Application Note:
TLT-0627-AN-CM_ARMCC-MicroUIDisplay

Adding Support For A Display

In relation to: CM_ARMCC products

Features

This Application Note explains how to create a Java Platform with support for a display, using the MicroEJ® environment and the ST Standard Peripherals Library, and how to build it using Keil µVision.

Description

This Application Note assumes the reader wishes to understand the steps involved in creating a Java Platform with support for a display. It explains in detail all the steps required to build the platform and test it with a simple application that draws on the display.

Table of Contents

List of Figures

1 Introduction

1.1 Intended audience

The intended audience for this Application Note are developers who wish to add support for a display to a MicroEJ® Java Platform (JPF).

1.2 Scope

This Application Note describes the steps required to add support for a display to an existing MicroEJ Java Platform that was built for an STM3220GEVAL or STM3240GEVAL board.

1.3 Prerequisites

This Application Note assumes that the reader has already created a Java Platform with no display support, as described in the *TLT-0625-AN-CM_ARMCC-FromScratch* Application Note. All the prerequisites specified in that Application Note apply equally to this note.

2 MicroUI displays: principles of operation

This section focuses on those aspects of MicroUI™ relevant to this Application Note. For full details about the operation of MicroUI please consult the *UI Pack Reference Manual*[1].

2.1 Flow of control

The diagram below shows the overall flow of control.

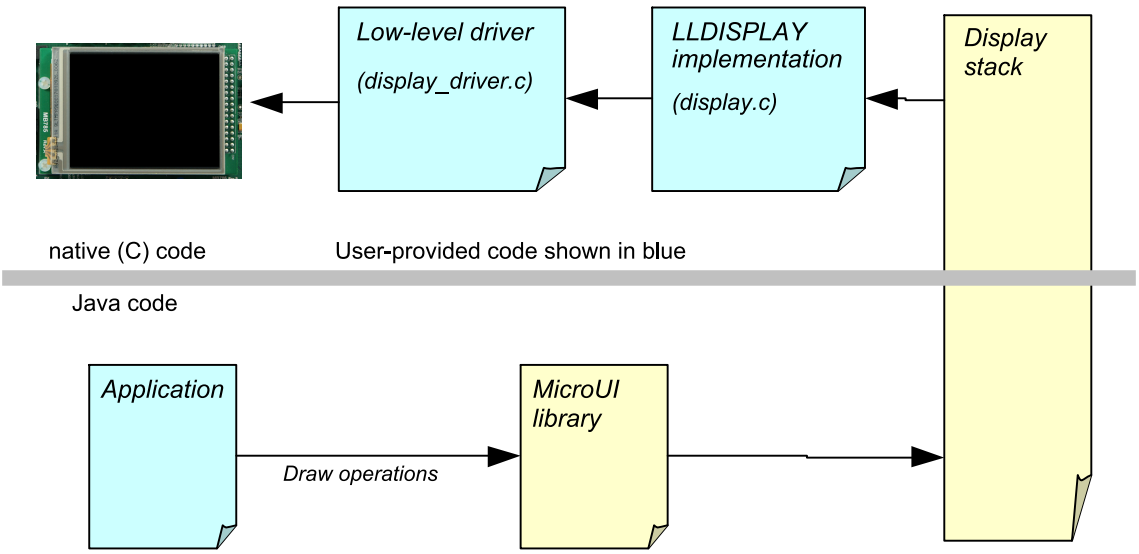

Figure 2.1. Overall Flow of Control

When the application draws on the display the MicroUI Java library interacts with the *display stack* to translate the draw requests into pixel updates. The display stack is code that bridges between the Java and C worlds, and implements the selected display policies. When the display stack wishes to update the physical display it makes to a request to the user-supplied driver, which is an implementation of an `LLDISPLAY` API. In this example the driver is divided into two parts: `display.c` is a mostly hardware-independent implementation of the `LLDISPLAY` API, while `display_driver.c` handles the interaction with the hardware.

A number of different display stack implementations are supplied with the MicroEJ Java platform, each tailored to different hardware capabilities and different buffering policies. This example uses the *16-xy-copy* stack implementation because:

- The display supports *16* bits-per-pixel
- The driver assumes pixels are laid out in the back buffer row-by-row (*xy*), rather than column-by-column (yx)
- The display cannot be switched dynamically to refresh from different buffers, so to achieve double-buffering the back buffer is *copied* to the display buffer when drawing is complete.

