

Cortex-M4原理与实践实验报告

题目： TFT液晶显示实验

**作者姓名 章金皓**

**指导教师**  **付明磊**

**专业班级 自动化1901**

**学 院 信息工程学院**

**提交日期** 2021年12月5日

**一、实验目的**

1. 了解 240x400TFT LCD 的工作原理。

2. 了解 TFT LCD 控制芯片 OTM4001A 的控制方式。

3. 学习并使用数学库 IQmathLib.h。

4. 学习 TM4C129x Series Cortex-M4 的 EPI 总线操作方式。

5. 学习 EPI 相关库函数的使用。

**二、实验设备**

1.计算机一台，操作系统为WindosXP或Windos7，装有CCSv11.0软件。

2.EK-TM4C1294XL实验开发板一块。

3.USB连接线一条。

**三、实验原理和流程**

**1. TFT 彩屏工作原理**

TFT(Thin Film Transistor)LCD 即薄膜场效应晶体管 LCD，是有源矩阵类型液晶显示器 (AM-LCD)中的一种。和 TN 技术不同的是，TFT 的显示采用“背透式”照射方式——假想的 光源路径不是像 TN 液晶那样从上至下，而是从下向上。这样的作法是在液晶的背部设置特 殊光管，光源照射时通过下偏光板向上透出。由于上下夹层的电极改成 FET 电极和共通电 极，在 FET 电极导通时，液晶分子的表现也会发生改变，可以通过遮光和透光来达到显示 的目的，响应时间大大提高到 80ms 左右。因其具有比 TN LCD 更高的对比度和更丰富的色 彩，荧屏更新频率也更快，故 TFT 俗称“真彩”。

LCD 是由二层玻璃基板夹住液晶组成的，形成一个平行板电容器，通过嵌入在下玻璃 基板上的 TFT 对这个电容器和内置的存储电容充电，维持每幅图像所需要的电压直到下一 幅画面更新。液晶的彩色都是透明的必须给 LCD 衬以白色的背光板上才能将五颜六色表达 出来，而要使白色的背光板有反射就需要在四周加上白色灯光。因此在 TFT LCD 的底部都 组合了灯具，如 CCFL 或 LED。

**2. OTM4001A 控制芯片简介**

OTM4001A是一款 262144 色，用于中小型 TFT LCD 显示屏的片上系统（SoC）驱动芯片，通过指定用于图形数据的RAM能支持达240xRGBx432分辨率。OTM4001A 内部的时序控制器能为不同的需求提供不同接口方式，OTM4001A提供了系统接口，包括 8/9/16/18 位并口和 SPI 串口方式（本实验采用 16 位并口方式）；OTM4001A 也提供了 6/16/18 位 RGB 接口，用于动态显示图片。OTM4001A 的主要特性还有窗口地址功能能限制数据重写区域， 并减少数据传输；内部 6 位 D/A 转换器输出的 64γ 颜色校准；内部 233280 字节的 RAM； 背光引脚输出控制的内置自适应背光控制功能（CABC）；逻辑供电电压范围2.5~3.6V，IO接口支持操作电压 1.65~3.6V，模拟供电电压范围 2.5~3.6V；内置的内部晶振与硬件复位。 本实验中，采用 80 系统总线的 16 位并口方式，对 TFT LCD 的常规写操作时序特征和时序图分别如下表 1 和图 1 所示：

表 1 16位并口方式时序特征

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Item** | **symbol** | | **Unit** | **Min.** | **Typ.** | **Max.** |
| Bus cycle time | Write | tCYCW | ns | 150 | - | - |
| Read | tCYCR | ns | 450 | - | - |
| Write low-level pulse width | | PWLW | ns | 55 | - | - |
| Read low-level pulse width | | PWLR | ns | 170 | - | - |
| Write high-level pulse width | | PWHW | ns | 70 | - | - |
| Read high-level pulse width | | PWHR | ns | 250 | - | - |
| Write/Read rise/ fall time | | tWRr， WRf | ns | - | - | 10 |
| Setup time | Write (RS to CS\*,WR\*) | tAS | ns | 0 | - | - |
| Read (RS to CS\*, RD\*) | ns | 10 | - | - |
| Address Hold Time | | tAH | ns | 2 | - | - |
| Write data setup time | | tDSW | ns | 25 | - | - |
| Write data hold time | | tH | ns | 10 | - | - |
| Read data delay time | | tDDR | ns | - | - | 150 |
| Read data hold time | | tDHR | ns | 5 | - | - |

图示, 工程绘图

描述已自动生成

图 1 16位并口方式时序图

为使液晶正常工作，我们需要在初始化之前先复位液晶，复位液晶的时序特征和时序图 分别如下表 2 和图 2 所示：

表 2 TFTLCD复位时序特征

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Item** | **Symbol** | **Unit** | **Min.** | **Typ.** | **Max.** |
| Reset low-level width | tRES | ms | 1 | － | － |
| Reset rise time | trRES | µs | － | － | 10 |



