摘　要

数码相框是时尚的电子消费品，也是家庭必备的装饰品。继承了数码的时尚和传统相框的温情，用途十分广泛。

本文设计的是基于大规模 FPGA 的 BMP 图库管理，完成了数码相框的一部分功能。并且本文详细地介绍了 BMP 图库管理的软硬件实现，即采用 Altera 的 CyclonII 系列 EP2C20F484C7 作为主控芯片，内嵌 32 位的 NiosII 软核，采用 SDRAM 作为内存，把存储在 SD 卡内的二进制图片信息读入内存，并控制 TFT 彩色液晶，读取图片数据送到液晶上显示。整个过程的所有设备都是通过 Avalon 总线挂在 NiosII 上，在 NiosII 的协调下正常工作。

本作品最终能显示存入 SD 卡内的彩色图片信息，图片显示很流畅，没有延时。并且能通过 4 个按键分别完成图片的上翻、下翻、放大和缩小。

关键词：数码相框 BMP；NiosII；SD 卡；TFT 液晶

Abstract

Digital Photo Frame is a stylish electronics for consumer, and it is also essential for household decorations. It inherits the digital fashion and the traditional frames' warmth, which is useing very extensive.

This design is based on the large-scale FPGA-BMP library management, and completed part of the features of digital photo frame. This paper describes the library management software and hardware to achieve BMP photos, that used the Altera's CyclonII series EP2C20F484C7 as the master chip, embedded soft-core 32-bit NiosII, the use of SDRAM for memory, SD card stored the binary picture information read into memory, and control TFT color LCD, read the image form the memory data to the LCD display. All equipment of the process hanging in the NiosII through Avalon bus, with the NiosII CPU and complete the coordination of work.

Eventually the work can show the color pictures of information stored into the SD card, pictures show smoothly, and with no delay. And with 4 keys, respectively, we can make the TFT display the previous image or the next image,and make the pictures zoom in or zoom out.

Key words： Digital photo frames;BMP;NiosII; SD card;TFT

目　录

引言

随着高清像素的数码照相机越来越普及，以及大多数手机都带有摄像摄影功能，人们开始慢慢习惯用各种存储器保存照片。有了具有摄像功能的设备，我们可以非常方便的拍摄自己喜欢的照片，在某个地方想留下自己美好的记忆。这样，拍摄下来的照片数量成几何增长。但如果像我们传统的方法那样都把它冲印出来回味、欣赏，不仅浪费金钱，而且传统的相框、相册也无法再承担起保存的重任。

因此，设计出一个具有照片的保存、回放和浏览的数码相册就会满足人们在数字化时代的特殊要求。并且人们日益感到数码相册的优势，它不会像传统相册那样照片随着时间的推迟会褪色，给人的记忆造成模糊，数码相册数据存储在外部介质中，可以一直存储。另外数码相册还具有图、文、声、像等各种表现手法，修改编辑的功能，快速检索方式，还可以很方便的复制分享。

为了最终用 FPGA 实现数码相册常见的全部功能，本课题完成了一个过渡性设计，即用 FPGA 来读取 SD 卡数据，并在 TFT 中显示。

1 系统方案确定

1.1题目设计任务和要求

BMP 图库管理设计任务：用 FPGA 完成 TFT 的驱动，能在 TFT 上显示彩色图片，并且能够通过按键放大、缩小、上翻、下翻。

BMP 图库管理设计要求：处理器采用 Altera 的 CyclonII 系列 FPGA，输入采用普通键，系统内存和图片显示缓冲采用 SDRAM，图片数据存储采用 SD 卡，图片的显示采用 TFT 真彩色液晶。用 SOPC_Builder 配置生成硬件系统，内部嵌入 NiosII 的 32 位软核处理器，用 NiosII C 编写相应的控制代码。图片信息存入 SD 卡中，SD 卡上移植 FAT32 文件系统方便文件的存取，将图片数据读出来放入缓存，然后再缓存中把数据送到 TFT 彩色液晶上显示。

1.2 系统设计思路简析

主控芯片采用 Altera 的 CyclonII 系列 EP2C20F484C7，该 FPGA 逻辑单元足够用，且性能稳定。在 FPGA 中嵌入 NiosII 软核，软核上挂有 SDRAM、EPCS-FLASH、SD 卡、JTAG 等控制器。另外还挂有 4 个按键，分别实现图片的上翻、下翻、放大、缩小。其中放大和缩小因为没有移植 UC/GUI,只能通过软件算法实现不连贯的放大和缩小。

该系统采用 240*320 的彩色 TFT 液晶,每一个点用 16 位真彩色表示,存储一张这样的彩色图片需要存储空间 240*320*2=153600(Byte),即 15.36K 字节的数据,为了实现大量的图片存储,单靠 FPGA 内部的存储单元是远远不够的。所以必须外扩存储器，当然这里一般选择的是 ROM。另外又因为 FPGA 内部是基于 SDRAM 存储结构的，掉电后数据就消失,所以也必须外扩 ROM。满足图片存储空间的扩展,我们一般可以选择 SD 卡或者 FLASH,但要考虑到存储用户程序，即数据掉电后不消失，一般就选用 FLASH 了，因此本系统中

装用户程序的就是用的 EPCS-FLASH,装图片数据的就是用 SD 卡。

另外由于图片的显示对数据读取要求高，如果单独从 SD 卡读取数据送给 TFT 控制器，则速度上跟不上去，使显示的图片不够顺畅。解决的方法之一是加大系统的主频，这里我们采用的 NiosII 核用了 100Mhz。解决方法之二是采用数据缓存，这里我们采用的是 64M 通用的 SDRAM。显示部分也采用通用的 ILI9325 作为控制 TFT 彩色液晶，通过对 TFT 控制器的初始化后就可以很方便的控制液晶的显示了。

1.3 系统硬件选取

系统硬件以及电路的选取的好坏将直接影响着整个系统实现效果的优劣。本节将介绍整个系统的硬件的选取和理由。

1.3.1中央处理器的选取

方案一：选用 89C51 单片机作为控制器核心。89C51 处理器是一种闪烁可编程可擦除只读存储器的低电压、高性能 CMOS8 位微处理器单片机。其软件编程自由度大，可用编程实现各种控制算法和逻辑控制，是应用规模最大的低价格 MCU，在低端电子产品领域有着广泛的应用。但 89C51 程序运行时串行处理,速度较慢。另外 51 系列单片机内部的 ROM 和 RAM 都非常少。如果选取 89C51，则在价格上可以低很多。

方案二：采用 FPGA 实现，FPGA（Field－Programmable Gate Array），即现场可编程门阵列，它是在 PAL、GAL、CPLD 等可编程器件的基础上进一步发展的产物。其超低功耗和超高的速度受到广大电子工程师的青睐。以硬件描述语言（Verilog 或 VHDL）所完成的电路设计，可以经过简单的综合与布局，快速的烧录至 FPGA 上进行测试; 它不需要编程人员非常熟悉器件内部结构,只要熟悉 Verilog 语言即可对 FPGA 进行操作。Verilog 程序采用并行处理,从现而提高了程序的处理速度,提高了器件的运行速度。另外还可以根据 SOPC 管理器自定义外设，嵌入 NiosII，方便快捷，减少了硬件开发成本和周期。32 位强大的 NiosII 核可以提供堪比 ARM 处理器的性能。此方案可以实现高速、低功耗和低成本的控制要求。此方案中外围电路简单，基本可在 FPGA 内实现，可避免传统设计中因电路老化和电路过于复杂等问题带来的系统不稳定。并且经过验证，NIOS II 软核运行起来非常的稳定。

对比上述两种方案，决定采用 FPGA 来实现本题目。因为如果用 89C51 来显示图片，则显示一幅 320*240 的图片需要 10 多秒。另外，51 单片机内部的 ROM 和 RAM 明显不够，要显示图片还需要外扩这些存储器。所以不管是从速度和功耗上，还是稳定性或者方便后续开发上 FPGA 都占有很大的优势，设计出来的系统大大增强了其稳定性，可支持更多的功能。所以最后决定使用 FPGA 作为该控制系统的核心。另外 FPGA 是一种比较前沿的控制器，拿来做毕设可以帮助对 FPGA 更深入的学习，这也是选用该处理器一个很重要的目的。

1.3.2 外部RAM的选取

方案一：使用FPGA内部的RAM，在Altera公司的FPGA内部嵌入的RAM块有三种，分别是M512RAM（512bit RAM）、M4K（4Kbit）和M-RAM（512Kbit RAM），其中M512主要用于大量分散的数据存储、浅FIFO、移位寄存器、时钟域隔离等功能。M4K通常用作芯片内部数据流的缓存、ATM信元的处理、信元FIFO接口以及CPU的程序存储器等。而M-RAM主要用于在大数据包的缓存（如以太网帧、IP包等大到几K字节的数据包），视频图像帧的缓存，回波抵消（Echo Canceller）数据存储等等。但是由于图片的信息量太大，内部RAM不够用，且因为配置了NiosII软核，就是用了很多FPGA资源，为了使系统达到更好的性能，一般就只剩下了很少用的RAM作为彩色图片显示的缓冲。

方案二：使用外部的SRAM。由于SRAM使用的是6管存储信息，是属于静态RAM，不需要刷新电路即能保存它内部存储的数据，且SRAM存储器的读写速度比较快，一般用在CPU与主存之间的高速缓存，或者CPU内部的L1/L2或外部的L2高速缓存之间，以及CPU外部扩充用的COAST高速缓存。所以是一个很好的选择。但是SRAM也有它的缺点，即它的集成度较低，相同容量的DRAM内存可以设计为较小的体积，但是SRAM却需要很大的体积，集成度低，且功耗较大。所以在主板上SRAM存储器要占用一部分面积集成度太低。且价格也非常昂贵，性价比不是很高。

方案三：采用外部的SDRAM。同步动态随机存储器，同步是指Memory工作需要同步时钟，内部的命令的发送与数据的传输都以它为基准；动态是指存储阵列需要不断的刷新来保证数据不丢失；随机是指数据不是线性依次存储，而是自由指定地址进行数据读写.因为SDRAM是基于电容式存储的，需要不断的刷新以保留数据，操作起来对时序要求比较严。但是其速度也比较快，且容量比SRAM大，价格比SRAM要便宜。且SOPC_Builder中集成了对SDRAM的控制，所以用Nios II核操作起来并不复杂。所以用SDRAM是个比较好的选择,性价比比较高。

通过比较以上三个方案，从速度、价格、容量和操作难易上考虑，选择了SDRAM作为图片显示的缓冲。

1.3.3 外部ROM的选取

方案一：使用FPGA内部的ROM。on-chip-memery就是可以作为FPGA内部的ROM。但是FPGA中其实并没有专用的ROM硬件资源，实现ROM的思路是对上电后RAM赋予初值，也就是用on-chip-memery资源搭成一个RAM或者ROM。但是由于一副图片的信息量太大，通过上述对外部RAM选取的方案一分析可知FPGA内部RAM存储图片很吃力，且因为配置了NiosII软核，就是用了很多FPGA资源，为了使系统达到更好的性能，一般很少用内部的ROM作为彩色图片数据的存储。

方案二：使用外部的FLASH闪存。闪存的英文名称是"Flash Memory"，一般简称为"Flash"，它属于内存器件的一种，是一种不挥发性（ Non-Volatile ）内存，在没有

电流供应的条件下也能够长久地保持数据，其存储特性相当于硬盘，这项特性使闪存被广泛用于移动存储、数码相机、MP3 播放器、掌上电脑等新兴数字设备中。由于受到数码设备强劲发展的带动，FLASH 闪存一直呈现指数级的超高速增长.。并且其能提供极高的单元密度,可以达到高存储密度,并且写入和擦除的速度也很快。是个不错的选择。

方案三：采用外部的 SD 卡。SD 卡（Secure Digital Memory Card）中文翻译为安全数码卡，是一种基于半导体快闪记忆器的新一代记忆设备，它被广泛地于便携式装置上使用，例如数码相机、个人数码助理(PDA)和多媒体播放器。其拥有高记忆容量、快速数据传输率、极大的移动灵活性以及很好的安全性。SD 卡共支持三种传输模式：SPI 模式，1 位的 SD 模式，4 位的 SD 模式。并且 SD 卡的操作一般可以涉及到文件系统，一旦移植上了文件系统，则 SD 卡上文件的读取就变得非常方便。因此用来存储文件是个不错的选择。

通过比较以上三个方案，从速度、价格、容量、稳定性和操作难易上考虑，选择了方案二中的 FLASH 作为系统的用户程序代码，并且采用的是 EPCS 串行 FLASH。另外因为图片数据非常大，选取了方案三种的 SD 卡作为外部图片数据的存储。

1.3.4 输入设备的选取

方案一：采用常见的电脑键盘。电脑键盘一般是 104 个按键的，作为日常接触最多的输入设备，手感是非常好的，大家用起来也非常的熟悉和方便。在高速敲击时也只产生较小的噪音，不影响到别人休息和工作。其程序设计简单，易于控制。而且按键非常多，可以实现很多功能。因此，无论在个人 PC，还是在工控行业，104 键盘都得到了广泛的使用。

方案二：采用自制的矩阵键盘。因为自制的矩阵键盘价格便宜，硬件电路简单，且编程很容易，在实现较少功能的前提下是个不错的选择，但是它的使用会占用 CPU 处理的时间。

通过考虑以上 2 个方案，本设计因为只需上下翻页和放大缩小，根本没有必要用到电脑的键盘，直接用自制的 4 个独立的按键就可以了，因此采用了方案二。

1.3.5 输出设备的选取

方案一：采用一般的点阵液晶显示器。点阵液晶显示器功耗低、体积小、操作方便，因此倍受工程师青睐，适用于使用电池的电子设备。普通的比如 LCD1602 或 LCD12864，其一般只能显示字符，要显示图片需要在程序上下很多功夫，占用 CPU 时间太多。并且其控制器内部功能不是很强，要显示高真彩色的图片还是有一定的困难。

方案二：采用 VGA 液晶显示器。常用于电脑显示的 VGA 液晶显示器为平面超薄的显示设备，它由一定数量的彩色或黑白像素组成，放置于光源或者反射面前方。功耗很低，它的主要原理是以电流刺激液晶分子产生点、线、面配合背部灯管构成画面。其驱动极其简单，只要 5 根线便可。VGA 液晶显示器能达到 262144 种颜色的色版，其显示的颜色接近于自然色，因此得到了广泛的使用。但是其体积太大，很少用于嵌入式系统。不

利于便携式处理数据。

方案三：采用 TFT 液晶显示。TFT（ThinFilmTransistor）是指薄膜晶体管，意即每个液晶像素点都是由集成在像素点后面的薄膜晶体管来驱动，从而可以做到高速度、高亮度、高对比度显示屏幕信息，是目前最好的 LCD 彩色显示设备之一，其效果接近 CRT 显示器 TFT 液晶比较小，TFT 的每个像素点都是由集成在自身上的 TFT 来控制，是有源像素点。因此，不但速度可以极大提高，而且对比度和亮度也大大提高了，同时分辨率也达到了很高水平。且容易便携，内部含有控制器，功能强大，可以显示彩色图片。

较以上三个方案，本设计决定采用 TFT 液晶显示器，可使显示的质量大大提高，且其较小的体积很容易便携，且显示真彩色的图片一点都不吃力。因此最终选择方案三。

2 系统硬件设计

在本章中先介绍系统总体设计框图，然后分别介绍各硬件的原理图，硬件芯片的具体参数介绍。

2.1 系统总体设计框图及说明

本系统采用 FPGA 内嵌的 NiosII 核，外挂 JTAG，SDRAM，SD 卡，TFT 和 SD 卡的控制。系统总体框图如图 2.1 所示：

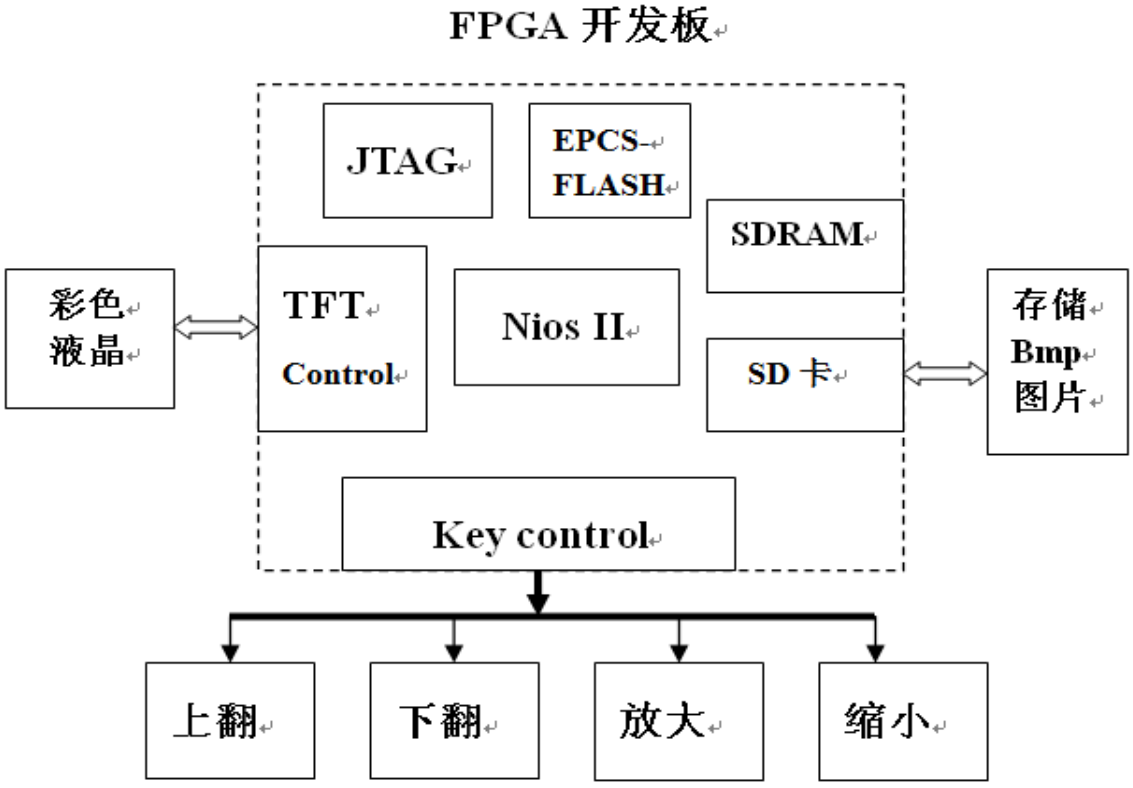


图 2.1 系统总体框图

2.2 中央处理器

2.2.1 CyclonII 系列 FPGA 介绍

Altera FPGA Cyclone II 系列 EP2C20F484C7 实物图如图 2.2 所示：•

图 2.2 Cyclone II EP2C20F484C7 实物图

FPGA（Field－Programmable Gate Array），即现场可编程门阵列，是在 PAL、GAL、CPLD 等可编程器件的基础上进一步发展的产物。它是作为专用集成电路（ASIC）领域中的一种半定制电路而出现的，不但解决了定制电路的不足，而且又克服了原有可编程器件门电路逻辑资源有限的缺点。它就相当于一个“万能芯片”，如果其逻辑资源够大的话，可以实现任何复杂的数字电路。它是集成电路发展的一个里程碑，可以使我们缩短了很多开发周期，并且现在 FPGA 内部资源越来越多，且价格越来越便宜，在市场上越来越受到 IT 行业厂商的青睐。