There are also different versions of the `LLDISPLAY` API, each tailored to a different buffering policy. In this case the driver must implement the `LLDISPLAY_COPY` API because double buffering is achieved by copying.

2.2 APIs, implementations and instances

The `LLDISPLAY_COPY` is defined by two files provided with the JPF:

- `LLDISPLAY_COPY.h`

[1] *ARM Cortex-Mx ARMCC - UI Reference Manual*

- `LLDISPLAY_COPY_IMPL.h`

The first of these defines the API that the display stack provides, the second defines the API that the driver has to implement (i.e. functions that are called by the display stack).

To avoid conflicts in platforms with two or more displays, each implementation of the API must be given a unique name. C macros are used to create names for the implemented functions that incorporate the implementation name. The implementation name is defined by the user in the driver source code using a `#define` statement.

The implementation must declare a data structure that represents the display at run-time. This data structure roughly corresponds to the concept of a run-time object instance.

The UML diagram below summarizes these concepts.

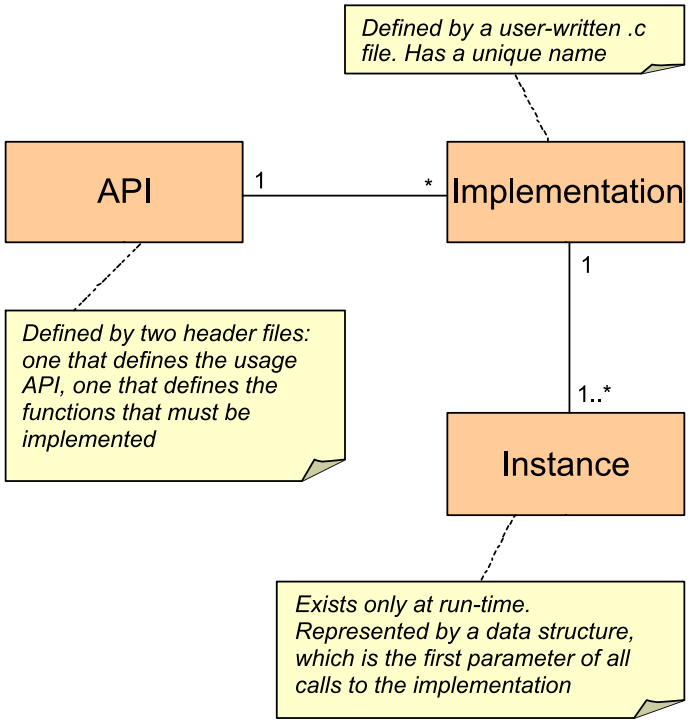

Figure 2.2. Native Implementation Model

The example in this Application Note creates a single implementation, and a single instance of the implementation.

3 An outline of the required steps

To add support for a display:

1. Enhance the JPF
2. Create the Java application that will test the JPF

   A Java project is created within the MicroEJ environment, and the required code written.
3. Build the Java application code

   The Java project is “run” in the MicroEJ environment to produce a binary object file that can be linked by µVision.
4. Write the required C files
5. Configure the Keil µVision project

   The properties of the µVision project are adjusted so that it includes all the required sources and libraries, and refers to the correct header files.
6. Build and deploy the C project

4 The steps in detail

4.1 Enhance the JPF

This Application Note assumes that you already have a basic JPF, like that created in the *TLT-0625-AN-CM_ARMCC-FromScratch* Application Note, and covers the enhancement of the JPF and board support package (BSP) to add support for a display.

4.1.1 Modules

A MicroEJ *module* is a group of related files (Java libraries, scripts, link files, native libraries, mocks, etc.) that together provide all or part of a platform functionality.

JPF architectures provide a set of modules that can be selected thanks to the JPF configuration file (eg `xxx.platform` file). The content tab of the latter provides facilities to select one or several modules. The **Build Platform** task then deploys selected modules on the JPF.

Module description can be seen by selecting it on JPF configuration interface. Most of modules needs to be configured ; if it's the case, module details contain a **Configuration** item explaining which file has to be created in order to configure the module.

4.1.2 Select the required modules

To support board screen, it is necessary to install and configure three modules: `bsp`, `microui` and `display`. Microui module is a Java library that exposes to application features in order to manipulate the screen (ex: `Screen.class`). Display module is a native library necessary for MicroUI internal functioning. BSP module is needed to do the link with C implementation. To select them,

- Open the `xxx.platform` file of your JPF configuration project,
- Go to the content tab,
- On Modules part, check **Build Support > Board Support Package**, **UI > Display** and **UI > MicroUI** items.