图 2 TFTLCD复位时序图

为减少数据访问次数，OTM4001A 还有窗口访问功能，能指定对液晶的操作区域，相关 寄存器如下：

1) 窗口水平 RAM 起始地址（R210h）

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| W | 1 |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | HSA7 (0) | HSA6 (0) | HSA5 (0) | HSA4 (0) | HSA3 (0) | HAS (0)2 | HSA1 (0) | HSA0 (0) |

2) 窗口水平 RAM 结束地址（R211h）

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| W | 1 |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | HEA7 (1) | HEA6 (1) | HEA5 (1) | HEA4 (1) | HEA3 (1) | HEA2 (1) | HEA1 (1) | HEA0 (1) |

3) 窗口垂直 RAM 起始地址（R212h）

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| W | 1 |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | VSA8 (0) | VSA7 (0) | VSA6 (0) | VSA5 (0) | VSA4 (0) | VSA3 (0) | VSA2 (0) | VSA1 (0) | VSA0 (0) |

4) 窗口垂直 RAM 结束地址（R213h）

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| W | 1 |  | 0 | 0 | 0 | 0 | 0 | 0 | 0 | VEA8 (1) | VEA7 (1) | VEA6 (1) | VEA5 (1) | VEA4 (1) | VEA3 (1) | VEA2 (1) | VEA1 (1) | VEA0 (1) |

A7-0 和 HEA7-0 代表了水平方向的窗口起始和结束地址，VSA8-0和 VEA8-0 代表垂直方向的窗口起始和结束地址。要使用窗口功能需满足下式：

“00”h≤ HSA7-0< HEA7-0 ≤ “EF”h and HEA-HAS>=“04h”

“00”h≤ VSA8-0< VEA8-0 ≤ 9’h1AF

对窗口访问功能定义如下图 3 所示：

图示

描述已自动生成

图 3 窗口访问功能定义

寄存器设置好访问的 RAM 之后，写入一个数据将会写到指定的区域，不需要完全擦除所有数据，这样大大减少了操作时间。

OTM4001A 的操作时序与窗口访问功能后，我们就能进行对 TFT LCD 进行初始化，让 LCD 显示我们需要的文字或图片。OTM4001A 有运行模式，睡眠模式，深度睡眠模式，睡眠模式和深度睡眠模式用于当不用显示时，节省电量开销，对于其他寄存器的操作，请参考 TFTdisplay.pdf。

**3. TivaWare C series 的数学库 IQmathLib.h**

Tiva™ IQmath.h 是一个高度优化和高精度的数学函数库，帮助 C/C++程序员在 Tiva 器 件上无缝地将浮点运算转化为定点计算。计算速度将显著快于浮点运算。IQmath 库采用 32 位定点带符号数作为基本数据类型。这些定点数的格式从 IQ1 到 IQ30，这里 IQ 数据格式代 表了数的小数位数。C 语言中要调用 IQmath 函数，需要先包含头文件“IQmath/IQmathLib.h”。 然后，才能使用\_iq 和\_iqN 的数据类型以及库中的函数，以下代码是对 IQmath 库函数的简 单调用：

#include "IQmath/IQmathLib.h" int main(void)

{

\_iq24 X, Y, Z; X = \_IQ24(1.0);

Y = \_IQ24(7.0);

Z = \_IQ24div(X, Y);

}

IQmath 库函数的具体使用请参阅 TivaWare™ IQmath Library USER’S GUIDE。

**4. TM4C129x Series Cortex-M4 的 EPI 工作原理与方式**

外围设备接口(EPI, External Peripheral Interface)，是一种用于外设或存储设备的高速并 行总线。有 3 种主要的工作方式，能无缝对接众多类型的外部设备。EPI 类似于标准的微处 理器地址/数据总线，通常必须和 1 种外设连接，不能一次和 2 种不同的外设连接。增强了 包括支持 uDMA，时钟控制和支持外部 FIFO 缓冲区等功能。

EPI 模块有以下主要特性：

用于外围设备和存储器的专用 8/16/32 位并行总线；

存储器接口支持连续存储访问，不限数据总线宽度，允许程序直接从 SDRAM， SRAM 和 Flash 存储器中运行；

阻断和非阻断读取；

使用内部写 FIFO 将处理器从严格时序要求中分离；

采用 uDMA 控制器高效地传输数据；

EPI 的 3 种主要的工作模式为 SDRAM 模式，传统的 Host-Bus 模式和通用模式，本实 验选用 EPI 的 Host-Bus 模式。Host-Bus 模式主要特点有：

支持 8 位，16 位 MCU 总线接口；

兼容类似 PIC，ATmega，8051 等其他类似设备；

访问 SRAM，NORFlash 和其他器件时，在非复用工作方式下，支持高达 1MB 的地 址访问，在多路复用工作方式下，支持 256MB 访问（16 位总线模式不带字节选择 的情况下，支持 512MB）；

