第1章　DCMI—OV2640 摄像头

本章参考资料：《STM32F4xx 参考手册》、《STM32F4xx 规格书》、库帮助文档《stm32f4xx_dsp_stdperiph_lib_um.chm》。

关于开发板配套的 OV2640 摄像头参数可查阅《ov2640datasheet》配套资料获知。

STM32F4 芯片具有浮点运算单元，适合对图像信息使用 DSP 进行基本的图像处理，其处理速度比传统的 8、16 位机快得多，而且它还具有与摄像头通讯的专用 DCMI 接口，所以使用它驱动摄像头采集图像信息并进行基本的加工处理非常适合。本章讲解如何使用 STM32 驱动 OV2640 型号的摄像头。

1.1 摄像头简介

在各类信息中，图像含有最丰富的信息，作为机器视觉领域的核心部件，摄像头被广泛地应用在安防、探险以及车牌检测等场合。摄像头按输出信号的类型来看可以分为数字摄像头和模拟摄像头，按照摄像头图像传感器材料构成来看可以分为 CCD 和 CMOS。现在智能手机的摄像头绝大部分都是 CMOS 类型的数字摄像头。

1.1.1 数字摄像头跟模拟摄像头区别

- 输出信号类型

  数字摄像头输出信号为数字信号，模拟摄像头输出信号为标准的模拟信号。

- 接口类型

  数字摄像头有 USB 接口(比如常见的 PC 端免驱摄像头)、IEE1394 火线接口(由苹果公司领导的开发联盟开发的一种高速度传送接口，数据传输率高达 800Mbps)、千兆网接口（网络摄像头）。模拟摄像头多采用 AV 视频端子（信号线+地线）或 S-VIDEO（即莲花头--SUPER VIDEO，是一种五芯的接口，由两路视频亮度信号、两路视频色度信号和一路公共屏蔽地线共五条芯线组成）。

- 分辨率

  模拟摄像头的感光器件，其像素指标一般维持在 752(H)*582(V)左右的水平，像素数一般情况下维持在 41 万左右。现在的数字摄像头分辨率一般从数十万到数千万。但这并不能说明数字摄像头的成像分辨率就比模拟摄像头的高，原因在于模拟摄像头输出的是模拟视频信号，一般直接输入至电视或监视器，其感光器件的分辨率与电视信号的扫描数呈一定的换算关系，图像的显示介质已经确定，因此模拟摄像头的感光器件分辨率不是不能做高，而是依据于实际情况没必要做这么高。

1.1.2 CCD 与 CMOS 的区别

摄像头的图像传感器 CCD 与 CMOS 传感器主要区别如下：

- 成像材料

CCD 与 CMOS 的名称跟它们成像使用的材料有关，CCD 是“电荷耦合器件”(Charge Coupled Device)的简称，而 CMOS 是“互补金属氧化物半导体”(Complementary Metal Oxide Semiconductor)的简称。

- 功耗

  由于 CCD 的像素由 MOS 电容构成，读取电荷信号时需使用电压相当大(至少 12V)的二相或三相或四相时序脉冲信号，才能有效地传输电荷。因此 CCD 的取像系统除了要有多个电源外，其外设电路也会消耗相当大的功率。有的 CCD 取像系统需消耗 2~5W 的功率。而 CMOS 光电传感器件只需使用一个单电源 5V 或 3V，耗电量非常小，仅为 CCD 的 1/8~1/10，有的 CMOS 取像系统只消耗 20~50mW 的功率。

- 成像质量

  CCD 传感器件制作技术起步早，技术成熟，采用 PN 结或二氧化硅(sio2)隔离层隔离噪声，所以噪声低，成像质量好。与 CCD 相比，CMOS 的主要缺点是噪声高及灵敏度低，不过现在随着 CMOS 电路消噪技术的不断发展，为生产高密度优质的 CMOS 传感器件提供了良好的条件，现在的 CMOS 传感器已经占领了大部分的市场，主流的单反相机、智能手机都已普遍采用 CMOS 传感器。

1.2 OV2640 摄像头

本章主要讲解实验板配套的摄像头，它的实物见图 1-1，该摄像头主要由镜头、图像传感器、板载电路及下方的信号引脚组成。

图 1-1 实验板配套的 OV2640 摄像头

镜头部件包含一个镜头座和一个可旋转调节距离的凸透镜，通过旋转可以调节焦距，正常使用时，镜头座覆盖在电路板上遮光，光线只能经过镜头传输到正中央的图像传感器，它采集光线信号，然后把采集得的数据通过下方的信号引脚输出数据到外部器件。

1.2.1 OV2640 传感器简介

图像传感器是摄像头的核心部件，上述摄像头中的图像传感器是一款型号为 OV2640 的 CMOS 类型数字图像传感器。该传感器支持输出最大为 200 万像素的图像 (1600x1200 分辨率)，支持使用 VGA 时序输出图像数据，输出图像的数据格式支持 YUV(422/420)、YCbCr422、RGB565 以及 JPEG 格式，若直接输出 JPEG 格式的图像时可大大减少数据量，方便网络传输。它还可以对采集得的图像进行补偿，支持伽玛曲线、白平衡、饱和度、色度等基础处理。根据不同的分辨率配置，传感器输出图像数据的帧率从 15-60 帧可调，工作时功率在 125mW-140mW 之间。

1.2.2 OV2640 引脚及功能框图

OV2640 传感器采用 BGA 封装，它的前端是采光窗口，引脚都在背面引出，引脚的分布见图 1-2。

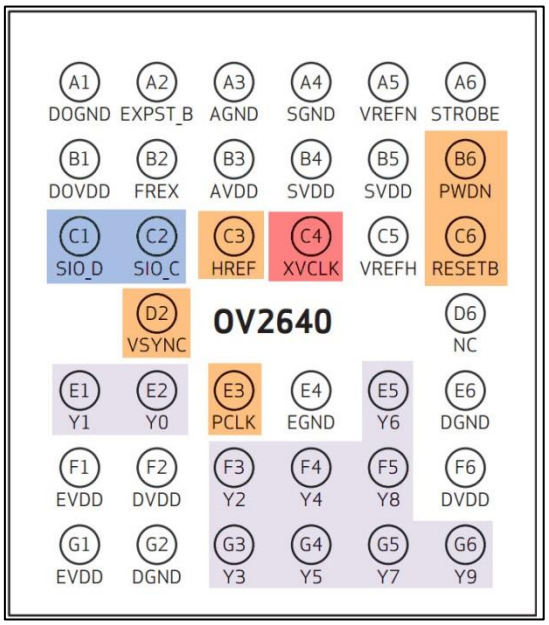

图 1-2 OV2640 传感器引脚分布图

图中的非彩色部分是电源相关的引脚，彩色部分是主要的信号引脚，介绍如下表 1-1。

表 1-1 OV2640 管脚

管脚名称	管脚类型	管脚描述
SIO_C	输入	SCCB 总线的时钟线，可类比 I2C 的 SCL
SIO_D	I/O	SCCB 总线的数据线，可类比 I2C 的 SDA
RESETB	输入	系统复位管脚，低电平有效
PWDN	输入	掉电/省电模式，高电平有效
HREF	输出	行同步信号
VSYNC	输出	帧同步信号
PCLK	输出	像素同步时钟输出信号
XCLK	输入	外部时钟输入端口，可接外部晶振
Y0…Y9	输出	像素数据输出端口

下面我们配合图 1-3 中的 OV2640 功能框图讲解这些信号引脚。

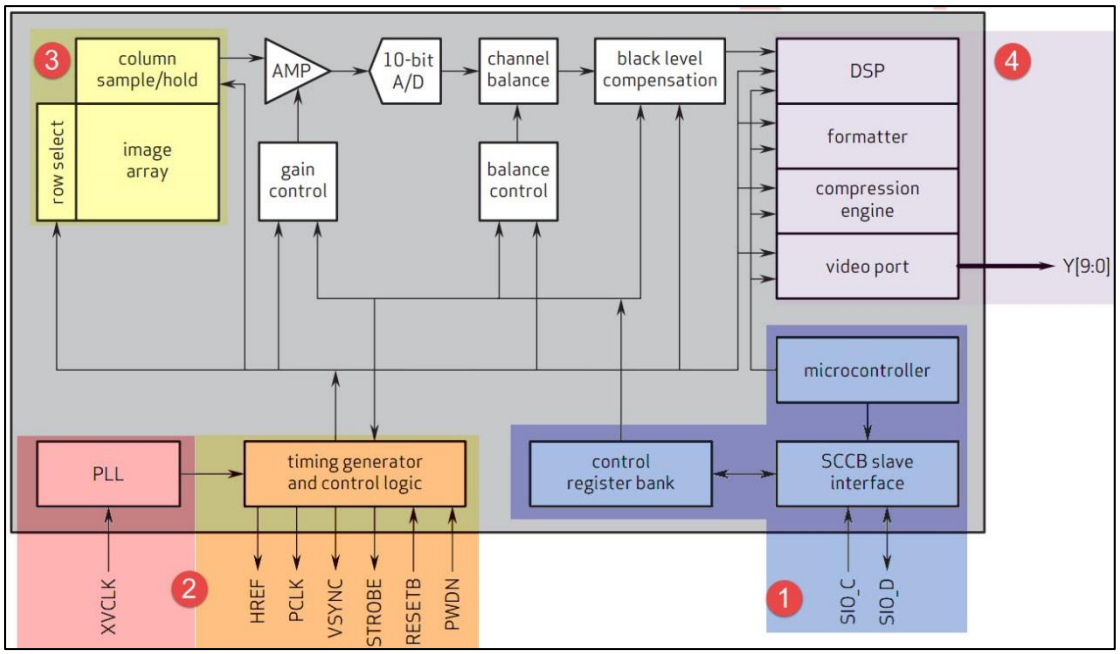

图 1-3 OV2640 功能框图

(1) 控制寄存器

标号①处的是 OV2640 的控制寄存器，它根据这些寄存器配置的参数来运行，而这些参数是由外部控制器通过 SIO_C 和 SIO_D 引脚写入的，SIO_C 与 SIO_D 使用的通讯协议 SCCB 跟 I2C 十分类似，在 STM32 中我们完全可以直接用 I2C 硬件外设来控制。

