Topic:	Digital Picture Frame CubeIDE
Author:	PARA
Date:	30.11.2021

0. Before you start

This document will give you an overview of the source code for the example package Digital Picture Frame. Before you can work with it you need to set up your working environment as explained in the document “examplepackage_cubeide_gettingstarted”. Make sure you have read this document beforehand and executed all the steps to configure your STM32CubeIDE.

For a more detailed explanation of how the microcontroller works, please refer to the STM32F429 Reference Manual (RM0090) by STMicroelectronics.

Note: When importing this project into your workspace you have to copy three folders “Core”, “FATFS” and “LIBJPEG” instead of just “Core”.

1. Introduction

In this example package we will implement a digital picture frame. The microcontroller will search an inserted micro-SD-card for JPG files. Afterwards it will decode each JPG one by one and show the picture on the display. The current decoded image will be stored on the external SDRAM. As stated in the Getting started document in this case you need to copy three folders (Core, FATFS and LIBJPEG) into your CubeIDE Project in order to test it.

For the access to the SD-card we use the Open-source FatFs library by ELM-Chan with addition from Tilen Majerle. For handling the JPGs we use the Open-source LibJPEG ibrary.

2. Explanation of Example Code

2.1 Main function

We will start our walk through the code at the main function since this gives a perfect overview of the steps that we will take. At the beginning, the hardware abstraction layer (HAL) and the system clocks are initialized. Then all the peripherals are initialized. A few variables are declared for later use.

The 3.5- and 5.0-inch displays need the pin PH6 to be set to high in order to enable the backlight. Those two displays are recognized by the define BACKLIGHT_EN which is defined in the global_Display_Touch_HAL.h file.

disk_initialize(0) will initialize the SD-card (physical drive number 0) which is connected via SDIO.

After that, the display startup sequence, consisting of filling the buffers/layers with the color white and displaying our 2 logos, is started.

After the startup sequence, we start with the actual task. Since we want to display JPGs from the SD-card, we need to find the full file paths of these. This is done with the function *search_for_jpeg(char*)* which needs a 2d-array of characters as parameter. This array will be filled with the file paths. As you can see, **we assumed that there won’t be more than 20 files** and that the **longest file path won’t exceed 50 characters**. *Search_for_jpeg()* returns the number of detected JPGs.

After the JPGs are found we enter the main loop. Here, we step through the file name array and give the current name to the function *paint_JPEG_file(char*, first, Buffer_e)*. This function connects again to the SD-card and obtains and decodes the JPG-file with the given name and displays it on the given buffer with or without a 10 second delay, according to the parameter first.

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA2D_Init();
  MX_FMC_Init();
  MX_LTDC_Init();
  MX_I2C1_Init();
  MX_USART1_UART_Init();
  MX_USART3_UART_Init();
  MX_USART6_UART_Init();
  MX_TIM2_Init();
  MX_SDIO_SD_Init();
  MX_FATFS_Init();
  MX_LIBJPEG_Init();
  /* USER CODE BEGIN 2 */
  /* Declaration of variables */
  Buffer_e activeBuffer;
  bool first = 0;
  /* Change the first dimension of the 2D-array if more files need to be displayed */
  uint8_t jpeg_filenames[20][50] = {0};

  /* Backlight enable */
  HAL_GPIO_WritePin(GPIOG, GPIO_PIN_9, GPIO_PIN_SET);
  if (BACKLIGHT_EN) {
    HAL_Delay(100);
    HAL_GPIO_WritePin(GPIOH, GPIO_PIN_6, GPIO_PIN_SET);
  }
  /* Initialize the SD card */
  disk_initialize(0);

  /* Display the logos and save active buffer */
  activeBuffer = Display_Startup_Sequence();
  HAL_Delay(250);

  /* Search the SD card for jpeg files, save count of files */
  int jpeg_count = SDIO_search_for_jpeg((char *)jpeg_filenames);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    for(int i=0; i<jpeg_count; i++){
      /* Invert active buffer, copy the next image into the buffer and switch buffer */
      activeBuffer = !activeBuffer;
      paint_JPEG_file((char *)&jpeg_filenames[i][0], first, activeBuffer);
      first = 1; // This adds a delay after first displayed image
      LTDC_Switch_Buffer(activeBuffer);
    }
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

