TjpgDec 技术手册

-------R0.01b 版

前言

相信大家对 FATFS 文件系统都不陌生了，2012 年 FATFS 的作者推出了 JPG/JPEG 图片的解码函数库 TJpgDec 的 R0.01b 版，使用方法和 FATFS 文件系统的使用一样，仅仅调用 2 个简单的库函数就能完成对 JPG/JPEG 图片的解码，而且输出的数据格式为 RGB888 或 RGB565。

本文档是根据 ChaN 的专用网页提供的英文版技术手册翻译而来，因个人水平有限以及时间仓促，错误之处在所难免。本文档原始版权归 TJpgDec 的作者所有，本人只是做了一下翻译的工作，把这篇文档献给所有嵌入式开发人员，希望能够帮到你们。

嵌入式奋勇前进

2013-10-20

一．前言：

TJpgDec 是一款为小型嵌入式系统服务的高效且完善的 JPEG 图片解码模块。它占用内存极少，因此可以移植入像 AVR,8051，PIC,Z80，Cortex-M0 等等小型单片机中。

二．特点：

- 库函数是按照 ANSI-C 规范编写的，所以应用平台不受约束。
- 易于使用的主模式操作方式。
- 完全可重入的架构。
- 非常小的内存占用：
  - RAM 仅占用 3KB，而不受图片大小的影响。
  - ROM 占用 3.5-8.5KB，主要用于存储代码和 const 常量。
- 输出格式:
  - 输出图片比例: 1/1, 1/2, 1/4 ，1/8 可选
  - 输出像素格式: RGB888/RGB565（可预设）

三．应用程序接口：

共有 2 个应用程序接口函数，用于分析和解码 JPEG 图片（译者注：移植 TJpgDec 时需要在主程序中调用这两个库函数）

- jd_prepare - 为解码一个 JPEG 图片做准备
- jd_decomp - 解码 JPEG 图片

四．I/O 接口函数：

TJpgDec 需要用户自定义 2 个 I/O 接口函数，用于输入 JPEG 数据和输出解码后得到的像素数据。

- Input funciotn - 从输入的数据流中读取 JPEG 的数据
- Output function - 把解码后得到的像素数据发送到输出设备

五．备注说明：

TJpgDec 应用模块是一款可用于教育和研发的开源软件。你完全可以根据自己的项目需要或者商业产品的需要，自由更改本软件，而不用担负任何个人责任。

六．ChaN 的个人网页：（即 TjpgDec 库函数下载地址）http://elm-chan.org/fsw/tjpg/00index.html

注：其他信息，如各版本信息，在此不作翻译了。

第一章 TjpgDec 使用方法

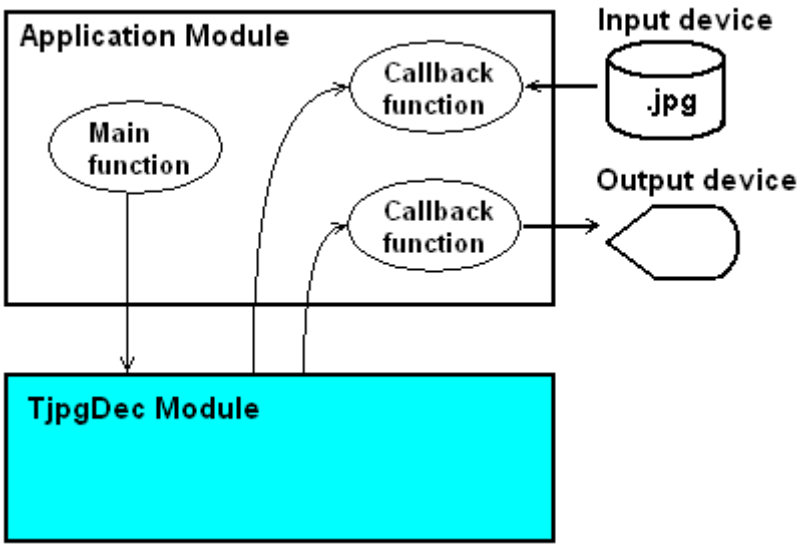
1. 如何使用 TjpgDec

首先创建并运行下面的例程中的程序，这是使用 TjpgDec 的基础，同时，这也是 TjpgDec 的典型用法，而且这种用法可以减少调试中的错误出现。TjpgDec 模块分为两个部分，第一个部分对 JPEG 图片的信息进行分析，第二个部分用于解码 JPEG 图片。（译者注：原技术手册中的例程是一个在 PC 机上运行的例程，能够作为 MCU 移植的一个参考。在本文最后面，本人根据自己对 TjpgDec 的移植情况写出来了一个移植到 STM32 单片机的例程。）