这种新的器件比第一代 Cyclone 产品具有两倍多的 I/O 引脚，且对可编程逻辑，存储块和其它特性进行了最优的组合，具有许多新的增强特性：

⑴Nios II 嵌入式软处理器

Altera 最近推出的 Nios II 系列软核处理器支持 Cyclone II FPGA 系列。在 Cyclone II FPGA 中实现 Nios II 的设计除了大幅度降低实现成本之外，还具有 100DMIP 的性能，大约比 Cyclone 器件和 Nios 处理器提升了 100%。设计者使用 Nios II 处理器，能够在任何一个 Cyclone II 器件上构建完整的可编程系统芯片(SOPC)，是中低规模 ASIC 的新的替代方案。

⑵低成本

Altera 为配置 Cyclone II FPGA 提供了低成本的串行配置器件。这些串行配置器件定价为批量应用，成本是相应 Cyclone II FPGA 的 10%。四个串行配置器件(1Mbit，4Mbit，16Mbit 和 64Mbit)提供了节省空间的 8 脚和 16 脚 SOIC 封装。器件中任何不用于配置的存储器可用于一般存储，进一步增强其价值。

⑶IP 核丰富

Altera 也为 Cylcone II 器件客户提供了 40 多个可定制 IP 核，Altera 和 Altera Megafunction 伙伴计划(AMPPSM)合作者提供的不同的 IP 核是专为 Cyclone II 架构优化的，包括：DDR SDRAM 控制器；PCI 编译器；Nios II 嵌入式处理器；FIR 编译器；NCO 编译器；FFT/IFFT； POS-PHY 编译器；Reed Solomon 编译器；Viterbi 编译器等等。当客户完全满意 IP 功能，再购买完全的许可。

2.2.2 FPGA 的工作原理

FPGA 采用了逻辑单元阵列 LCA（Logic Cell Array）这样一个概念，内部包括可配置逻辑模块 CLB（Configurable Logic Block）、输出输入模块 IOB（Input Output Block）和内部连线（Interconnect）三个部分。

（1）FPGA 的基本特点

①采用 FPGA 设计 ASIC 电路(特定用途集成电路)，用户不需要投片生产，就能得到合用的芯片；

②FPGA 可做其它全定制或半定制 ASIC 电路的中试样片；

③FPGA 内部有丰富的触发器和 I／O 引脚；

④FPGA 是 ASIC 电路中设计周期最短、开发费用最低、风险最小的器件之一；

⑤FPGA 采用高速 CHMOS 工艺，功耗低，可以与 CMOS、TTL 电平兼容。

FPGA 是由存放在片内 RAM 中的程序来设置其工作状态的，因此，工作时需要对片内的 RAM 进行编程。用户可以根据不同的配置模式，采用不同的编程方式。

可以说，FPGA 芯片是小批量系统提高系统集成度、可靠性的最佳选择之一。

掉电后，FPGA 恢复成白片，内部逻辑关系消失。加电时，FPGA 芯片将 EPROM 中数据读入片内编程 RAM 中，配置完成后，FPGA 进入工作状态。因此，FPGA 能够反复使用。FPGA 的编程只须用通用的 EPROM、PROM 编程器即可，无须专用的 FPGA 编程器。当需要修改 FPGA 功能时，只需换一片 EPROM 即可。这样，同一片 FPGA，可以产生不同的电路功能。可见，FPGA 的使用非常灵活。

（2）FPGA 配置模式

FPGA 有多种配置模式：串行模式可以采用串行 PROM 编程 FPGA；并行主模式为一片 FPGA 加一片 EPROM 的方式；主从模式可以支持一片 PROM 编程多片 FPGA；外设模式可以将 FPGA 作为微处理器的外设，由微处理器对其编程。

最近 FPGA 的配置方式已经多元化。

（3）FPGA 设计的注意事项

①I/O 信号分配

②信号完整性

③差分信号

④建立时间与保持时间

（4）FPGA 的应用

①电路设计中 FPGA 的应用

连接逻辑，控制逻辑是 FPGA 早期发挥作用比较大的领域也是 FPGA 应用的基石．事实上在电路设计中应用 FPGA 的难度还是比较大的这要求开发者要具备相应的硬件知识（电路知识）和软件应用能力（开发工具）这方面的人才总是紧缺的，往往都从事新技术，新产品的开发成功的产品将变成市场主流基础产品供产品设计者应用在不远的将来，通用和专用 IP 的设计将成为一个热门行业！搞电路设计的前提是必须要具备一定的硬件知识。

②产品设计

把相对成熟的技术应用到某些特定领域如通讯，视频，信息处理等等开发出满足行业需要并能被行业客户接受的产品这方面主要是 FPGA 技术和专业技术的结合问题，另外还有就是与专业客户的界面问题产品设计还包括专业工具类产品及民用产品，前者重点在性能，后者对价格敏感产品设计以实现产品功能为主要目的，FPGA 技术是一个实现手段在这个领域，FPGA 因为具备接口，控制，功能 IP，内嵌 CPU 等特点有条件实现一个构造简单，固化程度高，功能全面的系统产品设计将是 FPGA 技术应用最广大的市场，具有极大的爆发性的需求空间产品设计对技术人员的要求比较高，路途也比较漫长不过现在整个行业正处在组建＂首发团队＂的状态，只要加入，前途光明产品设计是一种职业发展方向定位，不是简单的爱好就能做到的！产品设计领域会造就大量的企业和企业家，是一个近期的发展热点和机遇 。

③系统级应用

系统级的应用是与传统的计算机技术结合，实现一种计算机系统如用 Xilinx V-5 系列的 FPGA，实现内嵌 POWER CPU, 然后再配合各种外围功能，实现一个基本环境，在这个平台上跑 LINUX 等系统这个系统也就支持各种标准外设和功能接口（如图象接口）了这对于快速构成 FPGA 大型系统来讲是很有帮助的。这种＂山寨＂味很浓的系统早期优势不很明显，类似 ARM 系统的境况但若能慢慢发挥出 FPGA 的优势，逐渐实现一些特色系统也是一种发展方向。若在系统级应用中，开发人员不具备系统的扩充开发能力，只是搞搞编程是没什么意义的，当然设备驱动程序的开发是另一种情况，搞系统级应用看似起点高，但不具备深层开发能力，很可能会变成爱好者，就如很多人会做网页但不能称做会编程类似以上是几点个人开发。这是一个不错的行业，有很好的个人成功机会。但也肯定是一个竞争很激烈的行业，关键看的就是深度和速度当然还有市场适应能力。

2.2.3 FPGA 硬件系统电路设计

本作品采用的是 Cyclon II 系列中的 EPCS20，属于 EPCS20 家族中的 EPCS20F484C7。其逻辑单元有 18742 个，内部嵌有 52 个 M4K 的 RAM，总共 RAM 的位数达到 239626 位。有 18 位*18 位的嵌入式乘法器 26 个。4 个 PLL，最大管脚数达到 315 个。芯片采用的 FBGA 封装。

由于其引脚太多，采用的是 FBGA 封装，所以画最小系统原理图时一定要小心，不能把引脚弄错了。

FPGA 系统的 EP2C20F484C7 芯片部分的原理图如图 2.3 所示：

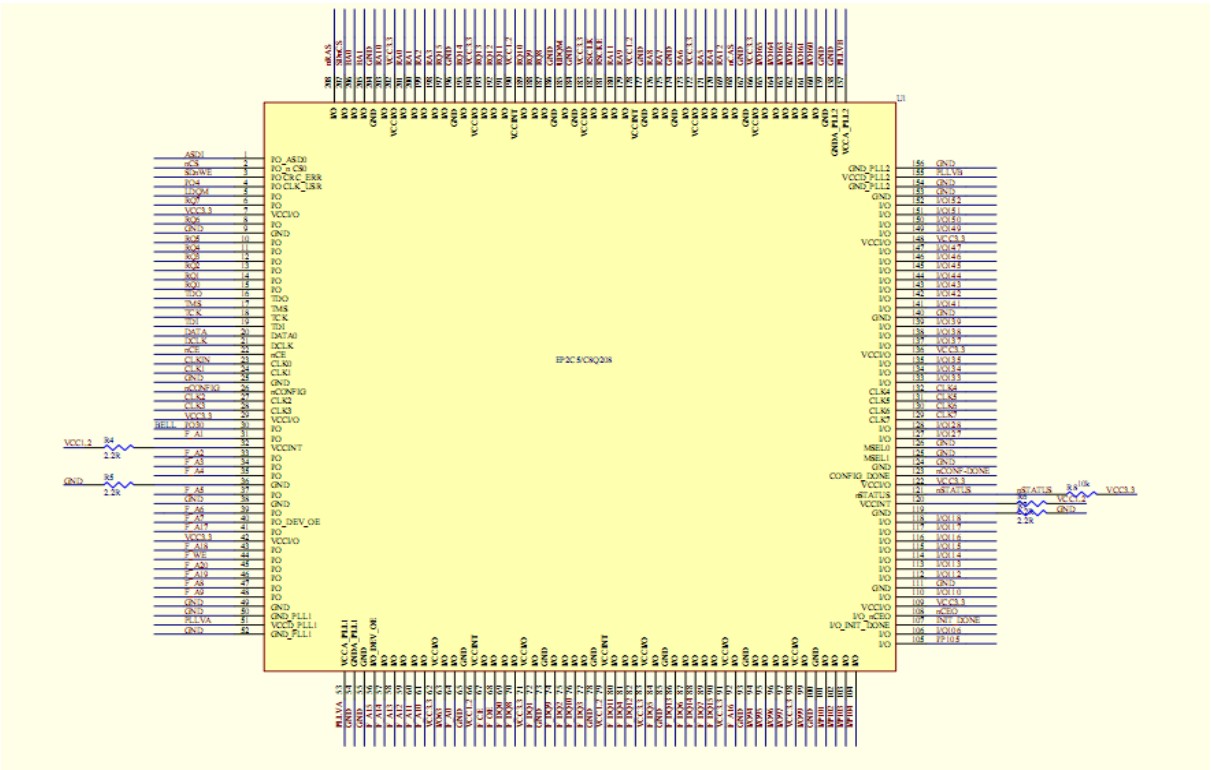


图 2.3 FPGA 系统原理图

2.3 SOPC_Builder

2.3.1 SOPC_Builder 介绍

在二十世纪九十年代末，可编程逻辑器件（PLD）的复杂度已经能够在单个可编程器件内实现整个系统，完整的单芯片系统（SOC）概念是指在一个芯片中实现用户定义的系统。在一个 SOC 设计中，将涵盖到包括微处理器、DSP 芯片、存储器件、I/O、控制逻辑、混合信号模块(Mixed-Signal Blocks)等在内的许多部分。

在系统设计复杂度不断的提高及新产品市场周期不断缩短的压力下， 把 FPGA 及微处理器的核心内嵌在同一芯片上，构建成为一个可编程的 SOC 系统体系框架结构，建成所谓的可编程芯片系统 SOPC(System on a Programmable Chip)，从而为系统设计者提供了又一灵活快捷的设计方法与途径。

SOPC（System-on-a-Programmable-Chip）即可编程片上系统，用可编程逻辑技术把整个系统放到一块硅片上，称作 SOPC。可编程片上系统（SOPC）是一种特殊的嵌入式系统：首先它是片上系统（SOC），即由单个芯片完成整个系统的主要逻辑功能；其次，

它是可编程系统，具有灵活的设计方式，可裁减、可扩充、可升级，并具备软硬件在系统可编程的功能。因此，著名的可编程逻辑器件生产厂家美国 Altera 公司提出了基于 PLD 的 SOC 设计方案——SOPC。

近年来 SOPC 技术已成为备受众多中小企业、研究所和大学院校青睐的设计技术。SOPC（System on a Programmable Chip）成为可编程片上系统，是 Altera 公司提出的一种灵活、高校的 SOC 解决方案，是一种新的软硬件协同设计的系统设计技术。SOPC 集成了硬核或软核 CPU、DSP、锁相环（PLL）、存储器、I/O 接口及可编程逻辑，可以灵活高效地解决 SOC 方案，而且设计周期短，设计成本低。Nios II 是一种软核（Soft-Core）处理器，软核处理器最大的特点就是可由用户需要进行设置。与专用 CPU 不同的是，Nios II 是一个用户可以自行定制的 CPU，用户可以增加新的外设、新的指令，分配外设的地址等。Nios II 的硬件开发就是由用户制定适合的 CPU 外设，Altera 公司的 SOPC Builder 提供了大量的 IP Core 来加快 Nios II 外设的开发速度。综合来看，SOPC 是 PLD 和 ASIC 技术融合的结果，可以认为 SOPC 代表了半导体产业未来的发展方向，对 SOPC 进行深入的研究不仅有利于半导体产业的发展，同时对微电子技术和计算机技术的发展也具有重要的意义。

SOPC 技术主要应用以下三个方向：

(1)基于 FPGA 嵌入 IP 硬核的应用。这种 SOPC 系统是指在 FPGA 中预先植入处理器。这使得 FPGA 灵活的硬件设计与处理器的强大软件功能有机地结合在一起，高效地实现 SOPC 系统。

(2)基于 FPGA 嵌入 IP 软核的应用。这种 SOPC 系统是指在 FPGA 中植入软核处理器，如：NIOS II 核等。用户可以根据设计的要求，利用相应的 EDA 工具，对 NIOS II 及其外围设备进行构建，使该嵌入式系统在硬件结构、功能特点、资源占用等方面全面满足用户系统设计的要求。

(3)基于 HardCopy 技术的应用。这种 SOPC 系统是指将成功实现于 FPGA 器件上的 SOPC 系统通过特定的技术直接向 ASIC 转化。把大容量 FPGA 的灵活性和 ASIC 的市场优势结合起来，实现对于有较大批量要求并对成本敏感的电子产品，避开了直接设计 ASIC 的困难。

2.3.2 SOPC_Builder 生成 BMP 管理系统

首先来看看本系统经过 SOPC_Builder 配置后的界面，如图 2.4 所示：

Use	Conne...	Module Name	Description	Clock	Base	End	Tags	IRQ
☑		sdram	SDRAM Controller					
		s1	Avalon Memory Mapped Slave	clk_0	0x02000000	0x027fffff		
☑		LCD_DATAH	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x04001050	0x0400105f		
☑		LCD_DATAL	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x04001060	0x0400106f		
☑		di_spi	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x04001070	0x0400107f		
☑		do_spi	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x04001080	0x0400108f		
☑		clk_spi	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x04001090	0x0400109f		
☑		cs_spi	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x040010a0	0x040010af		
☑		KEY_NEXT	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x040010b0	0x040010bf		
☑		KEY_UPPER	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x00000000	0x0000000f		
☑		KEY_BIG	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x00000010	0x0000001f		
☑		KEY_SMALL	PIO (Parallel I/O)					
		s1	Avalon Memory Mapped Slave	clk_0	0x00000020	0x0000002f		

图 2.4 SOPC_Builder 配置系统后的界面

下面就介绍怎样用 SOPC_Builder 配置成如图 4 的步骤一一介绍。

1.配置 CPU 过程

点击打开左侧 System Contents 目录，找到 Component Library，如下图 2.5 所示。

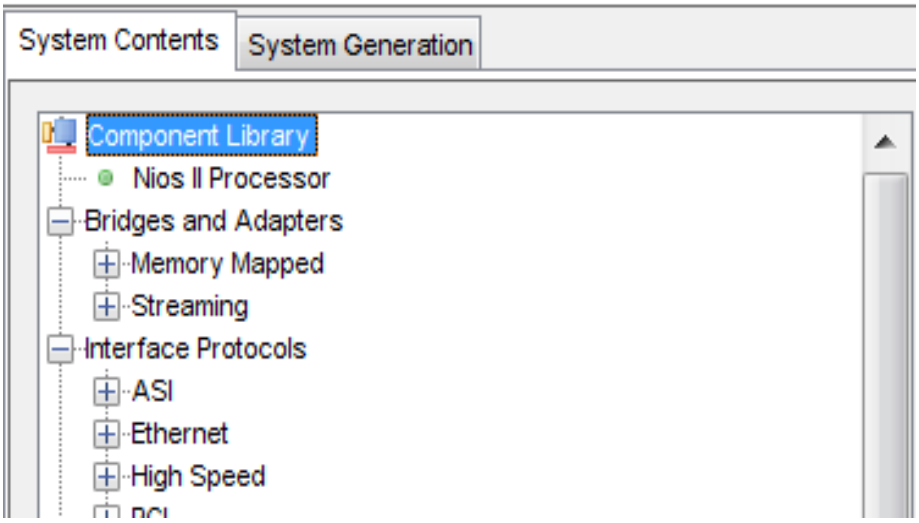


图 2.5 SOPC_Builde 配置 CPU 步骤 1

双击 NiosII Processor 后将 CPU 各项配置成如图 2.6 所示的参数。 NiosII 处理器因为在速度和占用逻辑资源上分为三种等级，分别为 e 型、s 型、f 型。为了兼顾这两者，这里选用的是 NiosII 的 s 型。配置好这个后，就点击 finish。

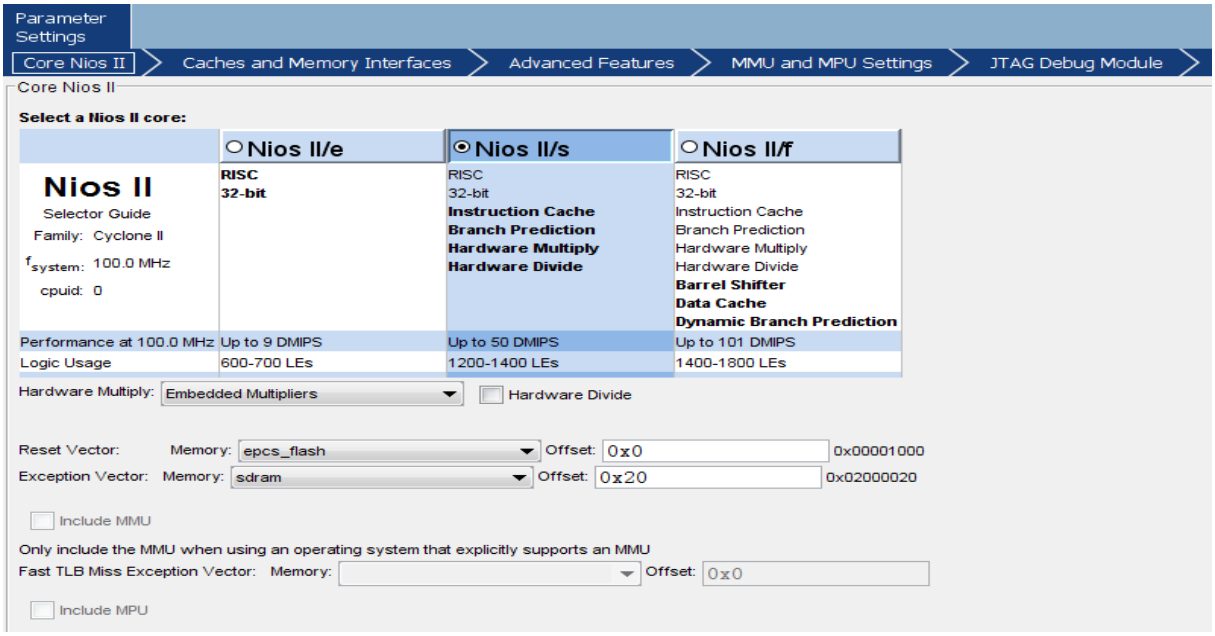


图 2.6 SOPC_Builde 配置 CPU 步骤 2

注意其中的 Reset Vector 地址和 Exception Vector 地址要在配置系统的 EPCS-FLASH 和 SDRAM 后才能选择。

2.配置液晶的控制端口 LCD_RESET,LCD_RD,LCD_CS,LCD_WR,LCD_RS 和 SD 卡的控制端口 di_spi,do_spi,clk_spi,cs_spi。