(2) 通信、控制信号及时钟

标号②处包含了 OV2640 的通信、控制信号及外部时钟，其中 PCLK、HREF 及 VSYNC 分别是像素同步时钟、行同步信号以及帧同步信号，这与液晶屏控制中的信号是很类似的。RESETB 引脚为低电平时，用于复位整个传感器芯片，PWDN 用于控制芯片进入低功耗模式。注意最后的一个 XCLK 引脚，它跟 PCLK 是完全不同的，XCLK 是用于驱动整个传感器芯片的时钟信号，是外部输入到 OV2640 的信号；而 PCLK 是 OV2640 输出数据时的同步信号，它是由 OV2640 输出的信号。XCLK 可以外接晶振或由外部控制器提供，若要类比 XCLK 之于 OV2640 就相当于 HSE 时钟输入引脚与 STM32 芯片的关系，PCLK 引脚可类比 STM32 的 I2C 外设的 SCL 引脚。

(3) 感光矩阵

标号③处的是感光矩阵，光信号在这里转化成电信号，经过各种处理，这些信号存储成由一个个像素点表示的数字图像。

(4) 数据输出信号

标号④处包含了 DSP 处理单元，它会根据控制寄存器的配置做一些基本的图像处理运算。这部分还包含了图像格式转换单元及压缩单元，转换出的数据最终通过

Y0-Y9 引脚输出，一般来说我们使用 8 根据数据线来传输，这时仅使用 Y2-Y9 引脚，OV2640 与外部器件的连接方式见图 1-4。

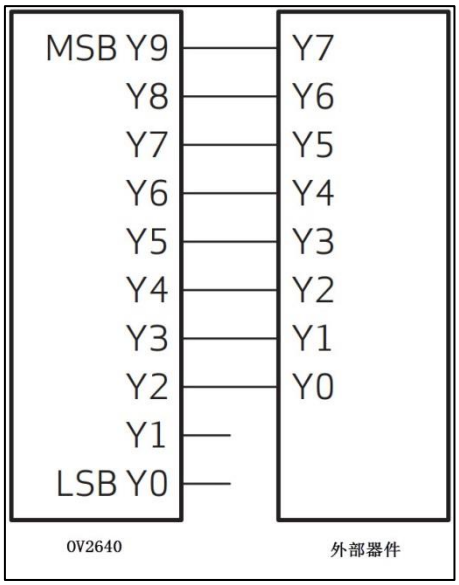

图 1-4 8 位数据线接法

1.2.3 SCCB 时序

外部控制器对 OV2640 寄存器的配置参数是通过 SCCB 总线传输过去的，而 SCCB 总线跟 I2C 十分类似，所以在 STM32 驱动中我们直接使用片上 I2C 外设与它通讯。SCCB 与标准的 I2C 协议的区别是它每次传输只能写入或读取一个字节的数据，而 I2C 协议是支持突发读写的，即在一次传输中可以写入多个字节的数据(EEPROM 中的页写入时序即突发写)。关于 SCCB 协议的完整内容可查看配套资料里的《SCCB 协议》文档，下面我们简单介绍下。

SCCB 的起始、停止信号及数据有效性

SCCB 的起始信号、停止信号及数据有效性与 I2C 完全一样，见图 1-5 及图 1-6。

- 起始信号：在 SIO_C 为高电平时，SIO_D 出现一个下降沿，则 SCCB 开始传输。
- 停止信号：在 SIO_C 为高电平时，SIO_D 出现一个上升沿，则 SCCB 停止传输。
- 数据有效性：除了开始和停止状态，在数据传输过程中，当 SIO_C 为高电平时，必须保证 SIO_D 上的数据稳定，也就是说，SIO_D 上的电平变换只能发生在 SIO_C 为低电平的时候，SIO_D 的信号在 SIO_C 为高电平时被采集。

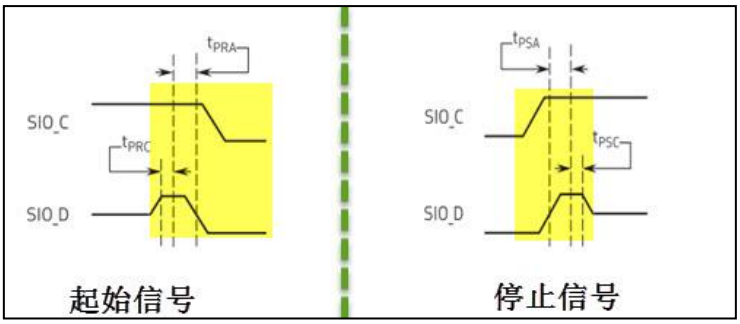

图 1-5 SCCB 停止信号

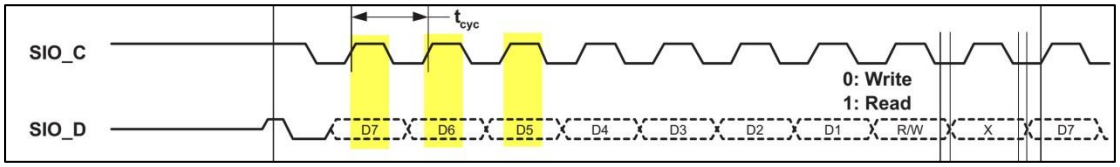

图 1-6 SCCB 的数据有效性

SCCB 数据读写过程

在 SCCB 协议中定义的读写操作与 I2C 也是一样的，只是换了一种说法。它定义了两种写操作，即三步写操作和两步写操作。三步写操作可向从设备的一个目的寄存器中写入数据，见图 1-7。在三步写操作中，第一阶段发送从设备的 ID 地址+W 标志(等于 I2C 的设备地址：7 位设备地址+读写方向标志)，第二阶段发送从设备目标寄存器的 8 位地址，第三阶段发送要写入寄存器的 8 位数据。图中的“X”数据位可写入 1 或 0，对通讯无影响。

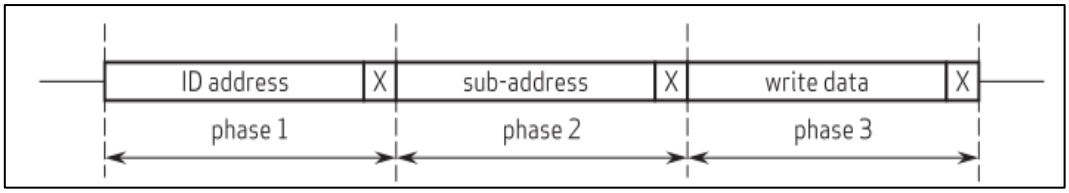

图 1-7 SCCB 的三步写操作

而两步写操作没有第三阶段，即只向从器件传输了设备 ID+W 标志和目的寄存器的地址，见图 1-8。两步写操作是用来配合后面的读寄存器数据操作的，它与读操作一起使用，实现 I2C 的复合过程。

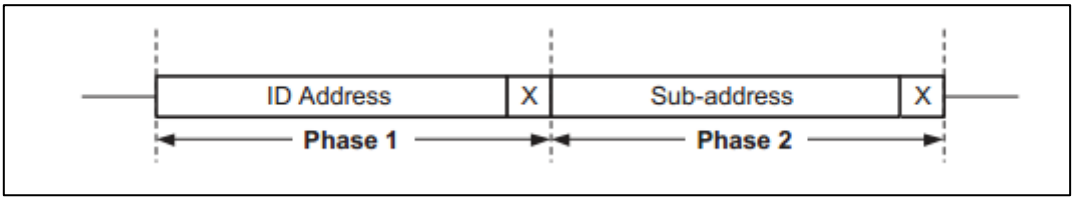

图 1-8 SCCB 的两步写操作

两步读操作，它用于读取从设备目的寄存器中的数据，见图 1-9。在第一阶段中发送从设备的设备 ID+R 标志(设备地址+读方向标志)和自由位，在第二阶段中读取寄存器中的 8 位数据和写 NA 位(非应答信号)。由于两步读操作没有确定目的寄存器的地址，所以在读操作前，必需有一个两步写操作，以提供读操作中的寄存器地址。

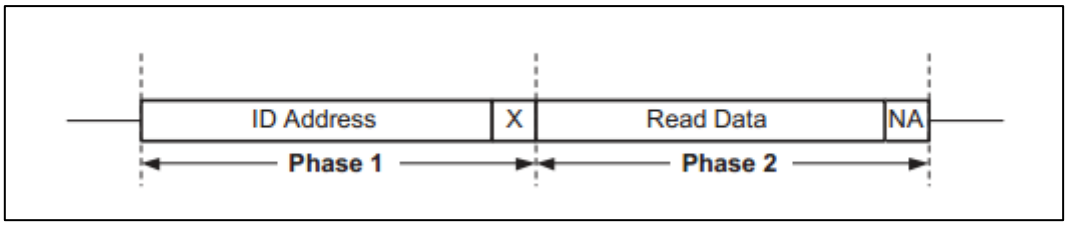

图 1-9 SCCB 的两步读操作

可以看到，以上介绍的 SCCB 特性都与 I2C 无区别，而 I2C 比 SCCB 还多出了突发读写的功能，所以 SCCB 可以看作是 I2C 的子集，我们完全可以使用 STM32 的 I2C 外设来与 OV2640 进行 SCCB 通讯。

1.2.4 OV2640 的寄存器

控制 OV2640 涉及到它很多的寄存器，可直接查询《ov2640datasheet》了解，通过这些寄存器的配置，可以控制它输出图像的分辨率大小、图像格式及图像方向等。要注意的是 OV2640 有两组寄存器，这两组寄存器有部分地址重合，通过设置地址为 0xFF 的 RA_DLMT 寄存器可以切换寄存器组，当 RA_DLMT 寄存器为 0 时，通过 SCCB 发送的寄存器地址在 DSP 相关的寄存器组寻址，见图 1-10；RA_DLMT 寄存器为 1 时，在 Sensor 相关的寄存器组寻址，图 1-10。

Table 12 Device Control Register List (when 0xFF = 00) (Sheet 1 of 4)

Address (Hex)	Register Name	Default (Hex)	R/W	Description
00-04	RSVD	XX	–	Reserved
05	R_BYPASS	0x1	RW	Bypass DSP Bit[7:1]: Reserved Bit[0]: Bypass DSP select 0: DSP 1: Bypass DSP, sensor out directly
06-43	RSVD	XX	–	Reserved
44	Qs	0C	RW	Quantization Scale Factor
45-4F	RSVD	XX	–	Reserved