使用步骤：

1. 初始化输入的数据流（例如：打开一个图片文件）。
2. 分配 JPEG 解压缩对象和工作区。
3. 调用函数 *jd_prepare()* ，用于分析 JPEG 图片的信息，并为接下来的解码做好准备。
4. 根据 JPEG 图片信息分析得到的数据，对输出设备进行初始化 。
5. 调用函数 *jd_decomp()* ，对 JPEG 图片进行解码。

2. 系统架构



3. 例程：

```
/*-----------------------------------------------*/
/* 可以在 PC 机上运行的 TjpgDec 模块快速评估测试例程*/
/*-----------------------------------------------*/
#include <stdio.h>
#include <string.h>
#include "tjpgd.h"
//用户自定义的解码工作中使用的应用信息的结构体（在 MCU 的使用中不需要此结构体）
typedef struct {
    FILE *fp;       /*指向输入文件的指针（JPEG 文件）*/
    BYTE *fbuf;     /*指向输出缓冲区的指针*/
    UINT wfbuf;     /*输出缓冲区的宽度[pix] */
} IODEV;

/*-----------------------------*/
/*用户自定义的输入函数          */
/*-----------------------------*/
//jd：指向需要解码的对象的指针，记录了需要解码的对象的应用信息
//buff：指向存储读取的数据的缓冲区的指针
//nbyte：将要从输入数据流中读取/删除的数据的字节数
```

```
UINT in_func (JDEC* jd, BYTE* buff, UINT nbyte)
{
    IODEV *dev = (IODEV*)jd->device;   /* jd_prepare 函数中使用的应用信息结构体 */
    if (buff)//输入数据缓冲区的指针不为 NULL 的情况下
     {
        /* 从数据输入流中读取数据，返回值为读取到的字节数目*/
        return (UINT)fread(buff, 1, nbyte, dev->fp);//从指定文件中读取指定数据量的数据到输入缓冲区中
     } else {    //当 buff 为 NULL 时，表示从输入流中删除数据
        /* 从数据输入流中删除数据*/
        return fseek(dev->fp, nbyte, SEEK_CUR) ? 0 : nbyte;//对文件中的数据进行重新定位，相当于删除相应的数据
    }
}
/*-----------------------------*/
/*用户自定义的输出函数          */
/*-----------------------------*/
//jd：指向需要解码的对象的指针，记录了需要解码的对象的应用信息
// bitmap：用于接收已解码好的 RGB 数据
// rect：像素块的大小（在 TjpgDec 中，图像的显示是以块的形式出现的，而不是一个一个像素点的形式）
// dev->fbuf：指向存储输出数据的缓冲区的指针
UINT out_func (JDEC* jd, void* bitmap, JRECT* rect)
{
    IODEV *dev = (IODEV*)jd->device;//从解码对象的信息中取出应用信息
    BYTE *src, *dst;
    UINT y, bws, bwd;

    /* Put progress indicator */
    if (rect->left == 0)
     {
        printf("\r%lu%%", (rect->top << jd->scale) * 100UL / jd->height);
      }

    /* 把解码后得到的数据转存到输出缓冲区中 (使用 RGB888 格式) */
    src = (BYTE*)bitmap;//指向已经解码好的 RGB 位图数据的指针
    dst = dev->fbuf + 3 * (rect->top * dev->wfbuf + rect->left);  /* 图片的矩形像素块的左上角的地址（图片起始点）*/
    bws = 3 * (rect->right - rect->left + 1);     /* 矩形像素块的一行数据的宽度[byte] */
    bwd = 3 * dev->wfbuf;                         /* 输出数据缓冲区的一行的宽度 [byte] */
    for (y = rect->top; y <= rect->bottom; y++)//从顶端开始取数据，一直取到最底端（bottom），每次取一行的数据
     {
        memcpy(dst, src, bws);   /* 存储矩形像素块一行的数据 */
        src += bws; dst += bwd;  /* 准备存储矩形像素块下一行的数据*/
      }

    return 1;    /* 返回值为 1，继续执行解码操作 */
}
/*-----------------------------*/
/* 主函数                       */
/*-----------------------------*/
int main (int argc, char* argv[])
{
    void *work;       /* 指向解码工作区的指针 */
    JDEC jdec;        /* 解码的对象 */
    JRESULT res;      /* TjpgDec 应用函数的返回值*/
    IODEV devid;      /* 用户自定义的应用信息结构体 ，包含待打开的文件以及输出缓冲区的指针等*/

    /* 打开一个 JPEG 文件 */
    if (argc < 2) return -1;
    devid.fp = fopen(argv[1], "rb");//打开待输入的文件
    if (!devid.fp) return -1;

    /* 为 TjpgDec 分配一个解码工作区 */
    work = malloc(3100);

    /* 做好准备解码的工作 */
    res = jd_prepare(&jdec, in_func, work, 3100, &devid);
```

```
    if (res == JDR_OK)
    {
       /* 准备解码. 此处的图片信息是有效的. jdec.width：图片的宽度， jdec.height 图片的高度 */
        printf("Image dimensions: %u by %u. %u bytes used.\n", jdec.width, jdec.height, 3100 - jdec.sz_pool);

        devid.fbuf = malloc(3 * jdec.width * jdec.height); /* 为输出缓冲区分配空间 (使用 RGB888 格式) */
        devid.wfbuf = jdec.width;

        res = jd_decomp(&jdec, out_func, 0);   /* 开始解码，图像输出比例为 1/1 */
        if (res == JDR_OK)
           {
            /* 解码成功. 解码的数据存储于输出缓冲区中. */
            printf("\rOK  \n");
           }
        else //解码失败
          {
            printf("Failed to decompress: rc=%d\n", res);
           }

        free(devid.fbuf);    /* 释放输出缓冲区内存 */

    }
 else //解码准备工作失败
    {
        printf("Failed to prepare: rc=%d\n", res);
    }

    free(work);             /*释放解码工作区 */
    fclose(devid.fp);       /*关闭 JPEG 文件 */
    return res;
}
```