因为它们的都是 1bit 的输出或输入脚。其配置图 2.7 所示。端口的 Width 选为 1，端口的 Direction 相应的选为 Output ports only 或者 Input ports only。

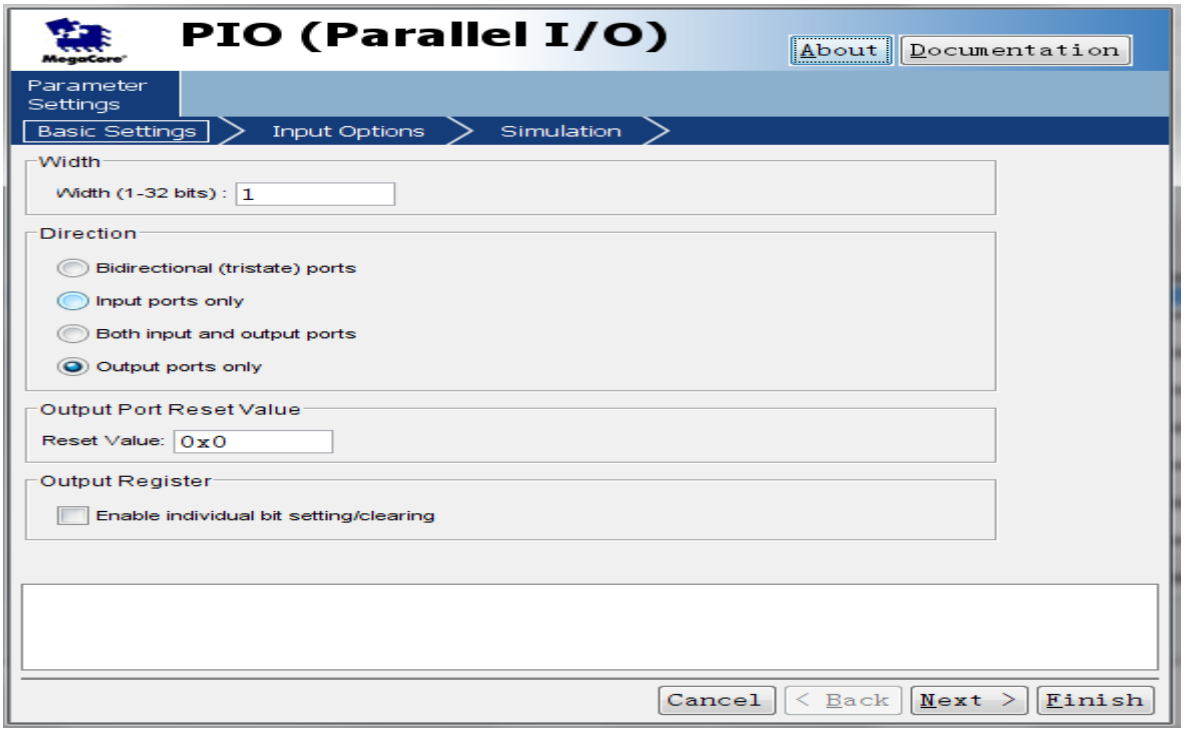


图 2.7 TFT 液晶和 SD 卡控制口的配置

LCD_DATAH 和 LCD_DATAL 都是输出 8 位的，其配置数据图 2.8 所示。Width 选 8，Direction 选为 Output ports only。

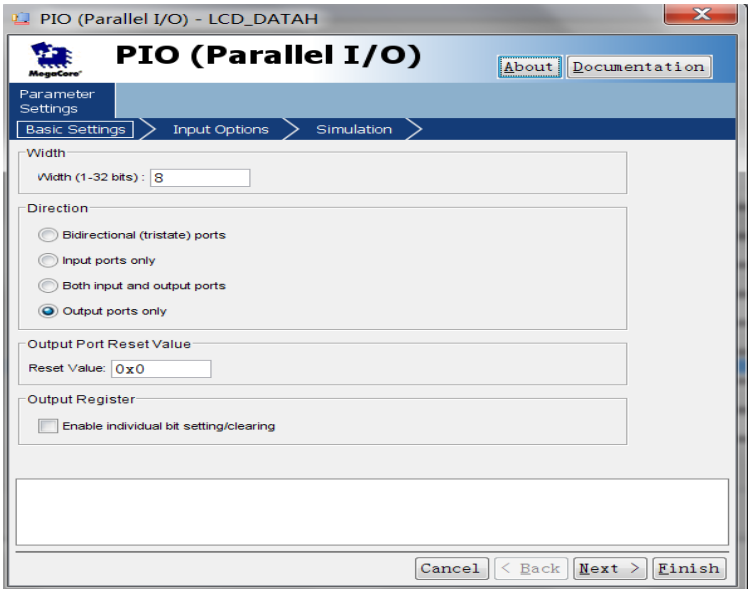


图 2.8 TFT 数据口的配置

3.配置 SDRAM。

系统采用的 SDRAM 是 64Mbit 的，数据位为 16 位，行地址线 12 根，列地址线 8 根，有 4 个逻辑 bank。其配置图形如图 2.9 所示：

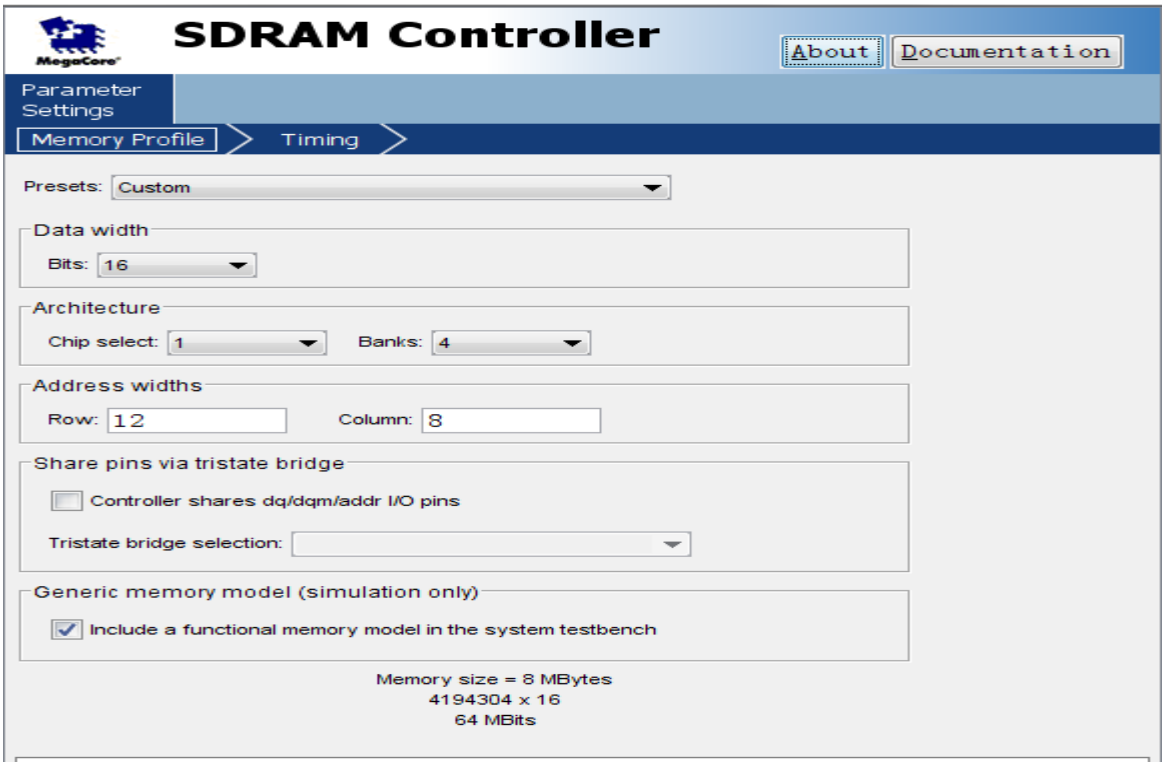


图 2.9　SDRAM 的配置

4.配置 JTAG 控制器。

为了方便系统的调试，在 NiosII 上配置了 JTAG 控制器，这样可以在调试程序的过程中不断的向 NiosIDE 的控制台打印信息，可以直观的看到程序运行的状态，为最程序的成功提供了不少方便，如图 2.10 所示：

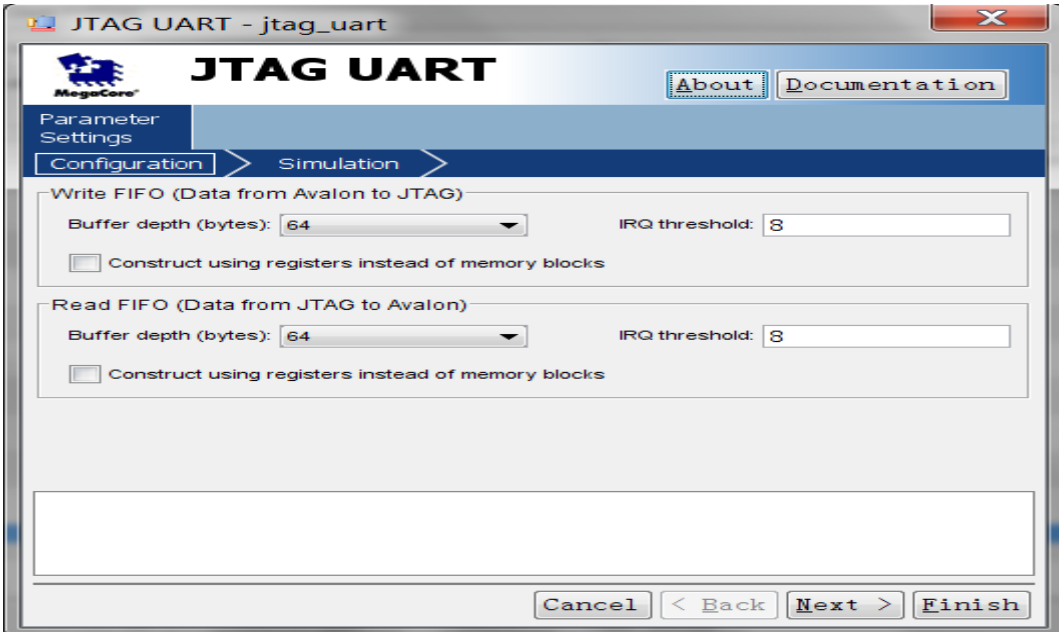


图 2.10　JTAG 的配置

4. 按键部分。

因为系统有 4 个按键，分别为 KEY_UPPER, KEY_NEXT, KEY_BIG, KEY_SMALL。其都是输入，配置如图 2.11 所示：

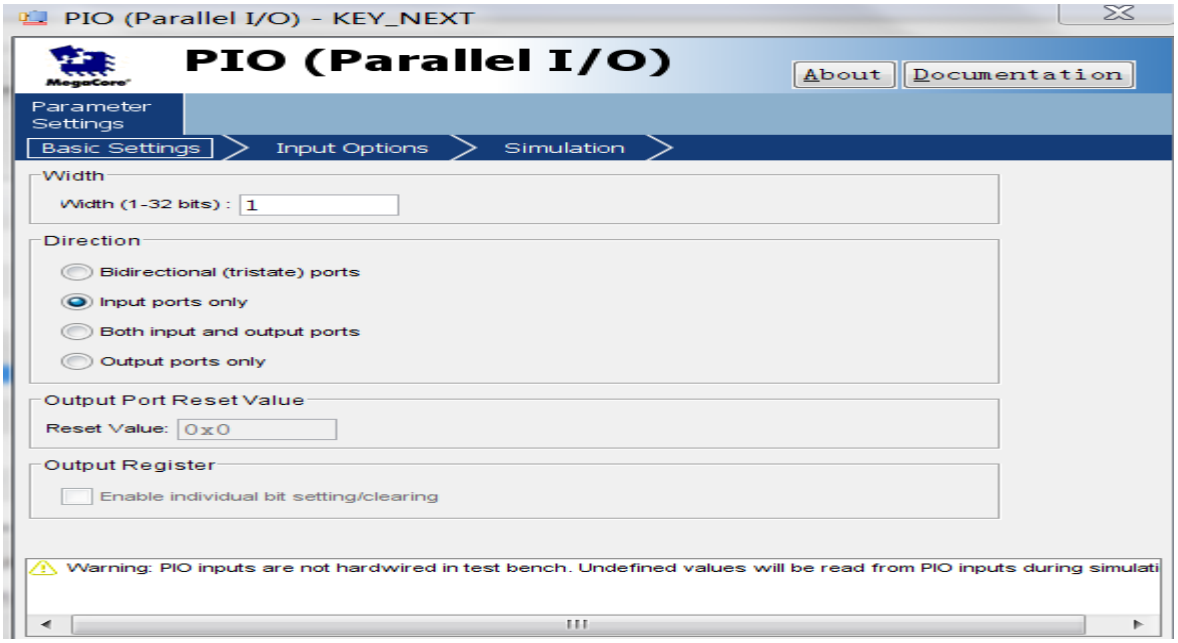


图 2.11　按键输入部分的配置

最后，为了时系统在运行时检测版本是否正确，添加一个 sysid，在 System Contens 里面找到 sysid，双击后点击 Finish 即可。如果 2.12 所示：

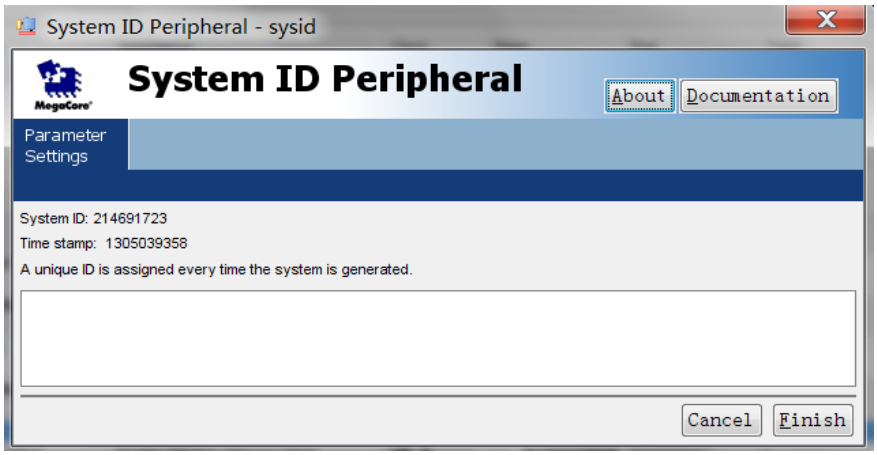


图 2.12 sysid 的配置

2.3.3 SOPC_Builder 编译整个系统

模块的库来生成系统。系统 PTF 文件不仅保存了用户定义的信息，而且也保存了 SOPC Builder 临时的内部结果。例如，系统生成程序的开始部分会向系统 PTF 文件写入数据，这些数据可能被系统生成程序的后续部分所使用。换句话说，系统 PTF 文件中的数据不全是根据用户输入产生的。

SOPC 动搜索已安装的 IP 模块。SOPC Builder 主窗口中的左侧模块池内显示了所有发现的 IP 模块列表。SOPC Builder 通过在一个搜索路径中的所有目录下搜索名为 class.ptf 的文件来获得 IP 模块列表。一般来说，和 IP 模块相关的所有文 C Builder 图形用户界面启动时，会自件都放置在 IP 模块的 class.ptf 文件所在的目录或其子目录下。

SOPC Builder 对于搜索到的 class.ptf 文件，要确定其是否包含一个有效的、语法正确的 IP 模块描述。class.ptf 文件至少要包含足够的信息以便 IP 模块能显示在图形用户界面的模块池内，模块池内显示的 IP 模块与搜索到的 class.ptf 文件是一一对应的。

SOPC_Builder 的编译流程如图 2.13 所示：

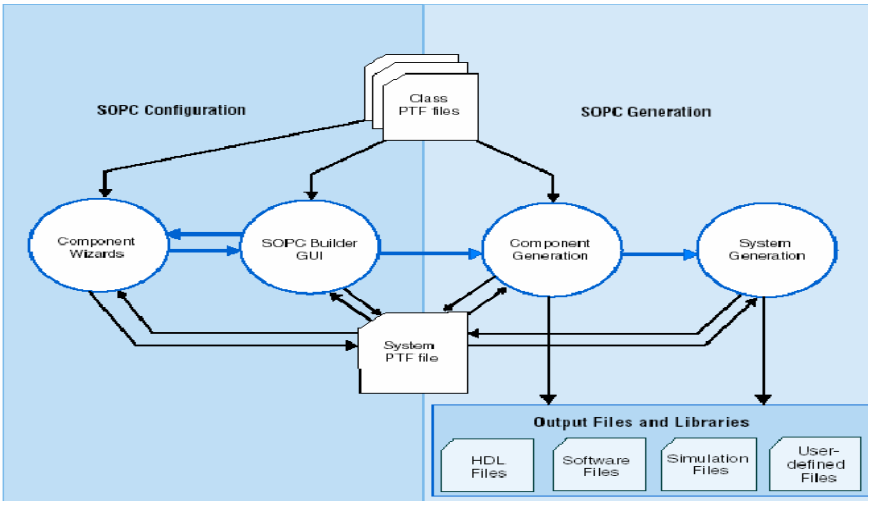


图 2.13 SOPC_Builder 编译流程

2.4 按键输入设计

此系统只有 4 个输入，上翻、下翻、放大、缩小，所以用到的按键很少，这里系统只采用 4 个 IO 口来接 4 个按键接地，并且每个按键都接 1 个 100 看的上拉电阻，最后经一个总线驱动器 74HC245。