图 1-10 0xFF=0 时的 DSP 相关寄存器说明(部分)

Table 13 Device Control Register List (when 0xFF = 01) (Sheet 1 of 6)

Address (Hex)	Register Name	Default (Hex)	R/W	Description
00	GAIN	00	RW	AGC Gain Control LSBs Bit[7:0]: Gain setting • Range: 1x to 32x Gain = (Bit[7]+1) x (Bit[6]+1) x (Bit[5]+1) x (Bit[4]+1) x (1+Bit[3:0]/16) ***Note:*** *Set COM8[2] = 0 to disable AGC.*
01-02	RSVD	XX	–	Reserved
03	COM1	0F (UXGA) 0A (SVGA), 06 (CIF)	RW	Common Control 1 Bit[7:6]: Dummy frame control 00: Reserved 01: Allow 1 dummy frame 10: Allow 3 dummy frames 11: Allow 7 dummy frames Bit[5:4]: Reserved Bit[3:2]: Vertical window end line control 2 LSBs (8 MSBs in VEND[7:0] (0x1A)) Bit[1:0]: Vertical window start line control 2 LSBs (8 MSBs in VSTRT[7:0] (0x19))

图 1-11 0xFF=1 时的 Sensor 相关寄存器说明(部分)

官方还提供了一个《OV2640_Camera_app》的文档，它针对不同的配置需求，提供了配置范例，见图 1-12。其中 write_SCCB 是一个利用 SCCB 向寄存器写入数据的函数，第一个参数为要写入的寄存器的地址，第二个参数为要写入的内容。

```
3.2 SVGA Preview, 15fps, 24 Mhz input clock

SCCB_salve_Address = 0x60;
write_SCCB(0xff, 0x01);
write_SCCB(0x11, 0x01);
write_SCCB(0x12, 0x40);
write_SCCB(0x2a, 0x00);
write_SCCB(0x2b, 0x00);
write_SCCB(0x46, 0x00);
write_SCCB(0x47, 0x00);
write_SCCB(0x3d, 0x38);
```

图 1-12 调节帧率的寄存器配置范例

1.2.5 像素数据输出时序

主控器控制 OV2640 时采用 SCCB 协议读写其寄存器，而它输出图像时则使用 VGA 时序(还可用 SVGA、UXGA，这些时序都差不多)，这跟控制液晶屏输入图像时很类似。OV2640 输出图像时，一帧帧地输出，在帧内的数据一般从左到右，从上到下，一个像素一个像素地输出(也可通过寄存器修改方向)，见图 1-13。

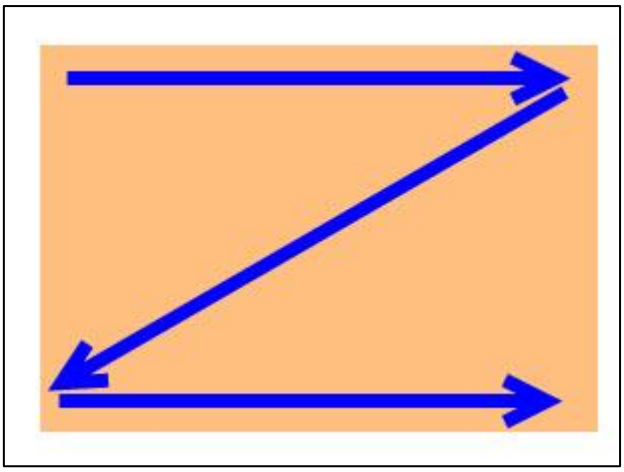

图 1-13 摄像头数据输出

例如，见图 1-14 和图 1-15，若我们使用 Y2-Y9 数据线，图像格式设置为 RGB565，进行数据输出时，Y2-Y9 数据线会在 1 个像素同步时钟 PCLK 的驱动下发送 1 字节的数据信号，所以 2 个 PCLK 时钟可发送 1 个 RGB565 格式的像素数据。像素数据依次传输，每传输完一行数据时，行同步信号 HREF 会输出一个电平跳变信号，每传输完一帧图像时，VSYNC 会输出一个电平跳变信号。

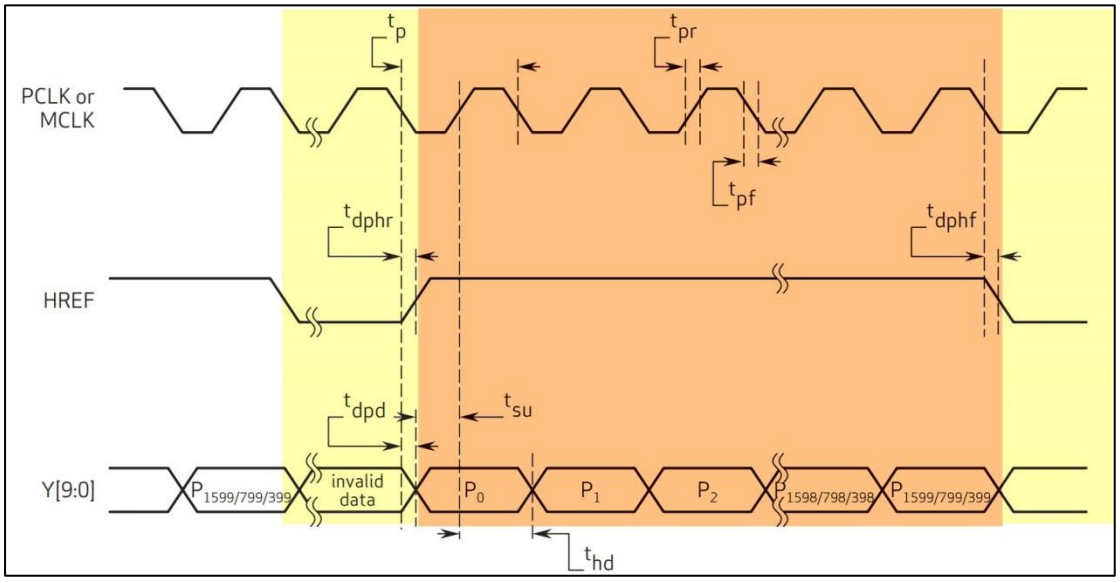

图 1-14 像素同步时序

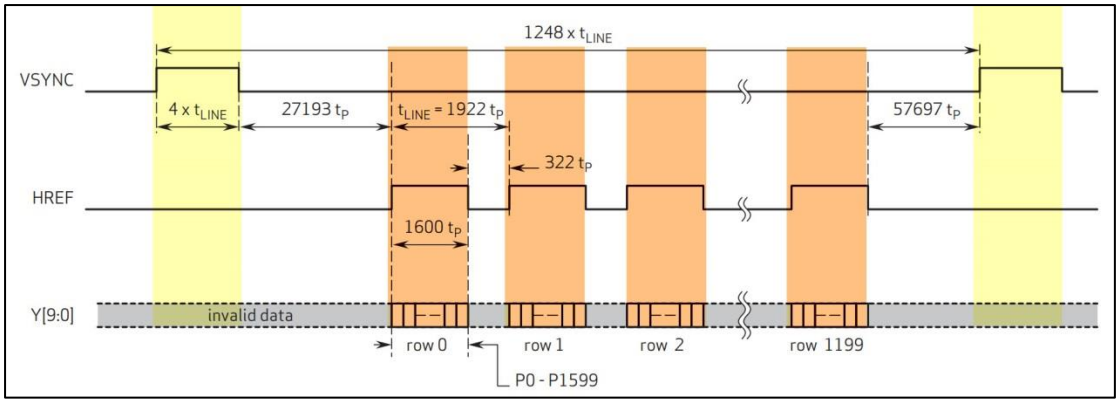

图 1-15 帧图像同步时序

1.3 STM32 的 DCMI 接口简介

STM32F4 系列的控制器包含了 DCMI 数字摄像头接口(Digital camera Interface)，它支持使用上述类似 VGA 的时序获取图像数据流，支持原始的按行、帧格式来组织的图像数据，如 YUV、RGB，也支持接收 JPEG 格式压缩的数据流。接收数据时，主要使用 HSYNC 及 VSYNC 信号来同步。

1.3.1 DCMI 整体框图

STM32 的 DCMI 接口整体框图见图 1-16。

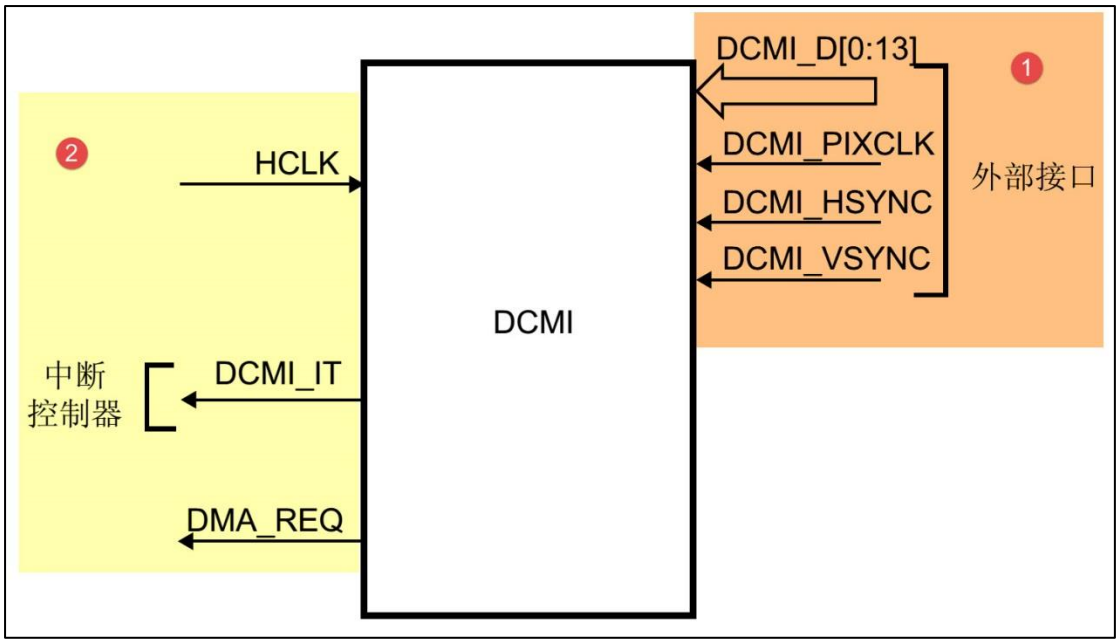

图 1-16 DCMI 接口整体框图

外部接口及时序

上图标号①处的是 DCMI 向外部引出的信号线。DCMI 提供的外部接口的方向都是输入的，接口的各个信号线说明见表 1-2。

表 1-2 DCMI 的信号线说明

引脚名称	说明
DCMI_D[0:13]	数据线
DCMI_PIXCLK	像素同步时钟
DCMI_HSYNC	行同步信号(水平同步信号)
DCMI_VSYNC	帧同步信号(垂直同步信号)