第二章．库函数解析

1. jd_prepare

函数功能：此函数用于分析 JPEG 图片的信息，并为后续的解码过程创建一个解压对象。

函数头：JRESULT jd_prepare （

```
JDEC* jdec,              /* 指向空白的待解压的对象的指针 */
 UINT(*infunc)(JDEC*,BYTE*,UINT), /* 指向输入函数的指针*/
void* work,              /* 指向本次解压工作区的指针 */
UINT sz_work,            /* 解压工作区的大小 */
 void* device             /* 应用信息接口 */
);
```

入口参数：

Jdec：用于后续进行解压的图片对象。

（*infunc）：用于读取 JPEG 图片数据的输入函数，需要用户自定义。

Work：指向一次解压时使用的工作区的指针，它必须是一个字对齐(4 字节对齐)的数据

sz_work：指定解压工作区的大小，根据 JPEG 内置参数表的不同，工作区最多需要 3092 字节的空间。

Device： 指向本次解码使用的用户定义的对象信息的指针. 它被存储到解压对象的成员中。在一次解码工作中可以参考 I/O 函数来定义它。如果 I/O 接口在一个项目中是固定的，或者这个功能不需要，那么可以把它设置为 NULL。

返回值：

JDR_OK：函数执行成功，并且解压对象有效。

JDR_INP：由于磁盘错误或者错误的数据结束而导致输入函数运行失败

JDR_MEM1：为解码工作配置的工作区大小不够

JDR_MEM2：输入缓冲区的容量过小， JD_SZBUF 数据过小。
JDR_PAR：入口参数错误，指定的工作区指针为 NULL.
JDR_FMT1: 入口参数错误，指定的 JPEG 数据无法读取
JDR_FMT2：非 JPEG 图像
JDR_FMT3：非标准格式的彩色 JPEG 图像
备注：
jd_prepare 函数是执行 JPEG 图片解码的第一步工作. 此函数用于分析 JPEG 图片的信息，并为后续的解码过程创建一个解码对象。此函数调用成功后,便可以调用 jd_decomp 函数执行解码的工作。此函数可以提取 JPEG 的尺寸参数，可用于配置用户自定义的对象信息，以及用于后续的解压工作。