4.1.3 Configure selected modules

The next step is to configure these modules.

- Display module can be configured using a `display/display.properties` file (cf module details).

  Create an `display` folder on the JPF configuration project and copy the `display.properties` file provided by the Application Note on it. This configuration file tells builder to deploy the "default" display module implementation. This "default" implementation also needs to be configured ; that's why a properties file is needed to define the number of bits per pixels, the pixel memory layout and the buffering mode. Those parameters are described in details on UI user manual. Please refer to it to further understanding.

- MicroUI module has to be configured using a `microui/microui.xml` file (cf module details).

  Create an `microui` folder on the JPF configuration project and copy the `microui.xml` file provided by the Application Note on it. For this example, only a very simple configuration is required ; the file contains the following:

```
<microui>
 <display name="DISPLAY"/>
</microui>
```

  This specifies that the MicroUI configuration will comprise a single display, whose name will be `DISPLAY`[2].

[2]The name has no significance when there is only one display.

- BSP module can be configured using a `bsp/bsp.xml` file (cf module details on §8.7 of UI Users Manual).

  Create an `bsp` folder on the JPF configuration project and copy the `bsp.xml` file provided by the Application Note on it.

4.1.4 Update the JPF

In order to add input and microui modules on the JPF, launch the **Build Platform** task. JPF source folder is modified to contain modules files.

4.2 Create the Java application that will test the JPF

To reduce typing, the Java application is provided with this Application Note as a ready-to-import MicroEJ project.

Create a new MicroEJ Java Project. Do this by selecting:

File → Import... → General → Existing Projects into Workspace

and selecting the `MicroUIDisplayApp.zip` archive file.

In the **Package Explorer**, browse to the `MicroUIDisplay` Java class, which looks like this:

```
package com.is2t.example;

import ej.microui.Colors;
import ej.microui.io.Display;
import ej.microui.io.GraphicsContext;

public class MicroUIDisplay {
    public static void main(String[] args) {
[1]     Display display = Display.getDefaultDisplay();
[2]     GraphicsContext gc = display.getNewGraphicsContext();
        int w = display.getWidth();
        int h = display.getHeight();
     gc.setColor(Colors.BLUE);
[3]  gc.fillRect(0, 0, w / 3 + 1, h);
  gc.setColor(Colors.WHITE);
  gc.fillRect(w / 3, 0, w / 3 + 1, h);
  gc.setColor(Colors.RED);
  gc.fillRect(2 * w / 3, 0, w / 3 + 1, h);
    }
}
```

The key lines in the application are marked with numbers:

1. The static method `getDefaultDisplay` is called to get a reference to the display (in this case there is only one, so the default display is the only display).

2. The display is used to get a reference to a `GraphicsContext`, which supports drawing commands. This example paints directly on the display; more typically the application would use the MicroUI framework where a `GraphicsContext` is passed as a parameter into a paint or repaint method of a `View` or `Viewable`, but a discussion of the MicroUI framework is beyond the scope of this document.

3. Having set the color that will be used in subsequent drawing commands, the third left part of screen is filled blue.

4.3 Build the Java application binary file

The next step is to build the Java application into a binary file that can be linked with the C parts of the platform.

To build the application a suitable launch configuration must be created.

Right-click in the **Package Explorer** on the `MicroUIDisplay` Java class, and select:

Run As → Run Configurations...

The **Run Configurations** dialog will open. Double-click the **MicroEJApplication** entry in the list to the left of the dialog to create a new configuration.

The details in the **Main** tab will be entered already, and should look like this:

