UM1743
User manual

STM32CubeF4 demonstration platform

Introduction

The STM32Cube™ initiative was originated by STMicroelectronics to ease developers' life by reducing development efforts, time and cost. STM32Cube™ covers the STM32 portfolio.

The STM32CubeF4 demonstration platform comes on top of the STM32Cube™ as a firmware package that offers a full set of software components based on a modules architecture allowing re-using them separately in standalone applications. All these modules are managed by the STM32CubeF4 demonstration kernel allowing to dynamically add new modules and access to common resources (storage, graphical components and widgets, memory management, Real-Time operating system)

The STM32CubeF4 demonstration platform is built around the powerful graphical library STemWin and the FreeRTOS real time operating system and uses almost the whole STM32 capability to offer a large scope of usage based on the STM32Cube HAL BSP and several middleware components.

The architecture was defined with the goal of making from the STM32CubeF4 demonstration core an independent central component which can be used with several RTOS and third party firmware libraries through several abstraction layers inserted between the STM32CubeF4 demonstration core and the several modules and libraries working around.

Contents

List of tables

List of figures

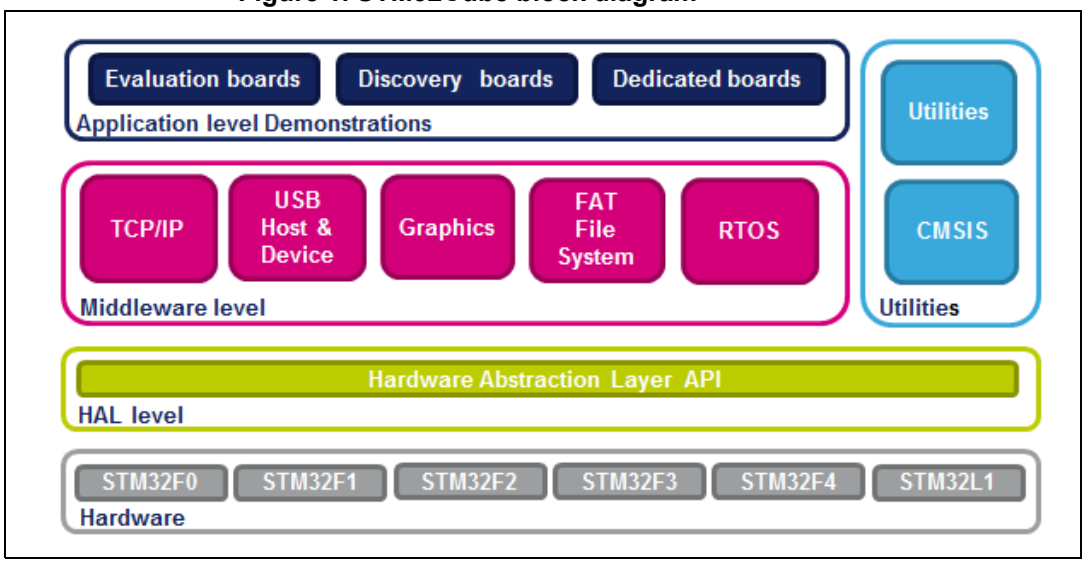
1 STM32Cube™ overview

The STM32Cube™ initiative was originated by STMicroelectronics to ease developers' life by reducing development efforts, time and cost. STM32Cube™ covers the STM32 portfolio.

STM32Cube™ Version 1.x includes:

- The STM32CubeMX, a graphical software configuration tool that allows to generate C initialization code using graphical wizards.
- A comprehensive embedded software platform, delivered per series (such as STM32CubeF4 for STM32F4 series)
  - The STM32CubeF4 HAL, an STM32 abstraction layer embedded software, ensuring maximized portability across STM32 portfolio
  - A consistent set of middleware components such as RTOS, USB, TCP/IP, Graphics
  - All embedded software utilities coming with a full set of examples.

Figure 1. STM32Cube block diagram



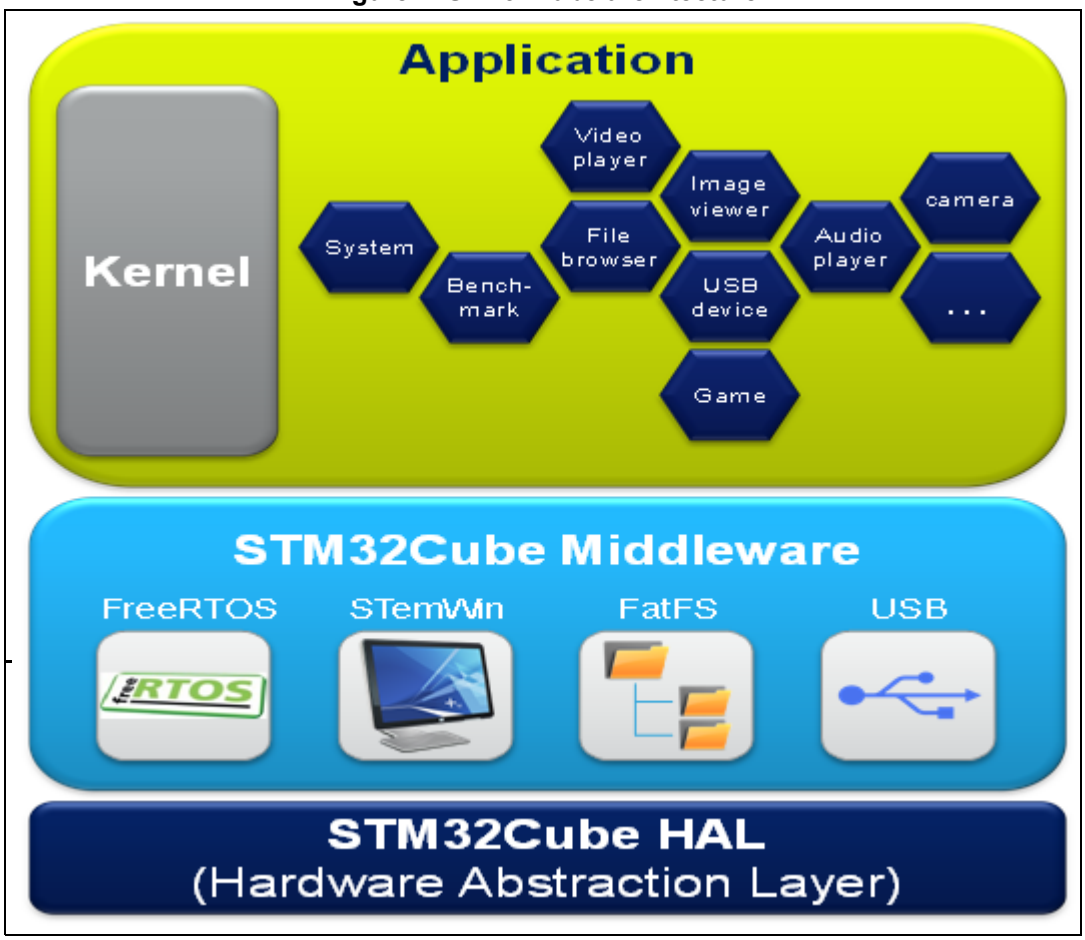
2 Global architecture

The STM32CubeF4 demonstration is composed of a central kernel based on a set of firmware and hardware services offered by the STM32Cube middleware and the several evaluation and discovery boards and a set of modules mounted on the kernel and built in a modular architecture. Each module can be reused separately in a standalone application. The full set of modules is managed by the Kernel which provides access to all common resources and facilitates the addition of new modules as shown in *Figure 2*.

Each module should provide the following functionalities and proprieties:

1. Icon and graphical aspect characteristics.
2. Method to startup the module.
3. Method to close down safety the module (example: Hot unplug for Unit Storage)
4. Method to manage low power mode
5. The module application core (main module process)
6. Specific configuration
7. Error management

Figure 2. STM32Cube architecture



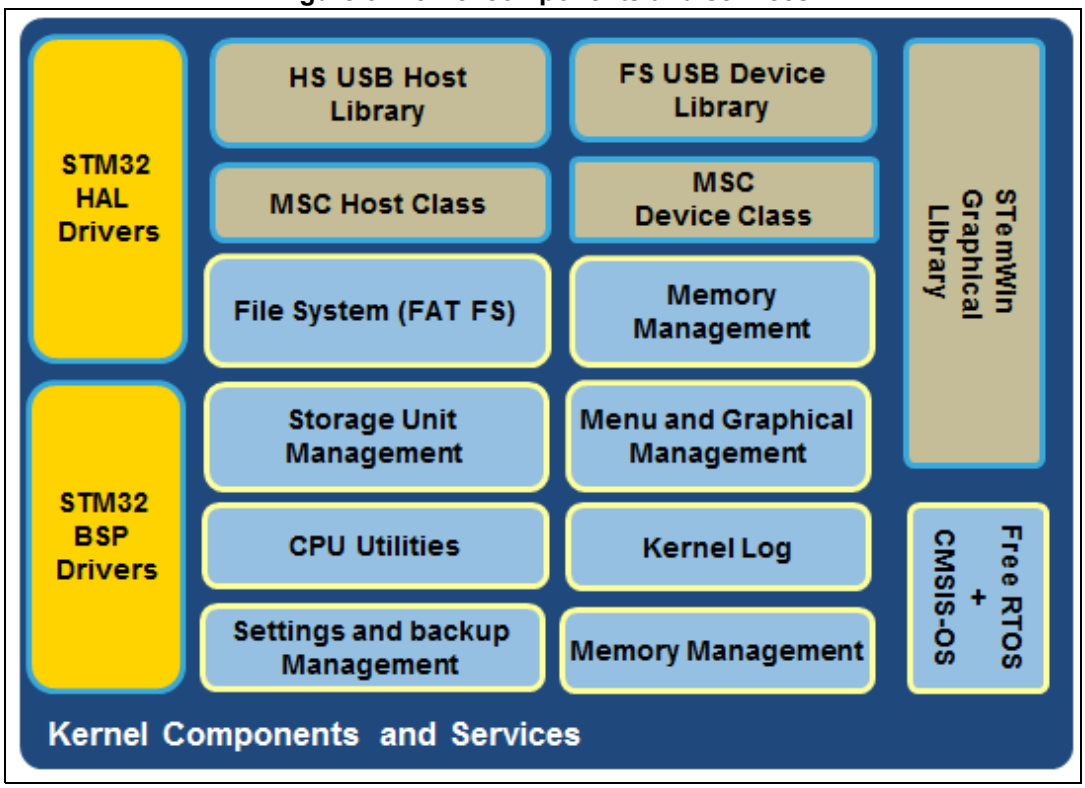
3 Kernel description

3.1 Overview

The role of the demonstration kernel is mainly to provide a generic platform that control and monitor all the application processes, the kernel provides a set of friendly user APIs and services that allow to the user modules to have access to all the hardware and firmware resources and provide the following tasks and services:

- Hardware and modules initialization:
  - BSP initialization (LEDs, SDRAM, Touch screen, CRC, SRAM, RTC and audio
  - GUI initialization and Touch screen calibration
- Memory management
- Kernel log
- Graphical resources and main menu management.
- Storage managements (USB Disk flash and microSD)
- System monitoring and settings
- Time and date resources management
- File browsing and contextual menu
- CPU utilities (CPU usage, running tasks)

Figure 3. Kernel components and services



3.2 Kernel Initialization

The first task of the kernel is to initialize the hardware and firmware resources to make them available to its internal processes and the modules around it. The kernel starts by initializing the HAL, system clocks and then the hardware resources needed during the middleware components:

- LEDs and Touchscreen
- SDRAM/SRAM
- Backup SRAM
- RTC
- Audio Interface

Once the low level resources are initialized, the kernel performs the STemWin GUI library initialization and prepares the following common services:

- Memory manager
- Storage units
- Modules manager
- Kernel Log

Upon full initialization phase, the kernel adds and links the system and user modules to the demonstration core.

3.3 Kernel processes and tasks

The kernel is composed of three main task managed by FreeRTOS through the CMSIS-OS wrapping layer:

- Start Thread: this is the main kernel task, it has a normal priority and manages the modules and some delayed common services including the startup window;
- GUI Thread: this task Initializes the demonstration main menu and then handles the graphical background task when requested by the STemWin;

```
234 /**
235   * @brief  Start task
236   * @param  argument: pointer that is passed to the thread function as start argument.
237   * @retval None
238   */
239 static void GUIThread(void const * argument)
240 {
241   (...)
242
243   /* Show the main menu */
244   k_InitMenu();
245
246   /* Gui background Task */
247   while(1)
248   {
249     GUI_Exec();
250     osDelay(30);
251   }
252 }
```

- Timer Callback: this is the callback of the Timer managing periodically the touch screen state, the Timer callback is called periodically each 40 milliseconds.

```
/**
  * @brief  Timer callbacsk (40 ms)
  * @param  n: Timer index
  * @retval None
  */
static void TimerCallback(void const *n)
{
  k_TouchUpdate();
}
```

3.4 Kernel graphical aspect

The STM32CubeF4 demonstration is built around the STemWin Graphical Library, based on SEGGER emWin one. STemWin is a professional graphical stack library, enabling Graphical User Interfaces (GUI) building up with any STM32, any LCD and any LCD controller, taking benefit from STM32 hardware accelerations, whenever possible.

The graphical aspect of the STM32CubeF4 demonstration is divided into two main graphical components:

- the startup window (*Figure 4*): showing the progress of the hardware and software initialization;
- the main desktop (shown in *Figure 5*), that handles the main demonstration menu and the numerous kernel and modules control.

Figure 4. Startup window

Figure 5. Main desktop window

3.5 Kernel menu management

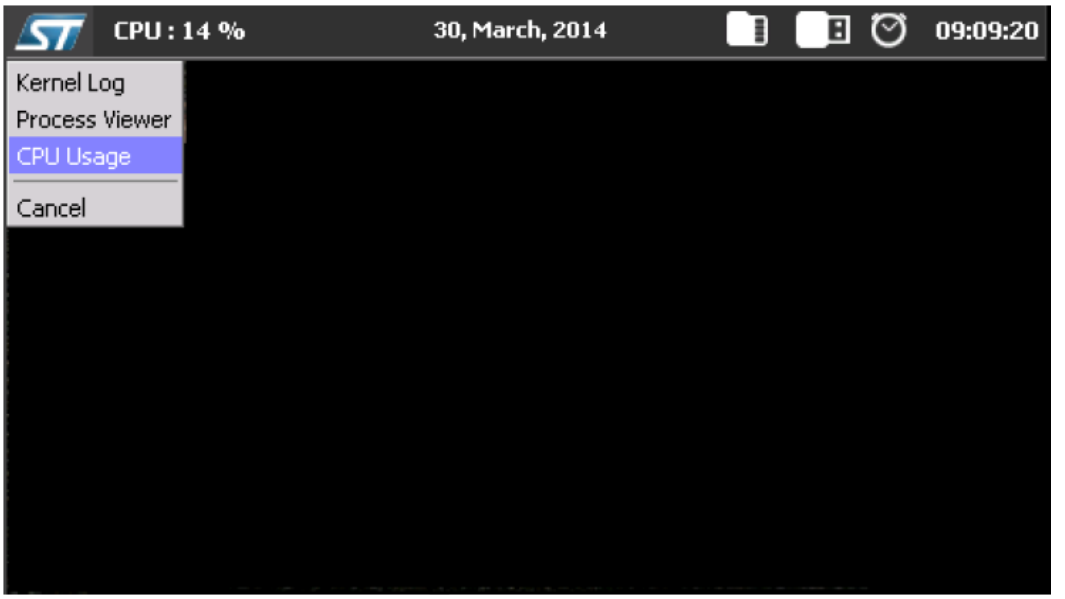
The main demonstration menu is initialized and launched by the GUI thread. Before the initialization of the menu the following actions are performed:

- Draw the background image.
- Create the status bar.
- Restore general settings from backup memory.
- Setup the main desktop callback to manage main window messages.

The main desktop is built around two main graphical components:

- The status bar (*Figure 6*): indicates the storage units connection status, current time and date and a system button to allow to get system information like (running task, CPU load, and kernel log).
- The icon view widget (*Figure 7*): contains the icons associated to added modules. User can launch a module by a simple click on the module icon.

Figure 6. Status bar



MS35206V1

Figure 7. Icon view widget

MS35207V1

A module is launched on simple click on the associated icon by calling to the startup function in the module structure; this is done when a WM_NOTIFICATION_RELEASED message arrives to the desktop callback with ID_ICONVIEW_MENU:

```
548 /**
549   * @brief  Callback routine of desktop window.
550   * @param  pMsg: pointer to data structure of type WM_MESSAGE
551   * @retval None
552   */
553 static void _cbBk(WM_MESSAGE * pMsg) {
554
555  (...)
556
557   switch (pMsg->MsgId)
558   {
559   case WM_NOTIFY_PARENT:
560     Id    = WM_GetId(pMsg->hWinSrc);
561     NCode = pMsg->Data.v;
562
563     switch (NCode)
564     {
565
566     case WM_NOTIFICATION_RELEASED:
567       if (Id == ID_ICONVIEW_MENU)
568       {
569
570         if(sel < k_ModuleGetNumber())
571         {
572           module_prop[sel].module->startup(pMsg->hWin, 0, 26);
573         }
574       }
575       break;
576
577       (...)
578
579     default:
580       break;
581     }
582     break;
583   default:
584     WM_DefaultProc(pMsg);
585   }
```

3.6 Modules manager

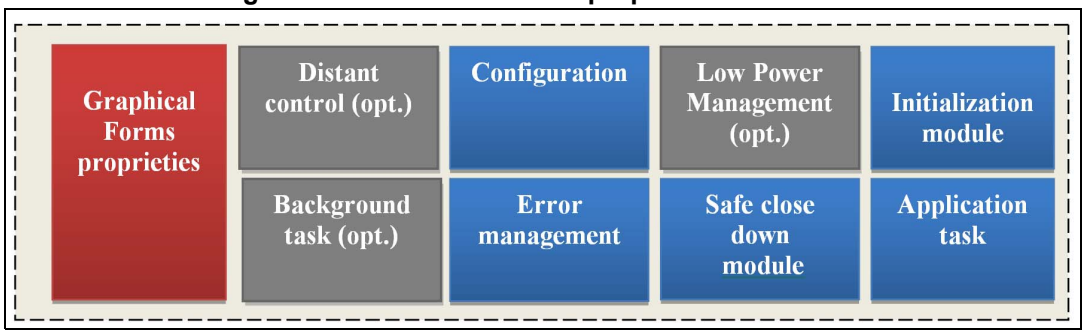
The modules are managed by the kernel; the latter is responsible of initializing the modules, initializing hardware and GUI resources relative to the modules and initializing the common resources such as the storage Unit, the graphical widgets and the system menu.