对于一系列无地址的 8 位/16 位 FIFO 接口，支持外部 FIFO（XFIFO）EMPTY 和FULL 信号；

带有读和写数据等待状态计数器；

多种片选方式，包括单芯片，双芯片，四芯片片选。下图 4 为 EPI 模块框图。

图示, 示意图

描述已自动生成

图 4 EPI模块框图

**5. 实验用到的主要库函数简介**

void EPIModeSet(uint32\_t ui32Base, uint32\_t ui32Mode)函数用于选择 EPI 的工作方式。 ui32Base 为 EPI 模块的基地址；ui32Mode 为 EPI 的工作模式，本实验选择为 EPI\_MODE\_HB16， 配置成 16 位总线模式。

void EPIConfigHB16Set(uint32\_t ui32Base, uint32\_t ui32Config, uint32\_t ui32MaxWait)函数用于 16 位总线工作模式的详细设置。ui32Base 为 EPI 模块的基地址；ui32Config 为 16 位总 线工作方式的配置参数；ui32MaxWait 为等待的最大外部时钟数。

void EPIAddressMapSet(uint32\_t ui32Base, uint32\_t ui32Map)函数用于配置外部设备的地 址映射。ui32Base 为 EPI 模块的基地址；ui32Map 是地址映射参数，由一些宏定义的参数通 过逻辑与组成。对于存储设备，EPI 将外设视为一块连续的存储空间；对于无地址设备，EPI 将外设视为一个地址。

**6. 简化的 EPI 模块配置步骤**



**流程图：**



**四、实验代码、注释**

**实验代码：**

/\*

\* Sine\_wave.c

\*

\* Created on: 2014年8月25日

\* Author: Sam

\*/

#include <stdbool.h>

#include <stdint.h>

#include <string.h>

#include "inc/hw\_memmap.h"

#include "inc/hw\_types.h"

#include "inc/hw\_epi.h"

#include "inc/hw\_ints.h"

#include "driverlib/epi.h"

#include "driverlib/gpio.h"

#include "driverlib/sysctl.h"

#include "driverlib/rom.h"

#include "driverlib/rom\_map.h"

#include "driverlib/pin\_map.h"

#include "driverlib/systick.h"

#include "driverlib/interrupt.h"

#include "driverlib/ssi.h"

#include "driverlib/fpu.h"

#include "utils/uartstdio.h"

#include "TFTinit/TFT\_400x240\_OTM4001A\_16bit.h"

#include "EPIinit/EPIinit.h"

#include "driverlib/debug.h"

#include "grlib/grlib.h"

#include "driverlib/gpio.h"

#include "driverlib/adc.h"

#include "inc/hw\_gpio.h"

#include "inc/tm4c1294ncpdt.h"

#include "driverlib/uart.h"

#include "IQmath/IQmathLib.h"

#include "utils/uartstdio.h"

#include "TFTinit/picture.h"

#include "TOUCHinit/TOUCH\_TSC2046.h"

//

// Select floating-point math.

//

//#define MATH\_TYPE FLOAT\_MATH

//#include "IQmath/IQmathLib.h"

// Provide a definition for M\_PI, if it was not provided by math.h.

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#ifndef M\_PI

#define M\_PI 3.14159265358979323846F

#endif

//系统时钟频率（Hz）

uint32\_t g\_ui32SysClock;

//用于跟踪时间的系统时钟计数器

static volatile unsigned long g\_ulTickCount;

#define SERIES\_LENGTH 240

//创建用于保存数据系列值的缓冲区。它必须足够大，以保持将在条形图上显示的系列中的数据点的最大数量

typedef struct \_Series

{

uint32\_t xAxis;

int32\_t data;

}tSeries;

static tSeries g\_cSeries[240];

//每秒SysTick滴答数

#define TICKS\_PER\_SECOND 40

#define FSECONDS\_PER\_TICK (1.0F / (float)TICKS\_PER\_SECOND)

//这是此SysTick中断的处理程序。它只是增加一个用于计时的计数器。

void

SysTickIntHandler(void)

{ //更新我们的计时计数器。

g\_ulTickCount++;

if(g\_ulTickCount>=320)

g\_ulTickCount = 0;

}

//配置UART及其管脚。这必须在UARTprintf()之前调用

void

ConfigureUART(void)

{//使能UART使用的GPIO外设。

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOA);

//使能UART0

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_UART0);

//为UART模式配置GPIO管脚

GPIOPinConfigure(GPIO\_PA0\_U0RX);

GPIOPinConfigure(GPIO\_PA1\_U0TX);

GPIOPinTypeUART(GPIO\_PORTA\_BASE, GPIO\_PIN\_0 | GPIO\_PIN\_1);

//初始化控制台I/O的UART

UARTStdioConfig(0, 115200, g\_ui32SysClock);

}

void DrawTriWave(void)

{//使能FPU，因为某些编译器将使用浮点寄存器，即使对于非浮点代码也是如此。

//如果未启用FPU，则会导致故障。这也确保了浮点操作可以添加到这个应用程序

//中，并且可以正常工作并使用硬件浮点单元。最后，为中断处理程序启用了延迟堆栈。

//这允许在中断处理程序中使用浮点指令，但要牺牲额外的堆栈使用。

uint16\_t ui32Loop = 0,ulItemCount=0,FinishCalculateFlag=0,ulLastTickCount=0;

FPUEnable();

FPULazyStackingEnable();

//锁相环以120兆赫的频率运行

g\_ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

SysTickPeriodSet(g\_ui32SysClock / TICKS\_PER\_SECOND);

IntMasterEnable();//启用对处理器的中断

SysTickIntEnable();//启用SysTick中断

SysTickEnable();//使能SysTick

ConfigureUART();

EPIGPIOinit();

UARTprintf("TFTLCD test\n");

UARTprintf("EPI Type: host-bus 16-bit interface\n");

TFT\_400x240\_OTM4001Ainit(g\_ui32SysClock);//打开背光

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_0);

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_0, GPIO\_PIN\_0);

TFTLCD\_ShowString(50,10,"GC211 TFTLCD test!",CYAN,LIGHTBLUE);//显示GC211 TFTLCD test!