其按键部分原理图如图 2.14 所示：

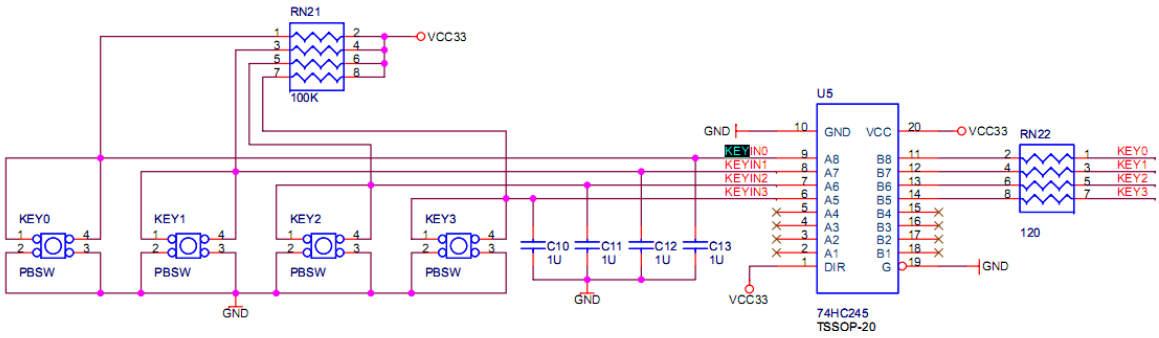


图 2.14 按键输入部分的原理图

2.5 SDRAM 电路设计

SDRAM 的型号为 IS42S16400，64Mbit,作为系统的外扩内存。其在电路中的芯片引脚图如图 2.15 所示：

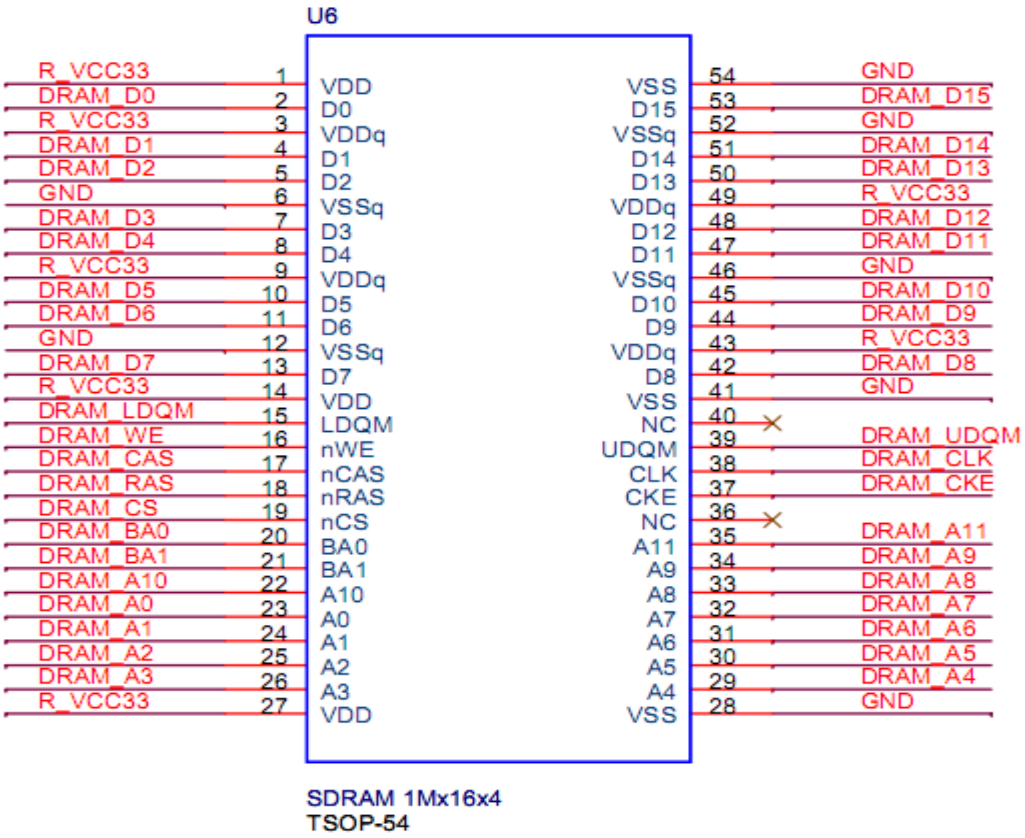


图 2.15 SDRAM 原理图

2.6 SD 卡

2.6.1 SD 卡介绍

SD 卡缩写为 Secure Digital，作为一种存储卡，全名应该是 Secure Digital Memory Card，中文翻译为安全数码卡或直接称为 SD 卡，是一种存储卡的标准，它被广泛地于便携式设备上使用，例如数字相机、个人数码助理（PDA）和多媒体播放器等。SD 卡的技术建是基于 MultiMedia 卡（MMC）格式上，但 SD 卡比 MMC 卡略厚。而 SD 卡也有较高的数据传送速度，而且不断地更新标准。大部份 SD 卡的侧面设有写保护控制，以避免一些数据意外地写入，而少部分的 SD 卡甚至支持数字版权管理（DRM）的技术。一般 SD 卡的大小约为 32mm × 24mm × 2.1mm，但可以薄至 1.4mm，与 MMC 卡相同。

SD 卡提供不同的速度，它是按 CD-ROM 的 150 KB/s 为 1 倍速（记作“1x”）的速率计算方法来计算的。基本上，它们能够比标准 CD-ROM 的传输速度快 6 倍（900KB/s），而高速的 SD 卡更能传输 66x（9900KB/s=9.66MB/s，标记为 10MB/s）以及 133x 或更高的速度。一些数字相机需要高速 SD 卡来更流畅地拍摄影片，以及使得相片连拍更为迅速。直至 2005 年 12 月，大部分设备跟从 SD 卡的 1.01 规格，而更高速至 133x 的设备亦跟从 1.1 规格。

设有 SD 卡插槽的设备能够使用较簿身的 MMC 卡，但是标准的 SD 卡却不能插入到 MMC 卡插槽。插上转接器后 SD 卡能够用于 CF 卡和 PCMCIA 卡上，；而 miniSD 卡和 microSD 卡亦能插上转接器在 SD 卡插槽使用。一些 USB 连接器能够插上 SD 卡，而且一些读卡器亦能够插上 SD 卡，并由许多连接端口，例如 USB、FireWire 等访问使用。

SD 卡应用于以下的手提数字设备：

数字相机存储相片及短片

数字摄录机存储相片及短片

个人数码助理（PDA）存储各类数据

手提电话存储相片、铃声、音乐、短片等数据

多媒体播放器

在 2006 年，SD 卡容量有 8/16/32/64/128/256/512MB，1GB/2GB，超过 2GB 容量的卡称为 SDHC(注：也有 4G 的普通 sd 卡)，是 SD 的升级版本。新一代 SDHC 2.0（SDXC）标准规范为 SD 卡的下一代标准，最大容量可高达 2TB。

SD/MMC 卡已经替代东芝开发的 SM 卡，成为了便携式数码相机使用最广泛的数字存储卡格式。2001 年 SM 卡的市场占有率超过 50%，但到了 2005 年下降到了 40%左右，并且还在快速滑落。大部分的数码相机生产商都提供了 SD 卡的支持，包括佳能、尼康、柯达、松下及柯尼卡美能达等。

三大主要厂商仍然在坚持使用自己的专利格式：奥林巴斯和富士使用 xD 卡，索尼使用 Memory Stick。

SD 卡是东芝在 MMC 卡技术中加入加密技术硬件而成，由于 MMC 卡可能会较易让用户复制数字音乐，东芝便加入这些技术希望令音乐业界安心。类似的技术包括索尼的 MagicGate，理论上加密技术可引入一些数字版权管理措施，但这功能甚少被应用。

用户可以使用一个 USB 的读卡器，在个人电脑上使用 SD 卡。某些新型电脑上已经内置了读卡装置。

最新的发展是 SD 内建了 USB 插口，省略了读卡器。晟碟的设计是使用一个可折叠的护套来保护 USB 插口。尽管 Sandisk 并不是第一家内建 USB 功能的 SD 卡生产商，但由于其在业内的重要地位。这一动作带动了其他厂商跟风。

“SD”商标实际上是用于另一个完全不同的用途：它最早是用在“超级密度光盘”上（Super-Density Optical Disk），这个由东芝开发的产品在 DVD 格式之争中败北。这就是为什么那个"D"字看起来像一张光盘。

SD 系列存储卡都是晟碟完成测试后送交 SD 卡协会认证规格，因此几乎所有专利权都掌控在晟碟手上。

SD 卡的性能指标

◎容量：32MB/64MB/128MB/256MB/512MB/1GByte

◎兼容规范版本 1.01

◎卡上错误校正

◎支持 CPRM

◎两个可选的通信协议：SD 模式和 SPI 模式

◎可变时钟频率 0－25MHz

◎通信电压范围：2.0-3.6V

◎工作电压范围:2.0-3.6V

◎低电压消耗：自动断电及自动睡醒，智能电源管理

◎无需额外编程电压

◎卡片带电插拔保护

◎正向兼容 MMC 卡

◎高速串行接口带随即存取

－－－支持双通道闪存交叉存取

－－－快写技术：一个低成本的方案，能够超高速闪存访问和高可靠数据存储

－－－最大读写速率：10Mbyte/s

◎最大 10 个堆叠的卡（20MHz,Vcc=2.7-3.6V）

◎数据寿命：10 万次编程/擦除

◎CE 和 FCC 认证

◎PIP 封装技术

◎尺寸：24mm 宽×32mm 长×1.44mm 厚

2.6.2 SD 卡电路设计

系统中的 SD 卡采用的是 SPI 模式，只用到了 4 个端口，片选、时钟、命令、数据。其中的片选端采用 SD 卡的数据口 Data3，SD 卡是采用 3.3V 供电的，命令口和数据口经过 4.7K 的电阻上拉。其电路原理图如图 2.16 所示：

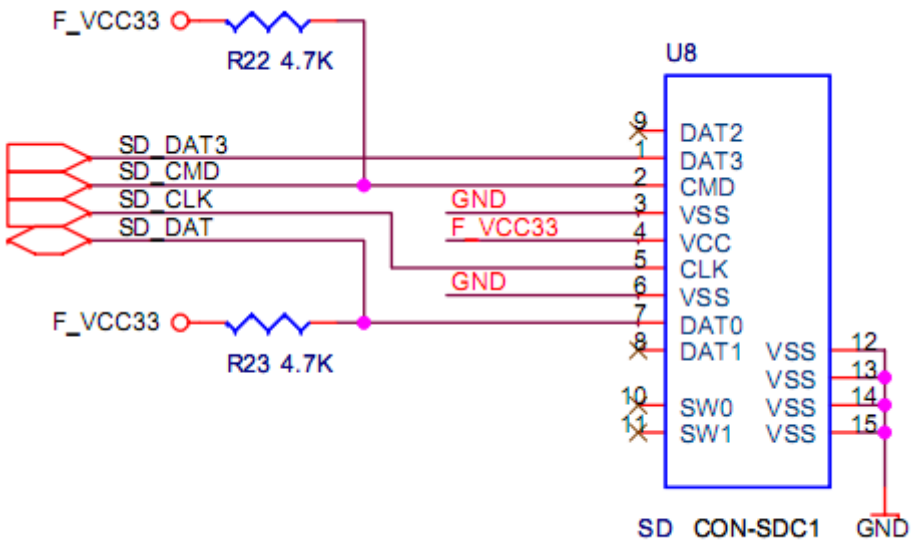


图 2.16 SD 卡电路原理图

2.6.3 SD 卡的操作规范

首先分析 SD 卡的内部结构，SD 卡通过外部引脚与 SD 卡的内部接口连在一起，然后到其存储介质，另外还有很多常见的寄存器，比如 OCR，CID，RCA，DSR，CSD,SCR 等，其内部结构如图 2.17 所示：

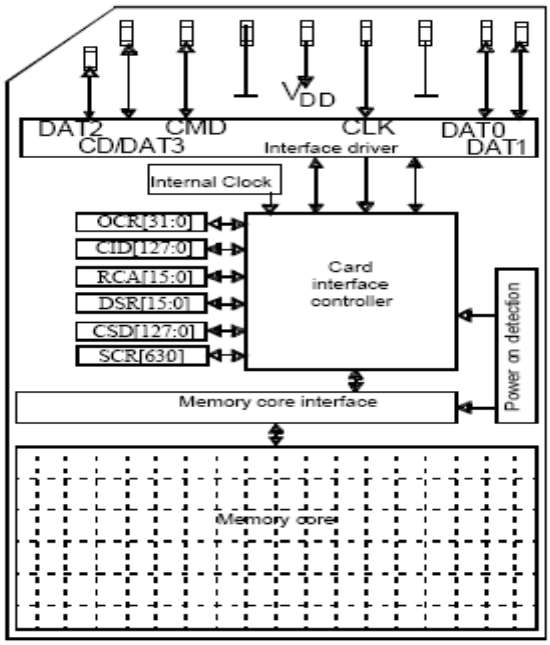


图 2.17 SD 卡的内部结构

SD 卡的操作一般是经过初始化，发送命令模式，比如复位，读写命令等。初始化的时序要求比较严，频率不得高于 400Khz，其操作过程全部流程如图 2.18 所示：

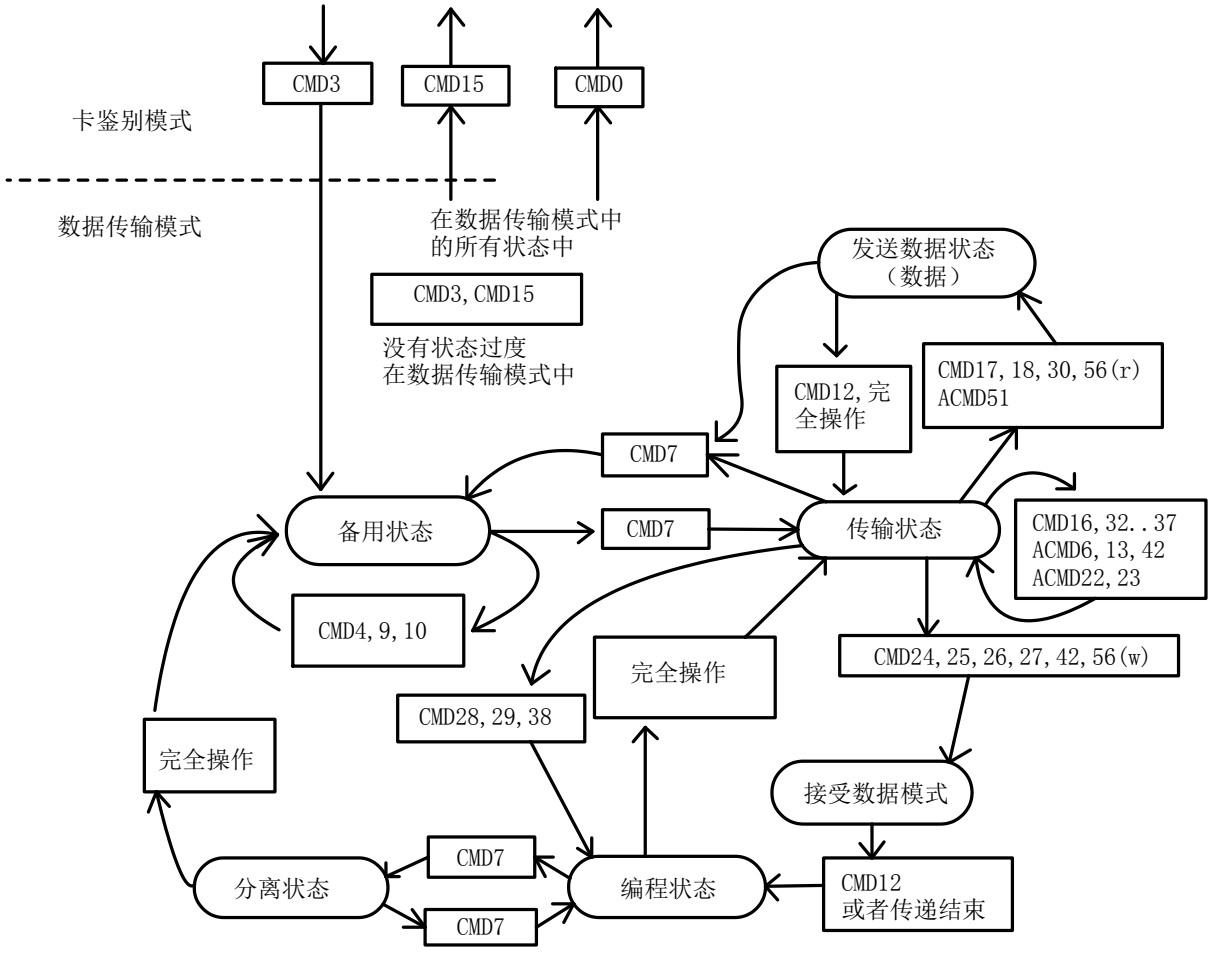


图 2.18 SD 卡流程状态

从上面的状态图中我们可以观察出如下信息：

所有的读数据指令艘可以被停止指令打断（CMD12）。数据传输将结束，卡将返回到传输模式。读命令是：读块(CMD17)，读多重块(CMD18)，发送写保护(CMD30)，发送 SCR(ACMD51)和读模式中的普通指令。

所有的写指令在任何时候都能被停止指令打断（CMD12）。写命令必须在未选择卡之前被 CMD7 指令停止。写指令是:写块(CMD24 和 CMD25) ,写 CID(CMD26), 写 CSD(CMD27), 锁定/解锁指令(CMD42) 和写模式下的普通指令(CMD56)。

一旦完成了数据传输，卡将退出写数据状态并在传输成功时转到编程状态（译者注：将数据写入卡的闪存时的状态）或在传输失败时转到传输状态。

如果读块操作停止，同时块的长度和最后一个块的 CRC 是有效的，数据将被规划（译者注：将数据写入卡的闪存中）。

卡可能会为写块提供缓存。这意味着下一个块可以在先前的数据被规划（译者注：将数据写入卡的闪存中）时发送给卡。如果所有的写缓存都满了，只要卡还在编程状态（见图 18），DAT0 线将报纸低电平（忙有效）。

在写 CSD、CID、写保护和擦除操作时不提供缓存。这意味着在卡忙着上述操作时，不可以执行其他的数据传输指令。只要卡是处于忙时或处于编程状态时，DAT0 线将保持为低电平。实际上，如果卡的 CMD(命令)和 DAT0 线保持分离并且主机保持忙着的 DAT0 线与其他的 DAT0 线(总线上)分离，主机可以在这张卡忙时操作其他的卡。