Each module should provide the following functionalities and proprieties:

1. Icon and graphical component structure.
2. Method to startup the module.
3. Method to close down safety the module (example: Hot unplug for MS flash disk)
4. Method to manage low power mode (optional)
5. The Application task
6. The module background process (optional)

7. Remote control method (optional)
8. Specific configuration
9. Error management

Figure 8. Functionalities and properties of modules



The modules could be added in run time to the demonstration and can use the common kernel resources. The following code shows how to add a module to the demonstration:

```
195   /* Add Modules*/
196   k_ModuleInit();
197
198   k_ModuleAdd(&video_player);
199   k_ModuleAdd(&image_browser);
200   k_ModuleAdd(&audio_player);
201   k_ModuleAdd(&camera_capture);
202   k_ModuleAdd(&system_info);
203   k_ModuleAdd(&file_browser);
204   k_ModuleAdd(&cpu_bench);
205   k_ModuleAdd(&game_board);
206   k_ModuleAdd(&usb_device);
207
```

A module is a set of function and data structures that are defined in a data structure that provides all the information and pointers to specific methods and functions to the kernel. This later checks the integrity and the validity of the module and inserts its structure into a module table.

Each module is identified by a unique ID. When two modules have the same UID, the Kernel rejects the second one. The module structure is defined as follows:

```
41  typedef struct
42  {
43   uint8_t     id;
44   const char  *name;
45   GUI_CONST_STORAGE GUI_BITMAP  *icon;
46   void        (*startup) (WM_HWIN , uint16_t, uint16_t );
47   void        (*DirectOpen) (char * );
48  }
49  K_ModuleItem_Typedef;
```

In this definition:

- Id: unique module identifier.
- Name: pointer to module name
- Icon: pointer to module icon (bitmap format)
- Startup: the function that create the module frame and control buttons
- DirectOpen: the function that create the module frame and launch the media associated to the file name selected in the file browser linked to a specific file extension.

3.7 Direct open feature

The direct open feature allows launching a media module directly from file browser when the extension file match with supported media type. The file extension should be previously associated to a module by using the following code:

```
195   /* Add Modules*/
196   k_ModuleInit();
197
198   k_ModuleAdd(&video_player);
199   k_ModuleOpenLink(&video_player, "emf");
```

When the file browser is opened, a simple click on a file will open a contextual menu, that direct file open can be executed, as shown in *Figure 9*:

Figure 9. Starting file execution

3.8 Backup and settings configuration

The STM32CubeF4 demonstration saves the kernel and modules settings in two different methods:

1. Using the RTC backup register (32 bits data width) , in this method the data to be saved should be a 32 bits data and could be defined as a bitfield structure, example:

```
typedef union
{
  uint32_t d32;
  struct
  {
    uint32_t repeat         : 2;
    uint32_t pause          : 2;
    uint32_t mute           : 1;
    uint32_t volume         : 8;
    uint32_t reserved       : 21;
  }b;
}
AudioSettingsTypeDef;
```

The structure could be handled than, by using the two following kernel APIs to save or restore it from the RTC backup registers.

```
void     k_BkupSaveParameter(uint32_t address, uint32_t data);
uint32_t k_BkupRestoreParameter(uint32_t address);
```

2. Using the backup SRAM: the backup SRAM is a memory that the content is not lost when the board is powered down. When available, the backup SRAM is 4 Kbytes size and located at address: BKPSRAM_BASE (0x40024000). The backup SRAM could be used as normal RAM to save file paths or big structure example:

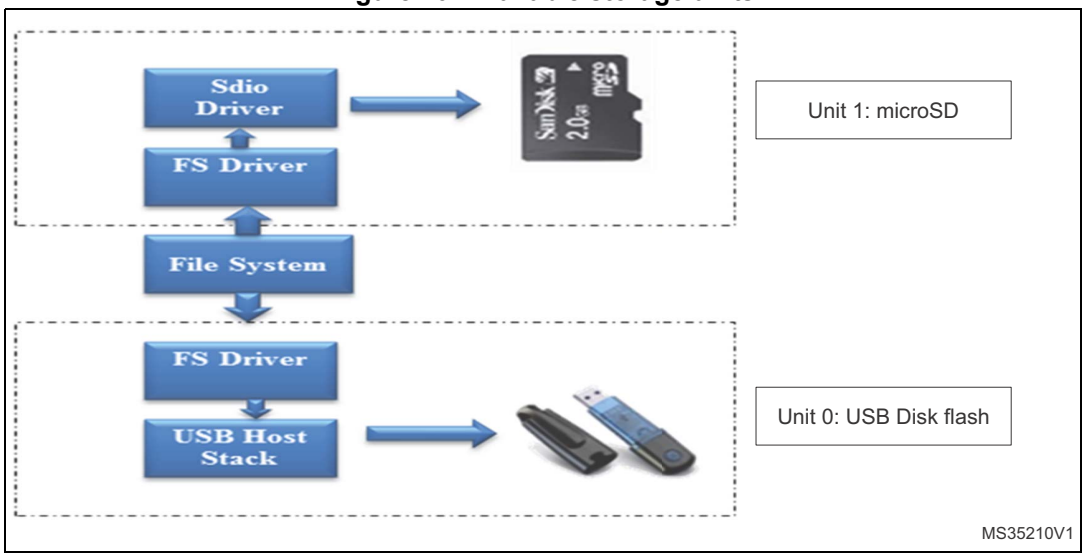
```
#define CAMERA_SAVE_PATH                    BKPSRAM_BASE + 0x000

/**
  * @brief  Set Default folder (saved in backup SRAM)
  * @param  path : pointer to the Default folder
  * @retval None
  */
static void _Check_DefaultPath (uint8_t *path)
{
  if ((*((char *)(CAMERA_SAVE_PATH )) == '0') || (*((char *)(CAMERA_SAVE_PATH)) == '1'))
  {
    strncpy((char *)path, (char *)(CAMERA_SAVE_PATH), FILEMGR_FULL_PATH_SIZE);
  }
  else
  {
    strcpy((char *)path, "0:");
  }
}
```

3.9 Storage units

The STM32CubeF4 demonstration kernel offers two storage units that can be used to retrieve audio, Image and Video media or to save captured images from the camera (*Figure 10*).

Figure 10. Available storage units



The two units are initialized during the platform startup and thus they are available to all the modules during the STM32CubeF4 demonstration run time. These two units are accessible through the standard I/O operations offered by the FatFS used in the development platform. The USB Disk flash unit is identified as the Unit 0 and available only if a USB disk flash is connected on the USB FS connector, while the microSD flash is identified as the Unit1 and available only if the microSD card is connected. The units are mounted automatically when the physical media are connected to the connector on the board.

The implemented functions in the file system interface to deal with the physical storage units are summarized in *Table 1*.

Table 1. File system interface: physical storage control functions

Function	Description
disk_initialize	Initialize disk drive
disk_read	Interface function for a logical page read
disk_write	Interface function for a logical page write
disk_status	Interface function for testing if unit is ready
disk_ioct	Control device dependent features

The full APIs functions set given by the file system interface are listed in *Table 2*:

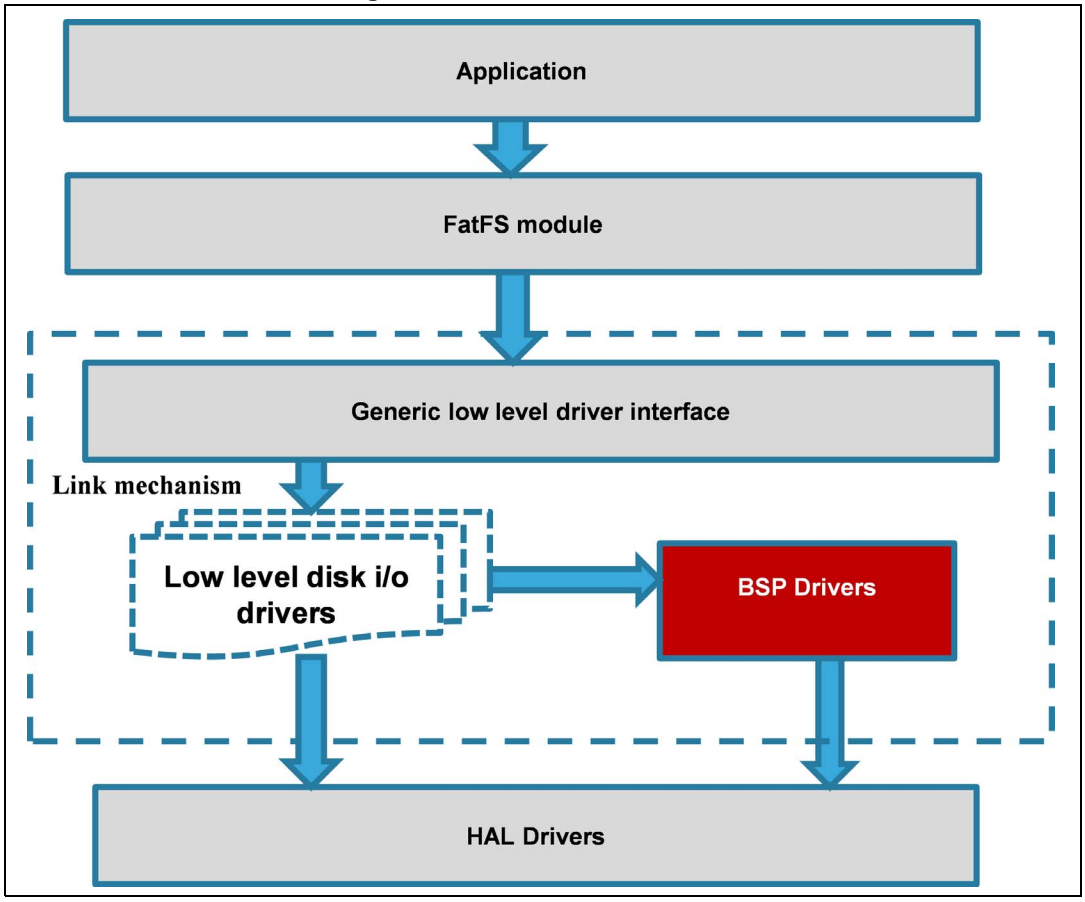
Table 2. File system interface APIs

Function	Description
f_mount	Register/Unregister a work area
f_open	Open/Create a file
f_close	Close a file
f_read	Read file
f_write	Write file
f_lseek	Move read/write pointer, Expand file size
f_truncate	Truncate file size
f_sync	Flush cached data
f_opendir	Open a directory
f_readdir	Read a directory item
f_getfree	Get free clusters
f_stat	Get file status
f_mkdir	Create a directory
f_unlink	Remove a file or directory
f_chmod	Change attribute
f_utime	Change timestamp
f_rename	Rename/Move a file or directory
f_mkfs	Create a file system on the drive
f_forward	Forward file data to the stream directly
f_chdir	Change current directory
f_chdrive	Change current drive
f_getcwd	Retrieve the current directory
f_gets	Read a string
f_putc	Write a character
f_puts	Write a string
f_printf	Write a formatted string

For the FAT FS file system, the page size is fixed to 512 bytes. USB disk flashes with higher page size are not supported.

The Storage units are built around the USB host library in high speed and the microSD BSP drivers; the software architecture is shown in *Figure 11*.

Figure 11. Software architecture



The FatFS is mounted upon the USB Host mass storage class and the SD BSP driver to allow an abstract access to the physical media through standard I/O methods.

The storage units' presence detection is handled internally by the kernel and the status bar shows the icons of the available media, as highlighted in *Figure 12*.

Figure 12. Detection of storage units

3.10 Clock and Date

The clock and date are managed by the RTC HAL driver, the RTC module initializes the LSE source clock and provides a set of methods to retrieve date and clock in addition to backup save and restore ones. *Table 3* shows the different APIs offered by the RTC module:

Table 3. APIs from the RTC module

Function	Description
k_calendarBkuplinit	Initialize RTC peripheral (clock and backup registers)
k_BkupSaveParameter	Save a 32bits word into backup registers
k_BkupRestoreParameter	Retrieve a saved 32bits word from backup registers
k_SetTime	Change system time through the RTC_TimeTypeDef
k_GetTime	Get system time into the RTC_TimeTypeDef structure
k_SetDate	Change system date through the RTC_DateTypeDef
k_GetDate	Get system date into the RTC_DateTypeDef structure

The following code shows an example of how to retrieve the system data:

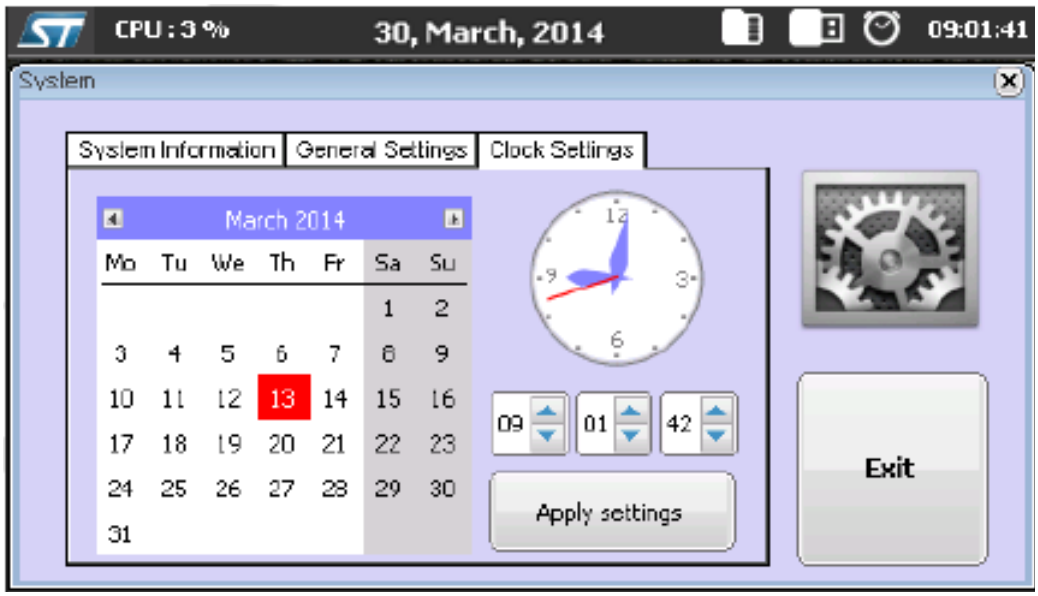
```
/**
* @brief  Save the data to specified file.
* @param  path: pointer to the saving path
* @retval File saved
*/
uint8_t  CAMERA_SaveToFile(uint8_t *path)
{
  RTC_TimeTypeDef   Time;
  RTC_DateTypeDef   Date;

  /* Create filename */
  k_GetTime(&Time);
  k_GetDate(&Date);

  sprintf((char *)filename, "/Camera_%02d%02d%04d_%02d%02d%02d.bmp",
          Date.Date,
          Date.Month,
          Date.Year + 2014,
          Time.Hours,
          Time.Minutes,
          Time.Seconds);
```

The kernel uses the RTC for modules settings saving and getting the time and date, displayed in the status bar of the main desktop. Time and date could be changed through the system module as shown in *Figure 13*.

Figure 13. Setting the time and date



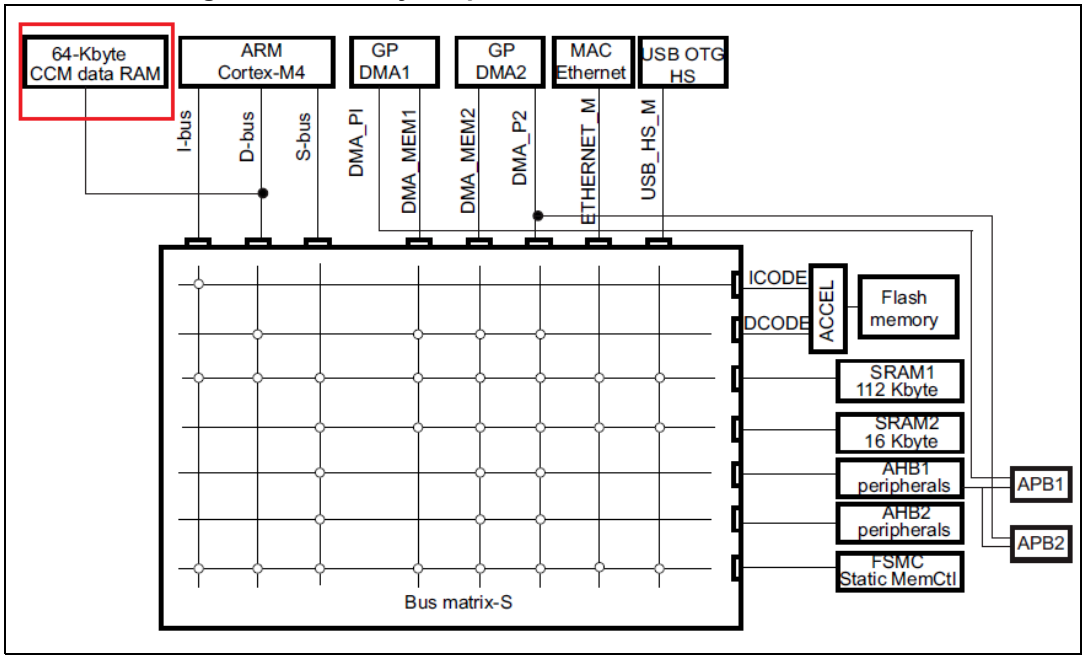
3.11 Memory Management

A huge amount of system RAM is allocated to the GUI internal heap, the kernel memory manager is used as a standalone memory allocator for some specific data blocks, like file lists and kernel log buffer.

The kernel memory manager is based on a single memory pool that could be placed anywhere in the additional internal or external memory resources. The memory heap is built on a contiguous memory blocks managed by the mem_Typedef structure through a pages table that gather the block status after each memory allocation or de-allocation operations.

For the STM32CubeF4 demonstration, the memory heap is located in the CCM data RAM.