TFTLCD\_ShowString(60,40,"TriWave",GREEN,BLACK);//显示TriWave

TFTLCD\_DrawHorizontalLine(0,240,63,GREEN);//显示横边界线

TFTLCD\_DrawHorizontalLine(0,240,128,GREEN);

TFTLCD\_DrawHorizontalLine(0,240,193,GREEN);

TFTLCD\_DrawVerticalLine(64,193,0,GREEN);//显示竖边界线

TFTLCD\_DrawVerticalLine(64,193,239,GREEN);

TFTLCD\_ShowString(12,46,"1",WHITE,BLACK);

TFTLCD\_ShowString(2,195,"-1",WHITE,BLACK);//显示1和-1

while(1)

{ int count;

int Tlenth=80;

while(ulLastTickCount == g\_ulTickCount)

{}//等待下一个计时器

TFTLCD\_ShowData(170,60,g\_ulTickCount,BLUE,BLACK);

ulLastTickCount = g\_ulTickCount;

// 准备将新数据点添加到条形图。如果条形图中的项目数达到最大值，

//则数据点需要在缓冲器中“滑下”，从而可以在末尾添加新的数据

if(ulItemCount == SERIES\_LENGTH)

{

FinishCalculateFlag = 1;

}

// 否则，序列数据缓冲区不足，只需增加数据点的计数

else

{

g\_cSeries[ulItemCount].xAxis = ++ulItemCount;

}

count=g\_ulTickCount;

while(count>=Tlenth)

{

count-=Tlenth;

}

if(count<Tlenth/2)

g\_cSeries[ulItemCount - 1].data =count;

else if(count<Tlenth)

g\_cSeries[ulItemCount - 1].data =(-1)\*count+Tlenth;

if(!FinishCalculateFlag)

{//如果还没有计算三角波，请继续计算。

TFTLCD\_DrawPoint(g\_cSeries[ulItemCount - 1].xAxis,g\_cSeries[ulItemCount - 1].data+110,RED);

}

else

{//我们需要更新每个系统周期的三角波

for(ui32Loop=2;ui32Loop<SERIES\_LENGTH;ui32Loop++)

{// 清除以前的数据

TFTLCD\_DrawPoint(g\_cSeries[ui32Loop-1].xAxis,g\_cSeries[ui32Loop-1].data+110,0);

g\_cSeries[ui32Loop-1].data = g\_cSeries[ui32Loop].data;

// 更新数据

TFTLCD\_DrawPoint(g\_cSeries[ui32Loop-1].xAxis,g\_cSeries[ui32Loop-1].data+110,RED);

}

//重写显示信息

TFTLCD\_DrawHorizontalLine(0,240,128,GREEN);

TFTLCD\_DrawVerticalLine(64,192,0,GREEN);

TFTLCD\_DrawVerticalLine(64,192,239,GREEN);

}

}

}

void TFTLCD\_DrawCircle(uint32\_t x,uint32\_t y,uint32\_t r,uint32\_t color)

{//使能FPU，因为某些编译器将使用浮点寄存器，即使对于非浮点代码也是如此。

//如果未启用FPU，则会导致故障。这也确保了浮点操作可以添加到这个应用程序

//中，并且可以正常工作并使用硬件浮点单元。最后，为中断处理程序启用了延迟堆栈。

//这允许在中断处理程序中使用浮点指令，但要牺牲额外的堆栈使用。

FPUEnable();

FPULazyStackingEnable();

//锁相环以120兆赫的频率运行

g\_ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

SysTickPeriodSet(g\_ui32SysClock / TICKS\_PER\_SECOND);

IntMasterEnable();//启用对处理器的中断

SysTickIntEnable();//启用SysTick中断

SysTickEnable();//使能SysTick

ConfigureUART();

EPIGPIOinit();

UARTprintf("TFTLCD test\n");

UARTprintf("EPI Type: host-bus 16-bit interface\n");

TFT\_400x240\_OTM4001Ainit(g\_ui32SysClock);//打开背光

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_0);

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_0, GPIO\_PIN\_0);\

int num=800;//画圆点数

int i;

float fr;

\_iq24 angle,xi,yi,sin\_angle,cos\_angle;

fr=\_IQ(r);

for(i=0;i<num;i++)

{

angle=\_IQ((float)M\_PI\*(float)i/((float)(num/2)));//计算角度

sin\_angle=\_IQ24sin(angle);//sin值

cos\_angle=\_IQ24cos(angle);//cos值

xi=\_IQ24mpy(cos\_angle,fr);//计算x值

yi=\_IQ24mpy(sin\_angle,fr);//计算y值

TFTLCD\_DrawPoint(x+\_IQint(xi),y+\_IQint(yi),color);//画圆

}

}

void DrawSinWave(void)