2. jd_decomp
函数功能：解压 JPEG 图片，并输出 RGB 格式的数据
函数头：JRESULT jd_decomp (
JDEC* *jdec*, /* 指向一个已定义的解码对象 */
UINT(**outfunc*)(JDEC*,void*,JRECT*), /* 指向输出函数的指针 */
BYTE *scale* /* 图像解压尺寸比例配置 */
);
入口参数：
Jdec：指向一个已定义的解码对象
（*outfunc）：指向一个用户定义的输出函数.用于以 RGB 格式输出图片的数据。
Scale：配置图像解压缩输出尺寸比例，输出图像的比例被配置为 1 / 2 ^ N (N = 0 to 3).这里需要配置的便是 N 的大小，如果输出比例功能被关闭 (JD_USE_SCALE == 0),那么这个数据必须是 0；
返回值：
JDR_OK：函数执行成功
JDR_INTR:解压缩的工作被输出函数中断,即输出函数中出现错误或者输出函数命令本函数解码中止。
JDR_INP：由于磁盘错误或者错误的数据结束标识而导致输入函数运行失败
JDR_PAR：给定的图像输出比例参数错误
JDR_FMT1：数据结构错误，无法加载 JPEG 图片的数据
备注：

jd_decomp 函数是解码一个 JPEG 图片的第二步工作.此函数解码 JPEG 图片,并通过用户提供的输出函数输出解码得到的 RGB 数据.此函数执行过以后，待解码对象自动失效。

图像输出比例参数可以被预设.它可以被预设为 1/2, 1/4 或者 1/8.例如： 当使用 1/4 的图像输出比例解码一个大小为 1024x768 的 JPEG 图像时,它的输出尺寸将会是 256x192. 一般情况下，图像输出比例为 1/2、1/4 时，与不进行比例输出时相比会轻微的降低解码的速度。但是当图像输出比例等于 1/8 时，将会比无输出比例时快 2/3 时间，因为解码的部分功能将会被省略. 这个特性被用于产生缩略图。

3.Input Function

函数功能：用户自定义的用于输入数据的函数

函数头：UINT in_func (
JDEC* *jdec*, /* 指向待解码对象的指针 */
BYTE* *buff*, /* 指向用于存储读取的数据的缓冲区的指针 */
UINT *ndata* /* 待读取数据的数量 */
);

入口参数：
Jdec：指向待解码对象的指针
Buff:指向用于存储读取的数据的缓冲区的指针. 当此指针为 NULL 时,将从输入数据流中删除数据。
Ndata：指定将要从输入数据流中读取/删除的数据的字节数。
返回值：
返回读取/删除的字节数目；如果返回值为 0，则 jd_prepare 和 jd_decomp 函数将返回 JDR_INP。
备注：
此函数是 TjpgDec 模块的数据输入接口. 相应的解码流程会使用被传递给函数 jd_prepare 的第 5 个入口参数的数据进行解码工作,此参数是用户自定义的解码对象信息,被存储于 jdec->device 。(译者注：在 MCU 的应用中，此数据类型仅使用 FIL 类型用以指向一个待解码的 JPEG 文件)
4. Output Function
函数功能：用户自定义的输出函数，用于向输出设备输出解码后的图片数据
函数头：UINT out_func (

```
JDEC* jdec,      /*指向一个已定义的解码对象*/
void* bitmap,    /*指向已解码好但等待输出的 RGB 位图数据的指针*/
JRECT* rect      /* bitmap 所代表的待输出的像素块的大小 */
);
```