其中 DCMI_D 数据线的数量可选 8、10、12 或 14 位，各个同步信号的有效极性都可编程控制。它使用的通讯时序与 OV2640 的图像数据输出接口时序一致，见图 1-17。

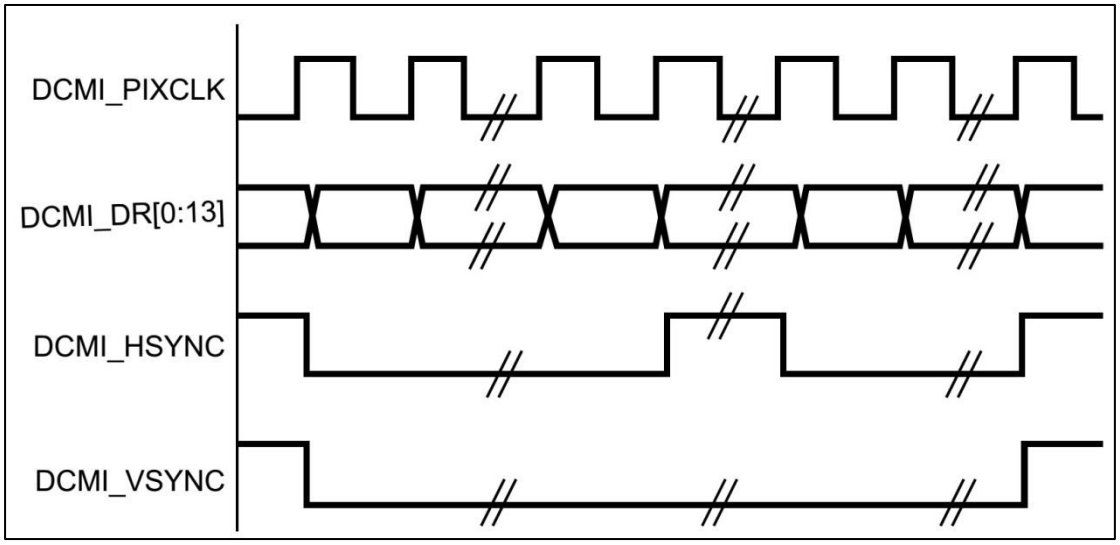

图 1-17 DCMI 时序图

内部信号及 PIXCLK 的时钟频率

图 1-16 的标号②处表示 DCMI 与内部的信号线。在 STM32 的内部，使用 HCLK 作为时钟源提供给 DCMI 外设。从 DCMI 引出有 DCMI_IT 信号至中断控制器，并可通过 DMA_REQ 信号发送 DMA 请求。

DCMI 从外部接收数据时，在 HCLK 的上升沿时对 PIXCLK 同步的信号进行采样，它限制了 PIXCLK 的最小时钟周期要大于 2.5 个 HCLK 时钟周期，即最高频率为 HCLK 的 1/4。

1.3.2 DCMI 接口内部结构

DCMI 接口的内部结构见图 1-18。

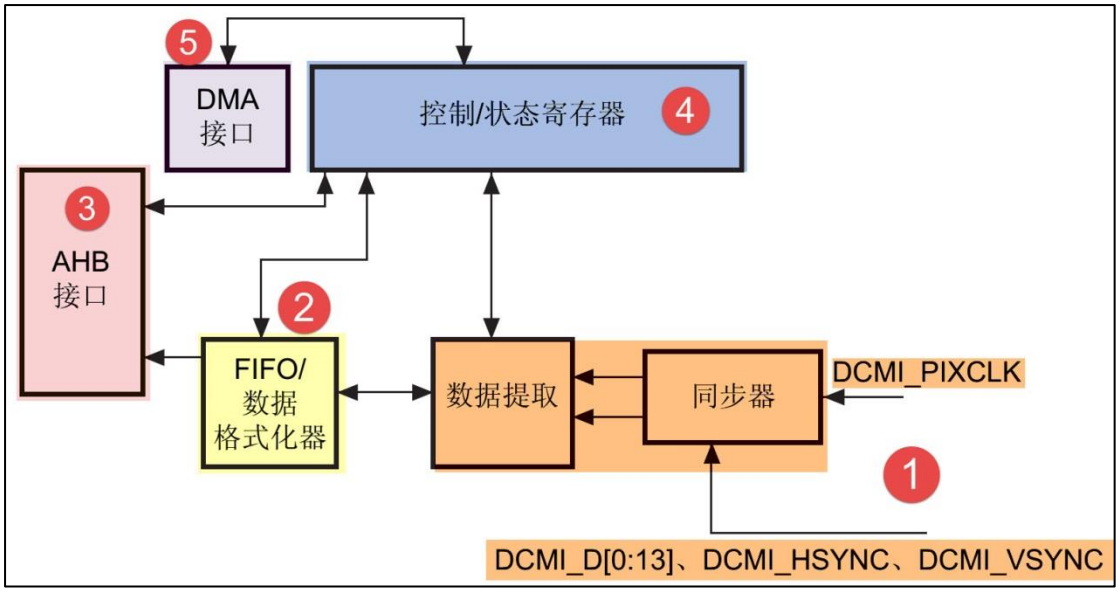

图 1-18 DCMI 接口内部结构

(1) 同步器

同步器主要用于管理 DCMI 接收数据的时序，它根据外部的信号提取输入的数据。

(2) FIFO/数据格式化器

为了对数据传输加以管理，STM32 在 DCMI 接口上实现了 4 个字(32bit x4)深度的 FIFO，用以缓冲接收到的数据。

(3) AHB 接口

DCMI 接口挂载在 AHB 总线上，在 AHB 总线中有一个 DCMI 接口的数据寄存器，当我们读取该寄存器时，它会从 FIFO 中获取数据，并且 FIFO 中的数据指针会自动进行偏移，使得我们每次读取该寄存器都可获得一个新的数据。

(4) 控制/状态寄存器

DCMI 的控制寄存器协调图中的各个结构运行，程序中可通过检测状态寄存器来获 DCMI 的当前运行状态。

(5) DMA 接口

由于 DCMI 采集的数据量很大，我们一般使用 DMA 来把采集得的数据搬运至内存。

1.3.3 同步方式

DCMI 接口支持硬件同步或内嵌码同步方式，硬件同步方式即使用 HSYNC 和 VSYNC 作为同步信号的方式，OV2640 就是使用这种同步时序。

而内嵌码同步的方式是使用数据信号线传输中的特定编码来表示同步信息，由于需要用 0x00 和 0xFF 来表示编码，所以表示图像的数据中不能包含有这两个值。利用这两个值，它扩展到 4 个字节，定义出了 2 种模式的同步码，每种模式包含 4 个编码，编码格式为 0xFF0000XY，其中 XY 的值可通过寄存器设置。当 DCMI 接收到这样的编码时，它不会把这些当成图像数据，而是按照表 1-3 中的编码来解释，作为同步信号。

表 1-3 两种模式的内嵌码

模式 2 的内嵌码	模式 1 的内嵌码
帧开始(FS)	有效行开始(SAV)
帧结束(FE)	有效行结束(EAV)
行开始(LS)	帧间消隐期内的行开始(SAV)，其中消隐期内的即为无效数据
行结束(LS)	帧间消隐期内的行结束(EAV)，其中消隐期内的即为无效数据

1.3.4 捕获模式及捕获率

DCMI 还支持两种数据捕获模式，分别为快照模式和连续采集模式。快照模式时只采集一帧的图像数据，连续采集模式会一直采集多个帧的数据，并且可以通过配置捕获率来控制采集多少数据，如可配置为采集所有数据或隔 1 帧采集一次数据或隔 3 帧采集一次数据。

1.4 DCMI 初始化结构体

与其它外设一样，STM32 的 DCMI 外设也可以使用库函数来控制，其中最主要的配置项都封装到了 DCMI_InitTypeDef 结构体，来这些内容都定义在库文件“stm32f4xx_dcmi.h”及“stm32f4xx_ dcmi.c”中，编程时我们可以结合这两个文件内的注释使用或参考库帮助文档。

DCMI_InitTypeDef 初始化结构体的内容见代码清单 1-1。

代码清单 1-1 DCMI 初始化结构体

```
/**
  * @brief   DCMI 初始化结构体
  */
typedef struct
{
    uint16_t DCMI_CaptureMode;       /*选择连续模式或拍照模式 */
    uint16_t DCMI_SynchroMode;       /*选择硬件同步模式还是内嵌码模式 */
    uint16_t DCMI_PCKPolarity;       /*设置像素时钟的有效边沿*/
    uint16_t DCMI_VSPolarity;        /*设置 VSYNC 的有效电平*/
    uint16_t DCMI_HSPolarity;        /*设置 HSYNC 的有效边沿*/
    uint16_t DCMI_CaptureRate;       /*设置图像的采集间隔 */
    uint16_t DCMI_ExtendedDataMode; /*设置数据线的宽度 */
} DCMI_InitTypeDef;
```

这些结构体成员说明如下，其中括号内的文字是对应参数在 STM32 标准库中定义的宏：

(1) DCMI_CaptureMode

本成员设置 DCMI 的捕获模式，可以选择为连续摄像(DCMI_CaptureMode_Continuous)或单张拍照 DCMI_CaptureMode_SnapShot。

(2) DCMI_SynchroMode

本成员设置 DCMI 数据的同步模式，可以选择为硬件同步方式(DCMI_SynchroMode_Hardware)或内嵌码方式(DCMI_SynchroMode_Embedded)。

(3) DCMI_PCKPolarity

本成员用于配置 DCMI 接口像素时钟的有效边沿，即在该时钟边沿时，DCMI 会对数据线上的信号进行采样，它可以被设置为上升沿有效(DCMI_PCKPolarity_Rising)或下降沿有效(DCMI_PCKPolarity_Falling)。