{//使能FPU，因为某些编译器将使用浮点寄存器，即使对于非浮点代码也是如此。

//如果未启用FPU，则会导致故障。这也确保了浮点操作可以添加到这个应用程序

//中，并且可以正常工作并使用硬件浮点单元。最后，为中断处理程序启用了延迟堆栈。

//这允许在中断处理程序中使用浮点指令，但要牺牲额外的堆栈使用。

uint16\_t ui32Loop = 0,ulItemCount=0,FinishCalculateFlag=0,ulLastTickCount=0;

float fElapsedTime;

\_iq24 fRadians,fSine;

FPUEnable();

FPULazyStackingEnable();

//锁相环以120兆赫的频率运行

g\_ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

SysTickPeriodSet(g\_ui32SysClock / TICKS\_PER\_SECOND);

//启用对处理器的中断

IntMasterEnable();

//启用SysTick中断

SysTickIntEnable();

//使能SysTick

SysTickEnable();

ConfigureUART();

EPIGPIOinit();

UARTprintf("TFTLCD test\n");

UARTprintf("EPI Type: host-bus 16-bit interface\n");

TFT\_400x240\_OTM4001Ainit(g\_ui32SysClock);

//打开背光

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_0);

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_0, GPIO\_PIN\_0);

TFTLCD\_ShowString(50,10,"GC211 TFTLCD test!",CYAN,LIGHTBLUE);//显示GC211 TFTLCD test!

TFTLCD\_ShowString(60,40,"y=sin(pi\*t/2)",GREEN,BLACK);//显示y=sin(pi\*t/2)

TFTLCD\_DrawHorizontalLine(0,240,63,GREEN);//显示横边界线

TFTLCD\_DrawHorizontalLine(0,240,128,GREEN);

TFTLCD\_DrawHorizontalLine(0,240,193,GREEN);

TFTLCD\_DrawVerticalLine(64,193,0,GREEN);//显示竖边界线

TFTLCD\_DrawVerticalLine(64,193,239,GREEN);

TFTLCD\_ShowString(12,46,"1",WHITE,BLACK);//显示1和-1

TFTLCD\_ShowString(2,195,"-1",WHITE,BLACK);

while(1)

{//等待下一个计时器

while(ulLastTickCount == g\_ulTickCount)

{}

ulLastTickCount = g\_ulTickCount;

// 准备将新数据点添加到条形图。如果条形图中的项目数达到最大值，

//则数据点需要在缓冲器中“滑下”，从而可以在末尾添加新的数据

if(ulItemCount == SERIES\_LENGTH)

{

FinishCalculateFlag = 1;

}

// 否则，序列数据缓冲区不足，只需增加数据点的计数。

else

{// 增加已添加到条形图系列数据缓冲区的项目数

g\_cSeries[ulItemCount].xAxis = ++ulItemCount;

}

// 以十进制秒为单位，以浮点格式计算经过的时间

fElapsedTime = (float)g\_ulTickCount \* FSECONDS\_PER\_TICK;

//将时间转换为弧度 //

fRadians = \_IQ24(fElapsedTime \* (float)M\_PI);//\_IQ24将浮点常量或变量转换为IQ数

// 调整波浪周期。这将给我们一个4秒或0.25赫兹的波周期。

//这个数字是任意选择，以提供一个好看的波形显示。

fRadians = \_IQdiv2(fRadians);//IQdiv2除以弧度2

// 计算正弦。乘以0.5以减小振幅

fSine = \_IQ24sin(fRadians);//计算输入值的正弦值

// 最后，将正弦值保存到序列数据点缓冲区的最后一个位置

g\_cSeries[ulItemCount - 1].data = \_IQ24mpyI32int(fSine,64); //\_IQ24mpyI32int multiplies an IQ number fSine by an integer 32, returning the integer portion of the result.

if(!FinishCalculateFlag)

{// 如果我们还没有计算正弦值，继续计算

TFTLCD\_DrawPoint(g\_cSeries[ulItemCount - 1].xAxis,g\_cSeries[ulItemCount - 1].data+128,BLUE);

}

else

{//我们需要更新每个Systick周期的正弦值

for(ui32Loop=2;ui32Loop<SERIES\_LENGTH;ui32Loop++)

{//清除以前的数据

TFTLCD\_DrawPoint(g\_cSeries[ui32Loop-1].xAxis,g\_cSeries[ui32Loop-1].data+128,0);

g\_cSeries[ui32Loop-1].data = g\_cSeries[ui32Loop].data;

//更新数据

TFTLCD\_DrawPoint(g\_cSeries[ui32Loop-1].xAxis,g\_cSeries[ui32Loop-1].data+128,BLUE);

}

//重新绘制线

TFTLCD\_DrawHorizontalLine(0,240,128,GREEN);

TFTLCD\_DrawVerticalLine(64,192,0,GREEN);

TFTLCD\_DrawVerticalLine(64,192,239,GREEN);

}

}

}

void Adc(void)

{

uint32\_t ulADC0\_Value;

uint16\_t ui32Loop = 0,ulItemCount=0,FinishCalculateFlag=0,ulLastTickCount=0;