入口参数：
Jdec：指向一个已定义的解码对象
Bitmap：指向已解码好但等待输出的 RGB 位图数据的指针
Rect: *bitmap* 所代表的待输出的像素块的尺寸大小（译者注：看其定义可知，此结构体使用“上下左右”的值来表示一个矩形图片像素块的四个点）
返回值：一般情况下返回 1.这个返回值将允许 TjpgDec 模块继续进行解码工作。当它的返回值为 0 时，jd_decomp 函数将会终止解码工作，并返回 JDR_INTR。这种利用 out_func 的返回值来终止解码的方式，是非常有用的。
备注：
此函数是 TjpgDec 模块的输出接口。相应的解码流程会使用被传递给函数 jd_prepare 的第 5 个入口参数的数据进行解码工作,此参数是用户自定义的解码对象信息,被存储于 jdec->device 。(译者注：在 MCU 的应用中，此数据类型仅使用 FIL 类型用以指向一个待解码的 JPEG 文件)
解压后得到的 RGB 位图数据可以在此函数中被发送到显示设备或者数据缓冲区,RGB 位图数据的第一个数据是图片的左上角的矩形块的像素数据，第二个数据是右边一个矩形块的像素数据，最后一个数据是图片右下角的矩形块的像素数据。
每一个矩形像素块的大小是根据裁剪方式和图像输出比例的不同而从 1x1 到 16x16 不等的（译者注：像素块的大小是 TjpgDec 自由选择的，用户无法指定）。如果矩形块的大小超过了数据缓冲区的容量，那么在这个输出函数中需要进行一定的裁剪。
（译者注：在 TjpgDec 模块中，图像的像素数据的输出是以像素块为单位的，例如：当使用 LCD 显示图片时，可以看到图片是以一块一块的形式从左上角到右下角把整个图片显示出来的。所以，在 MCU 的使用中，当使用 TFTLCD 显示图片时，需要在这个输出函数中先根据 *rect* 提供的值在 LCD 上定义出一个图片显示的矩形窗口，然后再把 *bitmap* 数据输出到这个矩形窗口中，而 TjpgDec 会根据图片的信息自动改变 *rect* 的值，用户不用计算下一个像素块的显示位置。）
RGB 位图数据的格式需要使用配置选项中的 JD_FORMAT 进行定义。（在文件 tjpgd.h 中）
当定义为 24 位真彩色（RGB888）时，RGB 位图数据使用 3 个字节表示一个像素点：RRRRRRRR, GGGGGGGG, BBBBBBBB, RRRRRRRR, GGGGGGGG, BBBBBBBB, ...
当定义为 RGB565 格式时，RGB 位图数据使用一个半字（16 位）的数据表示一个像素点：RRRRRGGGGGGBBBBB, RRRRRGGGGGGBBBBB, RRRRRGGGGGGBBBBB, ...

第三章 函数中使用的结构体

1. JDEC：此结构体存储有关解码使用的数据、指针和各种标志。（待解码对象信息记录表）

```
struct JDEC {
    UINT dctr;                  /* 输入缓冲区的容量 */
    BYTE* dptr;                 /* 指向当前正在读取的数据的指针 */
    BYTE* inbuf;                /* 指向输入缓冲区的指针 */
    BYTE dmsk;                  /* 当前正在读取的字节中的当前的一个位 */
    BYTE scale;                 /* 图像输出比例 */
    BYTE msx, msy;              /* 单片机一个内存块的大小（宽度和高度）*/
    BYTE qtid[3];               /* 每个组件的量化表的编号 */
    SHORT dcv[3];               /* Previous DC element of each component */
    WORD nrst;                  /* Restart inverval */
    UINT width, height;         /* 输入的图片的大小（像素） */
    BYTE* huffbits[2][2];         /*霍夫曼比特分布表 [yc][dcac] */
    WORD* huffcode[2][2];       /* 霍夫曼码字表 [yc][dcac] */
    BYTE* huffdata[2][2];       /* 霍夫曼解码数据表 [yc][dcac] */
    LONG* qttbl[4];             /* De-quaitizer tables [id] */
    void* workbuf;              /* 指向解码工作区的指针 */
    BYTE* mcubuf;               /* MCU 工作区的指针 */
    void* pool;                 /* 可用的内存指针 */
    UINT sz_pool;               /* 可用的内存的大小（以字节为单位） */
    UINT (*infunc)(JDEC*, BYTE*, UINT);    /* 指向数据输入函数的指针 */
    UINT (*outfunc)(JDEC*, void*, JRECT*); /* 指向数据输出函数的指针 */
    void* device;              /* 指向解码工作中用到的应用信息的指针 */
};
```

2. JRESULT：应用函数的返回值定义

```
typedef enum {
    JDR_OK = 0,         /* 0: 成功*/
    JDR_INTR,  /* 1: 被输出函数中断 */
    JDR_INP,   /* 2: 磁盘错误，或输入数据错误 */
    JDR_MEM1,  /* 3: 为解码工作配置的工作区大小不够*/
    JDR_MEM2,  /* 4: 输入缓冲区的容量过小，JD_SZBUF 数据过小*/
    JDR_PAR,   /* 5: 参数错误 */
    JDR_FMT1,  /* 6: 图片数据错误 */
    JDR_FMT2,  /* 7: 不支持的图片类型 */
    JDR_FMT3   /* 8: 不支持的 JPEG 格式 */
} JRESULT;
```