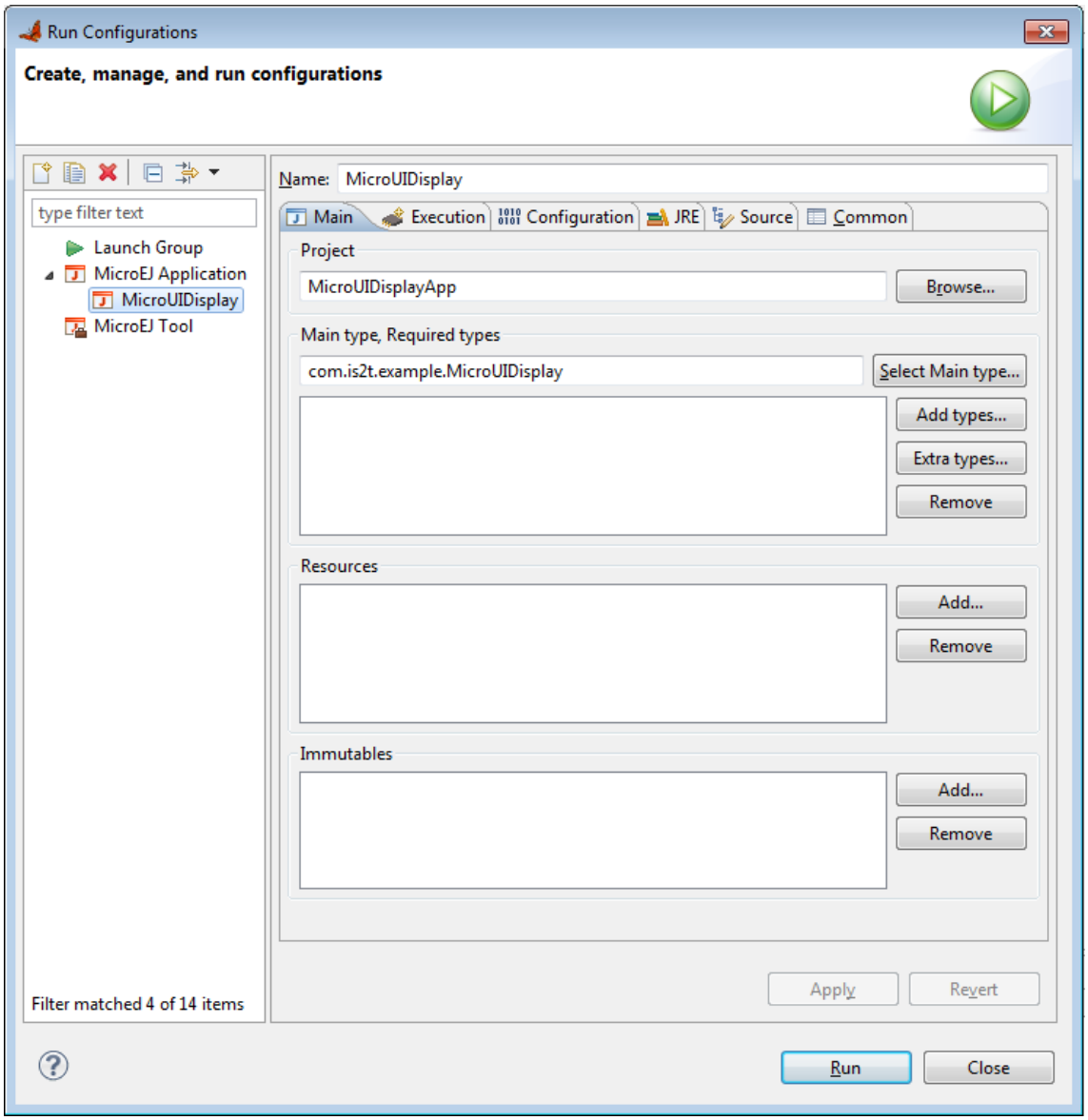

Figure 4.1. Launch Configuration - Main Tab

Select the **Execution** tab. Check the **Execute on EmbJPF** option, as shown below:

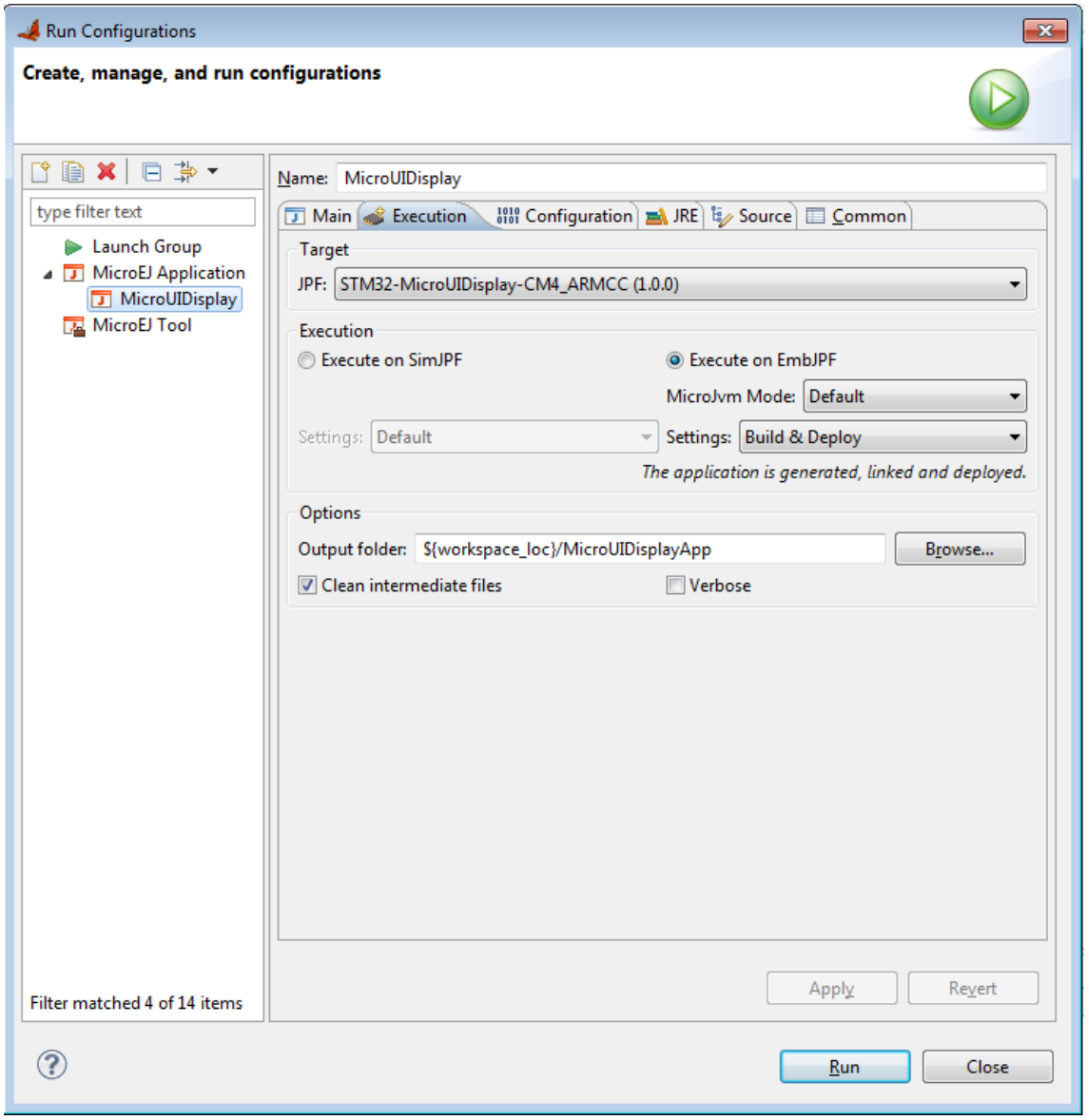

Figure 4.2. Launch Configuration - Execution Tab

Press **Run**. The application is built into a binary file called `SOAR.o`, located in the `com.is2t.example.MicroUIDisplay` folder of the `MicroUIDisplayApp` project.

4.4 Write the required C files

This Application Note does not cover creation of the most basic BSP, but assumes you already have a "java from scratch" (see *TLT-0625-AN-CM_ARMCC-FromScratch*) or similar BSP.

The JPF (as enhanced earlier) contains a number of C include files that are used by the user-supplied C implementation. These include files can be found in the `xxx-jpf/source/include` folder.

The user must provide C code that implements the `LLDISPLAY_COPY` API, as defined in the `LLDISPLAY_COPY_impl.h` include file.

Suitable C implementations are provided with this Application Note. These files are in addition to those required for a basic JPF, as supplied with the *TLT-0625-AN-CM_ARMCC-FromScratch* Application Note. This Application Note assumes that the:

`STM32Fxxx_StdPeriph_Lib_Vxxx/Project/FromScratch` folder, as defined in *JavaFromScratch*, is in place and correct.

The additional files supplied with this Application Note are:

- `LLDISPLAY_STM32x0GEVAL.c` – implements the `LLDISPLAY_COPY` API (which is defined in `LLDISPLAY_COPY_impl.h`)
- `LLDISPLAY_STM32x0GEVAL.h` – header file for LLDISPLAY_STM32x0GEVAL.c
- `microui_bsp.c` - defines the displays and image decoders

These files include comments which provide some explanation of their content. Nevertheless, some of these files are examined in further detail in the following section.