2.2 Function: SDIO_search_for_jpeg(char*)

```c
int SDIO_search_for_jpeg(char* name_array){
  // check if SD-card is inserted
  if(GPIOG -> IDR & GPIO_PIN_10){
    FRESULT res;
    FATFS sd_fs = {0};
    FATFS_SEARCH_t FindStruct = {0};
    char working_buffer[200];

    //Mount SD card
    res = f_mount(&sd_fs, "0", 1);

    if(res == FR_OK){
      FATFS_Search("/", working_buffer, 50, &FindStruct, name_array);
    }
    f_mount(0, "0", 1);
    return FindStruct.JpegCount;
  }
  else{// no SD-card detected
    return 0;
  }
}

```

This function starts the search process for the JPG files. At first you need to mount the SD-card. If this was successful we call the FATFS_Search function which checks if memory needs to be allocated and then scans the SD-card file by file by calling the *scan_files()* function. At the end, the SD-card will be unmounted by *f_mount(0, "0", 1)*.

```c
FRESULT FATFS_Search(char* dir, char* tmp_buffer, uint32_t tmp_buffer_size, FATFS_SEARCH_t* FindStructure, char* tmp_name_array) {
  uint8_t malloc_used = 0;
  FRESULT res;

  /* Reset values first */
  FindStructure->FilesCount = 0;
  FindStructure->DirCount = 0;
  FindStructure->JpegCount = 0;

  /* Check for buffer */
  if (tmp_buffer == NULL) {
    /* Try to allocate memory */
    tmp_buffer = (char *) malloc(tmp_buffer_size);

    /* Check for success */
    if (tmp_buffer == NULL) {
      return FR_NOT_ENOUGH_CORE;
    }
  }

  /* Check if there is a lot of memory allocated */
  if (strlen(dir) < tmp_buffer_size) {
    /* Reset TMP buffer */
    tmp_buffer[0] = 0;

    /* Format path first */
    strcpy(tmp_buffer, dir);

    /* Call search function */
    res = scan_files(tmp_buffer, tmp_buffer_size, FindStructure, tmp_name_array);
  } else {
    /* Not enough memory */
    res = FR_NOT_ENOUGH_CORE;
  }

  /* Check for malloc */
  if (malloc_used) {
    free(tmp_buffer);
  }

  /* Return result */
  return res;
}
```

Every time a file is detected we check the file ending to see if it is a JPG file. If it is, the file path is written in the name array. When we reached the end of the SD-card, we will return to the main function and the name array is filled with the file paths of all JPG files on the SD-card.

2.3 Function: paint_JPEG_file(char*)

The function *paint_JPEG_file()* takes the file path of the wanted picture as a parameter. It begins with mounting the SD-card and opening the wanted file on the SD-card, which creates a *FIL* object as a file handle. Then it goes on with initializing the standard error handler for the decoding process and the decompress struct which will contain all the necessary information and parameters for decompression. The function *jpeg_stdio_src()* defines our file handler as the source for the decompression process. Next, we read the header of our wanted JPG file. This will give us information about the dimension of the picture.

```
66 int paint_JPEG_file (char * filename)
67 {
68   struct jpeg_decompress_struct cinfo;
69   struct jpeg_error_mgr jerr;
70
71   FIL infile;       // source file
72   FATFS sd_fs;
73   JSAMPARRAY buffer;        // Output row buffer
74   int row_stride;       // physical row width in output buffer
75   tTimer refTime = timer_get_time_ms();
76   FRESULT fres = f_mount(&sd_fs, "0", 1);
77   if(fres != FR_OK){
78       return 1;
79   }
80   fres = f_open(&infile, filename, FA_READ);
81   if (fres != FR_OK) {
82     return 1;
83   }
84
85   cinfo.err = jpeg_std_error(&jerr);
86
87   jpeg_create_decompress(&cinfo);
88
89   jpeg_stdio_src(&cinfo, &infile);
90
91   jpeg_read_header(&cinfo, TRUE);
92
```