Figure 14. Memory heap for STM32CubeF4 demonstration



The memory manager offers a set of standard high level APIs to allocate and free memory block from the predefined pool. The granularity of the memory allocation is defined by the SIZE_OF_PAGE define, set to 1024 bytes by default and the total number of available blocks depending on the heap size, in the k_mem.h file as shown in the code below.

```
43  #define MEM_BASE                0x10000000
44  #define SIZE_OF_PAGE            1024    /* 1 KBytes pages */
45  #define MAX_PAGE_NUMBER         64     /* Maximum of 64 pages */
46
```

Table 4 shows the different APIs offered by the memory manager module.

Table 4. APIs from the memory manager module

Function	Description
void k_MemInit(void)	Initialize the memory heap (base address)
void * k_malloc (size_t s)	Allocate an amount of contiguous memory blocks
void k_free (void * p)	Free an already allocated amount of RAM blocks

3.12 Demonstration repository

The STM32Cube is a component in the STM32Cube package. *Figure 15* shows the demonstration folder organization:

Figure 15. Folder structure

The demonstration sources are located in the projects folder of the STM32Cube package for each supported board. The sources are divided into five groups described as follows:

1. Core: contains the kernel files
2. Modules: contains the system and user modules including the graphical aspect and the modules functionalities.
3. Binary: demonstration binary file in Hex format

4. Config: all middleware's components and HAL configuration files
5. Project settings: a folder per tool chain containing the project settings and the linker files.

3.13 Kernel components

Table 5. Kernel components list

Function	Description
Kernel core	Kernel core and utilities
Modules	User and system modules
STM32 HAL Drivers	STM32Cube HAL driver relative to the STM32 device under use
BSP Drivers	Evaluation board (or discovery kit) BSP drivers
CMSIS	CMSIS Cortex®-M3/4 Device Peripheral Access Layer System
FatFS	FATFS File system
FreeRTOS	FreeRTOS Real Time Operating System
STemWin	STemWin Graphical Library
USBD_Library	USB Device Library (Mass Storage Class)
USBH_Library	USB Host Library (Mass Storage Class)

3.14 Kernel core files

Table 6. Kernel core files list

Function	Description
main.c	Main program file
stm32fxxx_it.c	Interrupt handlers for the application
k_bsp.c	Provides the kernel BSP functions
k_calibration.c	Touch screen calibration processes
k_log.c	Kernel Log manager
k_mem.c	Kernel memory heap manager
k_menu.c	Kernel menu and desktop manager
k_module.c	Modules manager
k_modules_res.c	Common modules resources
k_rtc.c	RTC and backup manager
k_startup.c	Demonstration startup windowing process
k_storage	Storage units manager
startup_stm32fyyyxx.s	Startup file
cpu_utils.c	CPU load calculation utility

3.15 Hardware settings

The STM32CubeF4 demonstration supports STM32F4xx devices and runs on the following demonstration boards from STMicroelectronics:

- STM324x9I-EVAL
- STM324xG-EVAL
- STM32F429I-Discovery.

Figure 16. STM32Cube demonstration boards

Table 7. Jumpers for different demonstration boards

Board	Jumper	Position description
STM324x9I-EVAL	JP16	Not fitted (used for USB device module)
	JP4/JP5	<2-3> (used for Audio demonstration)
	JP8	<2-3> (used for backup domain on battery)
STM324xG-EVAL	JP16	<2-3> (used for Audio demonstration)
	JP19	<2-3> (used for backup domain on battery)
	JP31	<2-3> (used for USB device module)
STM32F429I-Discovery	JP3	ON (Power on MCU)
	CN4	ON (Discovery mode)

4 How to create a new module

A module is composed of two main parts:

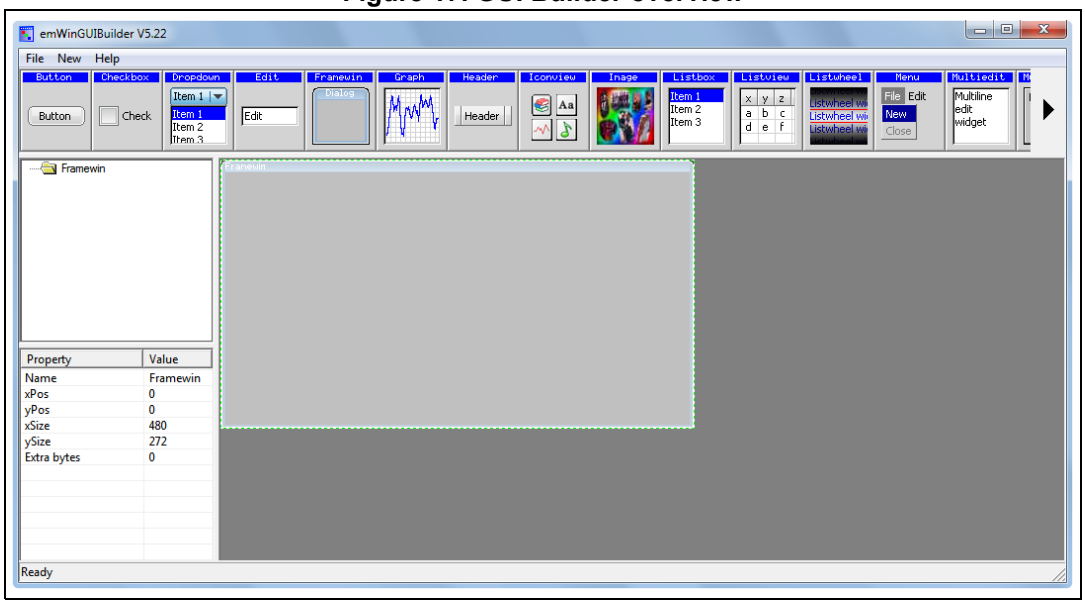
- Graphical aspect: the main window frame and module's controls
- Functionalities: module functions and internal processes

4.1 Creating the graphical aspect

The graphical aspect consists of the main frame window in addition to the set of the visual elements and controls (buttons, check boxes, progress bars…) used to control and monitor the module's functionalities.

The STM32CubeF4 demonstration package provides a PC tool; the *GUIBuilder* (*Figure 17*) that allows easily and quickly creating the module frame window and all its components in few steps. For more information about the GUI Builder, refer to the emwin User and reference guide (UM03001).

Figure 17. GUI Builder overview



The GUI Builder needs only a few minutes to totally design the module appearances using "drag and drop" commands and then generate the source code file to be included into the application.

The file generated is composed of the following main parts:

- A resource table: it's a table of type GUI_WIDGET_CREATE_INFO, which specifies all the widgets to be included in the dialog and also their respective positions and sizes.
- A dialog callback routine: described more in detail in section 4.3 (it is referred to as “main module callback routine”).

4.2 Graphics customization

After the basic module graphical appearance is created, it is then possible to customize some graphical elements, such as the buttons, by replacing the standard aspect by the user defined image. To do this, a new element drawing callback should be created and used instead of the original one.

Below an example of a custom callback for the Play button:

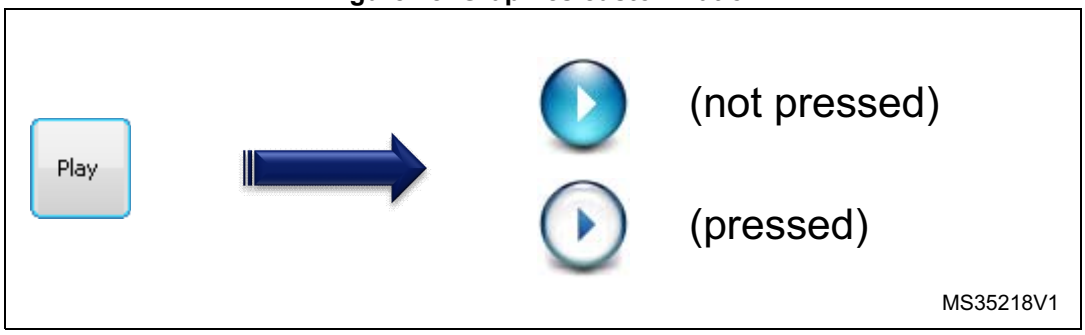
```
363 /**
364   * @brief  callback for play button
365   * @param  pMsg: pointer to data structure of type WM_MESSAGE
366   * @retval None
367   */
368 static void _cbButton_play(WM_MESSAGE * pMsg) {
369   switch (pMsg->MsgId) {
370     case WM_PAINT:
371       _OnPaint_play(pMsg->hWin);
372       break;
373     default:
374       /* The original callback */
375       BUTTON_Callback(pMsg);
376       break;
377   }
378 }
```

On the code portion above, the _OnPaint_play routine contains just the new button drawing command.

Note that the new callback should be associated to the graphical element at the moment of its creation, as shown below:

```
1034 hItem = BUTTON_CreateEx(148,140,50,50,pMsg->hWin,WM_CF_SHOW,0,ID_PLAY_BUTTON)
1035 WM_SetCallback(hItem,_cbButton_play);
```

Figure 18. Graphics customization



4.3 Module implementation

Once the graphical part of the module is finalized, the module functionalities and processes could be added then. It begins with the creation of the main module structure as defined in *Section 3.6: Modules manager*.

Then, each module has its own Startup function which simply consists of the graphical module creation, initialization and link to the main callback:

```
/**
  * @brief  Module window Startup
  * @param  hWin: pointer to the parent handle.
  * @param  xpos: X position
  * @param  ypos: Y position
  * @retval None
  */
static void Startup(WM_HWIN hWin, uint16_t xpos, uint16_t ypos)
{
  GUI_CreateDialogBox(_aDialogCreate, GUI_COUNTOF(_aDialogCreate), _cbDialog, hWin, xpos, ypos);
}
```

In the example above cbDialog refers to the main module callback routine. Its general skeleton is structured like the following:

```
/**
  * @brief  Callback routine of the dialog
  * @param  pMsg: pointer to data structure of type WM_MESSAGE
  * @retval None
  */
static void _cbDialog(WM_MESSAGE * pMsg){
  switch (pMsg->MsgId){
  case WM_INIT_DIALOG:
    /* Initialize graphical elements and restore backup parameters if any */
  case WM_NOTIFY_PARENT:
    Id    = WM_GetId(pMsg->hWinSrc);
    NCode = pMsg->Data.v;
    switch(Id) {
    case ID_BUTTON:
      switch (NCode) {
      case WM_NOTIFICATION_RELEASED:
        /* Operation associated to the button */
      }
      (...)
```

The list of windows messages presented in the code portion above (WM_INIT_DIALOG and WM_NOTIFY_PARENT) is not exhaustive, but represents the essential message IDs used:

- "WM_INIT_DIALOG: allows initializing the graphical elements with their respective initial values. It is also possible here to restore the backup parameters (if any) that will be used during the dialog procedure.
- "WM_NOTIFY_PARENT: describes the dialog procedure, for example: define the behavior of each button.

The full list of window messages can be found in the WM.h file.

4.4 Adding a module to the main desktop

Once the module appearance and functionality are defined and created, it still only to add the module to the main desktop view, this is done by adding it to the list (structure) of menu items: module_prop[ ], defined into k_module.h.

To do this, k_ModuleAdd() function should be called just after the module initialization into the main.c file.

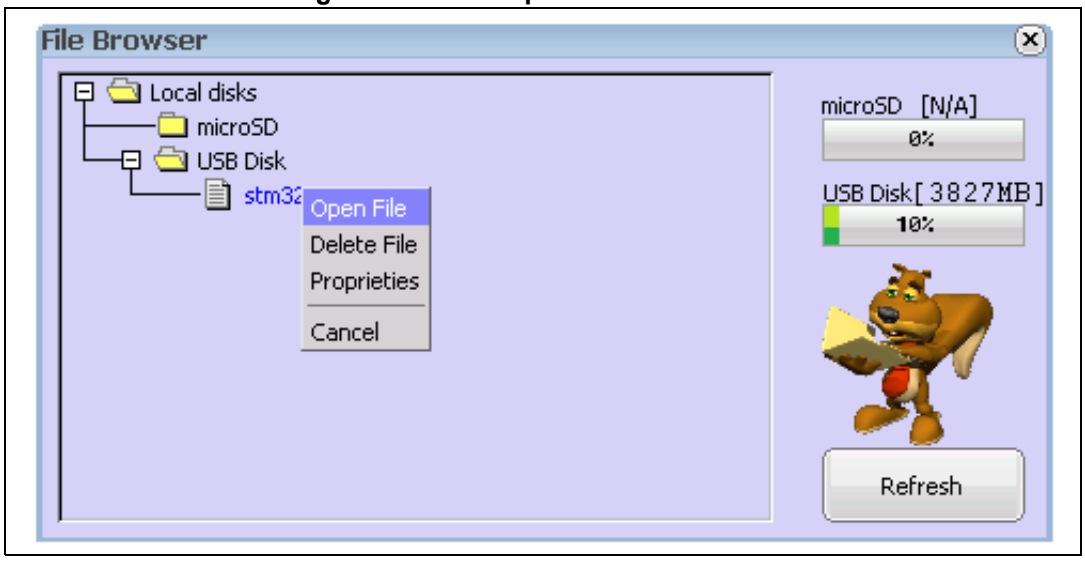
Note that the maximum modules number in the demonstration package is limited to 15; this value can be changed by updating MAX_MODULES_NUM defined into k_module.c.

4.5 Module's direct open

If there is a need to launch the module directly from the file browser contextual menu, an additional method should be added in the module structure for the direct open feature. This callback is often named _ModuleName_DirectOpen.

Figure 19 is an example of how to open a file using the adequate module from the file browser.

Figure 19. Direct open from file browser



In the STM32CubeF4 demonstration, there are three modules linked to the file browser contextual menu:

- The **video player**, supporting the format:
  - emf
- The **image browser**, supporting the formats:
  - jpg
  - bmp
- The **audio player**, supporting the format:
  - wav.

Then, to link the module to the file browser open menu, the command k_ModuleOpenLink() is called after the module is added.

5 Demonstration customization and configuration

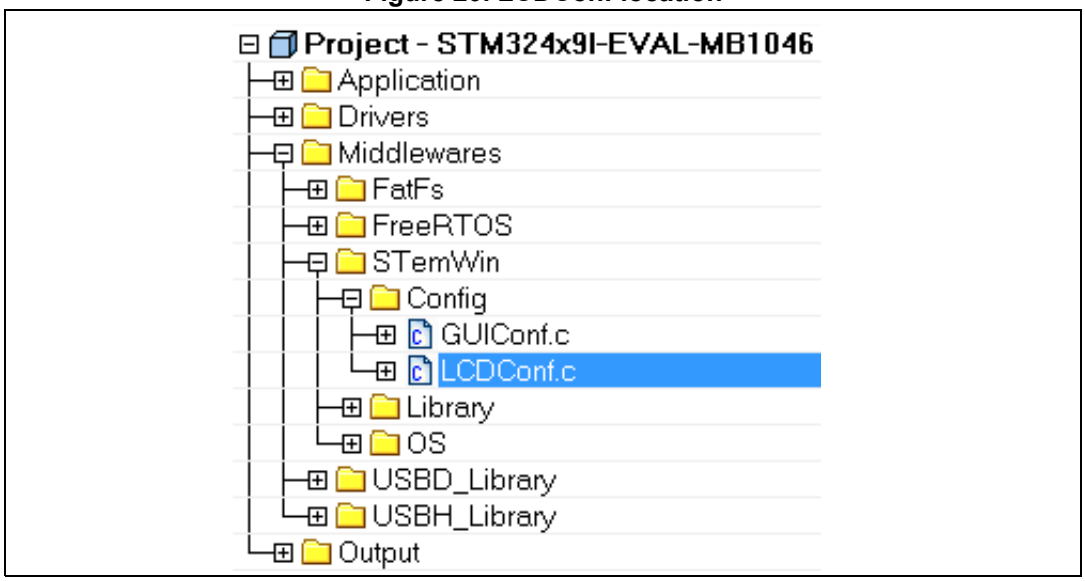
5.1 LCD configuration

The LCD is configured through the LCDConf.c file, see *Figure 20*. The main configuration items are listed below:

- Multiple layers:
  - The number of layers to be used defined using GUI_NUM_LAYERS.
- Multiple buffering:
  - If NUM_BUFFERS is set to a value "n" greater than 1, it means that "n" frame buffers will be used for drawing operation (see section 7.1 for impact of multiple buffering on performance).
- Virtual screens:
  - If the display area is greater than the physical size of the LCD, NUM_VSCREENS should be set to a value greater than 1. Note that virtual screens and multi buffers are not allowed together.
- Frame buffers locations:

The physical location of frame buffer is defined through LCD_LAYERX_FRAME_BUFFER.

Figure 20. LCDConf location



5.2 Layers management

In the STM32CubeF4 demonstration package with the STM324x9I-EVAL and Discovery Kit, GUI_NUM_LAYERS is set to 2 (both layers are used):

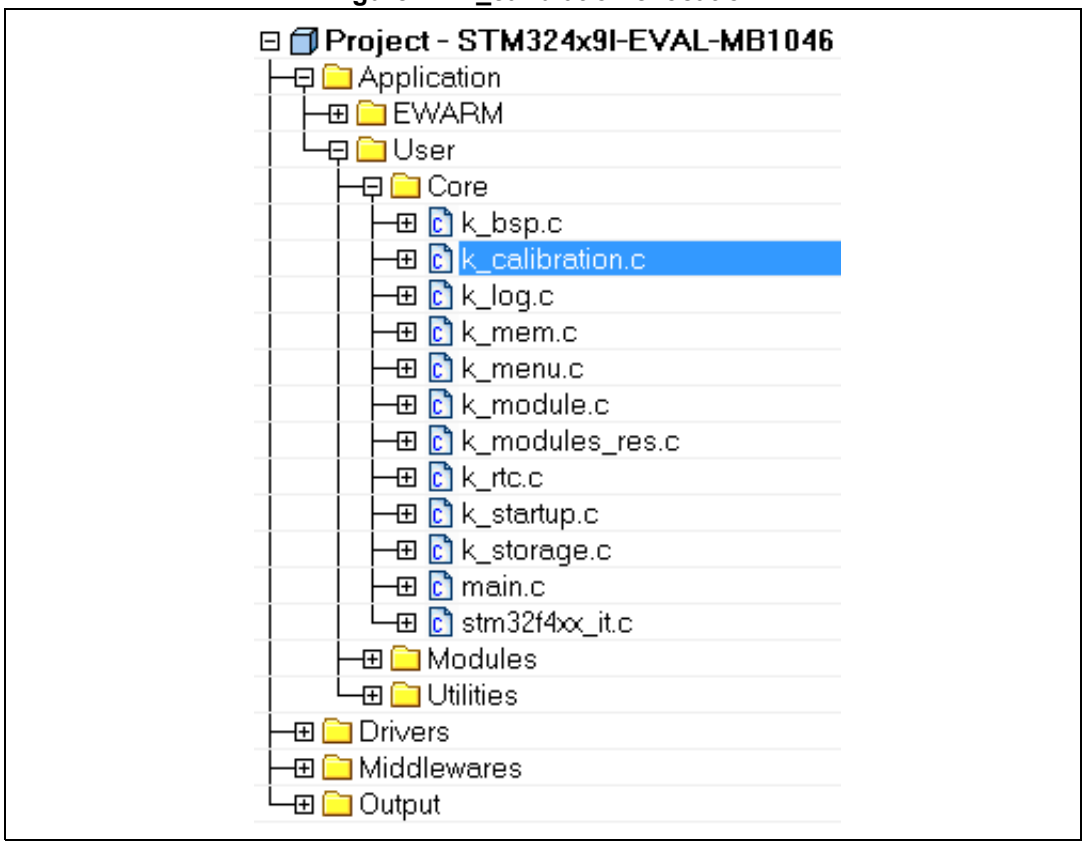
- "Layer 0 is dedicated to background display
- "Layer 1 is used for the main desktop display

Dedicated layers usage will lighten the CPU load during the refresh tasks.

