Topic:	Digital Picture Frame
Author:	JOKI
Date:	05.07.2021

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_GettingStarted”. Make sure you have read this document beforehand and executed all the steps to configure your Workbench. Especially, that you have added all the paths to the *Source Location* and *Includes* properties including those, who are specifically required for this Digital Picture Frame example.

For a more detailed explanation of how the microcontroller works, please refer to the STM32F429 Reference Manual (RM0090) and the Standard Peripheral Library documentation provided by STMicroelectronics.

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.

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 library.

2. Explanation of Example Code

2.1 Main function

```
38
39 /*****************************************************************
40  *
41  * Function: int main(void)
42  *
43  * Summary: Entry point of display control application
44  *
45  * Parameters:
46  *
47  * Return:
48  *
49  * Revision History:
50  * Creation Date:
51  *
52  *****************************************************************/
53 int main(void) {
54
55     /* Initialize the peripherals */
56     HW_Init();
57     display_init();
58     initMemBlock();
59     disk_initialize(0);
60
61     // 20 jpges with file path shorter than 50 characters (including filename).
62     //If longer file paths are expected the value most also be changed in "tm_stm32f4_fatfs.c" in the "scan_files"-function
63     char jpeg_filename_array[20][50];
64     int jpeg_count = SDIO_search_for_jpeg(&jpeg_filename_array[0][0]);
65     for(;;){
66         for(int i=0; i<jpeg_count; i++){
67             char jpeg_filename[50];
68             strcpy(jpeg_filename, (const char*)(jpeg_filename_array+i));
69             paint_JPEG_file(jpeg_filename);
70         }
71     }
72     return 0;
73 }
74
```

We will start our walkthrough in the code at the main function since this gives a perfect overview of the steps that we will take. At the beginning, the used peripherals are initialized. *HW_Init()* initializes and turns on the display backlight. Also, it sets up the different system clocks and the timer, which counts every microsecond, and the GPIO-Pin PG10, which detects if a SD-card is inserted.

Afterwards we initialize the display which also includes the initialization of the SDRAM, because this is where the buffer for the displayed images will be located. The DMA2D is used to send the image data directly from the buffer to the LCD, which speeds up the image manipulation. Part of the display initialization is the drawing of the EBS-SYSTART logo. The function that executes this task is explained in the Example Package-documentation for “Drawing Text and Images”.

The *initMemBlock()* function initializes a 8 Mbyte memory block on the external SDRAM for use with the self-implemented malloc function. This is needed for the JPG-decoding process since this will require an amount of memory that is too large for the internal RAM.

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

After the initialization process, 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*)*. This function connects again to the SD-card and obtains and decodes the JPG-file with the given name and displays it.

2.2 Function: SDIO_search_for_jpeg(char*)

d_SDIO.c

```c
int SDIO_search_for_jpeg(char* name_array){
    // check if SD-card is inserted
    if(GPIO_ReadInputDataBit(GPIOG, GPIO_Pin_10)==1){
        FRESULT res;
        FATFS sd_fs;
        TM_FATFS_Search_t FindStruct;
        char working_buffer[200];

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

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

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 TM_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)*.

tm_stm32f4_fatfs.c

```c

FRESULT TM_FATFS_Search(char* Folder, char* tmp_buffer, uint16_t tmp_buffer_size, TM_FATFS_Search_t* FindStructure, char* tmp_name_array) {
    uint8_t malloc_used = 0;
    FRESULT res;

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

    /* Check for buffer */
    if (tmp_buffer == NULL) {
        /* Try to allocate memory */
        tmp_buffer = (char *) LIB_ALLOC_FUNC(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(Folder) < tmp_buffer_size) {
        /* Reset TMP buffer */
        tmp_buffer[0] = 0;