3. JRECT：待输出的像素块的大小

```
typedef struct {
    WORD left, right, top, bottom;
} JRECT;
```

此结构体中的各个数值分别表示一个矩形图像的像素块的上下左右的点在整个图片中的位置

//移植 TJpgDec 时需要补充和更改的相关配置

```
// 以下 4 句在 tjpgd.h 中修改
#define        JD_SZBUF          2048      /* 输入缓冲区的大小（必须是 512 的倍数）*/
#define        JD_FORMAT          1        /* 输出 RGB 格式：0:RGB888 (3 字节/像素), 1:RGB565 (1 半字/) */
#define        JD_USE_SCALE       1        /* 使用去除图片冗余数据的功能 */
#define        JD_TBLCLIP         1       /* 使用快速裁剪换算表（表的大小为 1kB） */
//有关 TFTLCD 的尺寸的定义
#define TFTLCD_WIDTH     240          //2.4 寸 TFTLCD 屏幕的宽度
#define TFTLCD_HEIGHT    320          //2.4 寸 TFTLCD 屏幕的高度

static int Mask_Left  = 0;                    //TFTLCD 最左边的起始点
static int Mask_Right = TFTLCD_WIDTH - 1;     //TFTLCD 最右边的结束点
static int Mask_Top   = 0;                    //TFTLCD 最上方的起始点
static int Mask_Bottom= TFTLCD_HEIGHT - 1;    //TFTLCD 最下方的结束点
```

//用户自定义的用于输入文件数据的功能函数

```
UINT STM32_ in_func (
                        JDEC* jd,         /*储存待解码的对象信息的结构体 */
                        BYTE* buff,         /* 输入数据缓冲区 (NULL:删除数据) */
                        UINT nd             /*需要从输入数据流读出/删除的数据量*/
                    )
{
    UINT  rb;
    FIL  *fil = (FIL*)jd->device;    /* 待解码的文件的信息，使用 FATFS 中的 FIL 结构类型进行定义 */
    if (buff)     /*读取数据有效，开始读取数据 */
    {
        f_read(fil, buff, nd, &rb); //调用 FATFS 的 f_read 函数，用于把 jpeg 文件的数据读取出来
        return rb;        /* 返回读取到的字节数目*/
    }
    else
    {
        return (f_lseek(fil, f_tell(fil) + nd) == FR_OK) ? nd : 0;/* 重新定位数据点，相当于删除之前的 n 字节数据 */
    }
}
```