5.3 Touchscreen calibration

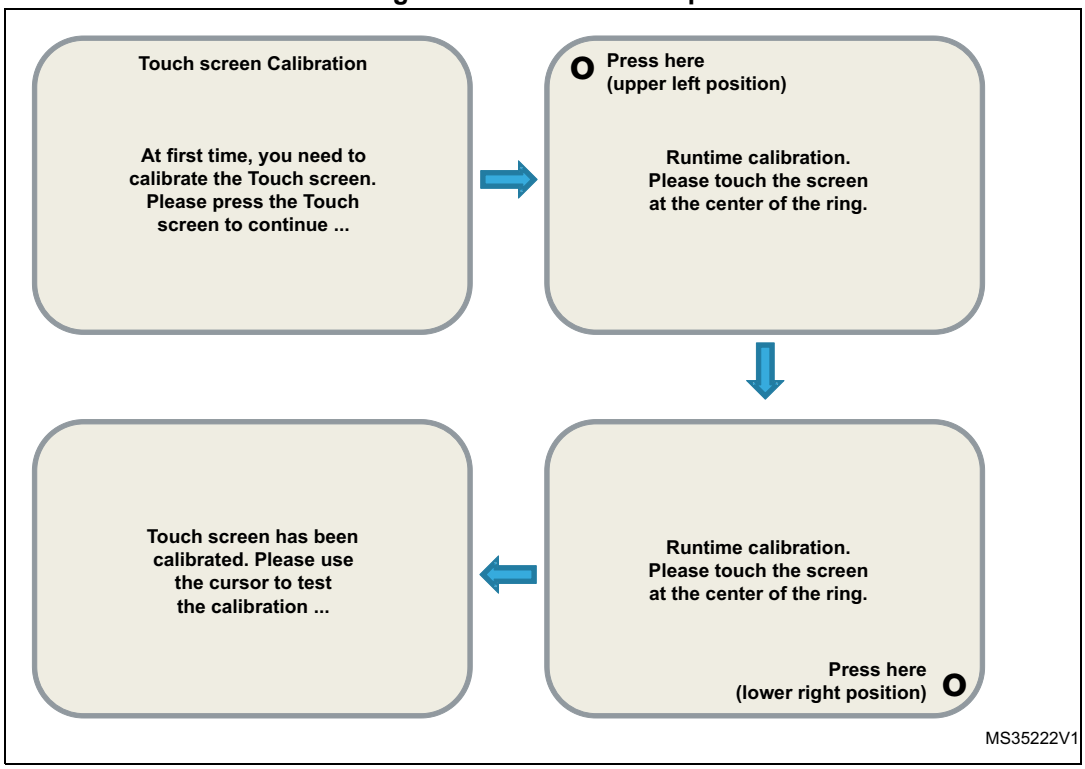
When the demonstration is launched for the first time, the touchscreen needs to be calibrated. A full set of dedicated routines is included in the demonstration package and regrouped into k_calibration.c file (*Figure 21*).

Figure 21. k_calibration.c location



To do this, after the startup screen is displayed, the user has to follow the displayed calibration instructions by touching the screen at the indicated positions (*Figure 22*). This will allow getting the physical Touch screen values that will be used to calibrate the screen.

Figure 22. Calibration steps



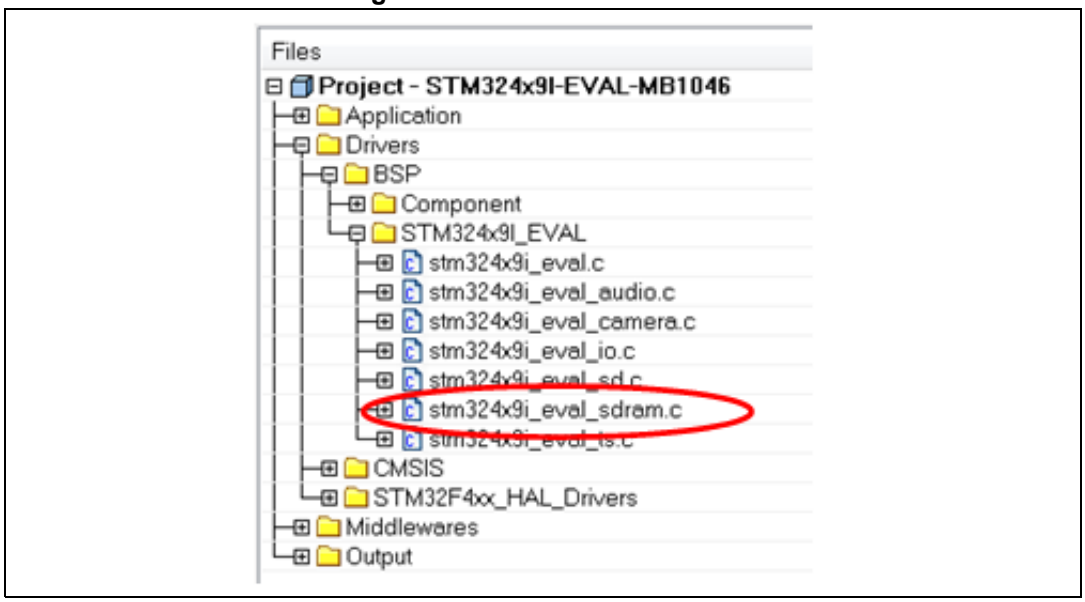
Once this runtime calibration is done, the touch screen calibration parameters are saved to the RTC Backup data registers: RTC_BKP_DR0 and RTC_BKP_DR1, so the next time the application is restarted, these parameters are automatically restored and there is no need to re-calibrate the touchscreen.

5.4 BSP customization

5.4.1 SDRAM configuration

The SDRAM capacity is 1 Mbyte x 32 bits x 4 banks. The BSP SDRAM driver offers a set of functions to initialize, read/write in polling or DMA mode.

Figure 23. SDRAM initialization



The SDRAM external memory must be initialized before the GUI initialization to allow his use as LCD layers frame buffer.

Table 8. LCD frame buffer locations

Layer	Address
LCD Layer0	0xC0200000
LCD Layer1	0xC0400000

The SDRAM is used also as DCMI output for camera module. The camera output is stored in camera frame buffer address as 16bpp (RGB565) and converted to 24bpp in the Camera converted frame before its stocking in the selected storage unit.

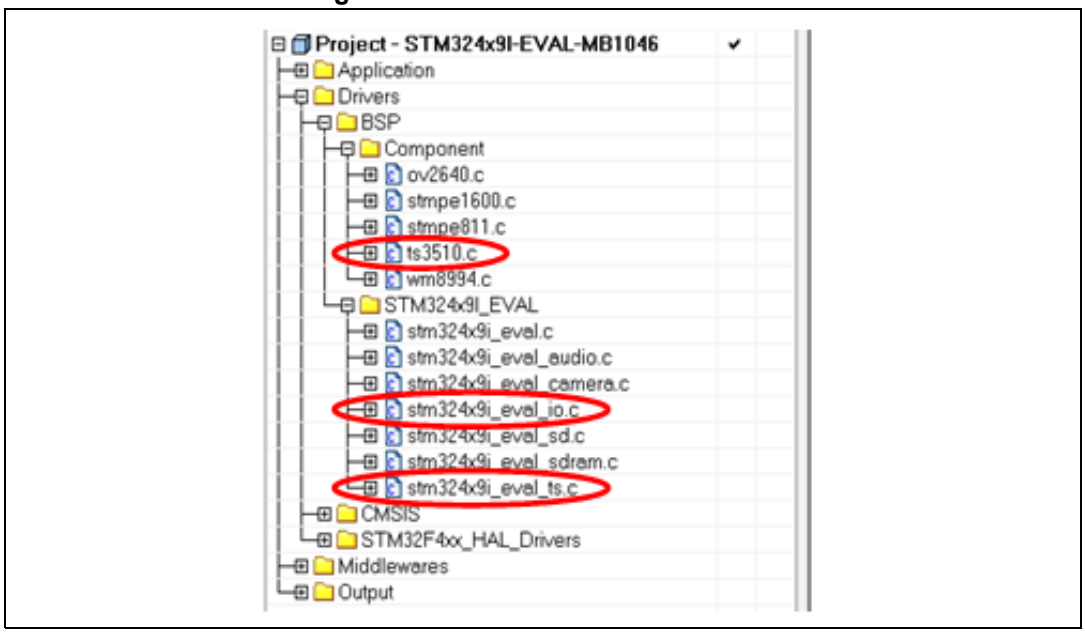
Table 9. Camera frame buffer locations

Camera	Address
Camera frame buffer	0xC0000000
Camera converted frame	0xC0025800

5.4.2 Touch screen configuration

The touch screen is controlled by the BSP TS driver which uses the BSP IO driver in case of STM32429 Eval board and TS3510 component in case of STM32439 Eval board.

Figure 24. Touch screen initialization



The touch screen is initialized in 'k_BspInit' following the used screen resolution as shown in the code below.

```
/**
  * @brief  Initializes LEDs, SDRAM, touch screen, CRC and SRAM.
  * @param  None
  * @retval None
  */
void k_BspInit(void)
{
  (...)
    /* Initialize the Touch screen */
  BSP_TS_Init(480, 272);
  (...)

}

/**
  * @brief  Read the coordinate of the point touched and assign their
  *         value to the variables u32_TSXCoordinate and u32_TSYCoordinate
  * @param  None
  * @retval None
  */
void k_TouchUpdate(void)
{
  GUI_PID_STATE TS_State;

  BSP_TS_GetState((TS_StateTypeDef *)&ts);
  (...)
  GUI_TOUCH_StoreStateEx(&TS_State);
  (...)
}
```

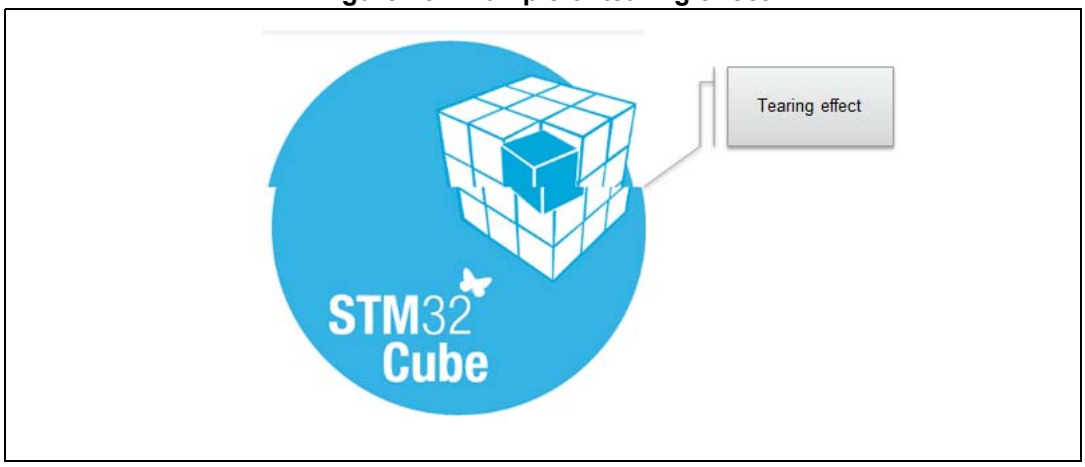
6 Performance

6.1 Multi buffering features

Multiple buffering is the use of more than one frame buffer, so that the display ever shows a screen which is already completely rendered, even if a drawing operation is in process. When starting the process of drawing the current content of the front buffer is copied into a back buffer. After that all drawing operations take effect only on this back buffer. After the drawing operation has been completed the back buffer becomes the front buffer. Making the back buffer the visible front buffer normally only requires the modification of the frame buffer start address register of the display controller.

Now it should be considered that a display is refreshed by the display controller approximately 60 times per second. After each period there is a vertical synchronization signal, known as VSYNC signal. The best moment to make the back buffer the new front buffer is this signal. If not considering the VSYNC signal tearing effects can occur, as shown in *Figure 25*.

Figure 25. Example of tearing effect

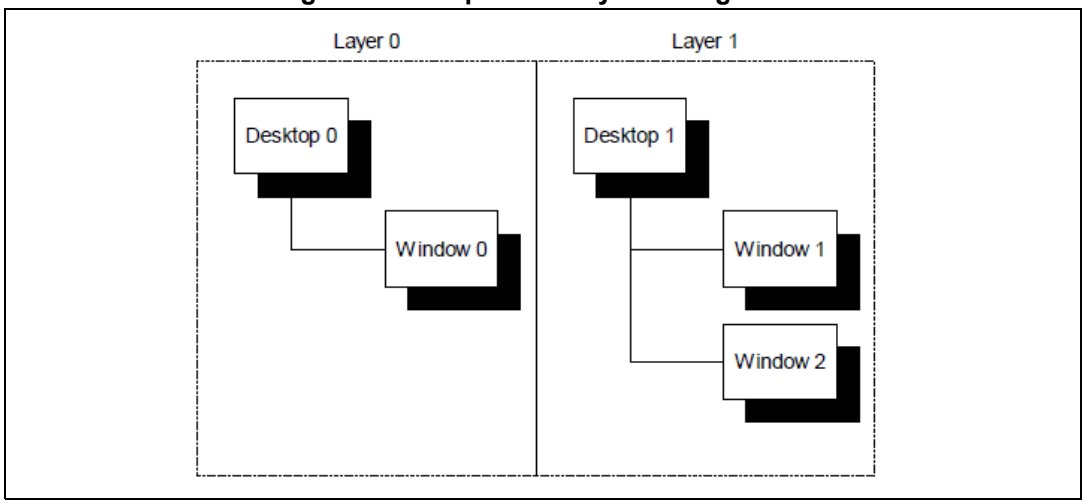


6.2 Multi layers feature

Windows can be placed in any layer or display, drawing operations can be used on any layer or display. Since there are really only smaller differences from this point of view, multiple layers and multiple displays are handled the same way (Using the same API routines) and are simply referred to as multiple layers, even if the particular embedded system uses multiple displays.

In the STM32CubeF4 demonstration, the layer 0 is dedicated for the background while the layer 1 with transparency activated is dedicated for the main desktop, this will allow to the kernel to keep the background unchanged during the desktop visual changes without refreshing the background image.

Figure 26. Independent layer management



6.3 Hardware acceleration

With the STM324x9I-EVAL and Discovery Kit demonstration, the hardware acceleration capabilities of the STM32F429/ STM32F439 cores are used. STemWin offers a set of customization callbacks to changes the default behavior based on the hardware capabilities, the optimized processes are implemented in the LCDConf.c file and implement the following features:

a) Color conversion

Internally STemWin works with logical colors (ABGR). To be able to translate these values into index values for the hardware and vice versa the color conversion routines automatically use the DMA2D for that operation if the layer work with direct color mode

This low level implementation makes sure that in each case where multiple colors or index values need to be converted the DMA2D is used.

b) Drawing of index based bitmaps

when drawing index based bitmaps STemWin first loads the palette of the bitmap into the DMA2Ds LUT instead of directly translating the palette into index values for the hardware. The drawing operation then is done by only one function call of the DMA2D.

c) Drawing of high color bitmaps

If the layer works in the same mode as the high color bitmap has its pixel data available, these bitmaps can be drawn by one function call of the DMA2D. The following function is used to set up such a function;:

LCD_SetDevFunc(LayerIndex, LCD_DEVFUNC_DRAWBMP_16BPP, pFunc);

d) Filling operations

Setting up the function for filling operations:

LCD_SetDevFunc(LayerIndex, LCD_DEVFUNC_FILLRECT, pFunc);

e) Copy operations

Setting up the functions for copy operations used by the function GUI_CopyRect():

LCD_SetDevFunc(LayerIndex, LCD_DEVFUNC_COPYRECT, pFunc);

f) Copy buffers

Setting up the function for transferring the front- to the back buffer when using multiple buffers:

LCD_SetDevFunc(LayerIndex, LCD_DEVFUNC_COPYBUFFER, pFunc);

g) Fading operations

Setting up the function for mixing up a background and a foreground buffer used for fading memory devices:

GUI_SetFuncMixColorsBulk(pFunc);

h) General alpha blending

The following function replaces the function which is used internally for alpha blending operations during image drawing (PNG or true color bitmaps) or semitransparent memory devices:

GUI_SetFuncAlphaBlending(pFunc);

i) Drawing antialiased fonts

Setting up the function for mixing single foreground and background colors used when drawing transparent ant aliased text:

GUI_SetFuncMixColors(pFunc).

7 Footprint

The purpose of the following sections is to provide the memory requirements for all the demonstration modules, including jpeg decoder and STemWin's main GUI components. The aim is to have an estimation of memory requirement in case of suppression or addition of a module or feature.

The footprint data are provided for the following environment:

- Tool chain: IAR 6.70.1
- Optimization: high size
- Board: STM32F429-EVAL.