(4) DCMI_VSPolarity

本成员用于设置 VSYNC 的有效电平，当 VSYNC 信号线表示为有效电平时，表示新的一帧数据传输完成，它可以被设置为高电平有效(DCMI_VSPolarity_High)或低电平有效(DCMI_VSPolarity_Low)。

(5) DCMI_HSPolarity

类似地，本成员用于设置 HSYNC 的有效电平，当 HSYNC 信号线表示为有效电平时，表示新的一行数据传输完成，它可以被设置为高电平有效(DCMI_HSPolarity_High)或低电平有效(DCMI_HSPolarity_Low)。

(6) DCMI_CaptureRate

本成员可以用于设置 DCMI 捕获数据的频率，可以设置为全采集、半采集或 1/4 采集(DCMI_CaptureRate_All_Frame/ 1of2_Frame/ 1of4_Frame)，在间隔采集的情况下，STM32 的 DCMI 外设会直接按间隔丢弃数据。

(7) DCMI_ExtendedDataMode

本成员用于设置 DCMI 的数据线宽度，可配置为 8/10/12 及 14 位数据线宽(DCMI_ExtendedDataMode_8b/10b/12b/14b)。

配置完这些结构体成员后，我们调用库函数 DCMI_Init 即可把这些参数写入到 DCMI 的控制寄存器中，实现 DCMI 的初始化。

1.5 DCMI—OV2640 摄像头实验

本小节讲解如何使用 DCMI 接口从 OV2640 摄像头输出的 RGB565 格式的图像数据，并把这些数据实时显示到液晶屏上。

学习本小节内容时，请打开配套的“DCMI—OV2640 摄像头”工程配合阅读。

1.5.1 硬件设计

摄像头原理图

本实验采用的 OV2640 摄像头实物见图 1-1，其原理图见图 1-19。

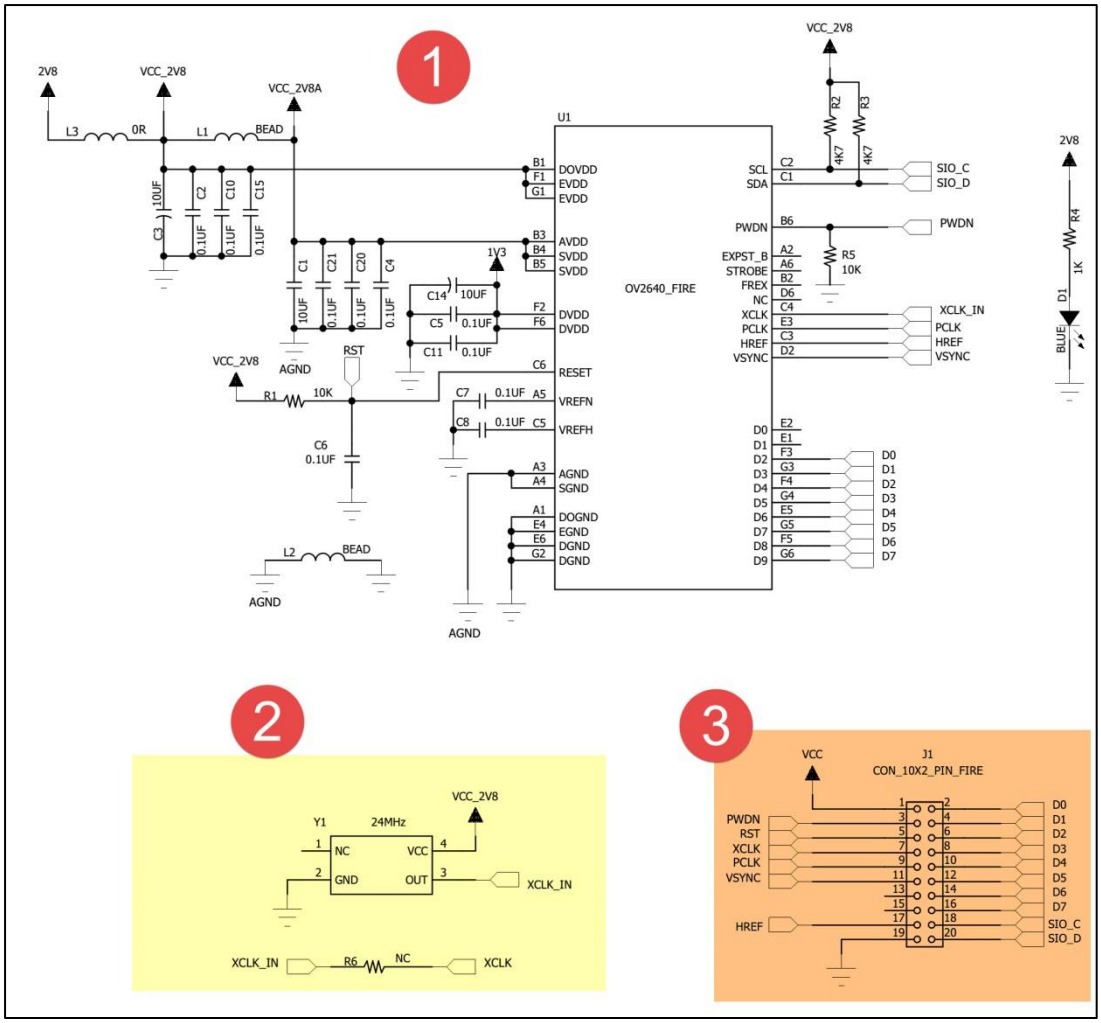

图 1-19 OV2640 摄像头原理图

错误!未找到引用源。标号①处的是 OV2640 芯片的主电路，在这部分中已对 SCCB 使用的信号线接了上拉电阻，外部电路可以省略上拉；标号②处的是一个 24MHz 的有源晶振，它为 OV2640 提供系统时钟，如果不想使用外部晶振提供时钟源，可以参考图中的 R6 处贴上 0 欧电阻，XCLK 引脚引出至外部，由外部控制器提供时钟；标号③处的是摄像头引脚集中引出的排针接口，使用它可以方便地与 STM32 实验板中的排母连接。

摄像头与实验板的连接

通过排母，OV2640 与 STM32 引脚的连接关系见图 1-20。 控制摄像头的部分引脚与实验板上的 RGB 彩灯共用，使用时会互相影响。

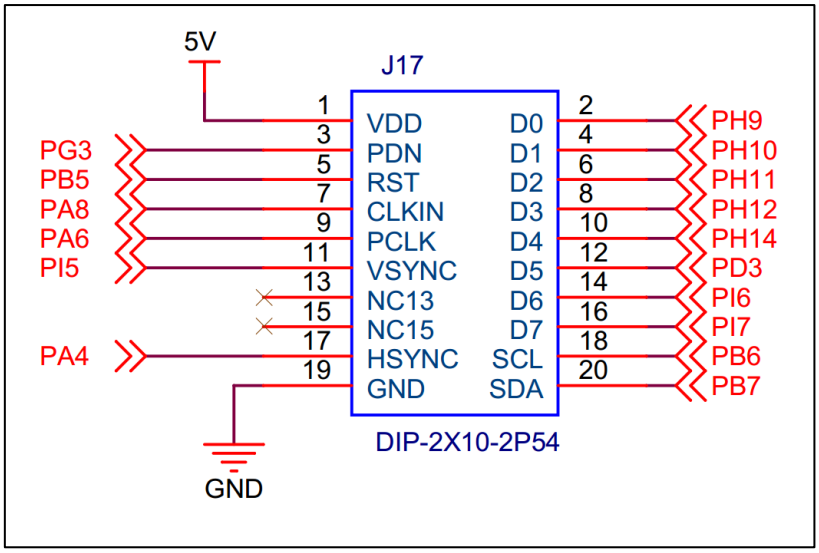

图 1-20 STM32 实验板引出的 DCMI 接口

以上原理图可查阅《ov2640—黑白原理图》及《野火 F429 开发板黑白原理图》文档获知，若您使用的摄像头或实验板不一样，请根据实际连接的引脚修改程序。

1.5.2 软件设计

为了使工程更加有条理，我们把摄像头控制相关的代码独立分开存储，方便以后移植。在“LTDC—液晶显示”工程的基础上新建“bsp_ov2640.c”及“bsp_ov2640.h”文件，这些文件也可根据您的喜好命名，它们不属于 STM32 标准库的内容，是由我们自己根据应用需要编写的。

1. 编程要点

(1) 初始化 DCMI 时钟，I2C 时钟；
(2) 使用 I2C 接口向 OV2640 写入寄存器配置；
(3) 初始化 DCMI 工作模式；
(4) 初始化 DMA，用于搬运 DCMI 的数据到显存空间进行显示；
(5) 编写测试程序，控制采集图像数据并显示到液晶屏。