//使能FPU，因为某些编译器将使用浮点寄存器，即使对于非浮点代码也是如此。

//如果未启用FPU，则会导致故障。这也确保了浮点操作可以添加到这个应用程序

//中，并且可以正常工作并使用硬件浮点单元。最后，为中断处理程序启用了延迟堆栈。

//这允许在中断处理程序中使用浮点指令，但要牺牲额外的堆栈使用。

FPUEnable();

FPULazyStackingEnable();

//锁相环以120兆赫的频率运行

g\_ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

SysTickPeriodSet(g\_ui32SysClock / TICKS\_PER\_SECOND);

IntMasterEnable();//启用对处理器的中断

SysTickIntEnable();//启用SysTick中断

SysTickEnable();//使能SysTick

ConfigureUART();

EPIGPIOinit();

UARTprintf("TFTLCD test\n");

UARTprintf("EPI Type: host-bus 16-bit interface\n");

TFT\_400x240\_OTM4001Ainit(g\_ui32SysClock);//打开背光

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_0);

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_0, GPIO\_PIN\_0);

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_ADC0);//开启ADC0时钟

SysCtlPeripheralEnable(SYSCTL\_PERIPH\_GPIOD);//使能GPIOD口

GPIOPinTypeADC(GPIO\_PORTD\_BASE, GPIO\_PIN\_7);//将PD7口配置成AD输入模式

TFTLCD\_ShowString(50,10,"GC211 TFTLCD test!",CYAN,LIGHTBLUE);//显示GC211 TFTLCD test!

TFTLCD\_ShowString(60,40,"ADC",GREEN,BLACK);//显示ADC

TFTLCD\_DrawHorizontalLine(0,240,63,GREEN);//显示横边界线

TFTLCD\_DrawHorizontalLine(0,240,128,GREEN);

TFTLCD\_DrawHorizontalLine(0,240,193,GREEN);

TFTLCD\_DrawVerticalLine(64,193,0,GREEN);//显示竖边界线

TFTLCD\_DrawVerticalLine(64,193,239,GREEN);

TFTLCD\_ShowString(12,46,"1",WHITE,BLACK);//显示1和-1

TFTLCD\_ShowString(2,195,"-1",WHITE,BLACK);

ADCSequenceConfigure(ADC0\_BASE, 0, ADC\_TRIGGER\_PROCESSOR, 0);// 配置ADC采集序列

ADCSequenceStepConfigure(ADC0\_BASE, 3, 0, ADC\_CTL\_CH4 | ADC\_CTL\_END | ADC\_CTL\_IE);

ADCSequenceEnable(ADC0\_BASE, 3);// 使能ADC采集序列

ADCIntClear(ADC0\_BASE, 3);//采样前清除采样序列3产生的中断

while(1)

{

ADCProcessorTrigger(ADC0\_BASE, 3);// 触发采集

while(!ADCIntStatus(ADC0\_BASE, 3, false)) ; // 等待采集结束

ADCSequenceDataGet(ADC0\_BASE, 3, &ulADC0\_Value);// 获取采集结果

ulADC0\_Value=ulADC0\_Value/32-45;//转化采集数据

SysCtlDelay(2\*50000000/3000);

UARTprintf("%04d \n ", ulADC0\_Value );

while(ulLastTickCount == g\_ulTickCount)

{}//等待下一个计时器

ulLastTickCount = g\_ulTickCount;

// 准备将新数据点添加到条形图。如果条形图中的项目数达到最大值，

//则数据点需要在缓冲器中“滑下”，从而可以在末尾添加新的数据

if(ulItemCount == SERIES\_LENGTH)

{

FinishCalculateFlag = 1;

}

// 否则，序列数据缓冲区不足，只需增加数据点的计数。

else

{

g\_cSeries[ulItemCount].xAxis = ++ulItemCount;

}

g\_cSeries[ulItemCount - 1].data =ulADC0\_Value;

if(!FinishCalculateFlag)

{//如果我们还没有计算正弦值，继续计算

TFTLCD\_DrawPoint(g\_cSeries[ulItemCount - 1].xAxis,g\_cSeries[ulItemCount - 1].data+110,BLUE);

}

else

{//我们需要更新每个Systick周期的ADC

for(ui32Loop=2;ui32Loop<SERIES\_LENGTH;ui32Loop++)

{ //清除以前的数据

TFTLCD\_DrawPoint(g\_cSeries[ui32Loop-1].xAxis,g\_cSeries[ui32Loop-1].data+110,0);

g\_cSeries[ui32Loop-1].data = g\_cSeries[ui32Loop].data;

//更新数据

TFTLCD\_DrawPoint(g\_cSeries[ui32Loop-1].xAxis,g\_cSeries[ui32Loop-1].data+110,BLUE);

}

//重新绘制线

TFTLCD\_DrawHorizontalLine(0,240,128,GREEN);

TFTLCD\_DrawVerticalLine(64,192,0,GREEN);

TFTLCD\_DrawVerticalLine(64,192,239,GREEN);

}

}

}

void showname(uint32\_t x, uint32\_t y, uint32\_t start, uint32\_t len, uint32\_t fColor, uint32\_t bColor){

FPUEnable();

FPULazyStackingEnable();