当卡正处于编程时，参数设定指令无效。参数设定指令包括：设定块长度(CMD16)，擦除块的开头(CMD32)和擦除块的末尾(CMD33)。

当卡正处于编程时，读指令无效。

将其他卡从备用状态转到传输状态时（使用 CMD7），无须结束擦除和编程操作。卡将转到分离状态并且释放 DAT 线。

卡处于分离状态时可以不被选择，使用 CMD7。在这种情况下，卡将会转到变成状态并恢复忙指示。

重新设定卡将会结束任何一种状态或者正处于操作中的编程操作。这将损坏卡的数据。应当尽量避免这样的主机操作。

当读数据时，DAT 总线线路位准是高的被那拉高当没有数据被传输。一个被传输的数据块由起始位（1 个或 4 个位的低电平）和后面的连续的数据流组成。数据流包含负载量数据(而且在使用了卡的 ECC 时包含错误校正位)。数据流以结束位（1 个或 4 个高电平）结束。数据的传输是与时钟信号保持同步的。(为定向数据传输的)块是由 1 或 4 位 CRC 校验和来保护的。

来自 SD 卡的读操作可以被关电源打断。SD 卡确保数据除了写或擦除操作外不被其他任何条件破坏，甚至主机突然停止运算或移除。

当要读块时，读块是快进行定向的数据转移。数据转移的基本单位是块，CSD(READ_BL_LEN)定义了最大值。起始和终止地址完全包含在一个实际块(如 READ_BL_LEN 所定义)中的较小的块也可以传输。能够传输 512 位的块对 SD 卡来说是一个强制性的标准。

为了保证数据传输的完整性，CRC 附加在每一个块的末尾。在读块的传输完成之后，CMD17(READ_SINGLE_BLOCK)介入读块，将会使卡返回传输状态。CMD18(READ_MULTIPLE_BLOCK)启动连续块的传输。块在接到 STOP_TRANSMISSION 命令(CMD12)前将持续的传输。由于持续的传输，停止命令执行将会延迟。数据传输将在接到结束位后停止。

如果主机仅使用部分块，累积后导致块的长度不齐，这是不允许的，卡将从第一个不齐的块开始删除这些块，并在状态寄存器中设一个 ADDRESS_ERROR 错误位。放弃传输并在数据状态中等待停止命令。

当写数据时，数据传输格式和读数据格式类似。由于块定向了写数据传输，CRC 检查位被加给了每个块。由于卡对接收到的每一个数据块都进行了 1 到 4 位的 CRC 对比(见 7.2 章)核查，所以避免了因写入造成的错误。

当要写数据块时，在区块书写的时候(CMD24-27,42,56(w))一或较多范围的数据从主机到卡被主机的每个区块的为那目的被附加的 1 或 4 位 CRC 转移。卡将总是能够接受并写入被 WRITE_BL_LEN 定义了的一个范围的数据，并且是 512 个字节的衍生（例如：如果写一个长度为 1024 的字节，那么可以写块为 1024 或 512 字节)。如果允许 WRITE_BL_PARTIAL（=1)，那么较小的块，直到 1 个字节，也可以使用。如果 CRC 失败，卡将在 DAT 线上指示失败（见下)；传输的数据将不做书面说明的被丢弃，并且后面的所有数据将被忽略（在写多重块模式中)。

为了使写操作更快速，写多重块优于写单一块。

如果主机使用累积长度的部分块而不是完整块，并且块没有对准，这是不允许的(CSD 参数 WRITE_BLK_MISALIGN 没有设定)。卡将在发现第一个没有对准的块时，开始放弃规划。卡将在状态寄存器中设定 ADDRESS_ERROR 错误位，然后忽略后面传输来的数据，在接受数据状态等待停止命令。

如果主机尝试对一个保护区进行写操作，写操作将失败。此时卡将设置 WP_VIOLATION 位。

2.7 TFT 液晶

2.7.1 TFT 液晶介绍

TFT 采用的是 ILI9325 芯片，可以显示 25 万多种颜色，16 位真彩的，另外还带有触摸芯片，在上面可以用触摸笔操作。其液晶外观如图 2.19 所示：

图 2.19 TFT 液晶外观图

其背面照片如图 2.20 所示。包过了 SD 卡控制电路部分（这里直接用的是开发板上的 SD 卡，不是这里的)，触摸屏控制芯片，TFT 控制芯片等。

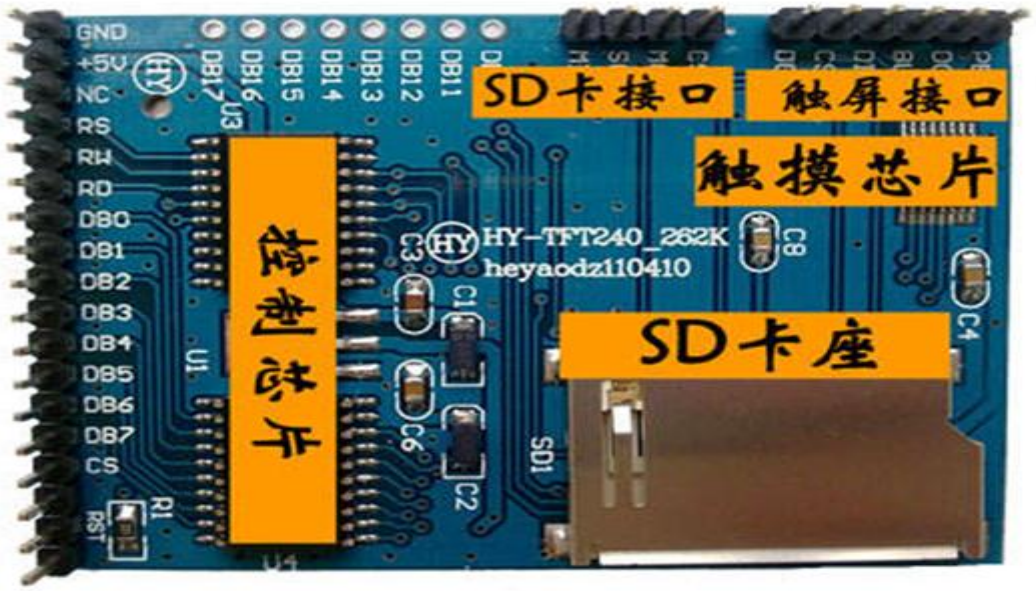


图 2.20 SD 卡模块背面图

2.7.2 TFT 液晶驱动电路设计

FPGA 与 TFT 液晶显示器采用普通 IO 接如图 2.21 所示，其中 TFT 液晶采用的是 8 位的，触摸屏控制引脚虽然引出了，但是在程序中没有利用该功能，所以也没有用到。

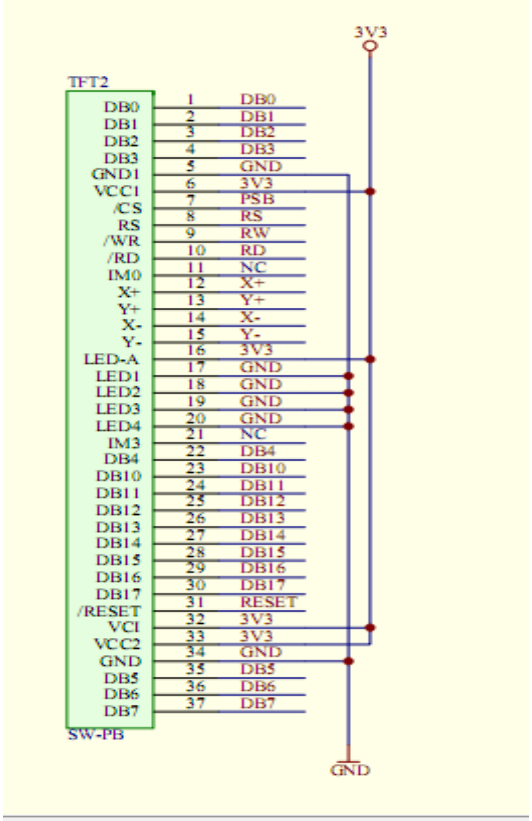


图 2.21 SD 卡硬件电路

2.7.3 ILI9325 控制器简介

（1） 关于坐标原点的确定

首先，买回来的液晶屏，有一个位置，就是 G1 和 S1 开始的位置，暂且把它称为物理地址。这个地址是没有办法改变的，所以就称之为物理地址，不过屏幕的坐标原点还是可以任意改的。关于 G1 和 S1 的描述可以参考图 2.22.

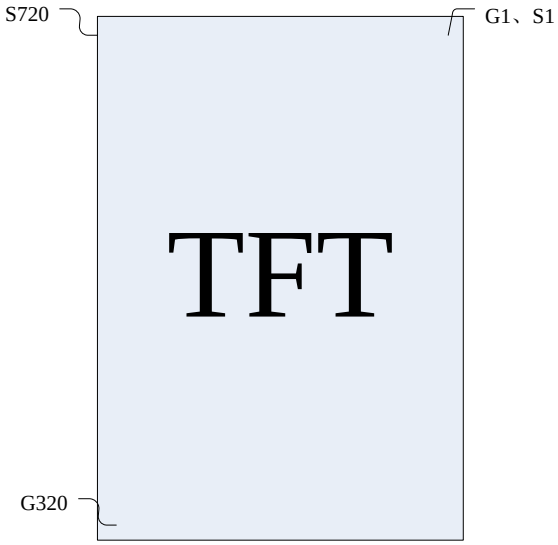


图 2.22 TFT 上坐标点示意图

（2） 屏上点和 GRAM 的对应关系。

从图 2.23 可以看出 TFT 屏幕显示的每一个点和 GRAM 的每一个点的对应关系。每一行的每一个点地址 2.5 个字节，所以用三个 S 表示一个点，720/3=240 ，且纵向是 320 行，所以是 G1 到 G320，所以正好是 320*240。

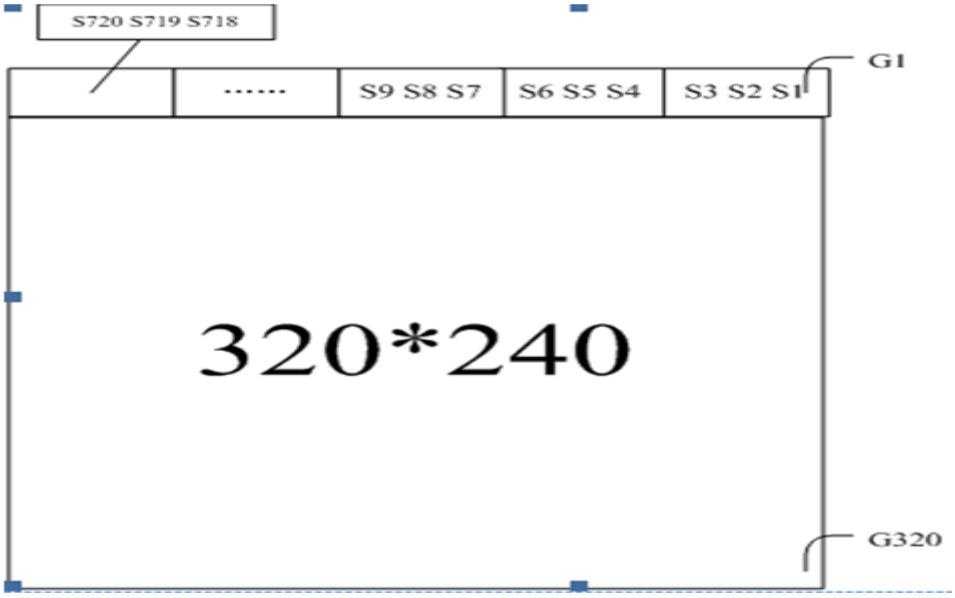


图 2.23 点和 GRAM 点的对应关系

（3） 关于寄存器 01H 的介绍

8.2.4. Driver Output Control (R01h)驱动器输出控制

R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	1	0	0	0	0	0	SM	0	SS	0	0	0	0	0	0	0	0

图 2.24 01H 寄存器

如图 2.24，其中 SS 为确定从源驱动器选择输出的转变方向。

当 SS=0，输出转变方向是从 S1 到 S720。

当 SS=1，输出转变方向是从 S720 到 S1。

除了改变方向 ，SS 和 BGR 位的设置都需要更改 R，G， B 三个点的分配到源驱动引脚。

从 S1 到 S720 指定 R，G ，B 点到源驱动引脚，设置 SS = 0 。

从 S720 到 S1 指定 R，G ，B 点到源驱动引脚，设置 SS =1 。

当改变 SS 或 BGR 位时，RAM 数据必须再次被写入。

设置栅极驱动引脚排列与 GS（R60H）相结合，以为模块选择最佳扫描。

其具体怎么扫描的可以参照下面的示意图 2.25：

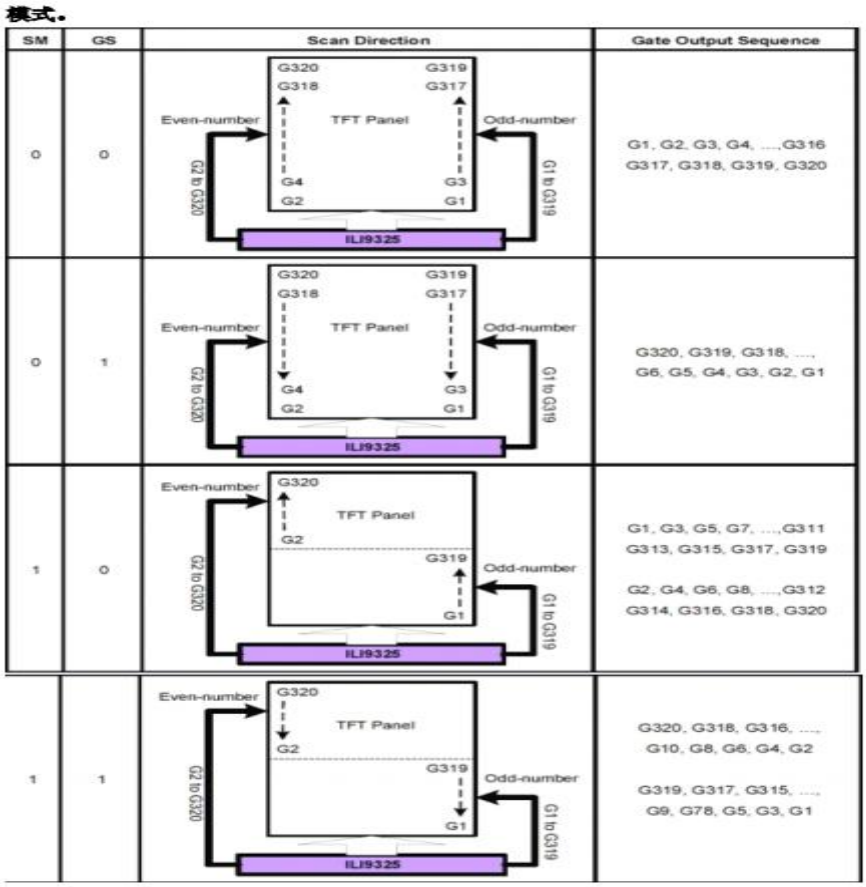


图 2.25 TFT 内部扫描控制图

（4） 关于寄存器 03H 的介绍

8.2.5. Entry Mode (R03h)

R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	1	TRI	DFM	0	BGR	0	0	HWM	0	ORG	0	I/D1	I/D0	AM	0	0	0

图 2.26 03H 寄存器

如图 2.26，其中 AM:控制 GRAM 更新方向的控制位

AM = 0:在水平方向更新地址

AM = 1:在垂直方向更新地址

这个地方对 AM 的选择将直接影响 img2lcd 软件的扫描方式控制项，这一位就是控制扫描方式的。

I/D[1:0]：当更新显示区域的一个像素点的时候，控制 AC 是增加 1 还是减少 1，参考下图 2.27:

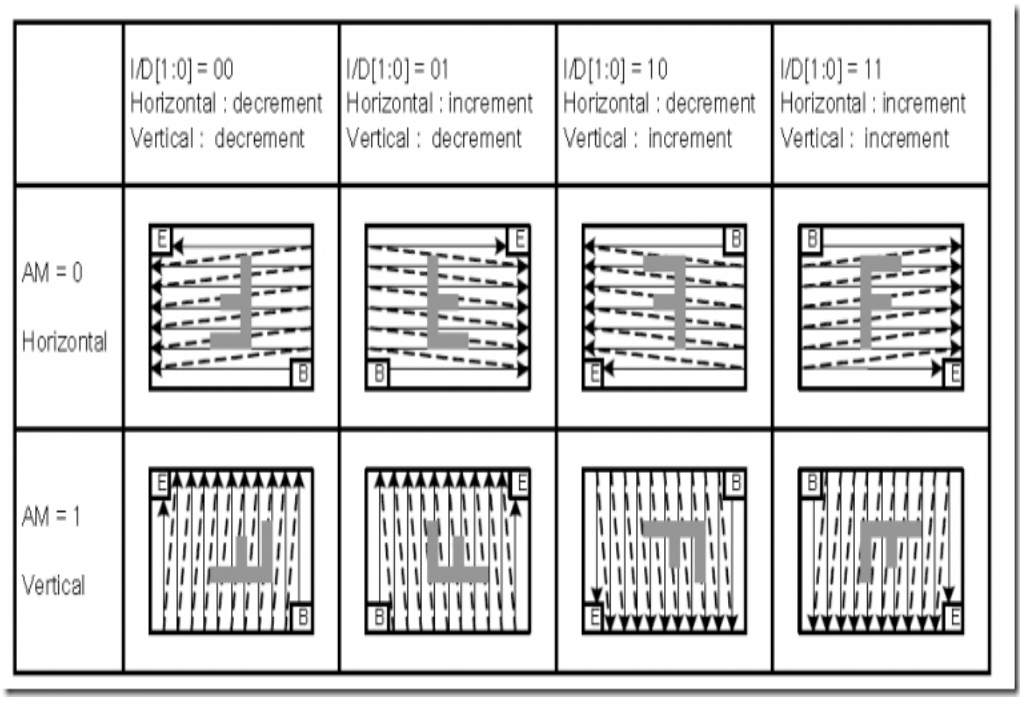


图 2.27 像素扫描方式

I/D[1:0] 的正确设置才能正确的显示图片，比如有时候发现显示出来的图片和输入 img2lcd 的图片方向是左右方向是反的，或者上下 或者都是反的，那就是
需要修改这个的地方了，可以根据上面的方向来选择合适的 I/D.

ORG：当一个窗口的地址区域确定以后，根据上面 I/D 的设置，来移动原始地址。当高速写窗口地址域时，这个功能将被使能。

ORG = 0: 原始地址是不移动的。这种情况下，是通过指定地址来启动写操作的，这个地址是根据窗口显示区域的 GRAM 的地址表。

ORG = 1:原始地址是更加 I/D 的设置相应的移动的。

注意：1、当 ORG =1 的时候，设置 R20H,R21H,的原始地址的时候，只能设置 0x0000

2、在 RAM 读操作时，要保证 ORG = 0;

BGR 交换写数据中红和蓝

BGR = 0 ： 根据 RGB 顺序写像素点的数据。

BGR = 1： 交换 RGB 数据为 BGR,写入 GRAM

TRI: 当 TRI = 1 的时候，在 8 位数据模式下是以 8bit * 3 传输的，也就是传输三个字节到内部的 RAM，同样也支持 16 位数据的模式，和使用 SPI 模式显示 26 万色，也就是说当 RTI = 1 的时候，传输的字节数基本上都是三个。这一位在显示 26 万色的时候有用的，或者使用 8 位数据接口的时候，这个要看具体的应用来设置，但是注意如果不需要的时候，要设置为 0.