The next part of the function *paint_JPEG_file()* checks if the dimension of the picture is compatible with our used display. Pictures that are too large for the display will be scaled down. Since the LibJPEG library only supports scaling ratios 1/8, 2/8, … 16/8 we need to find the right scaling parameters. Afterwards, the decompression is started.

```
75   // check if downscaling is needed. If so, set the parameter for decompression accordingly
76   if(cinfo.image_width > HDP || cinfo.image_height > VDP){
77     float fract;
78     float width_fract = ((float)HDP)/cinfo.image_width;
79     float height_fract = ((float)VDP)/cinfo.image_height;
80     if(width_fract < height_fract){
81       fract = width_fract;
82     }
83     else{
84       fract = height_fract;
85     }
86
87     //only supported scaling ratios are M/8 with all M from 1 to 16
88     if(fract >= 7.0/8){
89       cinfo.scale_num = 7;
90     }
91     else if(fract >= 6.0/8){
92       cinfo.scale_num = 6;
93     }
94     else if(fract >= 5.0/8){
95       cinfo.scale_num = 5;
96     }
97     else if(fract >= 4.0/8){
98       cinfo.scale_num = 4;
99     }
100    else if(fract >= 3.0/8){
101      cinfo.scale_num = 3;
102    }
103    else if(fract >= 2.0/8){
104      cinfo.scale_num = 2;
105    }
106    else if(fract >= 1.0/8){
107      cinfo.scale_num = 1;
108    }
109    else{ // jpeg is too large to be shown on this display
110      return 2;
111    }
112    cinfo.scale_denom = 8;
113  }
```

After the decompression is finished, we go through the whole decompressed file line by line with the *while* loop. Each loop starts by storing a line of the image in the variable *buffer*. Then we read the *buffer* byte by byte. Since one byte represents either red, green or blue and the structure of *buffer* is like *red, green, blue, red, green, blue, red, …,* we combine each group of three bytes to one RGB-pixel and store the pixels on the external SDRAM. After we read the whole decompressed file and stored all pixels, we finish the decompression task with *jpeg_finish_decompress()* and *jpeg_destroy_decompress()* which frees all the memory that may still be allocated by the previous decompression function. Then we close the file handler and unmount the SD-card.

```
117   row_stride = cinfo.output_width * cinfo.output_components;
118
119   buffer = (*cinfo.mem->alloc_sarray)
120     ((j_common_ptr) &cinfo, JPOOL_IMAGE, row_stride, 1);
121
122   int rgb_ctr;
123   int pixel_ctr = 0;
124   uint32_t pixel = 0;
125   uint32_t image_data_addr = 0xC0A00000;
126   while (cinfo.output_scanline < cinfo.output_height) {
127     jpeg_read_scanlines(&cinfo, buffer, 1);
128     rgb_ctr = 0;
129     for(int i=0; i<row_stride; i++){
130       uint32_t temp = (uint32_t)*(*buffer+i);
131       switch(rgb_ctr){
132       case 0: pixel = temp<<16;
133           rgb_ctr++;
134           break;
135       case 1: pixel |= temp<<8;
136           rgb_ctr++;
137           break;
138       case 2: pixel |= temp;
139           pixel = (pixel | (0xff << 24));
140           memcpy((void *)image_data_addr+pixel_ctr*4,(void *) &pixel, (size_t) 4);  //4 byte per pixel...
141           pixel_ctr++;
142           rgb_ctr = 0;
143           break;
144       }
145
146     }
147   }
148
149   jpeg_finish_decompress(&cinfo);
150   jpeg_destroy_decompress(&cinfo);
151   f_close(&infile);
152   f_mount(0, "0", 1);
```

At last, we wait if the parameter delay is true for 10 seconds and send the image to the display.

```
153   if(delay){
154   while(HAL_GetTick()-refTime < 10000){}
155   }
156   DMA2D_Fill_Color(0xFFFFFFFF, Layer_1, displayBuffer, true);
157   DMA2D_Draw_Image((HDP-cinfo.output_width)/2, (VDP-cinfo.output_height)/2, cinfo.output_width, cinfo.output_height,
158       0xFF, DMA2D_REPLACE_ALPHA, (uint32_t)image_data_addr, CM_ARGB8888, Layer_1, displayBuffer, false);
159
```