2. 代码分析

摄像头硬件相关宏定义

我们把摄像头控制硬件相关的配置都以宏的形式定义到 “bsp_ov2640.h”文件中，其中包括 I2C 及 DCMI 接口的，见代码清单 1-2。

代码清单 1-2 摄像头硬件配置相关的宏(省略了部分数据线)

```
1
2 /*摄像头接口 */
3 //IIC SCCB
4 #define CAMERA_I2C                          I2C1
5 #define CAMERA_I2C_CLK                      RCC_APB1Periph_I2C1
6
```

```
#define CAMERA_I2C_SCL_PIN                  GPIO_Pin_6
#define CAMERA_I2C_SCL_GPIO_PORT            GPIOB
#define CAMERA_I2C_SCL_GPIO_CLK             RCC_AHB1Periph_GPIOB
#define CAMERA_I2C_SCL_SOURCE               GPIO_PinSource6
#define CAMERA_I2C_SCL_AF                   GPIO_AF_I2C1

#define CAMERA_I2C_SDA_PIN                  GPIO_Pin_7
#define CAMERA_I2C_SDA_GPIO_PORT            GPIOB
#define CAMERA_I2C_SDA_GPIO_CLK             RCC_AHB1Periph_GPIOB
#define CAMERA_I2C_SDA_SOURCE               GPIO_PinSource7
#define CAMERA_I2C_SDA_AF                   GPIO_AF_I2C1

//VSYNC
#define DCMI_VSYNC_GPIO_PORT            GPIOI
#define DCMI_VSYNC_GPIO_CLK             RCC_AHB1Periph_GPIOI
#define DCMI_VSYNC_GPIO_PIN             GPIO_Pin_5
#define DCMI_VSYNC_PINSOURCE            GPIO_PinSource5
#define DCMI_VSYNC_AF                     GPIO_AF_DCMI
// HSYNC
#define DCMI_HSYNC_GPIO_PORT            GPIOA
#define DCMI_HSYNC_GPIO_CLK             RCC_AHB1Periph_GPIOA
#define DCMI_HSYNC_GPIO_PIN             GPIO_Pin_4
#define DCMI_HSYNC_PINSOURCE            GPIO_PinSource4
#define DCMI_HSYNC_AF                     GPIO_AF_DCMI
//PIXCLK
#define DCMI_PIXCLK_GPIO_PORT           GPIOA
#define DCMI_PIXCLK_GPIO_CLK            RCC_AHB1Periph_GPIOA
#define DCMI_PIXCLK_GPIO_PIN            GPIO_Pin_6
#define DCMI_PIXCLK_PINSOURCE           GPIO_PinSource6
#define DCMI_PIXCLK_AF                    GPIO_AF_DCMI
//PWDN
#define DCMI_PWDN_GPIO_PORT         GPIOG
#define DCMI_PWDN_GPIO_CLK          RCC_AHB1Periph_GPIOG
#define DCMI_PWDN_GPIO_PIN          GPIO_Pin_3

//数据信号线
#define DCMI_D0_GPIO_PORT           GPIOH
#define DCMI_D0_GPIO_CLK            RCC_AHB1Periph_GPIOH
#define DCMI_D0_GPIO_PIN            GPIO_Pin_9
#define DCMI_D0_PINSOURCE           GPIO_PinSource9
#define DCMI_D0_AF                    GPIO_AF_DCMI
/*....省略部分数据线*/
```

以上代码根据硬件的连接，把与 DCMI、I2C 接口与摄像头通讯使用的引脚号、引脚源以及复用功能映射都以宏封装起来。

初始化 DCMI 的 GPIO 及 I2C

利用上面的宏，初始化 DCMI 的 GPIO 引脚及 I2C，见代码清单 1-3。

代码清单 1-3 初始化 DCMI 的 GPIO 及 I2C(省略了部分数据线)

```

/**
  * @brief  初始化控制摄像头使用的 GPIO(I2C/DCMI)
  * @param  None
  * @retval None
  */
void OV2640_HW_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    I2C_InitTypeDef  I2C_InitStruct;

    /***DCMI 引脚配置***/
    /* 使能 DCMI 时钟 */
    RCC_AHB1PeriphClockCmd(DCMI_PWDN_GPIO_CLK|DCMI_VSYNC_GPIO_CLK |
                           DCMI_HSYNC_GPIO_CLK | DCMI_PIXCLK_GPIO_CLK|
```

```
                                DCMI_D0_GPIO_CLK| DCMI_D1_GPIO_CLK|, ENABLE);

    /*控制/同步信号线*/
    GPIO_InitStructure.GPIO_Pin = DCMI_VSYNC_GPIO_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP ;
    GPIO_Init(DCMI_VSYNC_GPIO_PORT, &GPIO_InitStructure);
    GPIO_PinAFConfig(DCMI_VSYNC_GPIO_PORT, DCMI_VSYNC_PINSOURCE, DCMI_VSYNC_AF);

    GPIO_InitStructure.GPIO_Pin = DCMI_HSYNC_GPIO_PIN ;
    GPIO_Init(DCMI_HSYNC_GPIO_PORT, &GPIO_InitStructure);
    GPIO_PinAFConfig(DCMI_HSYNC_GPIO_PORT, DCMI_HSYNC_PINSOURCE, DCMI_HSYNC_AF);

    GPIO_InitStructure.GPIO_Pin = DCMI_PIXCLK_GPIO_PIN ;
    GPIO_Init(DCMI_PIXCLK_GPIO_PORT, &GPIO_InitStructure);
     GPIO_PinAFConfig(DCMI_PIXCLK_GPIO_PORT, DCMI_PIXCLK_PINSOURCE, DCMI_PIXCLK_AF);

    GPIO_InitStructure.GPIO_Pin = DCMI_PWDN_GPIO_PIN ;
    GPIO_Init(DCMI_PWDN_GPIO_PORT, &GPIO_InitStructure);
    /*PWDN 引脚，高电平关闭电源，低电平供电*/
    GPIO_ResetBits(DCMI_PWDN_GPIO_PORT,DCMI_PWDN_GPIO_PIN);

    /*数据信号*/
    GPIO_InitStructure.GPIO_Pin = DCMI_D0_GPIO_PIN ;
    GPIO_Init(DCMI_D0_GPIO_PORT, &GPIO_InitStructure);
    GPIO_PinAFConfig(DCMI_D0_GPIO_PORT, DCMI_D0_PINSOURCE, DCMI_D0_AF);
    /*...省略部分数据信号线*/

    /****** 配置 I2C，使用 I2C 与摄像头的 SCCB 接口通讯*****/
    /* 使能 I2C 时钟 */
    RCC_APB1PeriphClockCmd(CAMERA_I2C_CLK, ENABLE);
    /* 使能 I2C 使用的 GPIO 时钟 */
RCC_AHB1PeriphClockCmd(CAMERA_I2C_SCL_GPIO_CLK|CAMERA_I2C_SDA_GPIO_CLK, ENABLE);
    /* 配置引脚源 */
GPIO_PinAFConfig(CAMERA_I2C_SCL_GPIO_PORT, CAMERA_I2C_SCL_SOURCE, CAMERA_I2C_SCL_AF);
GPIO_PinAFConfig(CAMERA_I2C_SDA_GPIO_PORT, CAMERA_I2C_SDA_SOURCE, CAMERA_I2C_SDA_AF);

    /* 初始化 GPIO */
    GPIO_InitStructure.GPIO_Pin = CAMERA_I2C_SCL_PIN ;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_OD;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_Init(CAMERA_I2C_SCL_GPIO_PORT, &GPIO_InitStructure);
GPIO_PinAFConfig(CAMERA_I2C_SCL_GPIO_PORT, CAMERA_I2C_SCL_SOURCE, CAMERA_I2C_SCL_AF);

    GPIO_InitStructure.GPIO_Pin = CAMERA_I2C_SDA_PIN ;
    GPIO_Init(CAMERA_I2C_SDA_GPIO_PORT, &GPIO_InitStructure);

    /*初始化 I2C 模式 */
    I2C_DeInit(CAMERA_I2C);

    I2C_InitStruct.I2C_Mode = I2C_Mode_I2C;
    I2C_InitStruct.I2C_DutyCycle = I2C_DutyCycle_2;
    I2C_InitStruct.I2C_OwnAddress1 = 0xFE;
    I2C_InitStruct.I2C_Ack = I2C_Ack_Enable;
     I2C_InitStruct.I2C_AcknowledgedAddress = I2C_AcknowledgedAddress_7bit;
    I2C_InitStruct.I2C_ClockSpeed = 40000;

    /* 写入配置 */
    I2C_Init(CAMERA_I2C, &I2C_InitStruct);

    /* 使能 I2C */
    I2C_Cmd(CAMERA_I2C, ENABLE);
```

```
82 }
```

与所有使用到 GPIO 的外设一样，都要先把使用到的 GPIO 引脚模式初始化，以上代码把 DCMI 接口的信号线全都初始化为 DCMI 复用功能，而 PWDN 则被初始化成普通的推挽输出模式，并且在初始化完毕后直接控制它为低电平，使能给摄像头供电。

函数中还包含了 I2C 的初始化配置，使用 I2C 与 OV2640 的 SCCB 接口通讯，这里的 I2C 模式配置与标准的 I2C 无异。

配置 DCMI 的模式

接下来需要配置 DCMI 的工作模式，我们通过编写 OV2640_Init 函数完成该功能，见代码清单 1-4。

代码清单 1-4 配置 DCMI 的模式(bsp_ov2640.c 文件)

```
1 #define FSMC_LCD_ADDRESS      ((uint32_t)0xD0000000)
2
3 /*液晶屏的分辨率，用来计算地址偏移*/
4 uint16_t lcd_width=800, lcd_height=480;
5
6 /*摄像头采集图像的大小，改变这两个值可以改变数据量，
7 但不会加快采集速度，要加快采集速度需要改成 SVGA 模式*/
8 uint16_t img_width=800, img_height=480;
9 /**
10   * @brief  配置 DCMI/DMA 以捕获摄像头数据
11   * @param  None
12   * @retval None
13   */
14 void OV2640_Init(void)
15 {
16     DCMI_InitTypeDef DCMI_InitStructure;
17     NVIC_InitTypeDef NVIC_InitStructure;
18
19     /*** 配置 DCMI 接口 ***/
20     /* 使能 DCMI 时钟 */
21     RCC_AHB2PeriphClockCmd(RCC_AHB2Periph_DCMI, ENABLE);
22
23     /* DCMI 配置*/
24     DCMI_InitStructure.DCMI_CaptureMode = DCMI_CaptureMode_Continuous;
25     DCMI_InitStructure.DCMI_SynchroMode = DCMI_SynchroMode_Hardware;
26     DCMI_InitStructure.DCMI_PCKPolarity = DCMI_PCKPolarity_Rising;
27     DCMI_InitStructure.DCMI_VSPolarity = DCMI_VSPolarity_Low;
28     DCMI_InitStructure.DCMI_HSPolarity = DCMI_HSPolarity_Low;
29     DCMI_InitStructure.DCMI_CaptureRate = DCMI_CaptureRate_All_Frame;
30     DCMI_InitStructure.DCMI_ExtendedDataMode = DCMI_ExtendedDataMode_8b;
31     DCMI_Init(&DCMI_InitStructure);
32
33     //开始传输，从后面开始一行行扫描上来，实现数据翻转
34     //dma_memory 以 16 位数据为单位， dma_bufsize 以 32 位数据为单位(即像素个数/2)
35      OV2640_DMA_Config(FSMC_LCD_ADDRESS+(lcd_height-1)*(lcd_width)*2,img_width*2/4);
36
37     /* 配置中断 */
38     NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
39
40     /* 配置中断源 */
41     NVIC_InitStructure.NVIC_IRQChannel = DMA2_Stream1_IRQn ;//DMA 数据流中断
42     NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
43     NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
44     NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
45     NVIC_Init(&NVIC_InitStructure);
46     DMA_ITConfig(DMA2_Stream1,DMA_IT_TC,ENABLE);
47
```

```
48      /* 配置帧中断，接收到帧同步信号就进入中断 */
49      NVIC_InitStructure.NVIC_IRQChannel = DCMI_IRQn ;     //帧中断
50      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority =0;
51      NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
52      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
53      NVIC_Init(&NVIC_InitStructure);
54      DCMI_ITConfig (DCMI_IT_FRAME,ENABLE);
55  }
```