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

        /* 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) {
        LIB_FREE_FUNC(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.

```
93    // check if downscaling is needed. If so, set the parameter for decompression accordingly
94    if(cinfo.image_width > HDP || cinfo.image_height > VDP){
95        float fract;
96        float width_fract = ((float)HDP)/cinfo.image_width;
97        float height_fract = ((float)VDP)/cinfo.image_height;
98        if(width_fract < height_fract){
99            fract = width_fract;
100       }
101       else{
102           fract = height_fract;
103       }
104
105       //only supported scaling ratios are M/8 with all M from 1 to 16
106       if(fract >= 7.0/8){
107           cinfo.scale_num = 7;
108       }
109       else if(fract >= 6.0/8){
110           cinfo.scale_num = 6;
111       }
112       else if(fract >= 5.0/8){
113           cinfo.scale_num = 5;
114       }
115       else if(fract >= 4.0/8){
116           cinfo.scale_num = 4;
117       }
118       else if(fract >= 3.0/8){
119           cinfo.scale_num = 3;
120       }
121       else if(fract >= 2.0/8){
122           cinfo.scale_num = 2;
123       }
124       else if(fract >= 1.0/8){
125           cinfo.scale_num = 1;
126       }
127       else{ // jpeg is too large to be shown on this display
128           return 2;
129       }
130       cinfo.scale_denom = 8;
131   }
132
133   jpeg_start_decompress(&cinfo);
```

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.

```
135   row_stride = cinfo.output_width * cinfo.output_components;
136
137   buffer = (*cinfo.mem->alloc_sarray)
138         ((j_common_ptr) &cinfo, JPOOL_IMAGE, row_stride, 1);
139
140   int rgb_ctr;
141   int pixel_ctr = 0;
142   uint32_t pixel = 0;
143   uint32_t image_data_addr = 0xC05DC000;
144   while (cinfo.output_scanline < cinfo.output_height) {
145     jpeg_read_scanlines(&cinfo, buffer, 1);
146     rgb_ctr = 0;
147     for(int i=0; i<row_stride; i++){
148         uint32_t temp = (uint32_t)*(*buffer+i);
149         switch(rgb_ctr){
150         case 0: pixel = temp<<16;
151                 rgb_ctr++;
152                 break;
153         case 1: pixel |= temp<<8;
154                 rgb_ctr++;
155                 break;
156         case 2: pixel |= temp;
157                 SDRAM_Write_32b((image_data_addr - 0xC0000000)+pixel_ctr*4, pixel); //4 byte per pixel...
158                 // note that SDRAM_Write_xxx and SDRAM_Read_xxx use the offset to SDRAM_START_ADDR(=0xC0000000) as parameter
159                 pixel_ctr++;
160                 rgb_ctr = 0;
161                 break;
162         }
163
164     }
165   }
166
167   jpeg_finish_decompress(&cinfo);
168   jpeg_destroy_decompress(&cinfo);
169   f_close(&infile);
170   f_mount(0, "0", 1);
```

At last, we wait for 7 seconds before the decompressed picture is send to the display.

```
172   while(timer_get_time_ms()-refTime < 7000){}
173   DMA2D_Fill_Color(0xFFFFFFFF, Layer_1, Buffer_1);
174   DMA2D_Draw_Image((HDP-cinfo.output_width)/2, (VDP-cinfo.output_height)/2, cinfo.output_width, cinfo.output_height,
175           0xFF, REPLACE_ALPHA_VALUE, (uint32_t)image_data_addr, CM_ARGB8888, Layer_1, Buffer_1);
176
177   return 0;
178 }
```

3. Ideas for Exercise Project

Here, we want to give you some suggestions how you could modify our example code and make your own little project.

In our example code we have created a digital picture frame, which changes automatically every 7 seconds the image. After reading the guide for our example package *Processing Touch Input*, you could try to implement a swipe function. Instead of automatically parsing through the list of found JPGs, the controller could wait until a touch input is detected and depending on whether the touch input is a left-swipe or a right-swipe it could swap to next or previous image.