//锁相环以120兆赫的频率运行

g\_ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

SysTickPeriodSet(g\_ui32SysClock / TICKS\_PER\_SECOND);

IntMasterEnable();//启用对处理器的中断

SysTickIntEnable();//启用SysTick中断

SysTickEnable();//使能SysTick

ConfigureUART();

EPIGPIOinit();

TFT\_400x240\_OTM4001Ainit(g\_ui32SysClock);//打开背光

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_0);

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_0, GPIO\_PIN\_0);

uint32\_t i;

for(i=0;i<len;i++){

TFTLCD\_ShowName(i\*32+x,y,start+i,fColor,bColor);

}

}

void main()

{

//DrawTriWave();//画三角波

DrawSinWave();//画正弦波

//TFTLCD\_DrawCircle(150,150,50,BLUE);//画圆

//showname(32,32,0,3,WHITE,BLACK);

//Adc();

}

**实验现象：**

**图形用户界面

描述已自动生成**

**五、思考题**

**如何在TFTLCD上画一个滚动的三角波？尝试通过使用TFTLCD\_DrawPoint 函数编写一个画圆函数 FTLCD\_DrawCircle()。**

**三角波**

实验代码：（为减小篇幅，只显示了画三角波的部分）

void DrawTriWave(void)

{//使能FPU，因为某些编译器将使用浮点寄存器，即使对于非浮点代码也是如此。

//如果未启用FPU，则会导致故障。这也确保了浮点操作可以添加到这个应用程序

//中，并且可以正常工作并使用硬件浮点单元。最后，为中断处理程序启用了延迟堆栈。

//这允许在中断处理程序中使用浮点指令，但要牺牲额外的堆栈使用。

uint16\_t ui32Loop = 0,ulItemCount=0,FinishCalculateFlag=0,ulLastTickCount=0;

FPUEnable();

FPULazyStackingEnable();

//锁相环以120兆赫的频率运行

g\_ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

SysTickPeriodSet(g\_ui32SysClock / TICKS\_PER\_SECOND);

IntMasterEnable();//启用对处理器的中断

SysTickIntEnable();//启用SysTick中断

SysTickEnable();//使能SysTick

ConfigureUART();

EPIGPIOinit();

UARTprintf("TFTLCD test\n");

UARTprintf("EPI Type: host-bus 16-bit interface\n");

TFT\_400x240\_OTM4001Ainit(g\_ui32SysClock);//打开背光

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_0);

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_0, GPIO\_PIN\_0);

TFTLCD\_ShowString(50,10,"GC211 TFTLCD test!",CYAN,LIGHTBLUE);//显示GC211 TFTLCD test!

TFTLCD\_ShowString(60,40,"TriWave",GREEN,BLACK);//显示TriWave

TFTLCD\_DrawHorizontalLine(0,240,63,GREEN);//显示横边界线

TFTLCD\_DrawHorizontalLine(0,240,128,GREEN);

TFTLCD\_DrawHorizontalLine(0,240,193,GREEN);

TFTLCD\_DrawVerticalLine(64,193,0,GREEN);//显示竖边界线

TFTLCD\_DrawVerticalLine(64,193,239,GREEN);

TFTLCD\_ShowString(12,46,"1",WHITE,BLACK);

TFTLCD\_ShowString(2,195,"-1",WHITE,BLACK);//显示1和-1

while(1)

{ int count;

int Tlenth=80;

while(ulLastTickCount == g\_ulTickCount)

{}//等待下一个计时器

TFTLCD\_ShowData(170,60,g\_ulTickCount,BLUE,BLACK);

ulLastTickCount = g\_ulTickCount;

// 准备将新数据点添加到条形图。如果条形图中的项目数达到最大值，

//则数据点需要在缓冲器中“滑下”，从而可以在末尾添加新的数据

if(ulItemCount == SERIES\_LENGTH)

{

FinishCalculateFlag = 1;

}

// 否则，序列数据缓冲区不足，只需增加数据点的计数

else

{

g\_cSeries[ulItemCount].xAxis = ++ulItemCount;

}

count=g\_ulTickCount;

while(count>=Tlenth)

{

count-=Tlenth;

}

if(count<Tlenth/2)

g\_cSeries[ulItemCount - 1].data =count;

else if(count<Tlenth)

g\_cSeries[ulItemCount - 1].data =(-1)\*count+Tlenth;

if(!FinishCalculateFlag)

{//如果还没有计算三角波，请继续计算。

TFTLCD\_DrawPoint(g\_cSeries[ulItemCount - 1].xAxis,g\_cSeries[ulItemCount - 1].data+110,RED);

}

else

{//我们需要更新每个系统周期的三角波

for(ui32Loop=2;ui32Loop<SERIES\_LENGTH;ui32Loop++)

{// 清除以前的数据

TFTLCD\_DrawPoint(g\_cSeries[ui32Loop-1].xAxis,g\_cSeries[ui32Loop-1].data+110,0);

g\_cSeries[ui32Loop-1].data = g\_cSeries[ui32Loop].data;

// 更新数据

TFTLCD\_DrawPoint(g\_cSeries[ui32Loop-1].xAxis,g\_cSeries[ui32Loop-1].data+110,RED);