7.1 Kernel footprint

Table 10 shows the code memory, data memory and the constant memory used for each kernel file.

Table 10. Kernel files footprint

File	code [byte]	data [byte]	const [byte]
k_bsp	260	8	0
K_calibration	972	28	48
k_Log	100	8(1)	0
k_mem	266	0	0
k_menu	3496	900	412089
k_module	214	244	0
k_module_res	98	0	207692
k_rtc	196	32	195529
k_startup	316	4	300064
k_storage	954	2844	24
main	614	4	44

1. The memory is allocated dynamically in some structures of this file.

7.2 Module footprint

Table 11 shows the code memory, data memory and the constant memory used for each kernel file.

Table 11. Modules footprint

File	code [byte]	data [byte]	const [byte]
Audio	6764	501(1)	33067
Benchmark	1320	36	32693

Table 11. Modules footprint (continued)

File	code [byte]	data [byte]	const [byte]
Camera	2467	213(1)	62629
File Browser	3062	516	69083
Game	4188	1916	33432
Image Browser	5308	1956(1)	32862
System	2486	89	33506
USB Device	540	29	195529
Video Player	6476	989(1)	32646

1. The memory is allocated dynamically in some structures of this file.

7.3 STemWin Features resources

7.3.1 JPEG decoder

The JPEG decompression uses approximately 33 Kbytes of RAM for decompression independently of the image size and a size dependent amount of bytes. The RAM requirement can be calculated as follows:

Approximate RAM requirement = X-Size of image * 80 bytes + 33 Kbytes

Table 12. RAM requirements for some JPEG resolutions

Resolutiont	RAM usage [kbyte]	RAM usage, size dependent [kbyte]
160x120	45.5	12.5
320x340	58.0	25.0
480x272	70.5	37.5
640x480	83.0	50.0

The memory required for the decompression is allocated dynamically by the STemWin memory management system. After drawing the JPEG image the complete RAM will be released.

7.3.2 GUI Components

The operation area of STemWin varies widely, depending primarily on the application and features used. In the following sections, memory requirements of various modules are listed, as well as the memory requirements of example applications.

Table 13 shows the memory requirements of the main components of STemWin. These values depend a lot on the compiler options, the compiler version and the used CPU. Note that the listed values are the requirements of the basic functions of each module.

Table 13. MemoSTemWin components memory requirements

Component	ROM	RAM	Description
Windows Manager	6.2 Kbytes	2.5 Kbytes	Additional memory requirements of basic application when using the Windows Manager
Memory Devices	4.7 Kbytes	7 Kbytes	Additional memory requirements of basic application when using memory devices
Antialiasing	4.5 Kbytes	2 * LCD_XSIZE	Additional memory requirements for the antialiasing software item
Driver	2-8 Kbytes	20 bytes	The memory requirements of the driver depend on the configured driver and whether a data cache is used or not. With a data cache, the driver requires more RAM
Multilayer	2-8 Kbytes	-	If working with a multi layer or a multi display configuration, additional memory is required for each additional layer, because each requires its own driver
Core	5.2 Kbytes	80 bytes	Memory requirements of a typical application without using additional software items
JPEG	12 Kbytes	36 Kbytes	Basic routines for drawing JPEG files
GIF	3.3 Kbytes	17 Kbytes	Basic routines for drawing GIF files
Sprites	4.7 Kbytes	16 bytes	Routines for drawing sprites and cursors
Font	1-4 Kbytes	-	Depends on the font size to be used

Table 14. Widget memory requirements

Component	ROM	RAM	Description
BUTTON	1.0 Kbytes	40 bytes	(1)
CHECKBOX	1.0 Kbytes	52 bytes	(1)
DROPDOWN	1.8 Kbytes	52 bytes	(1)
EDIT	2.2 Kbytes	28 bytes	(1)
FRAMEWIN	2.2 Kbytes	12 bytes	(1)
GRAPH	2.9 Kbytes	48 bytes	(1)
GRAPH_DATA_XY	0.7 Kbytes	-	(1)
GRAPH_DATA_XY	0.6 Kbytes	-	(1)
HEADER	2.8 Kbytes	32 bytes	(1)
LISTBOX	3.7 Kbytes	56 bytes	(1)
LISTVIEW	3.6 Kbytes	44 bytes	(1)
MENU	5.7 Kbytes	52 bytes	(1)
MULTIEDIT	7.1 Kbytes	16 bytes	(1)

Table 14. Widget memory requirements (continued)

Component	ROM	RAM	Description
MULTIPAGE	3.9 Kbytes	32 bytes	(1)
PROGBAR	1.3 Kbytes	20 bytes	(1)
RADIOBUTTON	1.4 Kbytes	32 bytes	(1)
SCROLLBAR	2.0 Kbytes	14 bytes	(1)
SLIDER	1.3 Kbytes	16 bytes	(1)
TEXT	1.0 Kbytes	16 bytes	(1)
CALENDAR	0.6 Kbytes	32 bytes	(1)

1. The listed memory requirements of the widgets contain the basic routines required for creating and drawing the widget. Depending on the specific widget there are several additional functions available which are not listed in the table

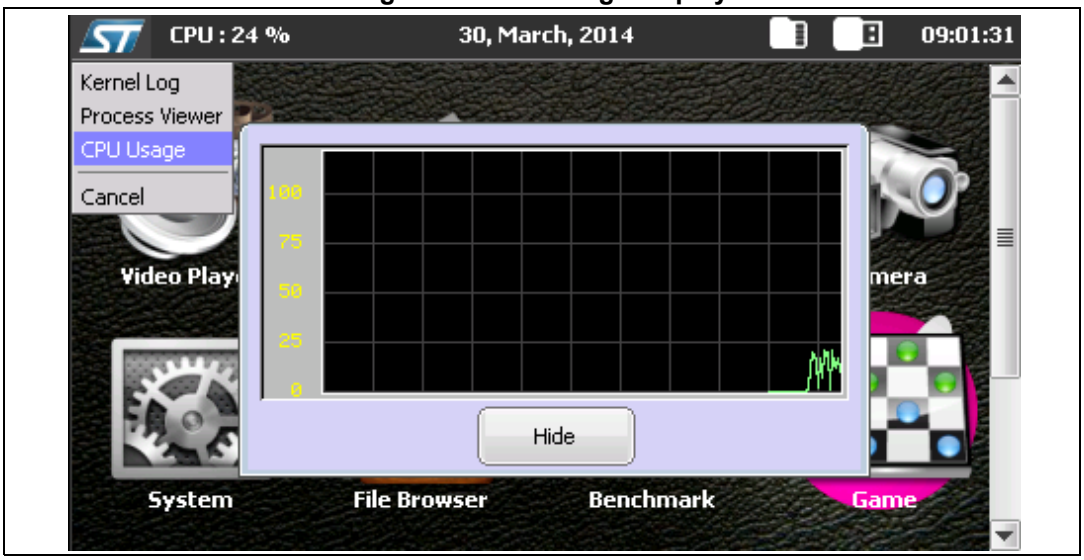
8 Demonstration functional description

8.1 Kernel

The main desktop is built around two main graphical components:

- The status bar: indicates the storage units' connection status, current time and date and a system utilities button to allow getting system information like (running task, CPU usage, and kernel log).
- The icon view widget: contains the icons associated to added modules. User can launch a module by a simple click on the module icon (see *Figure 27*).

Figure 27. CPU usage display



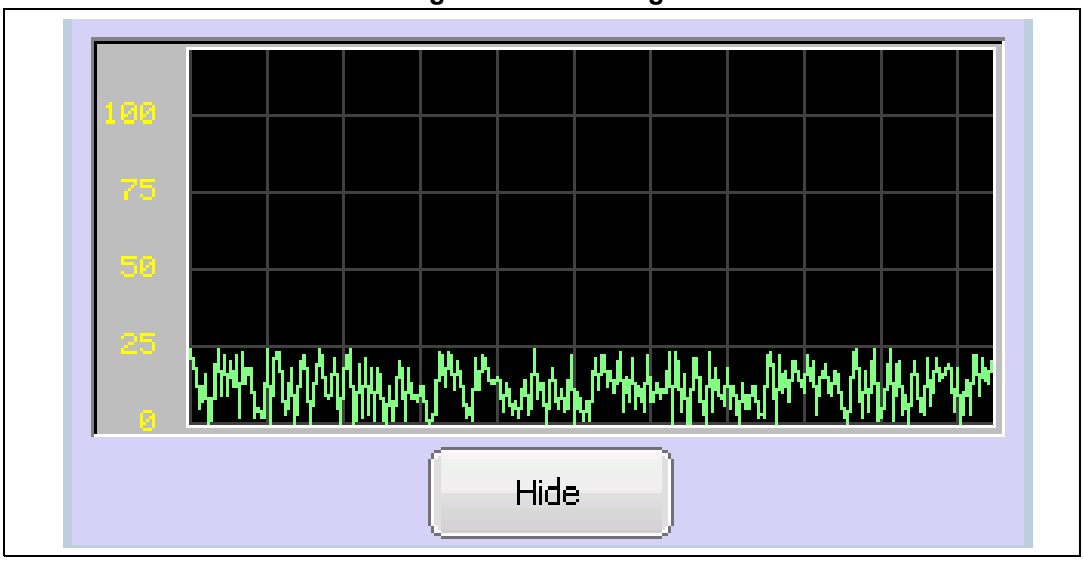
The system utilities are accessible during the STM32CubeF4 demonstration running time, using the system button (ST Logo) in top left of the main desktop. The system utilities button offers the following services:

- CPU Usage history
- Kernel log messages
- Current running processes viewer.

8.1.1 CPU Usage

The CPU Usage utility provides a graphical representation of the CPU usage evolution (*Figure 28*) during the demonstration run time starting for the first time it was launched. Note that once launched the CPU usage utilities keep running in background and can be restored in any time.

Figure 28. CPU usage



8.1.2 Kernel Log

The kernel log utility gather all the kernel and module messages and save them into a dedicated internal buffer. The Log messages can be visualized at any time during the demonstration run time, as shown in *Figure 29*.

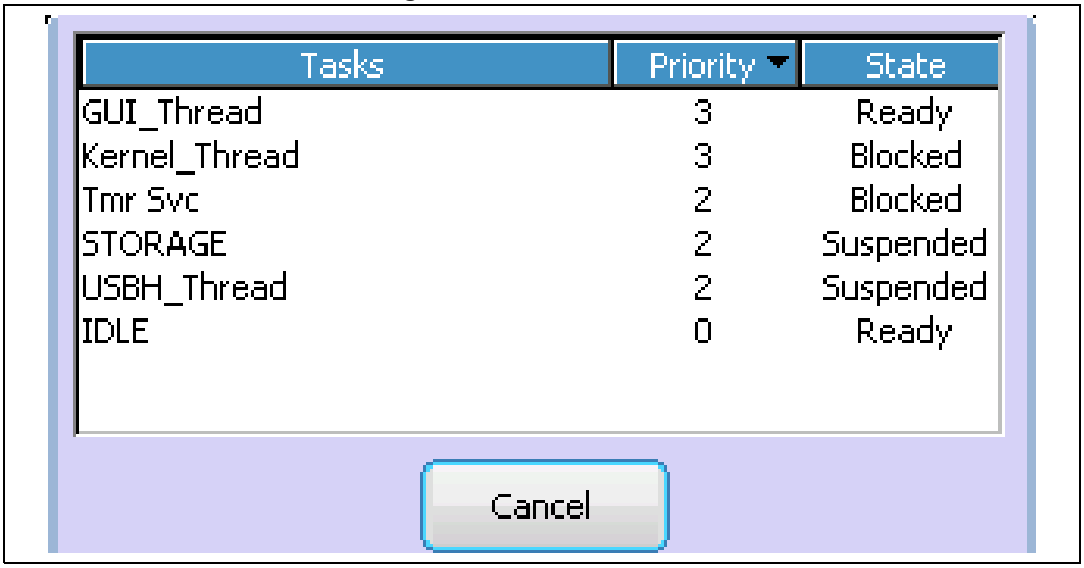
Figure 29. Example of Log messages

8.1.3 Process Viewer

The process viewer (*Figure 30*) allows to check and to display the status of the currently running tasks (FreeRTOS) at any time during the demonstration run time. It shows the following information:

1. Current running tasks names.
2. Current running tasks priorities
3. Running tasks states (FreeRTOS statics information).

Figure 30. Process viewer



8.2 Modules

8.2.1 System

Overview

The system module provides three control tabs: system information, general settings and clock settings to set the global demonstration settings. The system module retrieves demonstration information from internal kernel settings data structures and acts on the several kernel services to changes settings.

Functional description

The system module provides three graphical views:

a) Demonstration global Information (*Figure 31*)

This first page shows the main demonstration information such as: Used board, STM32 core part number, and current CPU clock and demonstration revision.

b) General settings (*Figure 32*)

The general settings tab permits to change the global demonstration configuration. Note that the new settings are not applied immediately; new settings take effect after restarting the demonstration.

Figure 31. Demonstration global information

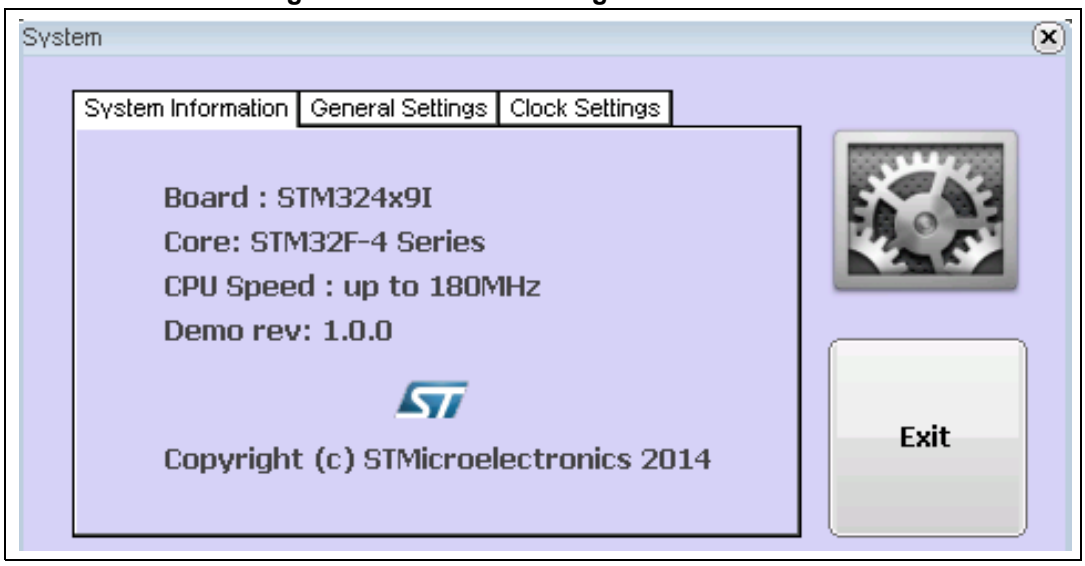


Figure 32. Demonstration general settings

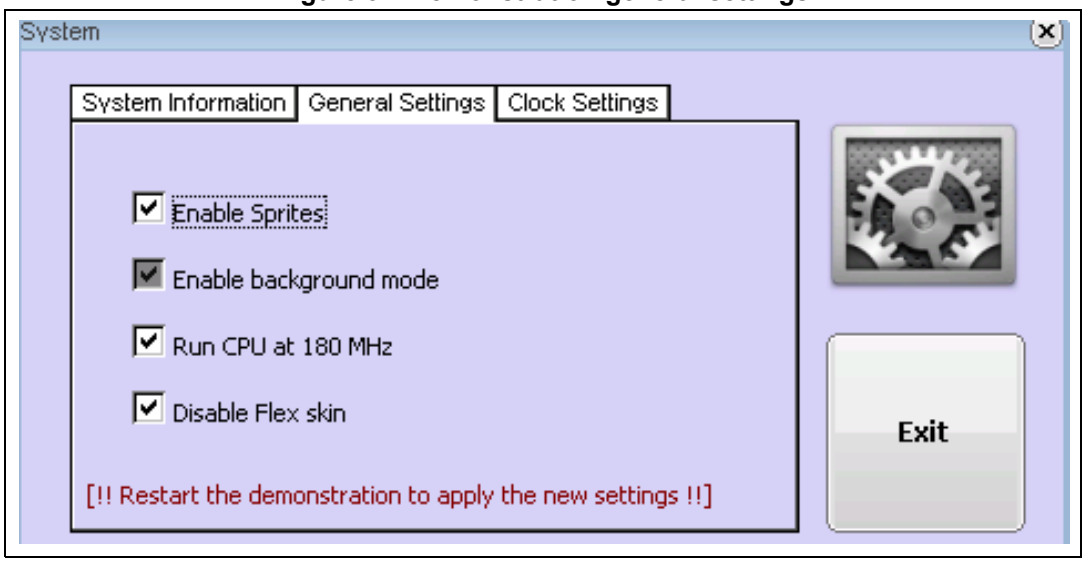
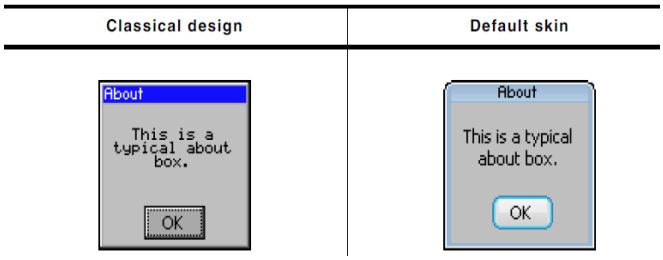


Table 15 shows the different settings that can be changed.