Install the supplied files into a new C project:

1. Create a new folder: `STM32Fxxx_StdPeriph_Lib_Vxxx/Project/MicroUIDisplay`
2. Copy the contents of the `STM32Fxxx_StdPeriph_Lib_Vxxx/Project/FromScratch` folder (including sub-folders) into the newly created folder.
3. Copy the files supplied with this Application Note into the newly created folder.

4.4.1 LLDISPLAY_STM32x0GEVAL.h

The file `LLDISPLAY_STM32x0GEVAL.h` includes the following lines:

```
[1] #define _LLDISPLAY_STM32x0GEVAL

 #include "LLDISPLAY_COPY.h"

[2] typedef struct LLDISPLAY_STM32x0GEVAL{
        struct LLDISPLAY_COPY header;
        // can add some fields
    } LLDISPLAY_STM32x0GEVAL;

[3] void LLDISPLAY_STM32x0GEVAL_new(LLDISPLAY_STM32x0GEVAL* env);

    #endif
```

The key lines are marked with numbers:

1. This defines the name of this implementation of the `LLDISPLAY_COPY` API. This has to match the mapping of the `nativeImplementation` name in `bsp.xml` file that we saw earlier.
2. The data structure for the run-time instance of the display. The header field must be present and be the first field in the structure, but other fields can be added if required.
3. Declares the `new` function that is called to initialize the run-time instance. Note that the name of this function must be `[implementation-name]_new`.

4.4.2 LLDISPLAY_STM32x0GEVAL.c

The file `LLDISPLAY_STM32x0GEVAL.c` implements the `LLDISPLAY_COPY` API as defined by `LLDISPLAY_COPY_impl.h`. These functions are called by the display stack. All the functions have as their first parameter a pointer to the instance data structure, the structure of which is defined in the `LLDISPLAY_STM32x0GEVAL.h` file.

The contents of `LLDISPLAY_STM32x0GEVAL.c` are described a section at a time.

```
uint8_t LLDISPLAY_COPY_IMPL_initialize(LLDISPLAY_COPY* env) {
    _STM32xxG_LCD_Init();
 LCD_Clear(White);
    return MICROEJ_TRUE;
}
```

The initialization function delegates to the display driver.

```
int32_t LLDISPLAY_COPY_IMPL_convertDisplayColorToRGBColor(
                        LLDISPLAY_COPY* env,
                        int32_t color) {
    /* The display pixel value encodes:
    0xf800 = 1111 1000 0000 0000 - the RED part
    0x07e0 =       111 1110 0000 - the GREEN part
    0x001f =             1 1111 - the BLUE part
    */
    return ((color & 0xf800) << 8) |
           ((color & 0x07e0) << 5) |
           ((color & 0x001f) << 3);

}

int32_t LLDISPLAY_COPY_IMPL_convertRGBColorToDisplayColor(
                        LLDISPLAY_COPY* env,
                        int32_t color) {
    uint32_t rgbColor = (uint32_t)color;
    /* The RGB value encodes:
    0xf80000 = 1111 1000 0000 0000 0000 0000 - the RED part
    0x00fc00 =           1111 1100 0000 0000 - the GREEN part
    0xf8     =                     1111 1000 - the BLUE part
    */
    return ((rgbColor & 0xf80000) >> 8) |
           ((rgbColor & 0x00fc00) >> 5) |
           ((rgbColor & 0xf8) >> 3);
}
```

These functions convert the pixel value used by the display into a MicroUI RGB value and vice-versa. For this display, there are 16 bits per pixel, but MicroUI RGB values are 24 bits per pixel.

The `LLDISPLAY_COPY_IMPL_convertDisplayColorToRGBColor` function uses the red, green and blue values from the display value as the most significant bits of the MicroUI RGB value ("expanding" the 16 bit representation into its 24 bit equivalent).