该函数的执行流程如下：

(1) 使能 DCMI 外设的时钟，它是挂载在 AHB2 总线上的；

(2) 根据摄像头的时序和硬件连接的要求，配置 DCMI 工作模式为：使用硬件同步，连续采集所有帧数据，采集时使用 8 根数据线，PIXCLK 被设置为上升沿有效，VSYNC 和 HSYNC 都被设置成低电平有效；

(3) 调用 OV2640_DMA_Config 函数开始 DMA 数据传输，每传输完一行数据需要调用一次，它包含本次传输的目的首地址及传输的数据量，后面我们再详细解释；

(4) 配置 DMA 中断，DMA 每次传输完毕会引起中断，以便我们在中断服务函数配置 DMA 传输下一行数据；

(5) 配置 DCMI 的帧传输中断，为了防止有时 DMA 出现传输错误或传输速度跟不上导致数据错位、偏移等问题，每次 DCMI 接收到摄像头的一帧数据，得到新的帧同步信号后(VSYNC)，就进入中断，复位 DMA，使它重新开始一帧的数据传输。

配置 DMA 数据传输

上面的 DCMI 配置函数中调用了 OV2640_DMA_Config 函数开始了 DMA 传输，该函数的定义见代码清单 1-5。

代码清单 1-5 配置 DMA 数据传输(bsp_ov2640.c 文件)

```
 1 /**
 2   * @brief  配置 DCMI/DMA 以捕获摄像头数据
 3   * @param  DMA_Memory0BaseAddr:本次传输的目的首地址
 4   * @param DMA_BufferSize: 本次传输的数据量(单位为字,即 4 字节)
 5   */
 6 void OV2640_DMA_Config(uint32_t DMA_Memory0BaseAddr,uint16_t
DMA_BufferSize)
 7 {
 8
 9     DMA_InitTypeDef  DMA_InitStructure;
10
11     /* 配置 DMA 从 DCMI 中获取数据*/
12     /* 使能 DMA*/
13     RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2, ENABLE);
14     DMA_Cmd(DMA2_Stream1,DISABLE);
15     while (DMA_GetCmdStatus(DMA2_Stream1) != DISABLE) {}
16
17     DMA_InitStructure.DMA_Channel = DMA_Channel_1;
18     DMA_InitStructure.DMA_PeripheralBaseAddr = DCMI_DR_ADDRESS; //DCMI 数据寄存器地址
19 DMA_InitStructure.DMA_Memory0BaseAddr = DMA_Memory0BaseAddr;//传输的目的地址(传入的参数)
20     DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;
21  DMA_InitStructure.DMA_BufferSize =DMA_BufferSize;//传输的数据大小(传入的参数)
22     DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
23     DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable; //寄存器地址自增
24     DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Word;
25     DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
26     DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;              //循环模式
27     DMA_InitStructure.DMA_Priority = DMA_Priority_High;
```

```
28      DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Enable;
29      DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_Full;
30      DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_INC8;
31      DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
32
33      /*DMA 中断配置 */
34      DMA_Init(DMA2_Stream1, &DMA_InitStructure);
35
36      DMA_Cmd(DMA2_Stream1,ENABLE);
37      while (DMA_GetCmdStatus(DMA2_Stream1) != ENABLE) {}
38  }
39
```

该函数跟普通的 DMA 配置无异，它把 DCMI 接收到的数据从它的数据寄存器搬运到 SDRAM 显存中，从而直接使用液晶屏显示摄像头采集得的图像。它包含 2 个输入参数 DMA_Memory0BaseAddr 和 DMA_BufferSize，其中 DMA_Memory0BaseAddr 用于设置本次 DMA 传输的目的首地址，该参数会被赋值到结构体成员 DMA_InitStructure.DMA_Memory0BaseAddr 中。DMA_BufferSize 则用于指示本次 DMA 传输的数据量，它会被赋值到结构体成员 DMA_InitStructure.DMA_BufferSize 中，要注意它的单位是一个字，即 4 字节，如我们要传输 60 字节的数据时，它应配置为 15。在前面的 OV2640_Init 函数中，对这个函数有如下调用：

```
1
2  #define FSMC_LCD_ADDRESS      ((uint32_t)0xD0000000)
3
4  /*液晶屏的分辨率，用来计算地址偏移*/
5  uint16_t lcd_width=800, lcd_height=480;
6
7  /*摄像头采集图像的大小，改变这两个值可以改变数据量，
8  但不会加快采集速度，要加快采集速度需要改成 SVGA 械*/
9  uint16_t img_width=800, img_height=480;
10
11
12 //开始传输，从后面开始一行行扫描上来，实现数据翻转
13 //dma_memory 以 16 位数据为单位， dma_bufsize 以 32 位数据为单位(即像素个数/2)
14 OV2640_DMA_Config(FSMC_LCD_ADDRESS+(lcd_height-1)*(lcd_width)*2,img_width*2/4);
15
```

其中的 lcd_width 和 lcd_height 是液晶屏的分辨率，img_width 和 img_heigh 表示摄像头输出的图像的分辨率，FSMC_LCD_ADDRESS 是液晶层的首个显存地址。另外，本工程中显示摄像头数据的这个液晶层采用 RGB565 的像素格式，每个像素点占据 2 个字节。

所以在上面的函数调用中，第一个输入参数：

【FSMC_LCD_ADDRESS+(lcd_height-1)*(lcd_width)*2】

它表示的是液晶屏最后一行的第一个像素的地址。

而第二个输入参数：

【img_width*2/4】

它表示表示摄像头一行图像的数据量，单位为字，即用一行图像数据的像素个数除以 2 即可。注意这里使用的变量是“img_width”而不是的“lcd_width”。

由于这里配置的是第一次 DMA 传输，它把 DCMI 接收到的第一行摄像头数据传输至液晶屏的最后一行，见图 1-21，再配合在后面分析的中断函数里的多次 DMA 配置，摄像头输出的数据会一行一行地“由下至上”显示到液晶屏上。

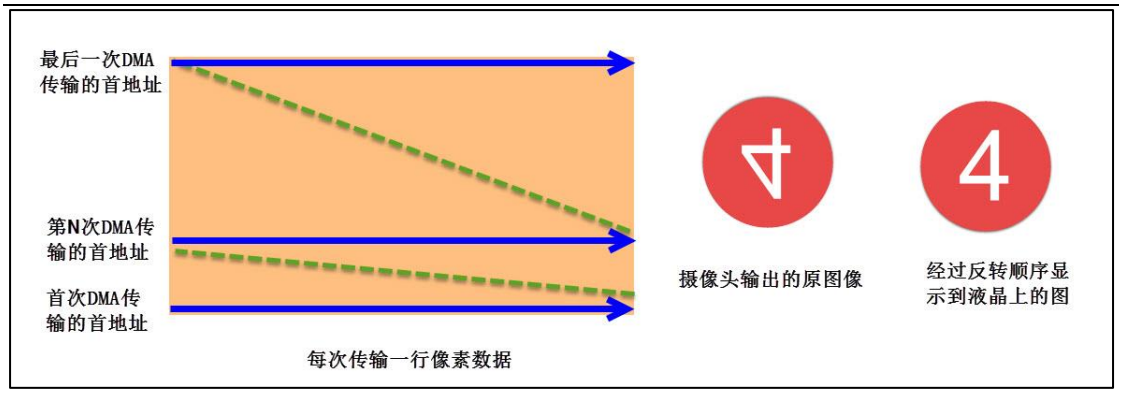

图 1-21 DMA 传输过程

把摄像头输出的第一行数据显示到液晶屏的最后一行，是因为摄像头输出的原图像是颠倒的，这样处理可方便观看实验现象(实际上 OV2640 可配置寄存器来直接输出镜像数据，但不知为何配置该功能后采集得的图像失真，所以我们就采用这样颠倒的方法处理了)。

DMA 传输完成中断及帧中断

OV2640_Init 函数初始化了 DCMI，使能了帧中断、DMA 传输完成中断，并使能了第一次 DMA 传输，当这一行数据传输完成时，会进入 DMA 中断服务函数，见代码清单 1-6 中的 DMA2_Stream1_IRQHandler。

代码清单 1-6 DMA 传输完成中断与帧中断(stm32f4xx_it.c 文件)

```
1 extern uint16_t lcd_width, lcd_height;
 2 extern uint16_t img_width, img_height;
 3
 4 //记录传输了多少行
 5 static uint16_t line_num =0;
 6 //DMA 传输完成中断服务函数
 7 void DMA2_Stream1_IRQHandler(void)
 8 {
 9     if (  DMA_GetITStatus(DMA2_Stream1,DMA_IT_TCIF1) == SET )
10     {
11         /*行计数*/
12         line_num++;
13         if (line_num==img_height)
14         {
15             /*传输完一帧,计数复位*/
16             line_num=0;
17         }
18         /*DMA 一行一行传输*/
19         OV2640_DMA_Config(FSMC_LCD_ADDRESS+(lcd_width*2*(lcd_height-line_num-
1)),img_width*2/4);
20         DMA_ClearITPendingBit(DMA2_Stream1,DMA_IT_TCIF1);
21     }
22 }
23
24
25 //帧中断服务函数，使用帧中断重置 line_num,可防止有时掉数据的时候 DMA 传送行数出现偏移
26 void DCMI_IRQHandler(void)
27 {
28     if (  DCMI_GetITStatus (DCMI_IT_FRAME) == SET )
29     {
30         /*传输完一帧，计数复位*/
31         line_num=0;
32         DCMI_ClearITPendingBit(DCMI_IT_FRAME);
33     }
34 }
```