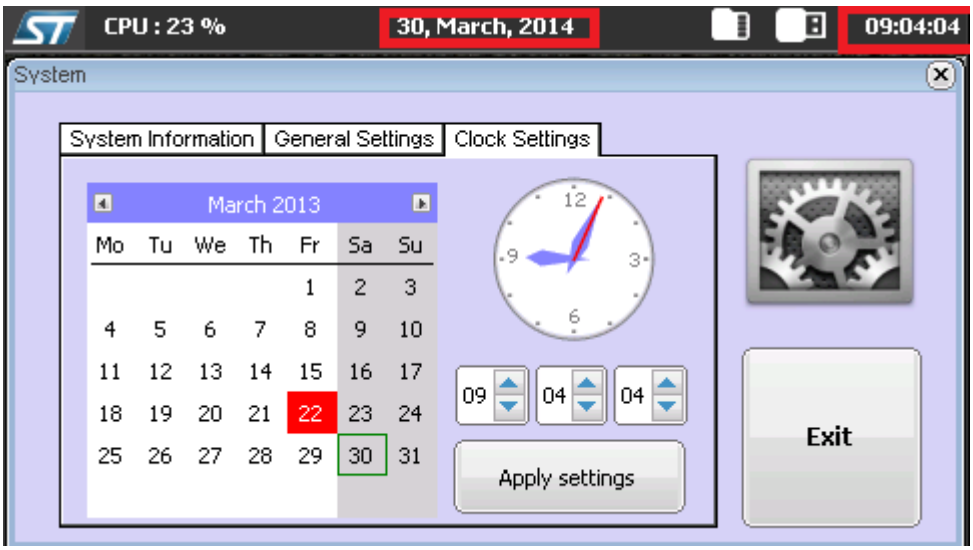
Table 15. Available settings

Configuration item	Description
Enable sprites	Checking this box allows the sprites to move on the background desktop 
Enable background mode	Not used (reserved for future use)
Run CPU at 180 MHz	Allow to run the demonstration at maximum speed. Note that the device USB clock is not at compliant clock with mode. To use the USB device mass storage module, it is recommended to use the default 168 MHz CPU clock
Disable Flex skin	Unchecking this box, classical GUI skin is used. 

c) Clock settings

The clock setting tab (*Figure 33*) allows to adjust the demonstration time and date by changing the RTC configuration of the kernel.

Figure 33. Clock setting

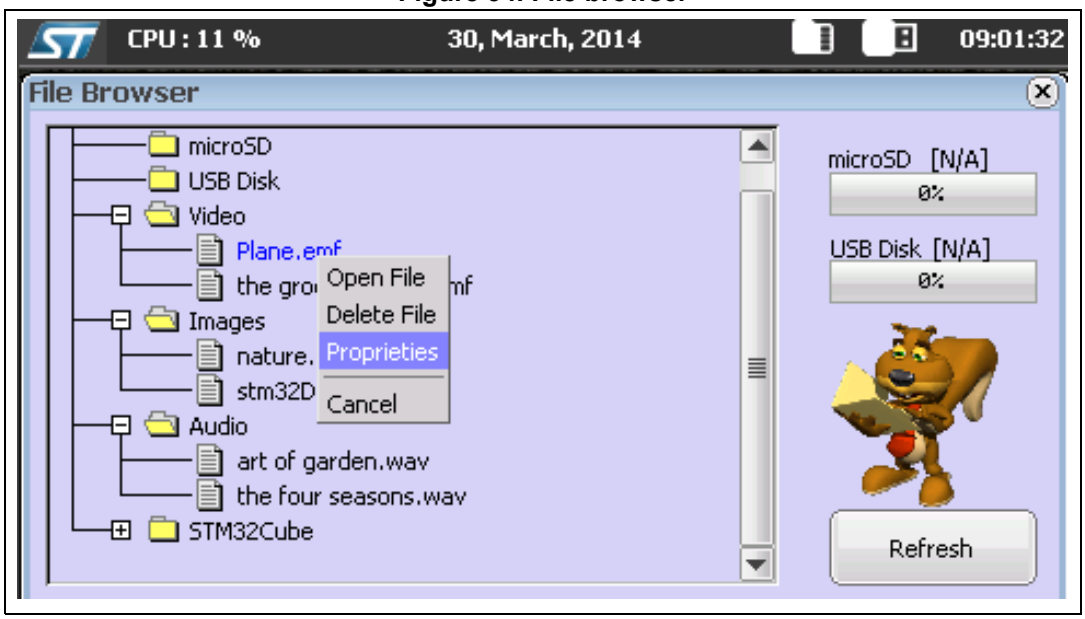


8.2.2 File Browser

Overview

The File browser module is a system module that allows to explore the connected storage unit(s), to delete or to open a selected file. The file list structure is built during the media connection and updated after a connection status change of one of the used media.

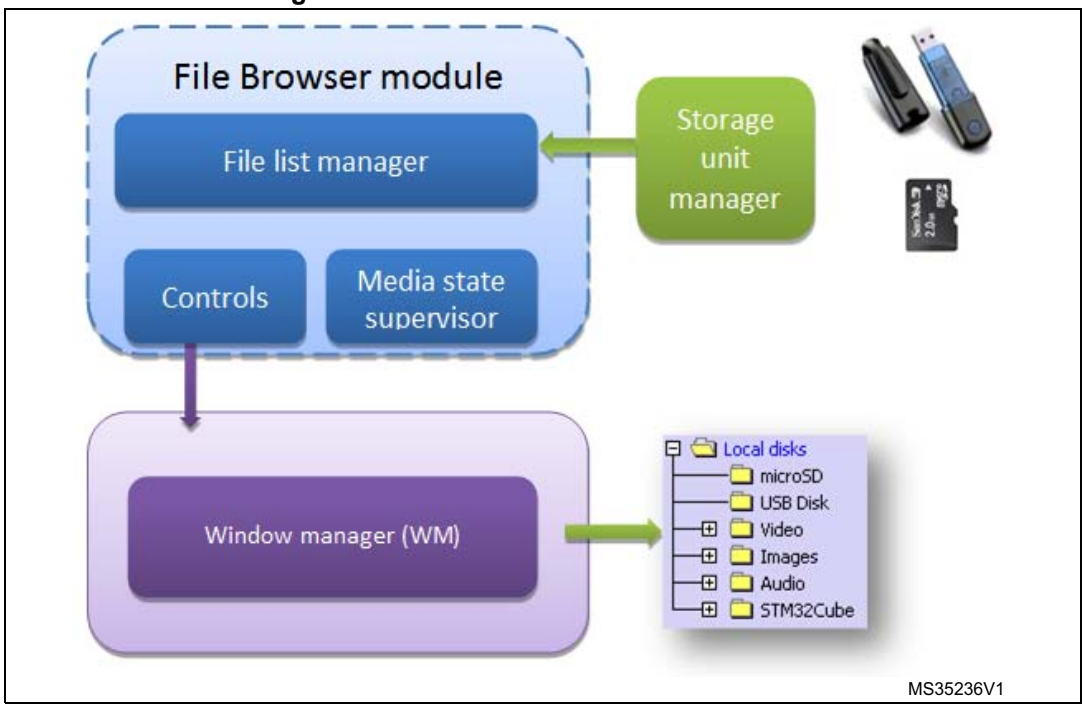
Figure 34. File browser



Functional description

The file browser is mainly used for standard file operations: explore folder, file information, file deletion and opening supported extension file when a file type is linked to the direct open file feature of the kernel (*Figure 35*). Note that Read-Only file cannot be deleted physically from media.

Figure 35. File browser module architecture



To open the contextual file menu, user has to select a file (selecting a folder has no effect).

The following actions are accessible through the contextual menu:

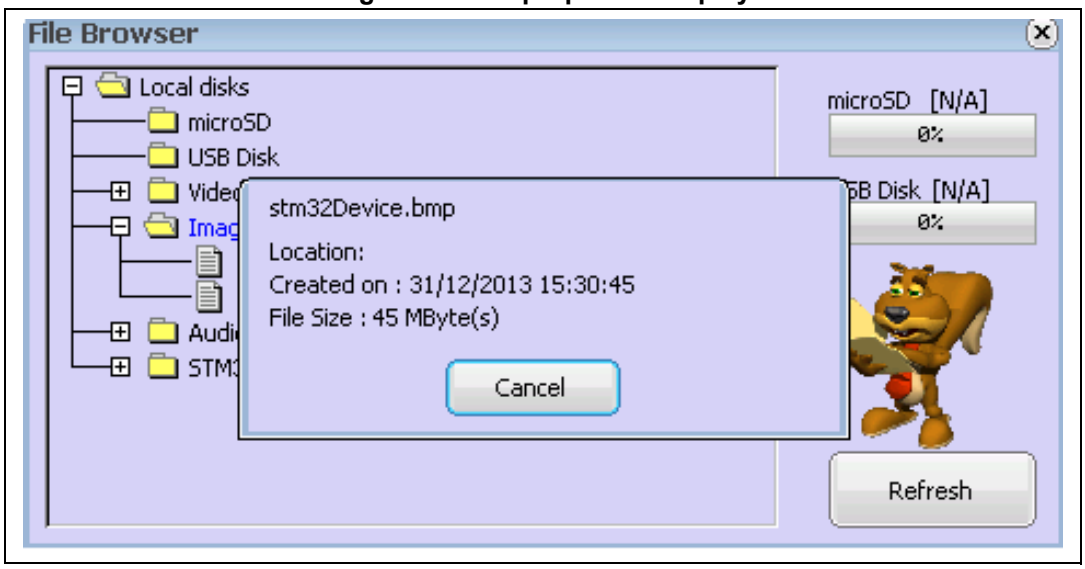
a) Open file: if a file extension is linked to the direct open file feature of the kernel, the associated application with this extension is launched and the file is opened automatically (*Figure 36*).

Figure 36. File opening from browser

b) Delete file: selecting a file for deletion will display a confirmation message box to confirm the deletion operation. Note that Read-Only file cannot be deleted physically from media.

c) Proprieties: the File browser can be used to check file proprieties such as current location, size, and creation date.

Figure 37. File properties display



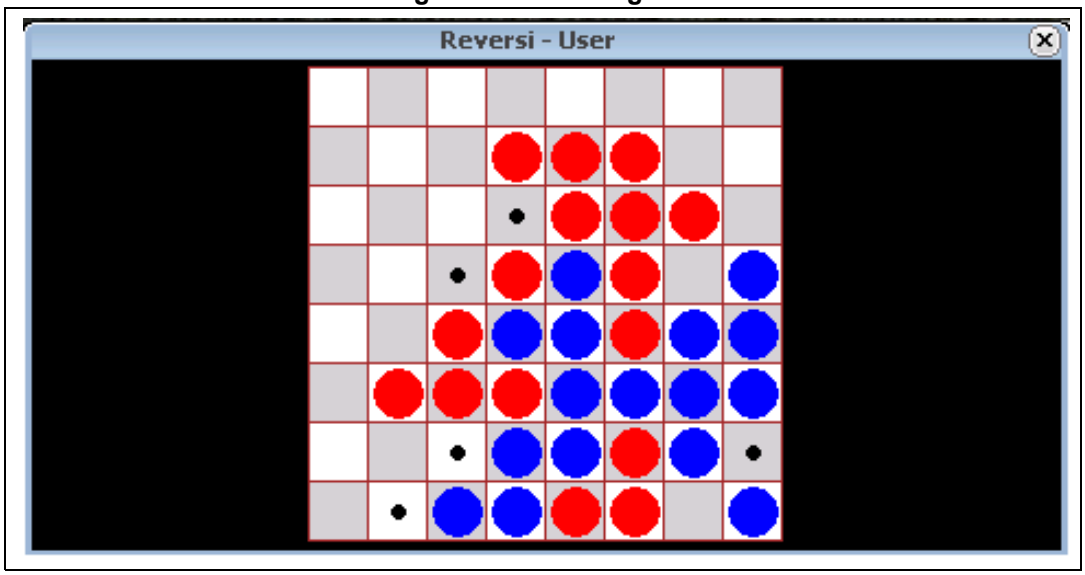
Note: The File browser can explore up to four levels, the maximum explorer level is defined in the kernel files (k_storage.h).

8.2.3 Game

The game coming in the STM32CubeF4 demonstration is based on the Reversi game. It is a strategy board game for two players, played on an 8×8 board. The goal of the game is to have the majority of disks turned to display your color when the last playable empty square is filled.

In this STM32CubeF4 demonstration STM32 MCU is one of the two players. The GUI will ask the user to start a new game when the ongoing one is over.

Figure 38. Reversi game



8.2.4 Benchmark

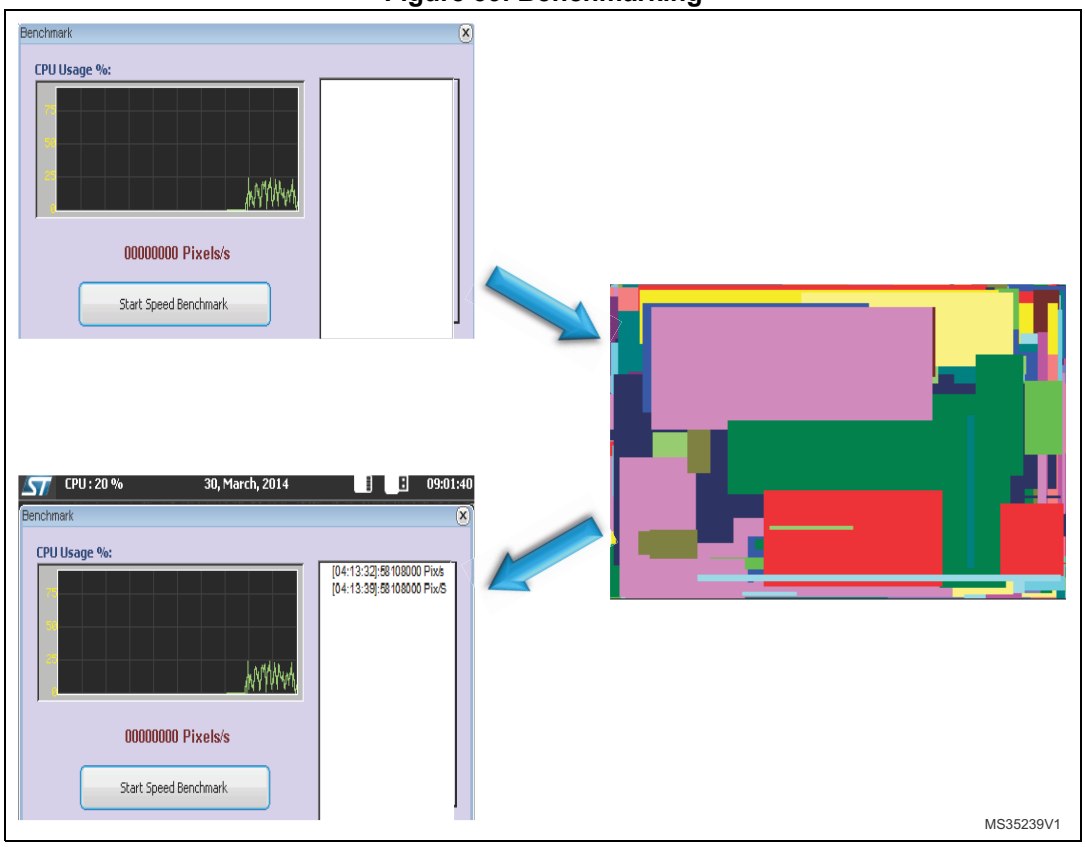
Overview

The Benchmark module is a system module that allows measure the graphical performance by measuring the time needed to draw several colored rectangles in random position with random size during a specific period. The result is given in pixel per second.

Functional description

The benchmark starts immediately once the start speed benchmark button is pressed. After few seconds the result is displayed in red below the CPU Usage graphical window and result is logged in the right list box with date and time stamp (*Figure 39*).

Figure 39. Benchmarking



8.2.5 Audio

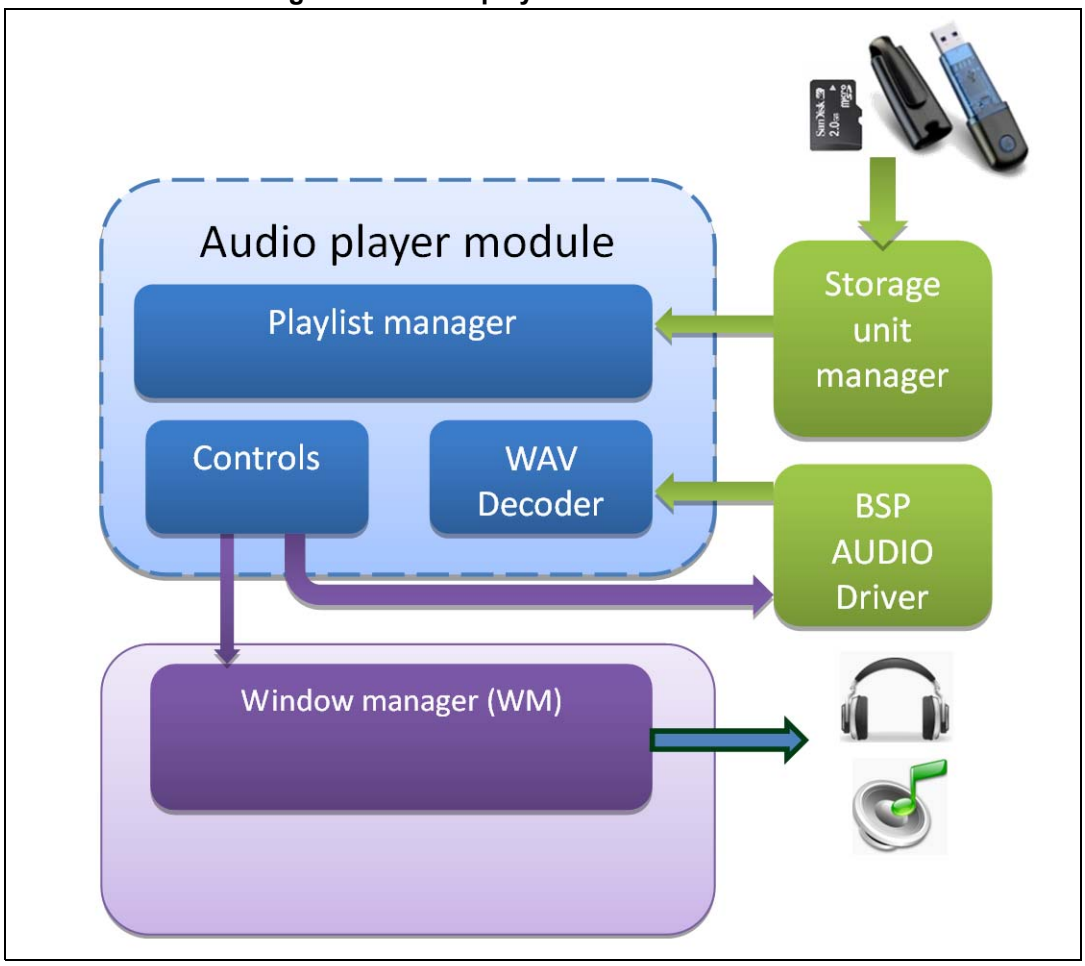
Overview

The audio player module provides a complete audio solution based on the STM32F4xx and delivers a high-quality music experience. It supports playing music in WAV format but may be extended to support other compressed formats such as MP3 and WMA audio formats.