The `LLDISPLAY_COPY_IMPL_convertRGBColorToDisplayColor` function uses the most significant bits of the MicroUI RGB value to create a 16 bit equivalent (which, of course, is an approximation – the least significant bits of the MicroUI RGB values are "lost" in the "compression" from 24 bits into 16 bits).

```
void LLDISPLAY_COPY_IMPL_copyBuffer(LLDISPLAY_COPY* env,
                    int32_t xmin, int32_t ymin, int32_t xmax, int32_t ymax) {
[1] uint16_t* addr = (uint16_t*)MEM_BUFFER + (ymin * WIDTH);
    // number of pixels to copy (row aligned)
[2] int32_t cpt = (ymax - ymin + 1) * WIDTH;
    LCD_SetCursor(ymin, WIDTH-1);
[3] LCD_WriteRAM_Prepare();

    // copy pixels
    while(--cpt>=0) {
[4]     LCD_BUFFER->LCD_RAM = *(addr);
        addr++;
    }
}
```

The function `LLDISPLAY_COPY_IMPL_copyBuffer` copies the rectangle of pixels defined by the opposite corners (`xmin`, `ymin`) to (`xmax`, `ymax`) from the off-screen section of back buffer into the display memory. In fact, this implementation ignores the `xmin` and `xmax` values because the hardware requires full rows to be updated.

The key lines in the function are marked with numbers:

1. This calculates the address (within the "back buffer") of the first pixel value to be copied. If `ymin == 0` then this is the address of the start of the "back buffer".

2. This calculates how many pixel values need to be copied.

3. This sets up the driver ready to copy the pixel values through successive calls of `writeLcdPixel`. That is, the way the driver works, we do not need to specify the location of each pixel whose value we are writing, but can instead stream the pixels in from a known starting position. Note that only the y starting position is specified because updating always commences at the beginning of a row.

4. This loop streams the pixel values into the display.

The functions related to the display's backlight and contrast do not need to be implemented for this example – they are just empty implementations so are not shown here.

4.4.3 microui_bsp.c

This source file contains code that defines the data structures for the display, and provides to the display stack a function that can be used to initialize the display.

```
[1] static struct LLDISPLAY_STM32x0GEVAL lcd_driver;

    // display creation function to be used in definition of LCD_TABLE below
[2] static void* createLCD(void) {
        LLDISPLAY_STM32x0GEVAL_new(&lcd_driver);
            return &lcd_driver;
    }

    // display stack tables - there is only one
[3] void* LCD_TABLE[] = {createLCD,0};

    // no image decoders are needed in this example, but
    // IMAGE_DECODERS_TABLE still needs to be declared
[4] void* IMAGE_DECODERS_TABLE[] = {0};
```

The key lines in the function are marked with numbers:

1. Declares the data structure for the display, using the struct defined in `display.h`.

2. Defines a function that will be used by the display stack to initialize the display data structure. It returns the address of the data structure.

3. Declares a data structure with the well-known name `LCD_TABLE` initialized with the address of the `createLCD` function. This null-terminated list is used by the display stack to initialize the display and obtain the address of its data structure.

4. Declares a data structure with the well-known name `IMAGE_DECODERS_TABLE` initialized with just a null termination because there are no image decoders in this example.

4.5 Configure the Keil µVision project

The Keil µVision project file `Project.uvproj` in the `MicroUIDisplay/MDK-ARM` folder must be changed to add the extra source files.

Double-click the `Project.uvproj` file to open it in the µVision environment. The contents of the project will look like this:

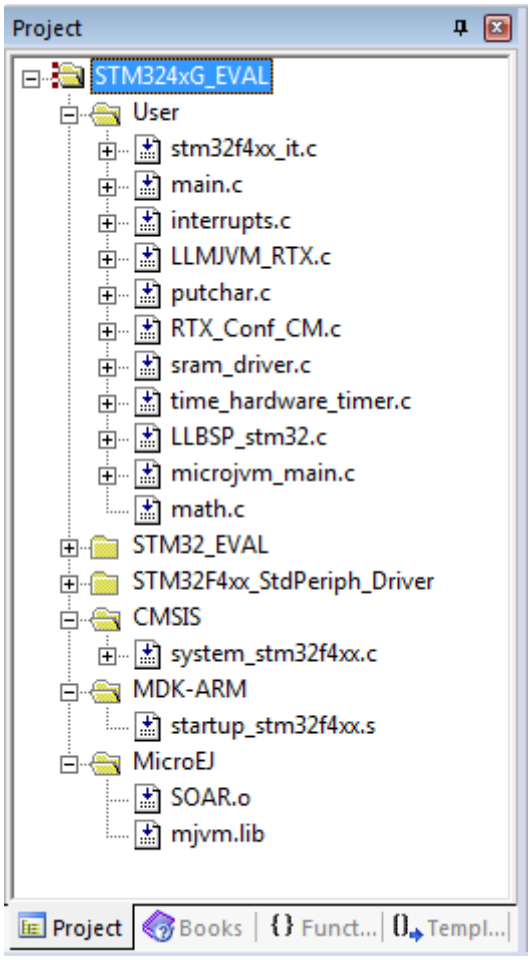

Figure 4.3. µVision Project Content Before Configuration

The actual files in the **User** group may vary.