}

//重写显示信息

TFTLCD\_DrawHorizontalLine(0,240,128,GREEN);

TFTLCD\_DrawVerticalLine(64,192,0,GREEN);

TFTLCD\_DrawVerticalLine(64,192,239,GREEN);

}

}

}

实验现象：

**电子器材

中度可信度描述已自动生成**

**圆圈**

实验代码：（为减小篇幅，只显示了画圆的部分）

void TFTLCD\_DrawCircle(uint32\_t x,uint32\_t y,uint32\_t r,uint32\_t color)

{//使能FPU，因为某些编译器将使用浮点寄存器，即使对于非浮点代码也是如此。

//如果未启用FPU，则会导致故障。这也确保了浮点操作可以添加到这个应用程序

//中，并且可以正常工作并使用硬件浮点单元。最后，为中断处理程序启用了延迟堆栈。

//这允许在中断处理程序中使用浮点指令，但要牺牲额外的堆栈使用。

FPUEnable();

FPULazyStackingEnable();

//锁相环以120兆赫的频率运行

g\_ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

SysTickPeriodSet(g\_ui32SysClock / TICKS\_PER\_SECOND);

IntMasterEnable();//启用对处理器的中断

SysTickIntEnable();//启用SysTick中断

SysTickEnable();//使能SysTick

ConfigureUART();

EPIGPIOinit();

UARTprintf("TFTLCD test\n");

UARTprintf("EPI Type: host-bus 16-bit interface\n");

TFT\_400x240\_OTM4001Ainit(g\_ui32SysClock);//打开背光

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_0);

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_0, GPIO\_PIN\_0);\

int num=800;//画圆点数

int i;

float fr;

\_iq24 angle,xi,yi,sin\_angle,cos\_angle;

fr=\_IQ(r);

for(i=0;i<num;i++)

{

angle=\_IQ((float)M\_PI\*(float)i/((float)(num/2)));//计算角度

sin\_angle=\_IQ24sin(angle);//sin值

cos\_angle=\_IQ24cos(angle);//cos值

xi=\_IQ24mpy(cos\_angle,fr);//计算x值

yi=\_IQ24mpy(sin\_angle,fr);//计算y值

TFTLCD\_DrawPoint(x+\_IQint(xi),y+\_IQint(yi),color);//画圆

}

}

实验现象：

电子设备

描述已自动生成

**额外部分：显示自己的名字**

实验代码：（为减小篇幅，只显示了画三角波的部分）

void showname(uint32\_t x, uint32\_t y, uint32\_t start, uint32\_t len, uint32\_t fColor, uint32\_t bColor){

FPUEnable();

FPULazyStackingEnable();

//锁相环以120兆赫的频率运行

g\_ui32SysClock = SysCtlClockFreqSet((SYSCTL\_XTAL\_25MHZ |

SYSCTL\_OSC\_MAIN | SYSCTL\_USE\_PLL |

SYSCTL\_CFG\_VCO\_480), 120000000);

SysTickPeriodSet(g\_ui32SysClock / TICKS\_PER\_SECOND);

IntMasterEnable();//启用对处理器的中断

SysTickIntEnable();//启用SysTick中断

SysTickEnable();//使能SysTick

ConfigureUART();

EPIGPIOinit();

TFT\_400x240\_OTM4001Ainit(g\_ui32SysClock);//打开背光

GPIOPinTypeGPIOOutput(GPIO\_PORTF\_BASE, GPIO\_PIN\_0);

GPIOPinWrite(GPIO\_PORTF\_BASE, GPIO\_PIN\_0, GPIO\_PIN\_0);

uint32\_t i;

for(i=0;i<len;i++){

TFTLCD\_ShowName(i\*32+x,y,start+i,fColor,bColor);

}

}

void TFTLCD\_ShowName(uint32\_t x, uint32\_t y, uint32\_t chr, uint32\_t fColor, uint32\_t bColor){

uint32\_t i,j;

uint8\_t m;

TFTLCD\_SetPos(x,x+32-1,y,y+32-1); //设置字符显示位置

for(i=0; i<128;i++) { //循环写入16字节，一个字符为16字节

m=asc2\_name[chr][i]; //提取c字符的第i个字节

for(j=0;j<8;j++) { //循环写入8位，一个字节为8位

if((m&0x80)==0x80) { //判断最高位是否为1

WriteData(fColor); //最高位为1，写入字符颜色

}

else {

WriteData(bColor); //最高位为0，写入背景颜色

}

m<<=1; //左移1位，准备写下一位

}

}

}

实验现象：

墙上的海报

描述已自动生成

**六、实验体会与心得**

本次实验中，使用了较多的浮点数计算，使用FPU有助于提高运算速度。

FPU是M4内核开始才特有的，有人曾经对比过FPU和CPU的浮点运算速度（使用的是同为M4内核的STM32）。比如float a=1.1,b=1.2,c=1.3,d=1.4;

执行c = b / d，FPU运算474566次，CPU运算64688次，除法速度快了7.3倍多。

执行c = b \* d，FPU运算722169次，CPU运算244271次，乘法运算快了3倍。FPU的乘法运算比除法运算快1.5倍。

由此可见，FPU在浮点运算中有很大的优势。