Architecture

Figure 40 shows the different audio player parts and their connections and interactions with the external components.

Figure 40. Audio player module architecture



Data structure used

Table 16 contains the different data structure used in audio player module and a brief description of each of them.

Table 16. Data structure for audio

Structure	Description
WAV_InfoTypedef	Contains the wave file information extracted from wave file header
AUDIOPLAYER_ProcessTypdef	Contains the audio player state, the speaker state, the volume value and the pointer to the audio buffer.
AUDIOPLAYER_StateTypdef	Contains the different audio player state: – AUDIOPLAYER_STOP – AUDIOPLAYER_START – AUDIOPLAYER_PLAY – AUDIOPLAYER_PAUSE – AUDIOPLAYER_EOF – AUDIOPLAYER_ERROR

Table 16. Data structure for audio (continued)

Structure	Description
AUDIOPLAYER_ErrorTypdef	Contains the different possible error. – AUDIOPLAYER_ERROR_NONE – AUDIOPLAYER_ERROR_IO – AUDIOPLAYER_ERROR_HW – AUDIOPLAYER_ERROR_MEM – AUDIOPLAYER_ERROR_FORMAT_NOTSUPPORTED
BUFFER_StateTypeDef	Contains the different Buffer state – BUFFER_OFFSET_NONE – BUFFER_OFFSET_HALF – BUFFER_OFFSET_FULL.

Functional description

The audio player initialization is done in startup step. In this step all the audio player states, the speaker and the volume value are initialized and only when the play button in the audio player interface is pressed to start the process.

There are two ways to start audio player module:

- From main desktop menu as shown in *Figure 41*
- Through the file browser contextual menu: direct open feature.

Figure 41. Audio player module startup

When the audio player is started, the following actions are executed:

- The graphical components are initialized:
  - The audio frame
  - The control buttons
  - The list box field

An additional memory is allocated to keep the audio list (pWavList) and the audio file information (pFileInfo).

Table 17. Audio module controls

Button	Preview	Brief description
Play button		Changes the audio player state to “AUDIOPLAYER_PLAY” Reads the wave file from storage unit Sets the frequency Starts or resumes the audio task Starts playing audio stream from a data buffer using “BSP_AUDIO_OUT_Play” function in BSP audio driver. Replaces play button by pause button
Pause button		Suspends the audio task Pauses the audio file stream Replaces pause button by play button
Stop button		Close the wave file from storage unit Suspends the audio task Stops audio playing Changes the audio player state to “AUDIOPLAYER_STOP”
Previous button		Point to the previous wave file Stops audio playing Starts playing the previous wave file if play button is pressed
Next button		Point to the next wave file Stops audio playing Starts playing the next wave file if play button is pressed
Add file to playlist		Open file browser window and choose wave file to be added to playlist
Add folders		Open file browser window and choose entire folder to be added to playlist
Repeat buttons		At the end of file: - If repeat all is selected next wave file is selected and played - If repeat once is selected the played wave file is repeated - If repeat off is selected the audio player stop
Speaker button		Sets the volume at mute (first press) Sets the volume at value displayed in volume slider (second press
Volume slider		Sets the volume value
Progress slider		Sets the desired position in the wave file
Close button		Close audio player module

8.2.6 Video

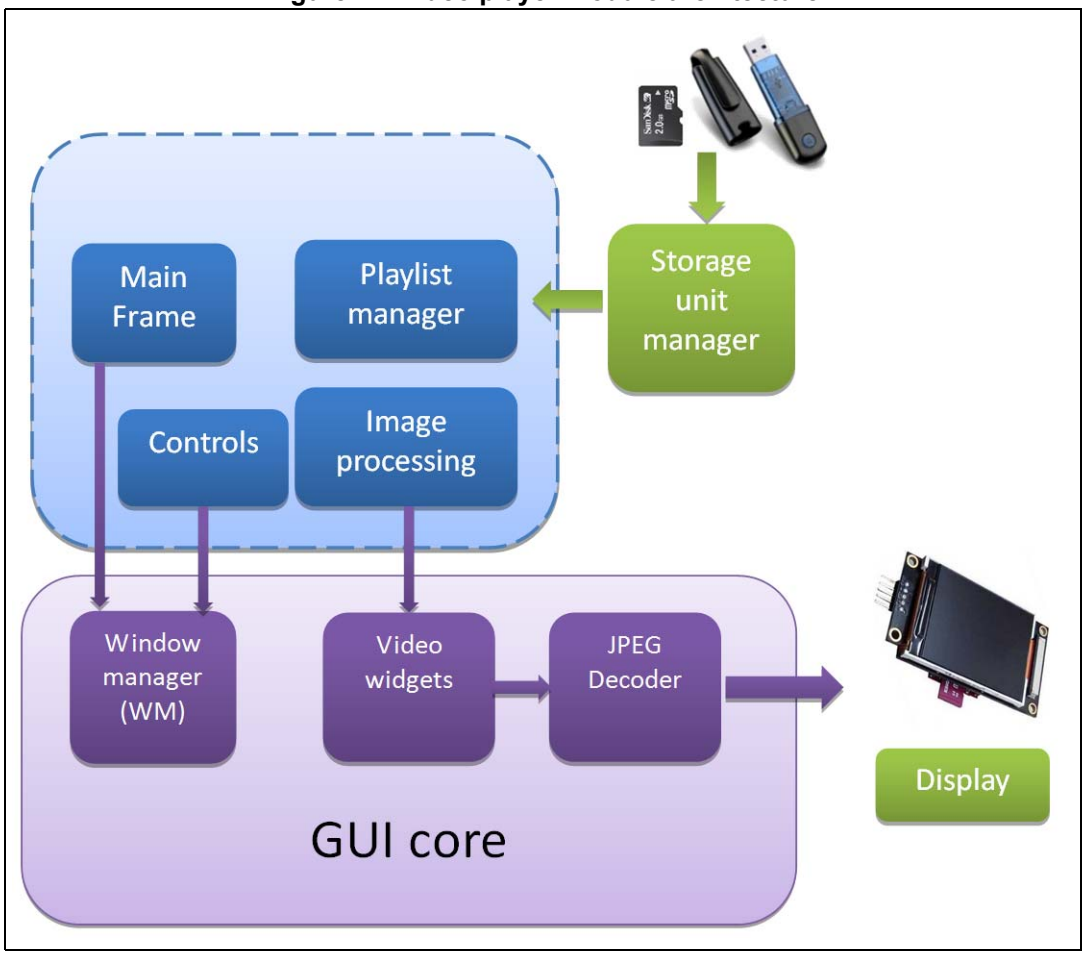
Overview

The video player module provides a video solution based on the STM32F4xx and STemWin movie API. It supports playing movie in emf format.

Architecture

Figure 42 shows the different video player modules and their connections and interactions with the external components.

Figure 42. Video player module architecture



Functional description

There are two ways to start Video player module:

- Either by touching the video player icon: *Figure 43*
- Or by using the file browser contextual menu: direct open feature.

When the video player is started, the following actions are executed:

- The graphical components are initialized:
  - The video frame
  - The control buttons
  - The list box field
- Memory is allocated to save the video list (pVideoList) and the file information (pFileInfo).

Figure 43. Video player module startup

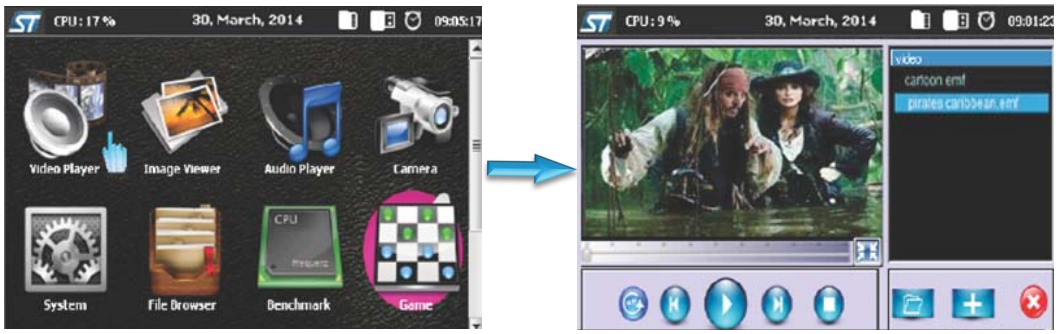


Table 18 summarizes the different actions behind each control button:

Table 18. Video module controls

Button	Preview	Brief description
Play button		Checks if the video size is not supported Supported video size: 0 < xSize < 1024 and 0 < ySize < 768 Changes the video player state to “VIDEO_PLAY” Reads the video file from storage unit Replaces play button by pause button
Pause button		Pauses the video file stream Changes the video player state to “VIDEO_PAUSE” Replaces pause button by play button
Stop button		Closes the video file from storage unit Stops video playing Changes the video player state to “VIDEO_IDLE”
Previous button		Points to the previous video file Stops video playing Changes the video player state to “VIDEO_IDLE”

Table 18. Video module controls (continued)

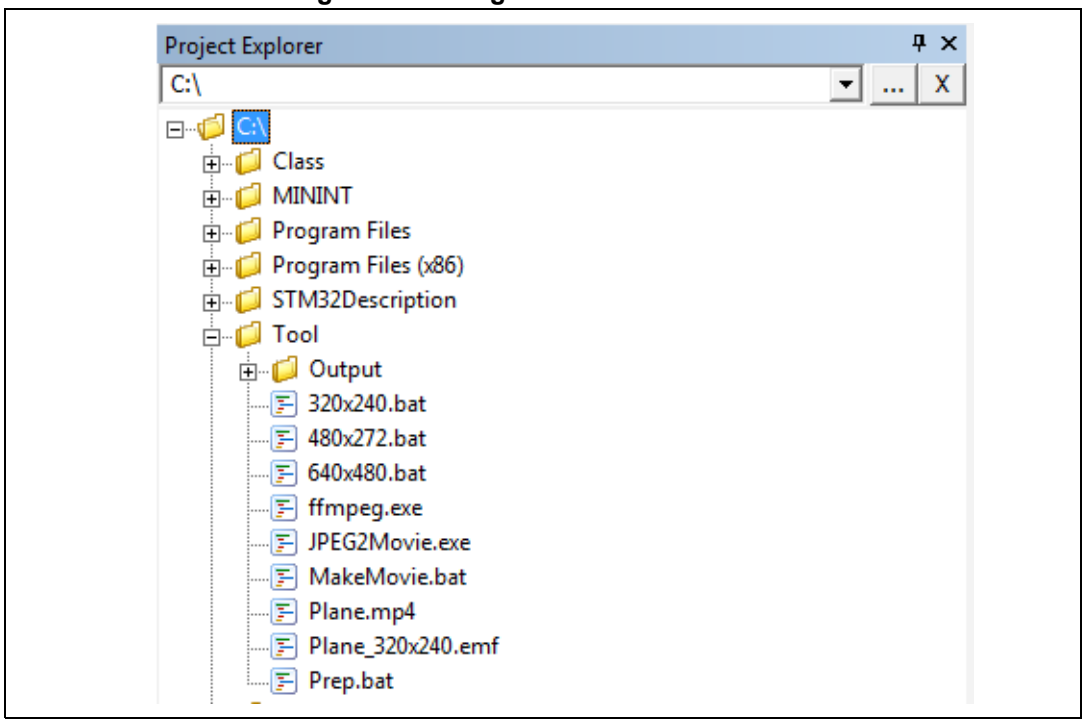
Button	Preview	Brief description
Next button		Points to the next video file Stops video playing Starts playing the next video file if play button is pressed
Add file to playlist		Opens file browser window and choose emf file from available storage unit to be added to playlist
Add folder		Opens file browser window and choose entire folder from available storage unit to be added to playlist
Repeat buttons		At the end of file: - If repeat all is selected next video file is selected and played - If repeat once is selected the played video file is repeated - If repeat off is selected the video player stops
Progress slider		Sets the desired position in the emf file
Full screen button		Scales the image to be showed on full screen mode
Close button		Closes video player module

Video file creation (emf)

To be able to play movies with the STemWin API functions it is required to create files of the STemWin specific EmWin movie file format. There are two steps to generate an emf file:

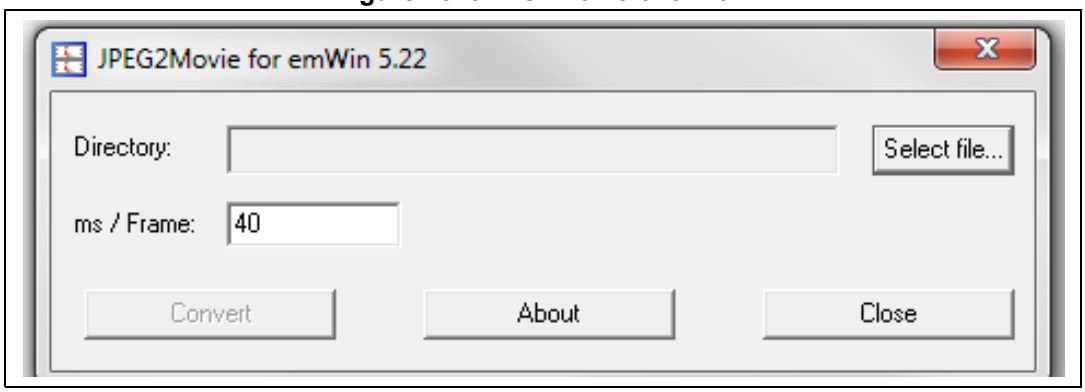
a) Convert files of any MPEG file format into a folder of single JPEG files for each frame (*Figure 44*). The free FFmpeg available at ffmpeg website can be used.

Figure 44. EMF generation environment



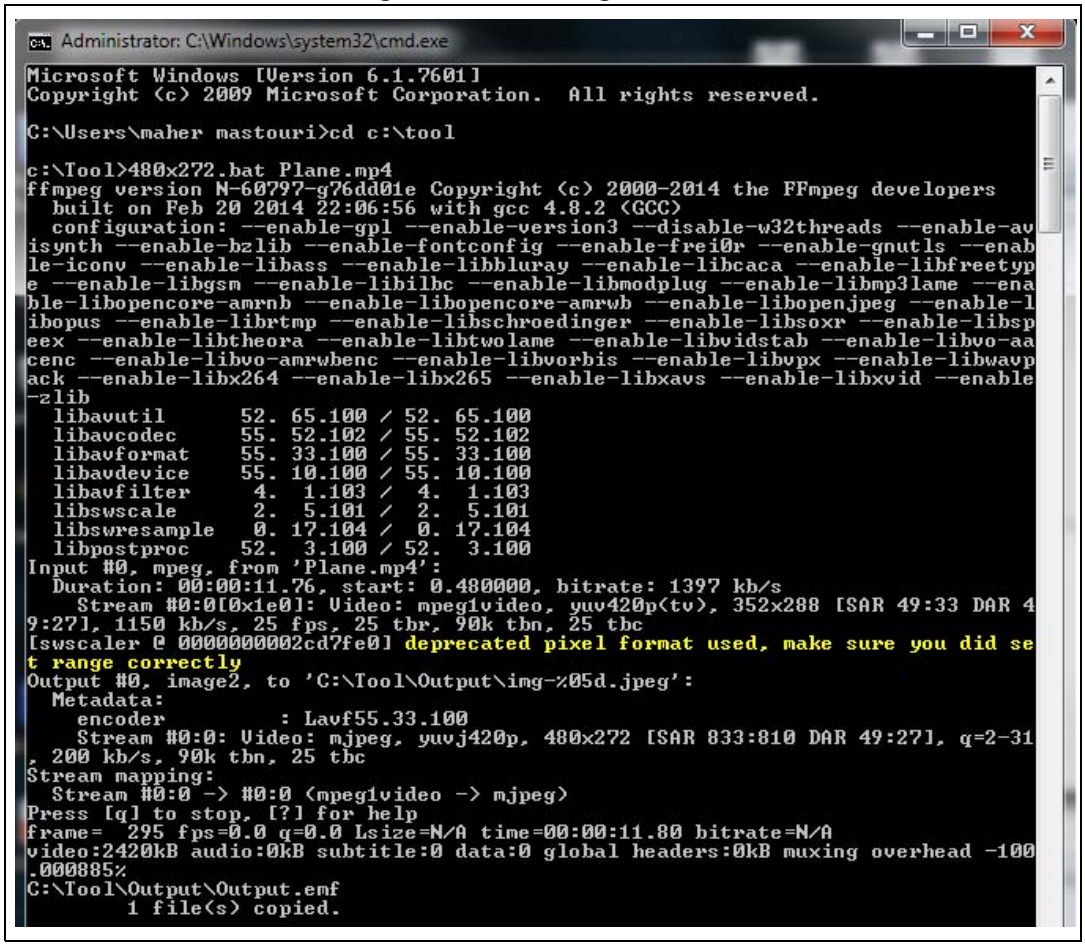
b) Create an emf file from JPEG file using JPEG2Movie tool available in STemWin package (see *Figure 45*).

Figure 45. JPEG2Movie overview



The above steps could be done once using a predefined batch (included in the STemWin package) as shown in *Figure 46*.

Figure 46. EMF file generation



For more information about how to use the emf generation batches, refer to the STemWin User and Reference Guide (UM3001).

Table 19. Batch files description

File	Explanation
Prep.bat	Sets some defaults to be used. Needs to be adapted as explained in *Prep.bat*.
MakeMovie.bat	Main conversion file. Not to be adapted normally.
<X_SIZE>x<Y_SIZE>.bat	Some helper files for different resolutions. Detailed explanation in *<X_SIZE>x<Y_SIZE>.bat*

Prep.bat

The Prep.bat is required to prepare the environment for the actual process. Calling it directly will not have any effect. It is called by the MakeMovie.bat. To be able to use the batch files it is required to adapt this file at first. This file sets variables used by the file MakeMovie.bat, they are listed in *Table 20*.

.