Several different configuration actions are required:

1. Configure the include path

   Right-click on the root of the project and select **Options for Target 'STM322xG_EVAL'...** On the **C/C++** tab press the **...** button to the right of the **Include Paths** box. Press the **New (Insert)** button. Press the **...** button to the right of the new entry and navigate to the `xxx-jpf\source\include` folder. (To find the location of this, in the MicroEJ environment, right click on the folder and select **Properties**). Press **OK**.

2. Add the extra C files

   Right-click on the **User** group of the project and select **Add Files to Group 'User'...**. Navigate to the `MicroUIDisplay` folder that holds the C files and select `LLDISPLAY_STM32x0GEVAL.c` and `microui_bsp.c`.

3. Remove the `FromScratch` application.

   Right-click on the `SOAR.o` file and select **Remove File ...**.

4. Add the required libraries

   Right-click on the **MicroEJ** group and select **Add Files to Group 'MicroEJ'...**. Navigate to the `com.is2t.example.MicroUIDisplay` folder of the `MicroUIDisplayApp` application, and select the `SOAR.o` file. It may be necessary to select **Files of type: All files (*.*)** in the dialog box to see the file. Press **Add**. If µVision asks for the type of the file, select **Object file**.

Now navigate to the `xxx-jpf\source\lib` folder of the `FromScratchJPF`, and select the `display.lib` file. Press **Add** and **Close**.

The project should now look like this:

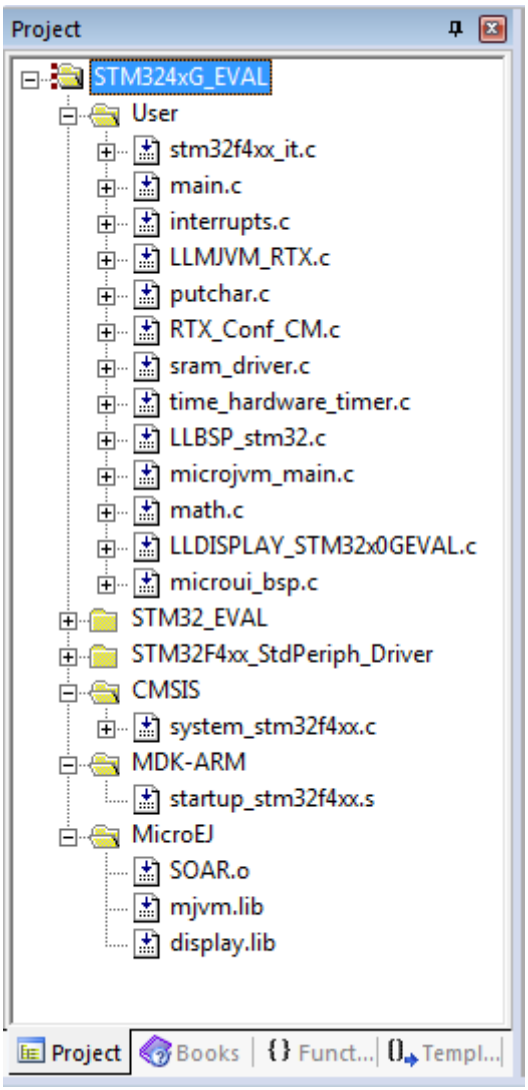

Figure 4.4. µVision Project Content After Configuration

4.6 Build and deploy the C project

The C code should now compile cleanly. Build it using **Project** → **Build target** (**F7**). It can now be downloaded to the target board using the ST-LINK connection. Reset the board to run the application and the display should show a red cross.

5 Document History

Date	Revision	Description
February 19th, 2013	A	First release
November 12th, 2013	B	MicroEJ ARM Cortex-M (Keil MDK-ARM) 2.0.0 compatibility
June 10th, 2014	C	MicroEJ ARM Cortex-M (Keil MDK-ARM) 3.0.0 compatibility

Headquarters
11, rue du chemin Rouge
44373 Nantes Cedex 3
FRANCE
Phone: +33 2 40 18 04 96
www.is2t.com