DFI ： 设置像内部 RAM 传输数据的的模式。这一位是要和 TRI 联合起来使用的

（5） 关于寄存器 04H 的介绍

8.2.6. Resizing Control Register (R04h)

R/W	RS	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
W	1	0	0	0	0	0	0	RCV1	RCV0	0	0	RCH1	RCH0	0	0	RSZ1	RSZ0

图 2.28 04H 寄存器

如图 2.28 所示 RSZ[1:0] ： 设置调整参数（RSZ 的意思就是 resizeing)。

当设置了 RSZ 后，ILI9325 将会根据 RSZ 设置的参数来调整图片的大小，这个时候水平和垂直方向的区域都会改变。在本系统中采用的缩小功能就是直接给 R04 寄存器赋值，硬件实现图片缩小功能，节省了 CPU 工作的时间。

RSZ[1:0]	Resizing factor
00	No resizing (x1)
01	x 1/2
10	Setting prohibited
11	x 1/4

图 2.29 RSZ 缩小的倍数

根据图 2.29，可以知道，设置 RSZ 相应的值就可以缩小为 1/(RSZ[1:0] - 1)。

RCH[1:0]: 当调整图像大小的时候设置水平余下的像素点的个数。实际上就是拿当前的图像的水平像素个数和缩小后水平像素个数取模，原因是由于你的图像不可能正好能被缩小 1/2，或者 1/4，比如你的图像水平像素点是 15 个，如果需要缩小为 1/2，但是 15 除以 2 是有余数的，余数为 1，RCH[1:0]这个时候就设置为 1，实际上就是保证你的原始图像水平减去几个像素点正好能被 RSZ 除尽。

RCV[1:0]:同上面的 RCH 原理是一样的，这个是来保证垂直方向上减去几个像素点正好能被 RSZ 除尽。

3 系统软件设计

软件是一个系统的灵魂，其灵活性使在硬件固定的情况下能实现很多功能。本章将介绍其程序代码开发的思想，流程以及涉及到的算法。

3.1程序总体框架结构

程序流程图如图 3.1 所示。在端口，液晶和 SD 卡初始化完成之后就一直扫描按键，根据按键按下的对应信息分别采用不同的方式显示图片。

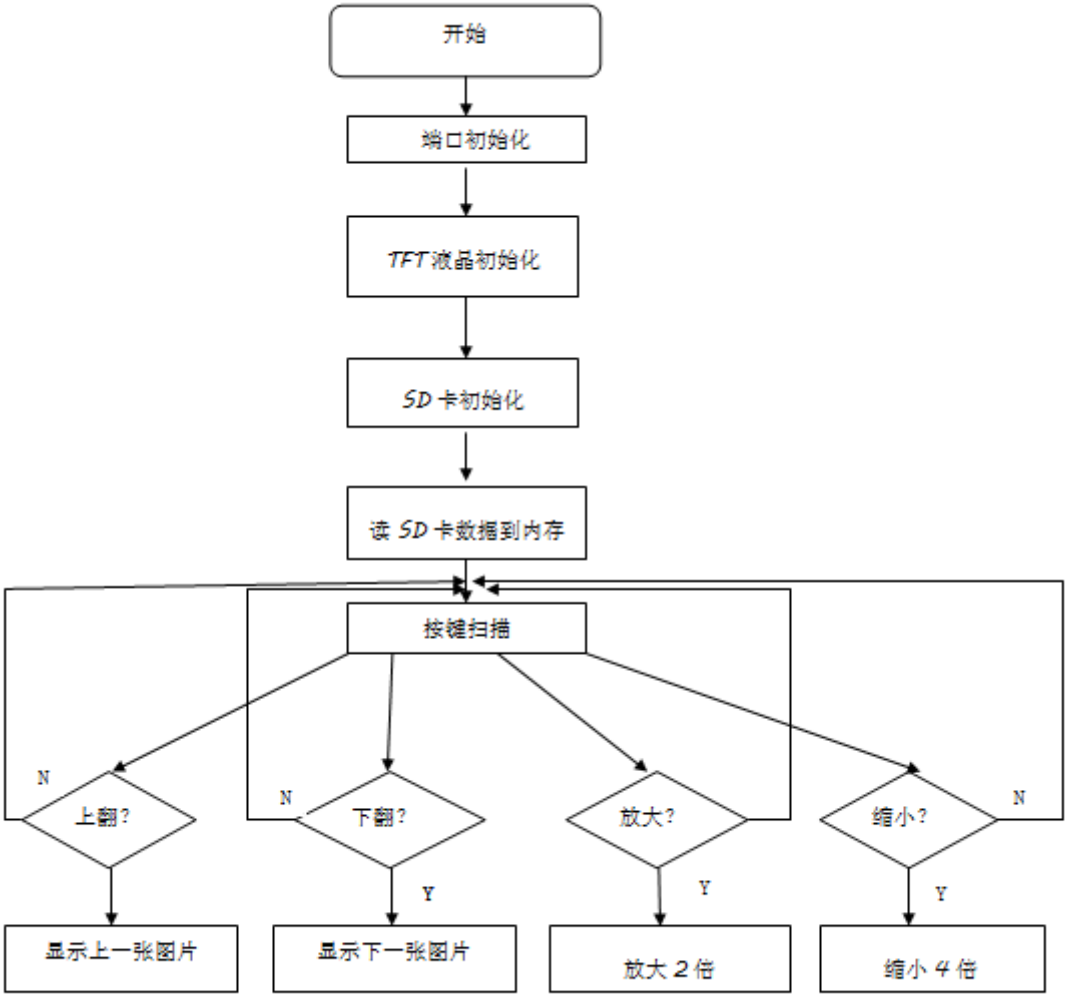


图 3.1 程序流程图

3.2Niosll 软核

本小节对本设计所使用的编程语言和软件进行了详细的介绍。

3.2.1NiosII 软核介绍

Nios Ⅱ嵌入式处理器是 Altera 公司推出的采用哈佛结构、具有 32 位指令集的第二代片上可编程的软核处理器，其最大优势和特点是模块化的硬件结构，以及由此带来的灵活性和可裁减性。相对于传统的处理器，Nios Ⅱ系统可以在设计阶段根据实际的需求来增减外设的数量和种类。设计者可以使用 Altera 提供的开发工具 SOPC Builder，在 PLD 器件上创建软硬件开发的基础平台，也即用 SOPC Builder 创建软核 CPU 和参数化的接口总线 Avalon。在此基础上，可以很快地将硬件系统(包括处理器、存储器、外设接口和用户逻辑电路)与常规软件集成在单一可编程芯片中。而且，SOPC Builder 还提供了标准的接口方式,以便用户将自己的外围电路做成 Nios Ⅱ软核可以添加的外设模块。这种设计方式，更加方便了各类系统的调试。

Nios II 处理器具有完善的软件开发套件，包括编译器、集成开发环境（IDE）、JTAG 调试器、实时操作系统（RTOS）和 TCP/IP 协议栈。Nios II 集成开发环境(integrated development environment, IDE)是 Nios II 系列嵌入式处理器的基本软件开发工具。所有软件开发任务都可以在 Nios II IDE 下完成，包括编辑、编译和调试程序。Nios II IDE 提供了一个统一的开发平台，用于所有 Nios II 处理器系统。

包括 3 种产品，分别是：Nios II/f（快速）——最高的系统性能，中等 FPGA 使用量；Nios II/s（标准）——高性能，低 FPGA 使用量；Nios II/e(经济)——低性能，最低的 FPGA 使用量。这 3 种产品具有 32 位处理器的基本结构单元——32 位指令大小，32 位数据和地址路径，32 位通用寄存器和 32 个外部中断源；使用同样的指令集架构（ISA），100%二进制代码兼容，设计者可以根据系统需求的变化更改 CPU，选择满足性能和成本的最佳方案，而不会影响已有的软件投入。

特别是，Nios II 系列支持使用专用指令。专用指令是用户增加的硬件模块，它增加了算术逻辑单元（ALU）。用户能为系统中使用的每个 Nios II 处理器创建多达 256 个专用指令，这使得设计者能够细致地调整系统硬件以满足性能目标。专用指令逻辑和本身 Nios II 指令相同，能够从多达两个源寄存器取值，可选择将结果写回目标寄存器。同时，Nios II 系列支持 60 多个外设选项，开发者能够选择合适的外设，获得最合适的处理器、外设和接口组合，而不必支付根本不使用的硅片功能。

Nios II 包括 3 种产品，分别是：Nios II/f（快速）——最高的系统性能，中等 FPGA 使用量；Nios II/s（标准）——高性能，低 FPGA 使用量；Nios II/e(经济)——低性能，最低的 FPGA 使用量。这 3 种产品具有 32 位处理器的基本结构单元——32 位指令大小，32 位数据和地址路径，32 位通用寄存器和 32 个外部中断源；使用同样的指令集架构（ISA），100%二进制代码兼容，设计者可以根据系统需求的变化更改 CPU，选择满足性能和成本的最佳方案，而不会影响已有的软件投入。

另外 Nios II 的 C 语言和 标准 C 语言 或者单片机 C 语言很相似，上层的标准 C 库函数都是一样的,区别在于与底层硬件相关的各个外设寄存器的结构不同。如果我们

把访问底层硬件寄存器的函数封装起来供上层调用，平台之间的移植就显得很容易了。所以我们一般把各个设备与其底层寄存器联系在一起，封装好。这样对习惯于其他嵌入式处理器编程的人员应用更方便。

以下图 3.6 就一个简单的 LED 口的封装。

```
#include "system.h"

#define _LED        1

typedef struct
{
    unsigned long int DATA;
    unsigned long int DIRECTION;
    unsigned long int INTERRUPT_MASK;
    unsigned long int EDGE_CAPTURE;

}PIO_STR;                                   2

#ifdef _LED
#define LED ((PIO_STR *)PIO_LED_BASE)
#endif                                      3
```

图 3.2 PORT 口的 LED 灯的程序封装

可以看到标有红圈的的地方定义了一个 PIO_STR 结构体，将其命名为 PIO_STR，这个结构体就是我们以后一直要操作的对象，它的来源也是很有说法的，它是根据芯片手册来写的。具体的要参考其 Nios II 9.0 的 datasheet。

3.2.2NiosII 软件设计流程

在进行 NIOS 软件开发时，通常按照下面的过程进行。

步骤 1：获得目标 N1OS 系统的 SDK

从 SOPC Builder 创建的工程目录中(或从负责 NIOS 处理器硬件系统的设计师那里)得到目标 NIOS 处理器系统的 SDK。软件开发环境的基础是由 SOPC Builder 生成的 SDK 目录。SDK 中包含的头文件和库文件，提供了硬件映像地址和一些基本的硬件访问子程序，有效地减少了原有分工模式下的软硬件衔接工作。

步骤 2：建立和编译应用软件

首先使用文本编辑器，用 C / C++或汇编语言编写应用程序的源代码(.c 或.s)；然后使用 nios-build 命令或 Makefile 将源代码编译成可执行代码。编译产生的二进制码以 S-record 的格式(. sere)存储。对于一个小型或中型的软件项目，使用 nios-build 就可以编译生成可执行代码：对于一个大型项目，则必须参照由 SOPC Builder 生成的示例 Makefile 编写自己的 Makefile，来编译生成可执行代码。

步骤 3：下载可执行代码到目标板

如果用户的目标系统中包含了 GERMS 监控程序，则它将扮演起调试器的角色，允许用户运行可执行代码、读写存储器、下载代码(或数据)段到存储器和擦除闪存。将 GERMS 监控程序分配到处理器的引导地址(通常位于片上 ROM)，用户可以立即进行代码开发、

下载和调试。

步骤 4：调试代码

如果采用了 GERMS 监控程序，那么调试信息通过 printf()函数由标准输入输出设备来发送。标准输入输出设备可以是 UART，也可以是 NIOS OCI 调试模块。nios. mn 命令可以用作一个终端来显示这些调试信息。

如果需要更细致的调试，可以通过 JTAG 下载电缆，用 NIOS OCI 调试模块来访问 NIOSCPU。通过 NIOS OCI 调试模块，能够执行单步调试代码，检查存储器和寄存器内容等。

也可以通过把编译器调试选项设成 ON，而重新编译代码，使用 GNu 调试器(GDB)。更为高级的调试 T 具可以从第三方供应商那里得到，在第八章里会对这些调试手段进行比较详细的介绍。

步骤 5：转变成目标代码

一旦应用程序代码经充分调试之后，用户可能希望把可执行代码存储到目标系统的非易失存储器里，以便在 NIOS CPU 复位后立刻执行用户程序。

3.2.3NiosII 软核的使用步骤

首先，将 NIOS II 9.0 IDE 软件打开，打开后 NIOS II IDE 的界面赫然显现在我们面前，界面很简单，跟其他的 IDE 没什么太大的区别，需要做的就是首先建立一个软件工程，操作方式如下图 3.2 所示，File->New->Project。

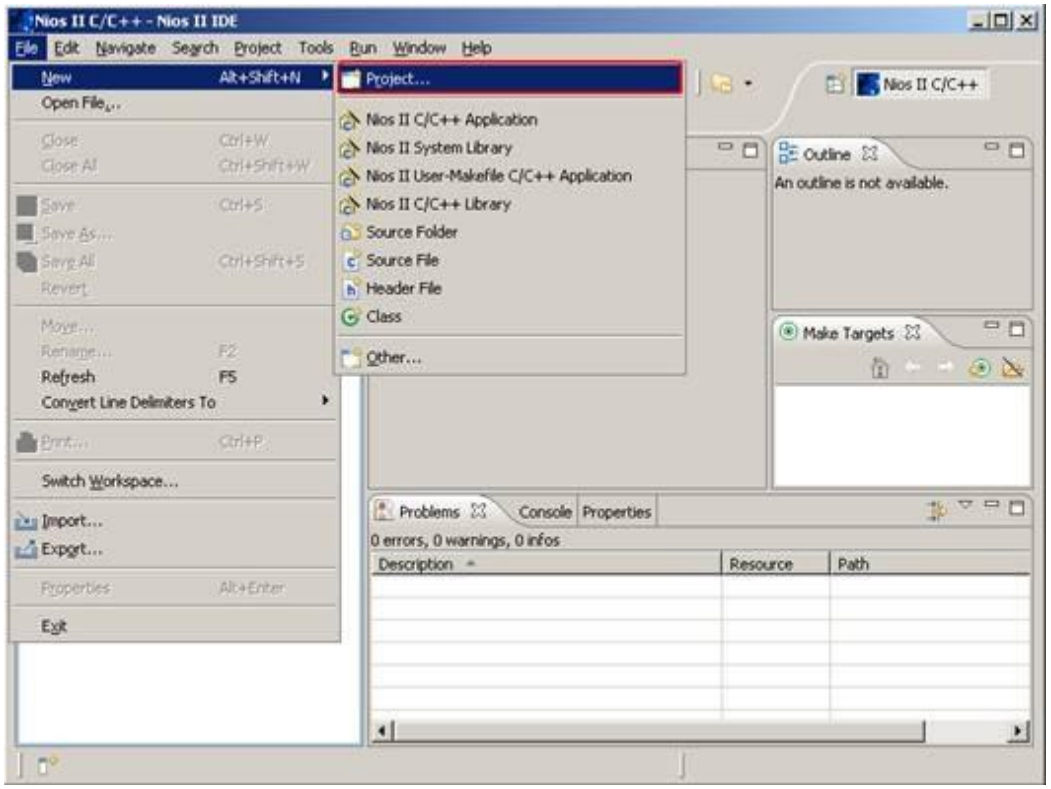


图 3.3 新建 NiosII 工程

点击后，会出现工程向导界面，如下图所示，选中红圈处的内容，NiosII C/C++ Application。如果 3.3 所示：

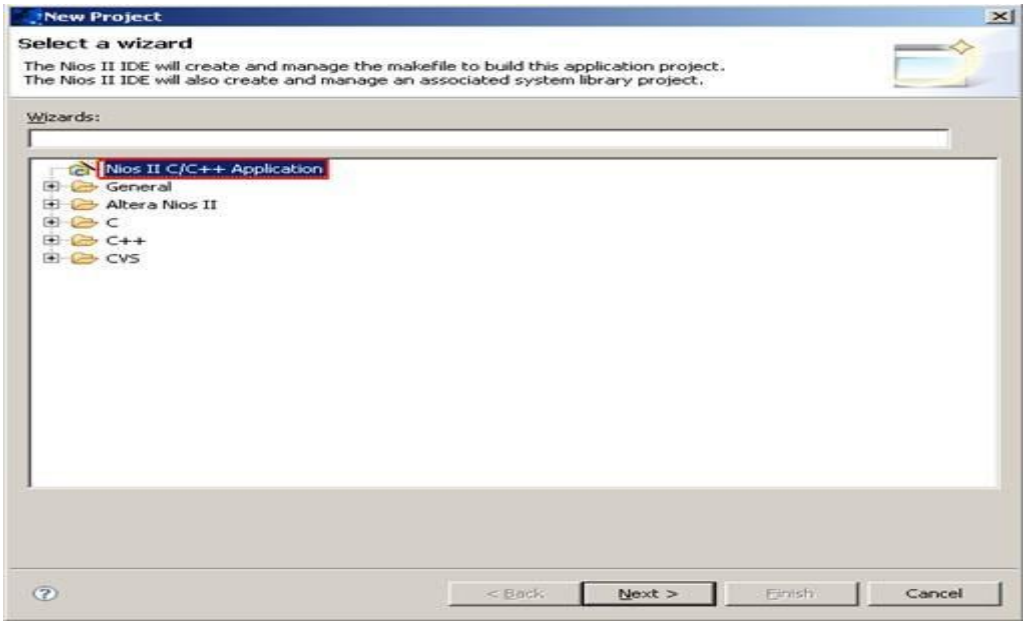


图 3.4 选择 NiosII C/C++ Application

点击 Next，会出现下图 3.4 所示内容，红圈 1 处是工程名，将其修改为 hello_world，红圈 2 处是目标硬件文件，点击 Browse，找到上一节生成的 NIOS 软核的位置，这个文件是以.ptf 为后缀的。在红圈 3 处选中 Hello World，这个地方是工程模版。再说说红圈 4，这个地方是改变工程所放位置的，如果不修改，软件工程的位置就在 Quartus 工程目录下的 software 下面。