Table 20. Variables description

Variable	Description
%OUTPUT%	Destination folder for the JPEG files. Will be cleared automatically when starting the conversion with MakeMovie.bat.
%FFMPEG%	Access variable for the FFmpeg tool. Should contain the complete path required to call FFmpeg.exe.
%JPEG2MOVIE%	Access variable for the JPEG2MOVIE tool. Should contain the complete path required to call JPEG2Movie.exe.
%DEFAULT_SIZE%	Default movie resolution to be used. Can be ignored if one of the <X-SIZE>x<Y-SIZE>.bat files are used.
%DEFAULT_QUALITY%	Default quality to be used by FFmpeg.exe for creating the JPEG files. The lower the number the better the quality. Value 1 indicates that a very good quality should be achieved, value 31 indicates the worst quality. For more details please refer to the FFmpeg documentation.
%DEFAULT_FRAMERATE%	Frame rate in frames/second to be used by FFmpeg. It defines the number of JPEG files to be generated by FFmpeg.exe for each second of the movie. For more details please refer to the FFmpeg documentation.

MakeMovie.bat

This is the main batch file used for the conversion process. Normally it is not required to be change this file, but it is required to adapt Prep.bat first. It could be called with the parameters listed in *Table 21*:

Table 21. Parameters description

Parameter	Description
%1	Movie file to be converted
%2 (optional)	Size to be used. If not given %DEFAULT_SIZE% of Prep.bat is used.
%3 (optional)	Quality to be used. If not given %DEFAULT_QUALITY% of Prep.bat is used.
%4 (optional)	Frame rate to be used. If not given %DEFAULT_FRAMERATE% of Prep.bat is used.

Since the FFmpeg output can differ strongly from the output of previous actions, the MakeMovie.bat deletes all output files in the first place. The output folder is defined by in the environmental variable %OUTPUT% in Prep.bat. After that it uses FFmpeg.exe to create the required JPEG files for each frame. Afterwards it calls JPEG2Movie to create a single EMF file which can be used by STemWin directly. After the conversion operation the result can be found in the conversion folder under FFmpeg.emf. It also creates a copy of that file into the source file folder. It will have the same name as the source file with a size-postfix and .emf extension.

<X_SIZE>x<Y_SIZE>.bat

These files are small but useful helpers if several movie resolutions are required. The filenames of the batch files itself are used as parameter '-s' for FFmpeg.exe. You can simply drag-and-drop the file to be converted to one of these helper files. After that an .emf file with the corresponding size-postfix can be found in the source file folder.

8.2.7 USB Mass storage Device

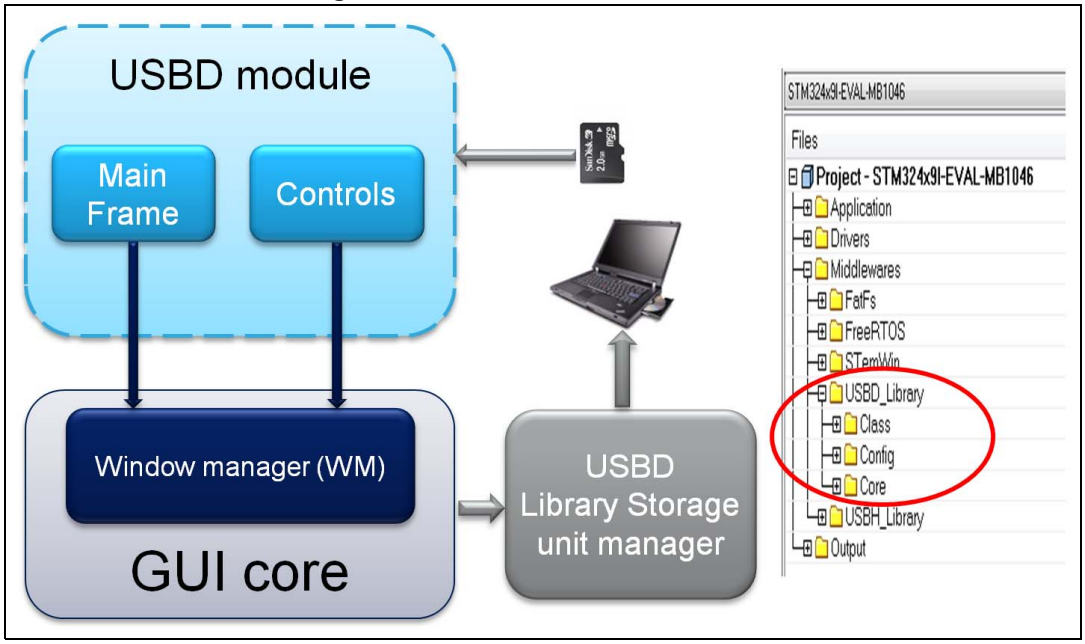
Overview

The USB device module includes mass storage device application using the MicroSD memory. It uses the USB OTG FS peripheral as the USB OTG HS is used for the USB disk Flash storage unit.

Architecture

Figure 47 shows the different USBD module components and their connections and interactions with the external components.

Figure 47. USBD module architecture



Data structure used

Table 22. Data structure for USBD module

Structure	Description
USBDSettingsTypeDef	sd_mounted: connection status

Functional description

Run USB Device demonstration by clicking USB device icon in the main desktop, as in *Figure 48*.

Figure 48. USBD module startup

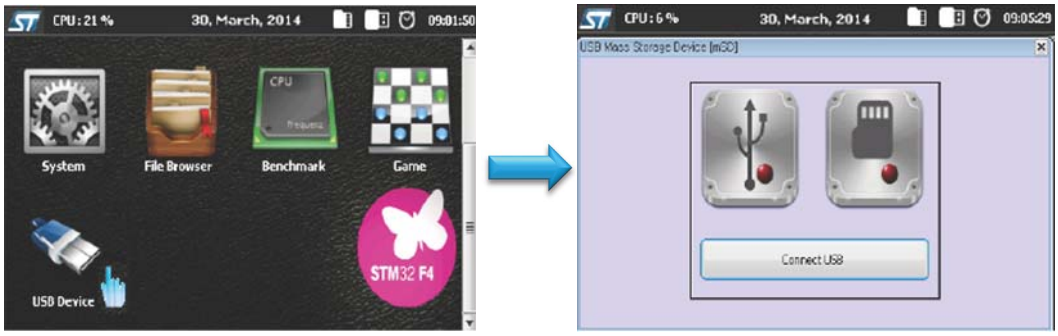


Table 23. USBD module controls

Button	Preview	Brief description
Connect USB	Connect USB	Changes the USB logo as follows:  Changes the USBD status as CONNECTED
Disconnect USB	Disconnect USB	Changes the USB logo as follows:  Changes the USBD status as DISCONNECTED

Table 23. USBD module controls (continued)

Button	Preview	Brief description
Insert microSD card	NA	Changes the microSD logo as follows: 
Remove microSD card	NA	Changes the microSD logo as follows: 
Close		Closes USBD module

8.2.8 Camera

Overview

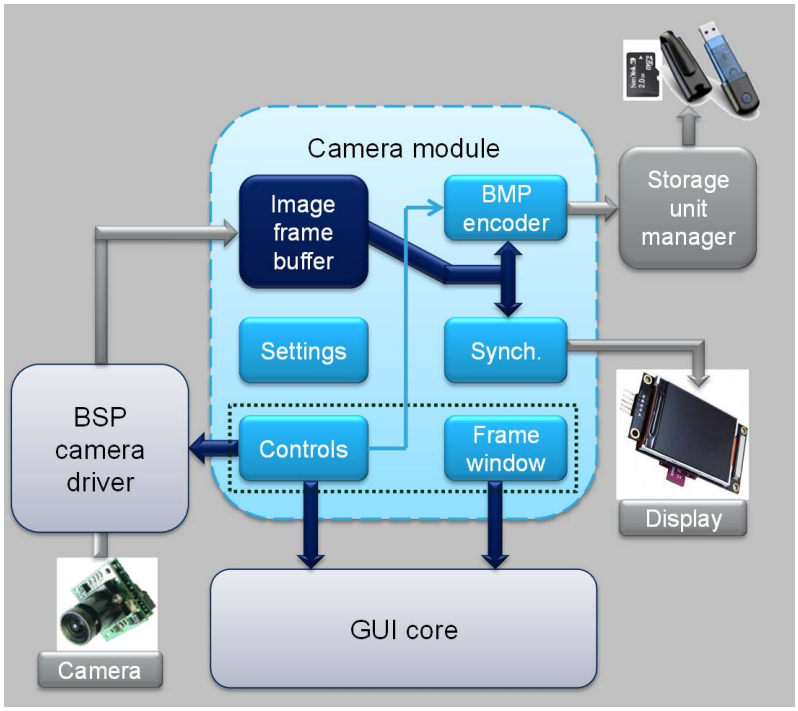
The camera application allows to directly and permanently display on the LCD the image captured using the camera module. It is also possible to take a snapshot and save it to a customizable location in the storage unit.

In addition to brightness and contrast which are adjustable, several effects can be applied to the output image: black and white, negative, antique...etc. Note that all these effects can be applied in runtime.

Architecture

Figure 49 shows the different camera module parts and their respective connections and interactions with the external components.

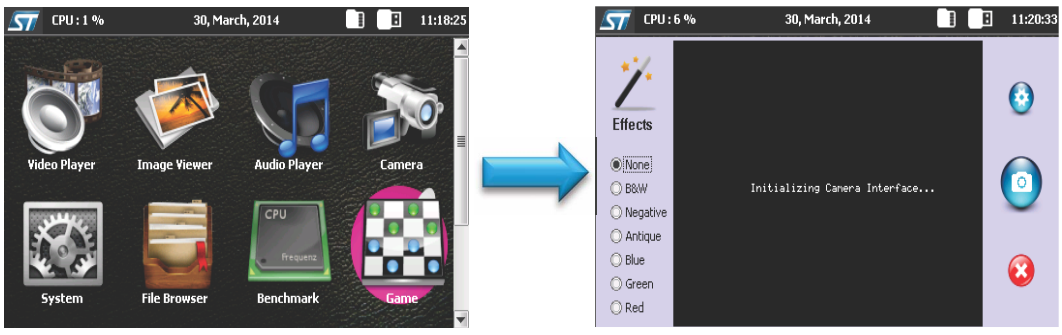
Figure 49. Camera module architecture



Functional description

To start the camera module click on the Camera icon, as indicated in *Figure 50*.

Figure 50. Camera module startup



When the camera module is started, the following actions are executed:

- The graphical components are initialized.
- Memory is allocated to save the capture folder location (pFileInfo).
- The saved parameters (brightness and contrast) are restored from the RTC backup register.

Table 24 summarizes the different actions behind each control button:

Table 24. Camera module controls

Button	Preview	Brief description
Settings		Creates and shows the settings dialog
Capture		Checks the camera state Displays a popup message ("Saving image...") Saves the current image to the specified file (default path is the root) Deletes the popup message
Close		Frees allocated memory during the initialization Stops the camera module Ends the module dialog
Effects		Applies the selected effect on the fly via the BSP camera driver commands

8.2.9 Image viewer

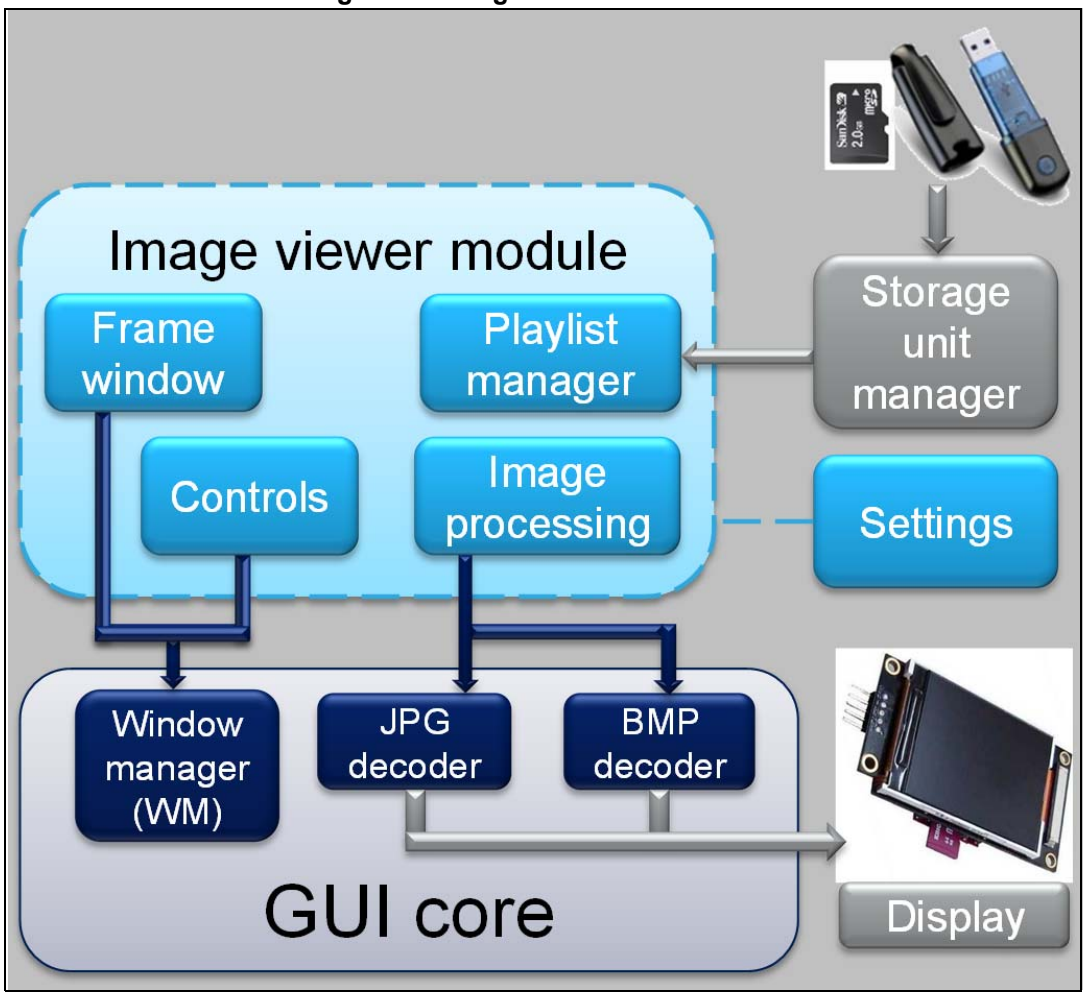
Overview

The Image viewer module allows displaying bmp and jpg pictures. It is possible to load the full images list from a folder or to add the images manually to the playlist. Once the playlist is created, navigation between pictures can be done either via Next and previous buttons or by enabling the slide show mode. The slide show timer can be changed on the fly (there is no need to restart the module).

Architecture

Figure 51 shows the different image viewer parts and their respective connections and interactions with the external components.

Figure 51. Image viewer architecture



Functional description

There are two ways to start Image viewer module:

- Either by touching the Image viewer icon (*Figure 52*);
- Or by using the file browser contextual menu: direct open feature.

When the image viewer is started, the following actions are executed:

- The graphical components are initialized:
  - The image frame
  - The control buttons
  - The list box field
- Memory is allocated to save the image list (pImageList) and the file information (pFileInfo).
- The saved parameters are restored from the RTC backup register.

Figure 52. Image viewer startup

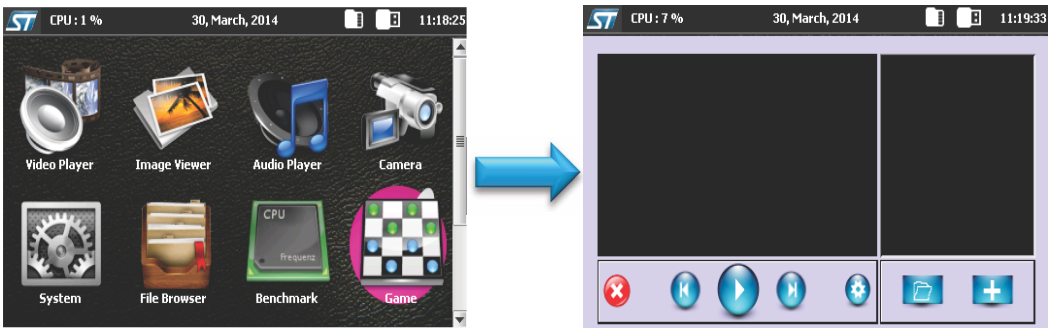


Table 25 summarizes the different actions behind each control button.

Table 25. Image viewer module controls

Button	Preview	Brief description
Close		Frees allocated memory Ends the module dialog
Previous		Closes the current image Opens the previous image Refreshes the image frame Updates the selection in the playlist
Start slideshow		Closes the current image Opens the next image Refreshes the image frame Creates the slideshow timer
Next		Closes the current image Opens the next image Refreshes the image frame Updates the selection in the playlist
Settings		Creates and shows the settings dialog
Add folder		Opens the directory chooser to allow selection of an entire folder and then adds all the images included in this folder to the playlist
Add file		Opens the file chooser to allow selection of an image which will be added to the playlist.

9 Revision history

Table 26. Document revision history

Date	Revision	Changes
24-Apr-2014	1	Initial release.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

ST PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN: (A) SAFETY CRITICAL APPLICATIONS SUCH AS LIFE SUPPORTING, ACTIVE IMPLANTED DEVICES OR SYSTEMS WITH PRODUCT FUNCTIONAL SAFETY REQUIREMENTS; (B) AERONAUTIC APPLICATIONS; (C) AUTOMOTIVE APPLICATIONS OR ENVIRONMENTS, AND/OR (D) AEROSPACE APPLICATIONS OR ENVIRONMENTS. WHERE ST PRODUCTS ARE NOT DESIGNED FOR SUCH USE, THE PURCHASER SHALL USE PRODUCTS AT PURCHASER’S SOLE RISK, EVEN IF ST HAS BEEN INFORMED IN WRITING OF SUCH USAGE, UNLESS A PRODUCT IS EXPRESSLY DESIGNATED BY ST AS BEING INTENDED FOR “AUTOMOTIVE, AUTOMOTIVE SAFETY OR MEDICAL” INDUSTRY DOMAINS ACCORDING TO ST PRODUCT DESIGN SPECIFICATIONS. PRODUCTS FORMALLY ESCC, QML OR JAN QUALIFIED ARE DEEMED SUITABLE FOR USE IN AEROSPACE BY THE CORRESPONDING GOVERNMENTAL AGENCY.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.
Information in this document supersedes and replaces all information previously supplied.
The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com