//用户自定义的用于输出 RGB 位图数据的功能函数

```
UINT STM32_ out_func (
                      JDEC* jd,         /*储存待解码的对象信息的结构体*/
                      void* bitmap,     /* 指向等待输出的 RGB 位图数据 的指针*/
                      JRECT* rect       /* 等待输出的矩形图像的参数 */
                      )
{
    jd = jd;    /* 说明：输出函数中 JDEC 结构体没有用到 */
    STM32_ Display (rect->left, rect->right, rect->top, rect->bottom, (uint16_t*)bitmap);
    return 1;    /*返回 1，使解码工作继续执行 */
}
```

```
// 在 TFTLCD 屏幕上显示图片
void STM32_ Display (
                          int left,                                /*图片左方起始点，即一行的起始点 */
                          int right,                               /*图片右方的结束点，即一行的结束点*/
                          int top,                                 /* 图片上方的起始点，即一列的起始点 */
                          int bottom,                              /*图像下方的结束点，即一列的结束点 */
                          const uint16_t * RGB_Data_Pointer        /* 待显示的图像数据，RGB 格式*/
                      )
{
    int  height_jpeg, width_jpeg , width_clip=0;
    u32  i, len, width_clip;
     if (left > right || top > bottom)
          {
              return;   //起始点错误，不执行显示功能
          }
    if (left > Mask_Right || right < Mask_Left  || top > Mask_Bottm || bottom < Mask_Top)
          {
              return;                 // 图像超出屏幕的显示范围，则不执行显示功能
          }
//开始对图片的宽度和高度进行裁剪和修改
    height_jpeg = bottom - top  + 1;    // 计算图像的高度
    width_jpeg = right  - left + 1;  //计算图像的宽度
    if (top < Mask_Top)          // 裁减掉在图片顶部超出显示范围的部分数据
          {
              RGB_Data_Pointer += width_jpeg * (Mask_Top- top); //计算出需要删除的数据量
              height_jpeg-= Mask_Top- top;
              top = Mask_Top;
          }
    if (bottom > Mask_Bottm)     // 裁减掉在图片底部超出显示范围的部分数据
          {
               height_jpeg-= bottom - Mask_Bottm;
               bottom = Mask_Bottm;
          }
    if (left < Mask_Left)          // 裁减掉在图片左边超出显示范围的部分数据
          {
              RGB_Data_Pointer += Mask_Left - left; //水平方向的数据指针右移
              width_jpeg -= Mask_Left - left;  //水平宽度减小
              width_clip+= Mask_Left - left;//记录裁减掉的宽度
              left = Mask_Left;
          }
    if (right > Mask_Right)          // 裁减掉在图片右边超出显示范围的部分数据
          {
              width_jpeg -= right - Mask_Right;
              width_clip+=right - Mask_Right;
              right = Mask_Right;
          }
//设置 TFTLCD 显示屏的显示参数
  LCD_Set_Window(left,top,right,bottom);//在此处加入设置 TFTLCD 屏幕显示图片的窗口的功能函数，设置矩形显示区域
  LCD_WR_REG(R32,left);
  LCD_WR_REG(R33,top); //在此处加入设置 TFTLCD 光标位置的功能函数，把像素点定位到起始点（左上角）
//向 TFTLCD 显示屏控制器写入数据
 LCD_WR_REG(R34); //在此处加入开始写入 TFTLCD 的 GRAM 功能的函数，
                  //根据 TFTLCD 的控制器的特性，接下来可以连续地向上面设置好的矩形区域丢数据
 while(height_jpeg --)  //控制显示区域的高度
     {
          x_width=width_jpeg;
         //在此处加入向 TFTLCD 写入数据的功能函数，在水平方向输出矩形区域一行的数据
         wehile(x_width --)   LCD_WR_DATA(*data_after_clip ++);
         //图片的水平方向的调整，去除裁减掉的像素数据
         RGB_Data_Pointer+= width_clip;
     }
}
```

```
//在主函数中调用的 JPEG 图片显示函数

BYTE Buff[4096] __attribute__ ((aligned(4)));    //定义全局数组变量，作为输入和输出的缓冲区，强制 4 字节对齐

uint32_t  STM32_JPEG_DISPLAY(FIL* f_Jpeg,const char* filename)
{
    FRESULT    Res_FatFs;//FATFS 系统函数返回值
    JDEC       jd;          /* 待解码的对象的信息记录表 */
    JRESULT    Res_TjpgDec; // TjpgDec 系统函数返回值
    BYTE       scale; //图像输出比例
     //使用 FATFS 的 f_open 函数以读方式打开文件
    Res_FatFs = f_open(f_Jpeg,filename,FA_READ);
    if(Res_FatFs!= FR_OK) //打开文件失败，返回
    {
        Delay(10000);
        f_close(f_Jpeg);
        return Res_FatFs;
    }
     //执行解码的准备工作，调用 TjpgDec 模块的 jd_prepare 函数
    Res_TjpgDec = jd_prepare(&jd, STM32_ in_func, Buff, sizeof(Buff), f_Jpeg);
    if (Res_TjpgDec == JDR_OK)
    {
        for (scale = 0; scale < 3; scale++) //确定输出图像的比例因子
        {
            if ((jd.width >> scale) <= 240 && (jd.height >> scale) <= 320)
            {
                break;
            }
        }
      //执行解码工作，调用 TjpgDec 模块的 jd_decomp 函数
       Res_TjpgDec = jd_decomp(&jd, STM32_ out_func, scale);
    }
    else    //解码的准备工作失败，关闭文件，返回
    {
         Delay(10000);
         f_close(f_Jpeg);
         return Res_FatFs; //返回错误代码
    }
     //解码工作执行成功，返回 0
    f_close(f_Jpeg);
    return 0;
}
```