContext, "mg",-1,280,LCD_ACCELEROMETER_X_LINE,0);
    GrStringDraw(&sContext, "Acceleration
Dir-Y:",-1,15,LCD_ACCELEROMETER_Y_LINE,0);
    GrStringDraw(&sContext, "mg",-1,280,LCD_ACCELEROMETER_Y_LINE,0);
    GrStringDraw(&sContext, "Acceleration
Dir-Z:",-1,15,LCD_ACCELEROMETER_Z_LINE,0);
    GrStringDraw(&sContext, "mg",-1,280,LCD_ACCELEROMETER_Z_LINE,0);

    NixieTubeSet_UnsignedInt(0);
    NixieTubeDropSet(3);

    TimerEnable(TIMER0_BASE,TIMER_A);
    PWMGenEnable(PWM_BASE,PWM_GEN_2);
    TimerEnable(TIMER1_BASE,TIMER_A);
    SystemActivatedFlag=1;
    while(1)
    {
        if (LCDDrawingFlag_Running)
        {
            GrContextForegroundSet(&sContext,ClrGreenYellow);
            GrContextFontSet(&sContext,&g_sFontCm40b);
            GrStringDrawCentered(&sContext,"Running",-1,
                                 GrContextDpyWidthGet(&sContext)/2,
                                 LCD_CLOCK_STATE_LINE,1);
            LCDDrawingFlag_Running=0;
        }
        if (LCDDrawingFlag_Pause)
        {
            GrContextForegroundSet(&sContext,ClrOrange);
            GrContextFontSet(&sContext,&g_sFontCm40b);
            GrStringDrawCentered(&sContext,"  Pause  ",-1,
                                 GrContextDpyWidthGet(&sContext)/2,
                                 LCD_CLOCK_STATE_LINE,1);
            LCDDrawingFlag_Pause=0;
        }
        if (LCDDrawingFlag_Thumbwheel)
```

```
        {
            sprintf(tMessage,"%04d ",(ThumbwheelResult*3000)/1023);
            GrContextForegroundSet(&sContext, ClrYellow);
            GrContextFontSet(&sContext, &g_sFontCm18b);
            GrStringDraw(&sContext, tMessage,-1,220,LCD_THUMBWHEEL_LINE,1);
            switch (ThumbwheelResult/256)
            {
                case 0:
                {
                    sprintf(tMessage," 1 Second    ");
                    SetBitFlag=0;
                    break;
                }
                case 1:
                {
                    sprintf(tMessage,"10 Seconds ");
                    SetBitFlag=1;
                    break;
                }
                case 2:
                {
                    sprintf(tMessage," 1 Minute " );
                    SetBitFlag=2;
                    break;
                }
                case 3:
                {
                    sprintf(tMessage,"10 Minutes");
                    SetBitFlag=3;
                    break;
                }
                default:
                {
                    sprintf(tMessage," 1 Second ");
                    SetBitFlag=0;
                    break;
                }
            }
            GrContextForegroundSet(&sContext, ClrCyan);
            GrStringDraw(&sContext, tMessage,-1,220,LCD_SETBIT_LINE,1);
            LCDDrawingFlag_Thumbwheel=0;
        }
        if (LCDDrawingFlag_Temperature)
        {
```

```
                sprintf(tMessage,"%0.2f ",((float)(151040-225*TemperatureResult)/1024));
                GrContextForegroundSet(&sContext, ClrRed);
                GrContextFontSet(&sContext, &g_sFontCm18b);
                GrStringDraw(&sContext, tMessage,-1,220,LCD_TEMPERATURE_LINE,1);
                LCDDrawingFlag_Temperature=0;
            }
            if (LCDDrawingFlag_AccelerationDirX)
            {
                sprintf(tMessage,"%03d ",AccelerationDirX*4);
                GrContextForegroundSet(&sContext, ClrLime);
                GrContextFontSet(&sContext, &g_sFontCm18b);
                GrStringDraw(&sContext,
tMessage,-1,220,LCD_ACCELEROMETER_X_LINE,1);
                LCDDrawingFlag_AccelerationDirX=0;
            }
            if (LCDDrawingFlag_AccelerationDirY)
            {
                sprintf(tMessage,"%03d ",AccelerationDirY*4);
                GrContextForegroundSet(&sContext, ClrBlue);
                GrContextFontSet(&sContext, &g_sFontCm18b);
                GrStringDraw(&sContext,
tMessage,-1,220,LCD_ACCELEROMETER_Y_LINE,1);
                LCDDrawingFlag_AccelerationDirY=0;
            }
            if (LCDDrawingFlag_AccelerationDirZ)
            {
                sprintf(tMessage,"%03d ",AccelerationDirZ*4);
                GrContextForegroundSet(&sContext, ClrSalmon);
                GrContextFontSet(&sContext, &g_sFontCm18b);
                GrStringDraw(&sContext,
tMessage,-1,220,LCD_ACCELEROMETER_Z_LINE,1);
                LCDDrawingFlag_AccelerationDirZ=0;
            }
        }
}
```

整体代码结构图略。

■ Step 9: 编译和下载