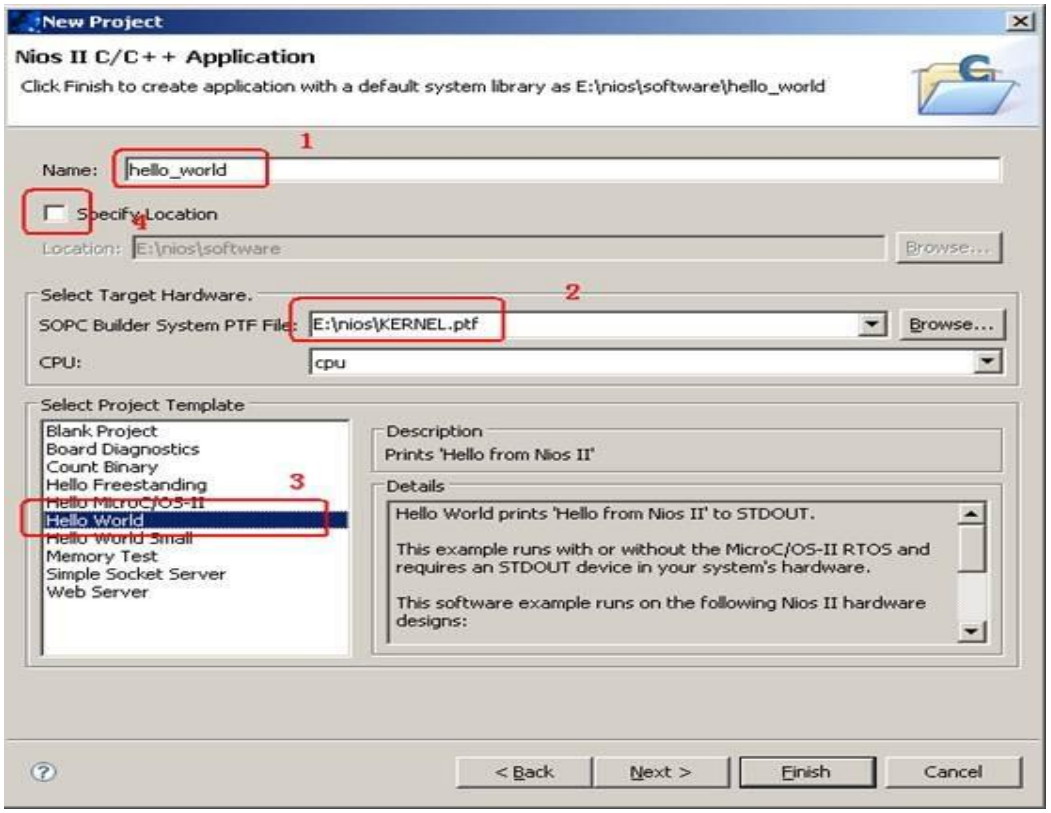


图 3.5 选择工程模板

点击 Next，这里不用修改，点击 Finish，完成工程向导。完成了上面的工程向导后，就可以正式进入 NIOS II IDE 的界面了，如下图 3.5 所示。就可以在代码区编辑代码了。

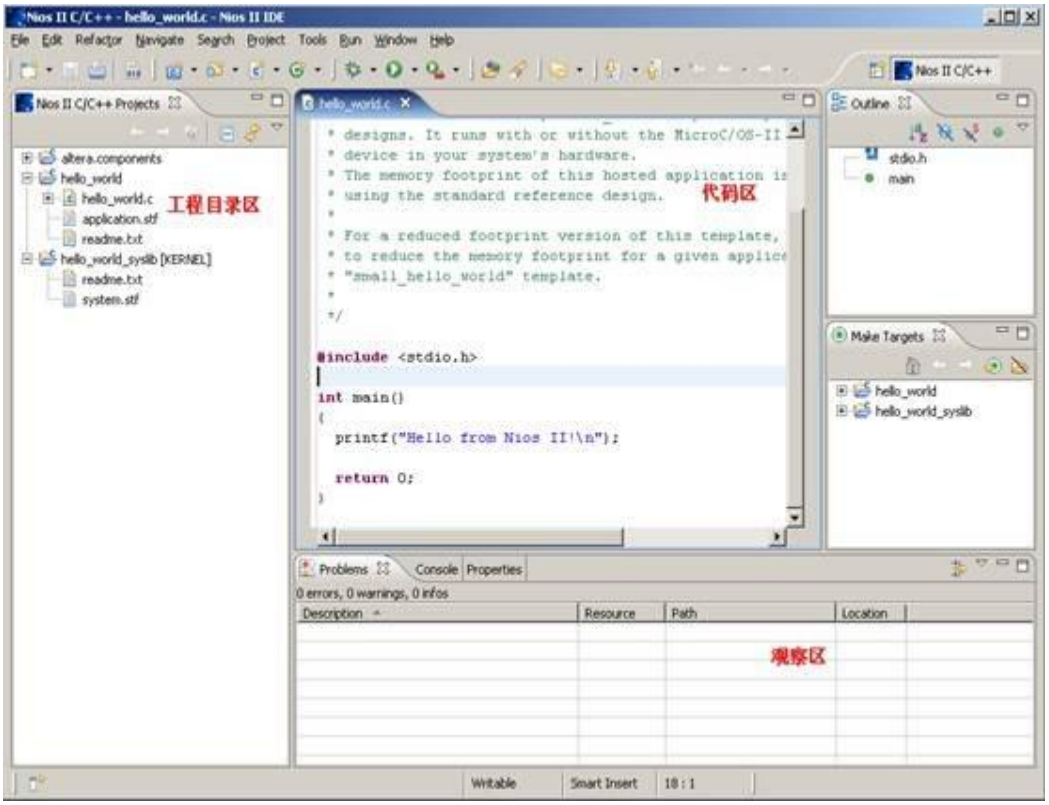


图 3.6 NiosII 工程新建后的界面

3.3 程序各部分解析

3.3.1系统预定义处理

因为 Nios II 工程建立后编译，对应的设备只提供了相应的寄存器，其相应的信息放在了 sysytem.h 中。查看其中的内容，比如 LCD_RS 这个端口，其与 niosII 打交道的信息如下：

```
#define LCD_RS_NAME "/dev/LCD_RS"
#define LCD_RS_TYPE "altera_avalon_pio"
#define LCD_RS_BASE 0x04001040
#define LCD_RS_SPAN 16
#define LCD_RS_DO_TEST_BENCH_WIRING 0
#define LCD_RS_DRIVEN_SIM_VALUE 0
#define LCD_RS_HAS_TRI 0
#define LCD_RS_HAS_OUT 1
#define LCD_RS_HAS_IN 0
```

```
#define LCD_RS_CAPTURE 0
#define LCD_RS_DATA_WIDTH 1
#define LCD_RS_RESET_VALUE 0
#define LCD_RS_EDGE_TYPE "NONE"
#define LCD_RS_IRQ_TYPE "NONE"
#define LCD_RS_BIT_CLEARING_EDGE_REGISTER 0
#define LCD_RS_BIT_MODIFYING_OUTPUT_REGISTER 0
#define LCD_RS_FREQ 100000000
#define ALT_MODULE_CLASS_LCD_RS altera_avalon_pio
```

其中最重要的就是 LCD_RS_BASE，也就是这个端口的地址。以后对这个端口进行输入输出或者中断操作，都要用到这个信息，这一点我们已经在上小节的 Nios IIC 语言的特点中已经有所介绍了，也就是对应在那个结构体重。所以在主函数 main()中一定要包含这个头文件，当然因为 Nios II 是软核，有一些地方是与其他 MCU 不同的，我们要把那些要包含在主函数前的宏定义中。其中用到的与 Nios II 核有关的头文件如下。

```
#include "sys/alt_irq.h"
#include "system.h"                    //包含基本的硬件描述信息
#include "altera_avalon_pio_regs.h"    //包含基本的 I/O 口信息
```

3.3.2 FAT32 在 SD 卡上的移植

FAT32 是 Windows 下的一种分区格式，它采用的 32 位的文件分配表，对磁盘的管理能力远远大于 FAT16 文件系统，因为 FAT16 文件系统最多只能管理 2GB 的文件。因为它的每个簇都是 4KB，比 FAT16 的每个簇 16KB 要节省很多，提高了磁盘的利用率。所以这里采用 FAT32 的文件系统。驱动 SD 卡采用文件系统比直接驱动 SD 卡从地址上开始读写要方便很多，因为这样不用每次都用 winhex 软件来查看哪个文件的首地址是多少，而可以根据文件的文件名来存储文件，方便了很多。关于 SD 卡上移植 FAT32 文件系统的代码在附录中给出了，其主要是关于 SD 初始化部分的程序。

3.3.3TFT 显示程序

TFT 显示程序的主要步骤在 TFT 的驱动，因为其内部有很多寄存器，必须一一设置好才行。当然 TFT 采用的是 8 个数据位的，写数据时采用 WR 的上升沿写入数据，读数据时采用 RD 的下降沿读取数据。写命令和写数据由 RS 引脚分开，当 RS=0 时为命令操作，当 RS=1 时为数据操作。其写入命令的函数如下：

```
void Write_Cmd(unsigned char DataH,unsigned char DataL)
{
    LCD_CS=0;
    LCD_RS=0;          //RS=0时位写命令
    LCD_DATAL=DataH;
```

```
    LCD_WR=0;          //WR的上升沿写入数据
    LCD_WR=1;
    LCD_DATAL=DataL;   //先写高位，再写低位
    LCD_WR=0;
    LCD_WR=1;
    LCD_CS=1;
}
```

其写入数据的函数如下：

```
void Write_Data(unsigned char DataH,unsigned char DataL)
{
    LCD_CS=0;
    LCD_RS=1;          //RS=1时位写数据
    LCD_DATAL=DataH;
    LCD_WR=0;          //WR的上升沿写入数据
    LCD_WR=1;
    LCD_DATAL=DataL;   //先写高位，再写低位
    LCD_WR=0;
    LCD_WR=1;
    LCD_CS=1;
}
```

因为在初始化时一般是写命令后马上就接着写数据，为了方便，可以写在一个函数中。如下：

```
void Write_Cmd_Data(unsigned int Cmd,unsigned int Data)
{
    unsigned char x,y,z;
    x=(unsigned char)Cmd;              //因为实际的命令高8位肯定位00，所以只需取其低8位
    y=(unsigned char)(Data>>8);          //取数据的高8位
    z=(unsigned char)Data;              //取数据的低8位
    Write_Cmd(0x00,x);
    Write_Data(y,z);
}
```

TFT 的初始化一定要对，包过扫描方式的设置，GRAM 的设置等，对每一个寄存器先写命令再发数据，这里一定要按照自己想显示的意图来初始化，每一个位的含义要弄清楚，否则会出现一些莫名奇妙的显示。

3.3.4上下翻页处理

上下翻页处理是利用一个变量 count 计数，当下翻键按下时，count 就加 1，当下翻键按下时，count 就减 1。当然这里还涉及到按键的消抖程序，即检测到对应 io 口出现低电平后延时 1ms 再判断是否仍为低电平。其代码如下：

```
void key_scan( )
{
    if(0==KEY_NEXT){         //下翻键
        Delay_1ms(1);
        if(0==KEY_NEXT){
            count++;
            if(101==count)count=1;
        }
    }
     else if(0==KEY_UPPER){   //上翻键
        Delay_1ms(1);
        if(0==KEY_UPPER){
            count--;
            if(0==count)count=100;
              }
    }
}
```

3.3.5放大部分处理

因为一幅图片是 240*320 个点，每个点对应 2 个字节，要把图片放大处理，肯定只能选取其一部分数据拿来在全屏显示。这里我们统一将 1 副图片放大 2 倍，即截取图片 240 宽度中的正中间的 120 个数据，将每一行的这 120 个数据显示在 240 个点上，所以每两个点显示的数据是取出的 120*320 个数据中的同一数据。

其核心处理算法如下面的程序所示：

```
for(j=0;j<320;j++)
    for(i=0;i<240;i++)
     {
        if(i>=60&&i<180)
        {
              Write_Data(image[(240*j+i-1)*2+1],image[(240*j+i)*2]);

              Write_Data(image[(240*j+i-1)*2+1],image[(240*j+i)*2]);
```

```
            }
        }
```

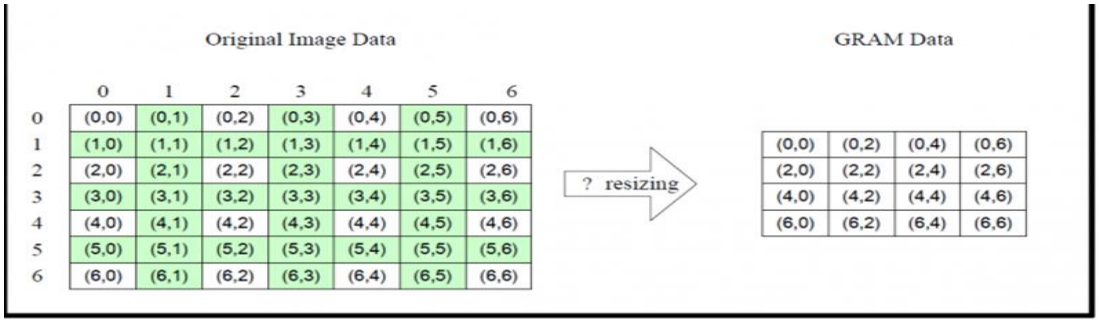
表达式中的(240*j+i-1)*2+1 是图片数组中提取出的我们要显示的数据。

3.3.6缩小部分处理

由于图片缩小后，有很大一部分是空闲中的，如果开始的图片是满屏原样大小显示的，改为缩小显示后，有一部分图片就是本图片原本大小的显示，这样就把多余的信息显示出来了，因此每次使图片缩小后都要先用白色的颜色清除屏幕一次，但是如果用白色清全屏一次后，中间又显示缩小的图片，给用户的感觉就是一直在刷新，图片会有明显的闪动，不是静态的。要消除这种在软件上要花费很多功夫，并且太占用 CPU 时间了。

后面看到了 ILI 的英文手册最后几页，发现其内部就有将图片缩小的硬件实现，只需要设置好寄存器 R04H，主要是设置其 RSH[1:0],其寄存器各位含义已经在 TFT 液晶硬件电路中作了详细的分析。

下面来看看 ILI9325 其内部到底是怎么实现数据的缩小的呢？如图 3.7 所示，它是把原始数据按照行列交换间隔放入 GRAM 中。



	0	1	2	3	4	5	6
0	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)
1	(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)
2	(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)
3	(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)
4	(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)
5	(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)
6	(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)

(0,0)	(0,2)	(0,4)	(0,6)
(2,0)	(2,2)	(2,4)	(2,6)
(4,0)	(4,2)	(4,4)	(4,6)
(6,0)	(6,2)	(6,4)	(6,6)

图 3.7 ILI9325 内部 GRAM 缩小示意图

所以当扫描到按键需要将图片缩小时，只需在程序中加入一个 R04H 的设置就可以了，然后把要显示的区域重新定位一下，包过显示数据的起始坐标。例如放屏幕正中间的话就需要添加以下代码：

```
Write_Cmd_Data(0x0004,0x0001);
LCD_Set_Pos(60,180,80,240);
```

4 系统的测试和总结

4.1系统的测试

经过反复测试，在 SD 卡内共放入了 50 张 240*320 的图片数据，可以通过上翻和下翻流畅的更新图片，通过放大键可以查看图片放大 2 倍后的信息，缩小键可以查看图片缩小 4 倍的信息。系统运行的效果总体如图 4.1 所示：

图 4.1 系统运行的总体效果图

经过测试，可以发现每显示一张图片所用的时间不超过 0.2 秒，4 个按键的灵敏读也非常高，一按下去就有反应。

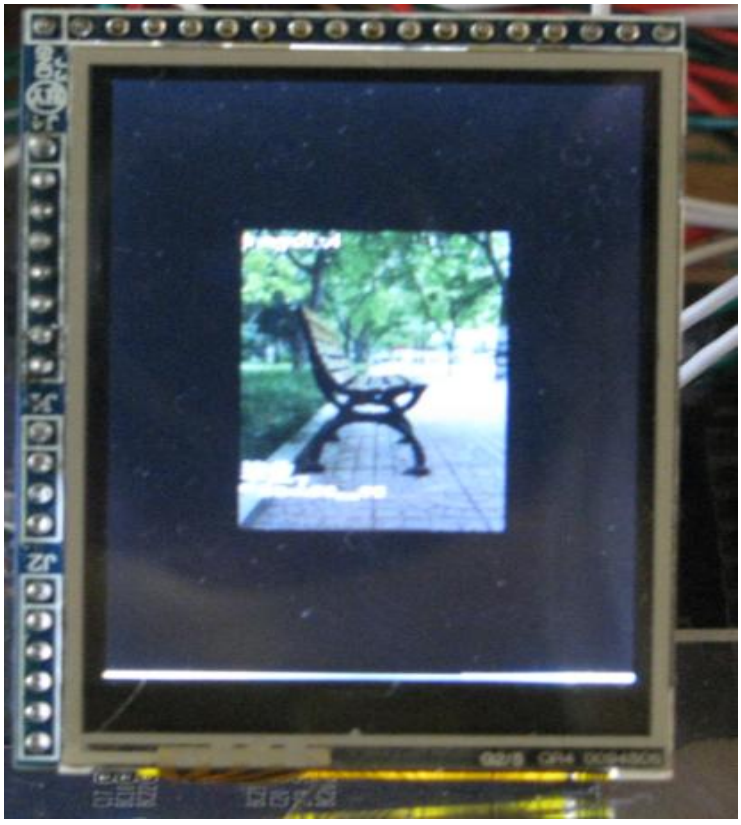


图 4.2 照片缩小 4 倍

当按下缩小键时，照片缩小 4 倍的图片如图 4.2 所示。
当按下放大键后，系统放大 2 倍的照片如图 4.3 所示。

图 4.3 照片放大 2 倍

4.2总结

系统最终达到了预期的的效果，即能大量存储和显示彩色照片，并且能放大和缩小。在完成作品的同时，遇到了不少困难。首先是对 FPGA 不熟，由于以前接触 FPGA 非常少，拿到这个题目时就开始自学 FPGA 了，从学 QuartusII 软件到 Verilog 描述语言，从学 SOPC_Builder 到学 NiosII 软，花了不少时间。在此过程中，经常访问 Altera 官网和进英文技术论坛，看英文资料和文档。慢慢开始对这些开发软件和语言熟悉起来。

接着就是开始学习系统中用到的外围器件了。因为用到了 SDRAM、FLASH、TFT、SD 卡、JTAG 调试器。慢慢开始学这些器件的时序和操作，学会用 Verilog 去驱动它们。一大部分时间都花在这上面，因为把这些理解透了后面的工作就好做了。在经过一段漫长的时间的摸索和实验后，就开始利用 NiosII 这个软核实现整个系统的功能了。NiosII 软核功能很强大，32 位的，性能堪比 ARM，且稳定性非常好，可以自定义。其编程比普通的 MCU 有一定区别，在学习的过程中慢慢体会到了其独特之处，最终用这个软核完成了整个系统。

数码相框的应用越来越多，且功能也越来越强大。在这背后需要更成熟的技术支持。为了完成最终用 FPGA 完成数码相框的功能，本次的实验完成了其初步工作，即可以正常显示 BMP 图片，并且能实现上翻、下翻、放大、缩小。因此还有很大的空间可以改进。