35

DMA 中断服务函数中主要是使用了一个静态变量 line_num 来记录已传输了多少行数据，每进一次 DMA 中断时自加 1，由于进入一次中断就代表传输完一行数据，所以 line_num 的值等于 lcd_height 时(摄像头输出的数据行数)，表示传输完一帧图像，line_num 复位为 0，开始另一帧数据的传输。line_num 计数完毕后利用前面定义的 OV2640_DMA_Config 函数配置新的一行 DMA 数据传输，它利用 line_num 变量计算显存地址的行偏移，控制 DCMI 数据被传送到正确的位置，每次传输的都是一行像素的数据量。

当 DCMI 接口检测到摄像头传输的帧同步信号时，会进入 DCMI_IRQHandler 中断服务函数，在这个函数中不管 line_num 原来的值是什么，它都把 line_num 直接复位为 0，这样下次再进入 DMA 中断服务函数的时候，它会开始新一帧数据的传输。这样可以利用 DCMI 的硬件同步信号，而不只是依靠 DMA 自己的传输计数，这样可以避免有时 STM32 内部 DMA 传输受到阻塞而跟不上外部摄像头信号导致的数据错误。

使能 DCMI 采集

以上是我们使用 DCMI 的传输配置，但它还没有使能 DCMI 采集，在实际使用中还需要调用下面两个库函数开始采集数据。

```
1 //使能 DCMI 采集数据
2 DCMI_Cmd(ENABLE);
3 DCMI_CaptureCmd(ENABLE);
```

读取 OV2640 芯片 ID

配置完了 STM32 的 DCMI，还需要控制摄像头，它有很多寄存器用于配置工作模式。利用 STM32 的 I2C 接口，可向 OV2640 的寄存器写入控制参数，我们先写个读取芯片 ID 的函数测试一下，见代码清单 1-7。

代码清单 1-7 读取 OV2640 的芯片 ID(bsp_ov2640.c 文件)

```
 1
 2 //存储摄像头 ID 的结构体
 3 typedef struct
 4 {
 5     uint8_t Manufacturer_ID1;
 6     uint8_t Manufacturer_ID2;
 7     uint8_t PIDH;
 8     uint8_t PIDL;
 9 } OV2640_IDTypeDef;
10 /*寄存器地址*/
11 #define OV2640_DSP_RA_DLMT        0xFF
12 #define OV2640_SENSOR_MIDH        0x1C
13 #define OV2640_SENSOR_MIDL        0x1D
14 #define OV2640_SENSOR_PIDH        0x0A
15 #define OV2640_SENSOR_PIDL        0x0B
16 /**
17   * @brief  读取摄像头的 ID.
18   * @param  OV2640ID: 存储 ID 的结构体
19   * @retval None
20   */
21 void OV2640_ReadID(OV2640_IDTypeDef *OV2640ID)
22 {
23     /*OV2640 有两组寄存器，设置 0xFF 寄存器的值为 0 或为 1 时可选择使用不同组的寄存器*/
24     OV2640_WriteReg(OV2640_DSP_RA_DLMT, 0x01);
25
26     /*读取寄存芯片 ID*/
```

```
27      OV2640ID->Manufacturer_ID1 = OV2640_ReadReg(OV2640_SENSOR_MIDH);
28      OV2640ID->Manufacturer_ID2 = OV2640_ReadReg(OV2640_SENSOR_MIDL);
29      OV2640ID->PIDH = OV2640_ReadReg(OV2640_SENSOR_PIDH);
30      OV2640ID->PIDL = OV2640_ReadReg(OV2640_SENSOR_PIDL);
31  }
```

在 OV2640 的 MIDH 及 MIDL 寄存器中存储了它的厂商 ID，默认值为 0x7F 和 0xA2；而 PIDH 及 PIDL 寄存器存储了产品 ID，PIDH 的默认值为 0x26，PIDL 的默认值不定，可能的值为 0x40、0x41 及 0x42。在代码中我们定义了一个结构体 OV2640_IDTypeDef 专门存储这些读取得的 ID 信息。

由于这些寄存器都是属于 Sensor 组的，所以在读取这些寄存器的内容前，需要先把 OV2640 中 0xFF 地址的 DLMT 寄存器值设置为 1。

OV2640_ReadID 函数中使用的 OV2640_ReadReg 及 OV2640_WriteReg 函数是使用 STM32 的 I2C 外设向某寄存器读写单个字节数据的底层函数，它与我们前面章节中用到的 I2C 函数大同小异，就不展开分析了。

向 OV2640 写入寄存器配置

检测到 OV2640 的存在后，向它写入配置参数，见代码清单 1-8。

代码清单 1-8 向 OV2640 写入寄存器配置

```
1  /* OV2640 的 SVGA 是 600 列*800 行的，在 800 列*480 行的液晶屏上不能全屏*/
2  /* 所以直接用 UXGA 模式，再根据所需的图像窗口裁剪 */
3  const unsigned char OV2640_UXGA[][2]=
4  {
5      0xff, 0x00,
6      0x2c, 0xff,
7      0x2e, 0xdf,
8      0xff, 0x01,
9      0x3c, 0x32,
10     0x11, 0x00,
11     0x09, 0x02,
12     0x04, 0x20|0x80,     //水平翻转
13     0x13, 0xe5,
14
15     /*....以下省略*/
16 }
17
18
19 /**
20   * @brief  配置 OV2640 为 UXGA 模式，并设置输出图像大小
21   * @param  None
22   * @retval None
23   */
24 void OV2640_UXGAConfig(void)
25 {
26     uint32_t i;
27
28     /* 写入寄存器配置 */
29     for (i=0; i<(sizeof(OV2640_UXGA)/2); i++)
30     {
31         OV2640_WriteReg(OV2640_UXGA[i][0], OV2640_UXGA[i][1]);
32     }
33     /*设置输出的图像大小*/
34     OV2640_OutSize_Set(img_width,img_height);
35 }
```

这个 OV2640_UXGAConfig 函数简单粗暴，它直接把一个二维数组 OV2640_UXGA 使用 I2C 传输到 OV2640 中，该二维数组的第一维存储的是寄存器地址，第二维存储的是对应寄存器要写入的控制参数。

如果您对这些寄存器配置感兴趣，可以一个个对着 OV2640 的寄存器说明来阅读，阅读时要注意区分 DLMT 寄存器(地址 0xFF)为 1 或为 0 时的寄存器组。总的来部，这些配置主要是把 OV2640 配置成了 UXGA 时序模式，并使用 8 根数据线输出格式为 RGB565 的图像数据。

其中 UXGA 时序是指它最大可输出 1600x1200 分辨率的图像，OV2640 还支持使用 SVGA 时序输出最大分辨率为 800x600 的图像，相对于 UXGA，它可使用更高的帧率输出，但由于它规定好了数据是 800 行、600 列，而我们的液晶屏在横屏状态下，无法直接全屏显示(800 列、480 行)，所以就把 OV2640 配置成 UXGA 模式了。通过修改 COM7 寄存器(地址 0x12)可改变时序模式。

main 函数

最后我们来编写 main 函数，利用前面讲解的函数，控制采集图像，见**错误!未找到引用源。**。

代码清单 1-9 main 函数

```
/**
  * @brief  主函数
  * @param  无
  * @retval 无
  */
int main(void)
{

    /*摄像头与 RGB LED 灯共用引脚，不要同时使用 LED 和摄像头*/
    Debug_USART_Config();

    /*初始化液晶屏*/
    LCD_Init();
    LCD_LayerInit();
    LTDC_Cmd(ENABLE);

    /*把背景层刷黑色*/
    LCD_SetLayer(LCD_BACKGROUND_LAYER);
    LCD_SetTransparency(0xFF);
    LCD_Clear(LCD_COLOR_BLACK);

    /*初始化后默认使用前景层*/
    LCD_SetLayer(LCD_FOREGROUND_LAYER);
    /*默认设置不透明 ，该函数参数为不透明度，范围 0-0xff ，
    0 为全透明，0xff 为不透明*/
    LCD_SetTransparency(0xFF);
    LCD_Clear(TRANSPARENCY);

    LCD_SetColors(LCD_COLOR_RED,TRANSPARENCY);

    LCD_ClearLine(LINE(18));
    LCD_DisplayStringLine_EN_CH(LINE(18),(uint8_t* )" 模式:UXGA 800x480");

    CAMERA_DEBUG("STM32F429 DCMI 驱动 OV2640 例程");

    /* 初始化摄像头 GPIO 及 IIC */
```

```
37      OV2640_HW_Init();
38
39      /* 读取摄像头芯片 ID，确定摄像头正常连接 */
40      OV2640_ReadID(&OV2640_Camera_ID);
41
42      if (OV2640_Camera_ID.PIDH  == 0x26)
43      {
44          CAMERA_DEBUG("%x %x",OV2640_Camera_ID.Manufacturer_ID1 ,
45                       OV2640_Camera_ID.Manufacturer_ID2);
46      }
47      else
48      {
49          LCD_SetTextColor(LCD_COLOR_RED);
50      LCD_DisplayStringLine_EN_CH(LINE(0),(uint8_t*) "没有检测到 OV2640，请重新检查连接。
");
51          CAMERA_DEBUG("没有检测到 OV2640 摄像头，请重新检查连接。");
52          while (1);
53      }
54
55      OV2640_Init();
56      OV2640_UXGAConfig();
57
58      //使能 DCMI 采集数据
59      DCMI_Cmd(ENABLE);
60      DCMI_CaptureCmd(ENABLE);
61
62      /*DMA 直接传输摄像头数据到 LCD 屏幕显示*/
63      while (1)
64      {
65      }
66  }
67
```

在 main 函数中，首先初始化了液晶屏，注意它是把摄像头使用的液晶层初始化成 RGB565 格式了，可直接在工程的液晶底层驱动解这方面的内容。

摄像头控制部分，首先调用了 OV2640_HW_Init 函数初始化 DCMI 及 I2C，然后调用 OV2640_ReadID 函数检测摄像头与实验板是否正常连接，若连接正常则调用 OV2640_Init 函数初始化 DCMI 的工作模式及 DMA，再调用 OV2640_UXGAConfig 函数向 OV2640 写入寄存器配置，最后，一定要记住调用库函数 DCMI_Cmd 及 DCMI_CaptureCmd 函数使能 DCMI 开始捕获数据，这样才能正常开始工作。

3. 下载验证

把 OV2640 接到实验板的摄像头接口中，用 USB 线连接开发板，编译程序下载到实验板，并上电复位，液晶屏会显示摄像头采集得的图像，通过旋转镜头可以调焦。