第一：图片数据的存储可以不是直接放入 BIN 数据，而是可以直接放入 BMP 图片，甚至是 JEPG 等其他格式的，因为这些格式的图片背压缩了，占用的空间少。所以后面可以采用在 FPGA 中用 Verilog 实现 JEPG 的编码解码，因为是硬件实现的，所以其速度非常快。

第二：液晶显示的界面可以不仅仅是显示一张图片。我们可以在 NiosII 软核上移植微型的嵌入式操作系统，然后在系统上移植 MiniGui。有了嵌入式操作系统的嵌入，我们可以完成更复杂更多的功能。有了 MiniGui 后，我们的界面管理就变得非常方便，界面更友好和更人性化。

第三：可以实现更多的功能。因为是实现数码相册，可以在系统中加入音乐播放的功能，这样当人们在浏览图片的时候可以同时享受音乐带来的美感。另外还可以加入图片编辑功能，像 photoshop 一样。这样就可以脱离 PC 机来方便的处理图片了。

第四：在图片更新方面，可以不用拔掉 SD 卡来更新。即利用网口或者串口连接到 PC 上动态更新图片，甚至可以可以通过无线用手机发送图片到数码相册。

总之，通过此次作品的制作，学到了一些开发工具的简单使用和完成了一些基本功能，在以后的时间中会逐步完成剩余的可以改进的功能。

谢 辞

毕业设计已经接近尾声，我的大学生涯也马上结束了，回顾大学四年，还算过得充实，感谢我的母校桂林电子科技大学提供给了我们一个这么好的学习平台，感谢桂林电子科技大学的校领导，院领导对我们衷心的栽培，感谢自动化全体教师全心的培养！

从开始拿到毕设题目起，自己脑袋里没有什么概念，因为以前没有接触过图片处理方面的信息，且开发平台 FPGA 只懂点皮毛。SOPC_Builder 和 Nios II 处理器更是完全没听说过，所以当时完全没底。因此自己从网上慢慢开始找资料学习，开始研究 JTAG、SRAM、SDRAM、FLASH、SD 卡、TFT、SOPC_Builde、Nios II 处理器等等。慢慢开始有概念了。在动手做毕设的过程中也走了不少弯路，比如对 SD 卡的操作一开始打算自己用 Verilog 描述出来一个 ip 核，但后面发现这个工作量太大，且还不如直接在 NiosII 中用 io 口来驱动，因为 Nios II 跑在 100Mhz 的频率下，而 SD 卡最大速率就 25Mhz，所以不影响其速度。另外在做毕设的过程中一开始用的开发板由于做板子有问题，在调试代码时出现很多莫名其妙的 bug，系统运行时好时坏。花了不少时间，后面改用友晶的 DE1 就没有了那些现象。

总之，通过本次毕设，我学会了很多知识，更重要的是学会了很多学习方法。在这里，衷心感谢毕设老师的指导和同学的帮助，谢谢！

参考文献

[1] 潘松，黄继业.EDA 技术实用教程[M].北京:北京科学出版社,2005.

[2] 李洪伟,袁斯华.基于 QuartusII 的 FPGA/CPLD 设计[M].北京:电子工业出版社,2006.

[3] EDA 先锋工作室. Altera FPGA/CPLD 设计(高级篇) [M]. 北京：人民邮电出版社,2005.

[4] EDA 先锋工作室. Altera FPGA/CPLD 设计(基础篇) [M].北京： 人民邮电出版社,2005.

[5] 周立功. SOPC 嵌入式系统基础教程[M]. 北京:北京航空航天大学出版社, 2006.

[6] 周立功. SOPC 嵌入式系统实验教程（一）[M]. 北京：北京航空航天大学出版社, 2006.

[7] 侯伯亨,顾新.硬件描述语言与数字电路设计[M].西安:西安电子科技大学出版社,2004.

[8] 洪城，邓锐，叶永鑫.基于 NiosII 的多功能数字相册[D].Altera,2007.

[9] 江国强.EDA 技术与应用[M].北京：电子工业出版社，2004.

[10] Cyclone Device Handbook,Volume 1.ALTERA Corporation.August 2005.

附 录

在 SD 卡上移植 FAT32 文件系统关于 SD 卡初始化部分的代码：

```
#define CMD0       (0)             /* GO_IDLE_STATE */
#define CMD1       (1)             /* SEND_OP_COND */
#define ACMD41     (0x80+41)       /* SEND_OP_COND (SDC) */
#define CMD8       (8)             /* SEND_IF_COND */
#define CMD9       (9)             /* SEND_CSD */
#define CMD10      (10)            /* SEND_CID */
#define CMD12      (12)            /* STOP_TRANSMISSION */
#define ACMD13     (0x80+13)       /* SD_STATUS (SDC) */
#define CMD16      (16)            /* SET_BLOCKLEN */
#define CMD17      (17)            /* READ_SINGLE_BLOCK */
#define CMD18      (18)            /* READ_MULTIPLE_BLOCK */
#define CMD23      (23)            /* SET_BLOCK_COUNT */
#define ACMD23     (0x80+23)       /* SET_WR_BLK_ERASE_COUNT (SDC) */
#define CMD24      (24)            /* WRITE_BLOCK */
#define CMD25      (25)            /* WRITE_MULTIPLE_BLOCK */
#define CMD41      (41)            /* SEND_OP_COND (ACMD) */
#define CMD55      (55)            /* APP_CMD */
#define CMD58      (58)            /* READ_OCR */
/* Card type flags (CardType) */
#define CT_MMC          0x01        /* MMC ver 3 */
#define CT_SD1          0x02        /* SD ver 1 */
#define CT_SD2          0x04        /* SD ver 2 */
#define CT_SDC          (CT_SD1|CT_SD2) /* SD */
#define CT_BLOCK        0x08        /* Block addressing */
static
DSTATUS Stat = STA_NOINIT;    /* Disk status */
static
BYTE CardType;                /* b0:MMC, b1:SDv1, b2:SDv2, b3:Block addressing */
/*init_port()*/
static void init_port (void)
```

```
{
   IOWR(DO_SPI_BASE, 1, 0);
   IOWR(DI_SPI_BASE, 1, 1);
   IOWR(CS_SPI_BASE, 1, 1);
   IOWR(CLK_SPI_BASE, 1, 1);
}
static
void xmit_mmc (
      const BYTE* buff,     /* Data to be sent */
      UINT bc                    /* Number of bytes to send */
)
{
      BYTE d;
      do {
            d = *buff++;       /* Get a byte to be sent */
            if (d & 0x80) DI_H(); else DI_L();    /* bit7 */
            CK_H(); CK_L();
            if (d & 0x40) DI_H(); else DI_L();    /* bit6 */
            CK_H(); CK_L();
            if (d & 0x20) DI_H(); else DI_L();    /* bit5 */
            CK_H(); CK_L();
            if (d & 0x10) DI_H(); else DI_L();    /* bit4 */
            CK_H(); CK_L();
            if (d & 0x08) DI_H(); else DI_L();    /* bit3 */
            CK_H(); CK_L();
            if (d & 0x04) DI_H(); else DI_L();    /* bit2 */
            CK_H(); CK_L();
            if (d & 0x02) DI_H(); else DI_L();    /* bit1 */
            CK_H(); CK_L();
            if (d & 0x01) DI_H(); else DI_L();    /* bit0 */
            CK_H(); CK_L();
      } while (--bc);
}
static
void rcvr_mmc (
```

```
    BYTE *buff, /* Pointer to read buffer */
    UINT bc         /* Number of bytes to receive */
)
{
    BYTE r;
    DI_H(); /* Send 0xFF */
    do {
        r = 0;    if (DO) r++;    /* bit7 */
        CK_H(); CK_L();
        r <<= 1; if (DO) r++;    /* bit6 */
        CK_H(); CK_L();
        r <<= 1; if (DO) r++;    /* bit5 */
        CK_H(); CK_L();
        r <<= 1; if (DO) r++;    /* bit4 */
        CK_H(); CK_L();
        r <<= 1; if (DO) r++;    /* bit3 */
        CK_H(); CK_L();
        r <<= 1; if (DO) r++;    /* bit2 */
        CK_H(); CK_L();
        r <<= 1; if (DO) r++;    /* bit1 */
        CK_H(); CK_L();
        r <<= 1; if (DO) r++;    /* bit0 */
        CK_H(); CK_L();
        *buff++ = r;                    /* Store a received byte */
    } while (--bc);
}
static
int wait_ready (void)    /* 1:OK, 0:Timeout */
{
    BYTE d;
    UINT tmr;
    for (tmr = 5000; tmr; tmr--) {    /* Wait for ready in timeout of 500ms */
        rcvr_mmc(&d, 1);
        if (d == 0xFF) return 1;
        dly_us(100);
```

```
	}


	return 0;
}
static
void deselect (void)
{
	BYTE d;
	CS_H();
	rcvr_mmc(&d, 1);
}
static
int select (void)	/* 1:OK, 0:Timeout */
{
	CS_L();
	if (!wait_ready()) {
		deselect();
		return 0;
	}
	return 1;
}
static
int rcvr_datablock (	/* 1:OK, 0:Failed */
	BYTE *buff,		/* Data buffer to store received data */
	UINT btr		/* Byte count */
)
{
	BYTE d[2];
	UINT tmr;
	for (tmr = 1000; tmr; tmr--) {	/* Wait for data packet in timeout of 100ms */
		rcvr_mmc(d, 1);
		if (d[0] != 0xFF) break;
		dly_us(100);
	}
	if (d[0] != 0xFE) return 0;		/* If not valid data token, retutn with error */
```

```
    rcvr_mmc(buff, btr);                /* Receive the data block into buffer */
    rcvr_mmc(d, 2);                     /* Discard CRC */
    return 1;                           /* Return with success */
}
static
int xmit_datablock (    /* 1:OK, 0:Failed */
    const BYTE *buff,   /* 512 byte data block to be transmitted */
    BYTE token          /* Data/Stop token */
)
{
    BYTE d[2];
    if (!wait_ready()) return 0;
    d[0] = token;
    xmit_mmc(d, 1);                     /* Xmit a token */
    if (token != 0xFD) {                /* Is it data token? */
        xmit_mmc(buff, 512);            /* Xmit the 512 byte data block to MMC */
        rcvr_mmc(d, 2);                 /* Dummy CRC (FF,FF) */
        rcvr_mmc(d, 1);                 /* Receive data response */
        if ((d[0] & 0x1F) != 0x05)      /* If not accepted, return with error */
            return 0;
    }
    return 1;
}
static
BYTE send_cmd (         /* Returns command response (bit7==1:Send failed)*/
    BYTE cmd,           /* Command byte */
    DWORD arg           /* Argument */
)
{
    BYTE n, d, buf[6];
    if (cmd & 0x80) {   /* ACMD<n> is the command sequense of CMD55-CMD<n> */
        cmd &= 0x7F;
        n = send_cmd(CMD55, 0);
        if (n > 1) return n;
    }
```

```
    /* Select the card and wait for ready */
    deselect();
    if (!select()) return 0xFF;
    /* Send a command packet */
    buf[0] = 0x40 | cmd;                /* Start + Command index */
    buf[1] = (BYTE)(arg >> 24);         /* Argument[31..24] */
    buf[2] = (BYTE)(arg >> 16);         /* Argument[23..16] */
    buf[3] = (BYTE)(arg >> 8);          /* Argument[15..8] */
    buf[4] = (BYTE)arg;                 /* Argument[7..0] */
    n = 0x01;                           /* Dummy CRC + Stop */
    if (cmd == CMD0) n = 0x95;          /* (valid CRC for CMD0(0)) */
    if (cmd == CMD8) n = 0x87;          /* (valid CRC for CMD8(0x1AA)) */
    buf[5] = n;
    xmit_mmc(buf, 6);
    /* Receive command response */
    if (cmd == CMD12) rcvr_mmc(&d, 1);  /* Skip a stuff byte when stop reading */
    n = 10;                             /* Wait for a valid response in timeout of 10 attempts */
    do
        rcvr_mmc(&d, 1);
    while ((d & 0x80) && --n);

    return d;               /* Return with the response value */
}
DSTATUS disk_status (
    BYTE drv                /* Drive number (0) */
)
{
    DSTATUS s = Stat;
    if (drv || !INS) {
        s = STA_NODISK | STA_NOINIT;
    } else {
        s &= ~STA_NODISK;
        if (WP)
            s |= STA_PROTECT;
        else
```

```
            s &= ~STA_PROTECT;
        }
        Stat = s;
        return s;
}
DSTATUS disk_initialize (
        BYTE drv            /* Physical drive nmuber (0) */
)
{
        BYTE n, ty, cmd, buf[4];
        UINT tmr;
        DSTATUS s;
        init_port();                    /* Initialize control port */
        s = disk_status(drv);           /* Check if card is in the socket */
        if (s & STA_NODISK) return s;
        CS_H();
        for (n = 10; n; n--) rcvr_mmc(buf, 1);    /* 80 dummy clocks */
        ty = 0;
        if (send_cmd(CMD0, 0) == 1) {                   /* Enter Idle state */
                if (send_cmd(CMD8, 0x1AA) == 1) {       /* SDv2? */
                        rcvr_mmc(buf, 4);                               /* Get trailing return value of R7 resp */
                        if (buf[2] == 0x01 && buf[3] == 0xAA) {         /* The card can work at vdd range of
 2.7-3.6V */
                                for (tmr = 1000; tmr; tmr--) {                  /* Wait for leaving idle state (ACMD41 with
 HCS bit) */
                                        if (send_cmd(ACMD41, 1UL << 30) == 0) break;
                                        dly_us(1000);
                                }
                                if (tmr && send_cmd(CMD58, 0) == 0) {       /* Check CCS bit in the OCR */
                                        rcvr_mmc(buf, 4);
                                        ty = (buf[0] & 0x40) ? CT_SD2 | CT_BLOCK : CT_SD2;    /* SDv2 */
                                }
                        }
                } else {                                        /* SDv1 or MMCv3 */
                        if (send_cmd(ACMD41, 0) <= 1)     {
```

```
                ty = CT_SD1; cmd = ACMD41;    /* SDv1 */
            } else {
                ty = CT_MMC; cmd = CMD1;      /* MMCv3 */
            }
            for (tmr = 1000; tmr; tmr--) {            /* Wait for leaving idle state */
                if (send_cmd(ACMD41, 0) == 0) break;
                dly_us(1000);
            }
            if (!tmr || send_cmd(CMD16, 512) != 0)    /* Set R/W block length to 512 */
                ty = 0;
        }
    }
    CardType = ty;
    if (ty)         /* Initialization succeded */
        s &= ~STA_NOINIT;
    else            /* Initialization failed */
        s |= STA_NOINIT;
    Stat = s;
    deselect();
    return s;
}
DRESULT disk_read (
    BYTE drv,               /* Physical drive nmuber (0) */
    BYTE *buff,             /* Pointer to the data buffer to store read data */
    DWORD sector,           /* Start sector number (LBA) */
    BYTE count              /* Sector count (1..128) */
)
{
    DSTATUS s;
    s = disk_status(drv);
    if (s & STA_NOINIT) return RES_NOTRDY;
    if (!count) return RES_PARERR;
    if (!(CardType & CT_BLOCK)) sector *= 512;    /* Convert LBA to byte address if needed */

    if (count == 1) {      /* Single block read */
```

```
        if ((send_cmd(CMD17, sector) == 0)    /* READ_SINGLE_BLOCK */
            && rcvr_datablock(buff, 512))
            count = 0;
    }
    else {                      /* Multiple block read */
        if (send_cmd(CMD18, sector) == 0) { /* READ_MULTIPLE_BLOCK */
            do {
                if (!rcvr_datablock(buff, 512)) break;
                buff += 512;
            } while (--count);
            send_cmd(CMD12, 0);                 /* STOP_TRANSMISSION */
        }
    }
    deselect();
    return count ? RES_ERROR : RES_OK;
}
DRESULT disk_write (
    BYTE drv,               /* Physical drive nmuber (0) */
    const BYTE *buff,       /* Pointer to the data to be written */
    DWORD sector,           /* Start sector number (LBA) */
    BYTE count              /* Sector count (1..128) */
)
{
    DSTATUS s;
    s = disk_status(drv);
    if (s & STA_NOINIT) return RES_NOTRDY;
    if (s & STA_PROTECT) return RES_WRPRT;
    if (!count) return RES_PARERR;
    if (!(CardType & CT_BLOCK)) sector *= 512;    /* Convert LBA to byte address if needed */
    if (count == 1) {       /* Single block write */
        if ((send_cmd(CMD24, sector) == 0)    /* WRITE_BLOCK */
            && xmit_datablock(buff, 0xFE))
            count = 0;
    }
    else {                      /* Multiple block write */
```

```
        if (CardType & CT_SDC) send_cmd(ACMD23, count);
        if (send_cmd(CMD25, sector) == 0) { /* WRITE_MULTIPLE_BLOCK */
            do {
                if (!xmit_datablock(buff, 0xFC)) break;
                buff += 512;
            } while (--count);
            if (!xmit_datablock(0, 0xFD))   /* STOP_TRAN token */
                count = 1;
        }
    }
    deselect();
    return count ? RES_ERROR : RES_OK;
}
/*-----------------------------------------------------------------------*/
/* Miscellaneous Functions                                               */
/*-----------------------------------------------------------------------*/

DRESULT disk_ioctl (
    BYTE drv,           /* Physical drive nmuber (0) */
    BYTE ctrl,          /* Control code */
    void *buff          /* Buffer to send/receive control data */
)
{
    DRESULT res;
    BYTE n, csd[16];
    WORD cs;
    if (disk_status(drv) & STA_NOINIT)                  /* Check if card is in the socket */
        return RES_NOTRDY;
    res = RES_ERROR;
    switch (ctrl) {
        case CTRL_SYNC :            /* Make sure that no pending write process */
            if (select()) {
                deselect();
                res = RES_OK;
            }
```

```
            break;
        case GET_SECTOR_COUNT : /* Get number of sectors on the disk (DWORD) */
            if ((send_cmd(CMD9, 0) == 0) && rcvr_datablock(csd, 16)) {
                if ((csd[0] >> 6) == 1) {      /* SDC ver 2.00 */
                    cs= csd[9] + ((WORD)csd[8] << 8) + 1;
                    *(DWORD*)buff = (DWORD)cs << 10;
                } else {                                /* SDC ver 1.XX or MMC */
                    n = (csd[5] & 15) + ((csd[10] & 128) >> 7) + ((csd[9] & 3) << 1) + 2;
                    cs = (csd[8] >> 6) + ((WORD)csd[7] << 2) + ((WORD)(csd[6] & 3) << 10) + 1;
                    *(DWORD*)buff = (DWORD)cs << (n - 9);
                }
                res = RES_OK;
            }
            break;
        case GET_BLOCK_SIZE :      /* Get erase block size in unit of sector (DWORD) */
            *(DWORD*)buff = 128;
            res = RES_OK;
            break;
        default:
            res = RES_PARERR;
    }
    deselect();
    return res;
}
```