Application Note:
TLT-0628-AN-MICROEJ-FrontPanelMock

Adding a Front Panel Mock

In relation to: MICROEJ products

Features

This Application Note explains how to add a Front Panel Mock to a JPF (Java Platform), so that applications which use a display or react to input devices events can be run in the Simulator.

Description

This Application Note assumes the reader wishes to understand the steps involved in adding a Front Panel Mock, which a the visual component of the Simulator, to a Java Platform. It explains in detail all the steps required to create the Front Panel Mock and test it with a simple application that draws on the display and reacts to input devices events.

Table of Contents

List of Figures

1 Introduction

1.1 Intended audience

The intended audience for this Application Note are developers who wish to run applications which use a display, hardware buttons or touch screen in the simulator.

1.2 Scope

This Application Note describes the steps required to add a Front Panel Mock, supporting buttons and touch screen, to a JPF.

1.3 Prerequisites

The MicroEJ environment for MicroEJ® (MICROEJ-PKG-STD-MICROEJ-3.1.0 or later) must be installed and any required licenses obtained.

This Application Note assumes that the reader has already read the *TLT-0626-AN-MICROEJ_MicroUIButtons* and *TLT-0627-AN-MICROEJ_MicroUIDisplay* Application Notes in order to understand about displays and inputs management. However, adding a Front Panel Mock to a JPF does not require a C environment, so has fewer prerequisites than either of those Application Notes.

Furthermore, one of the strengths of MicroEJ® is that it allows applications to be developed against a simulator even before the hardware is available.

For convenience, this Application Note will start from scratch, but it is straightforward to apply the same steps to an existing JPF.

2 An Outline of the Steps

To create and use a JPF that includes a Front Panel Mock, following steps needs to be done :

1. Create a JPF configuration project.

   A JPF Configuration project is created within the MicroEJ environment. It is based on a supplied JPF Architecture.

2. Configure and build the JPF.

   A JPF is created within the MicroEJ environment. The JPF is built in function of its configuration.

3. Define the Front Panel content and behavior.

   This involves creating a definition of the visual and event handling characteristics of the hardware.

4. Import the Java application that will test the input support and use the display of the JPF.

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

5. Run the application in the simulator.

3 The Steps In Detail

3.1 Create the JPF configuration

The first step is to create a JPF Configuration based on one of delivered JPF architectures. Here we will assume use of the CORTEX-M3-based JPF Architecture. Create the JPF Configuration by selecting, in the MicroEJ environment : **File** → **New** → **Java Platform**

A dialog box appears in order to provide the JPF Architecture selection and to suggest to start from an JPF example. For the application note, select the CORTEX-M3-based JPF Architecture. To create a JPF "from scratch", uncheck the option "Create a platform from an example or a template".

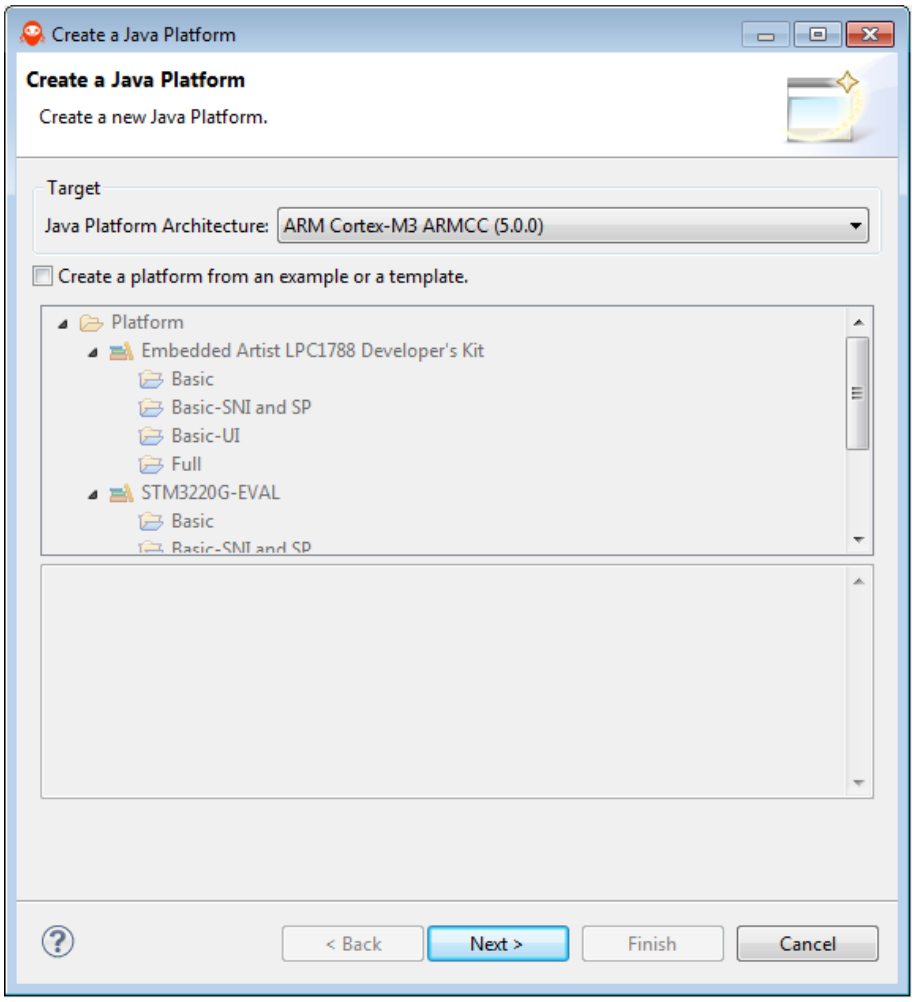

Figure 3.1. Java Platform Configuration (architecture selection)

Click on **Next.** JPF creation wizard continues asking to set a name for the future project and properties for the created JPF. Please fill the form like following:

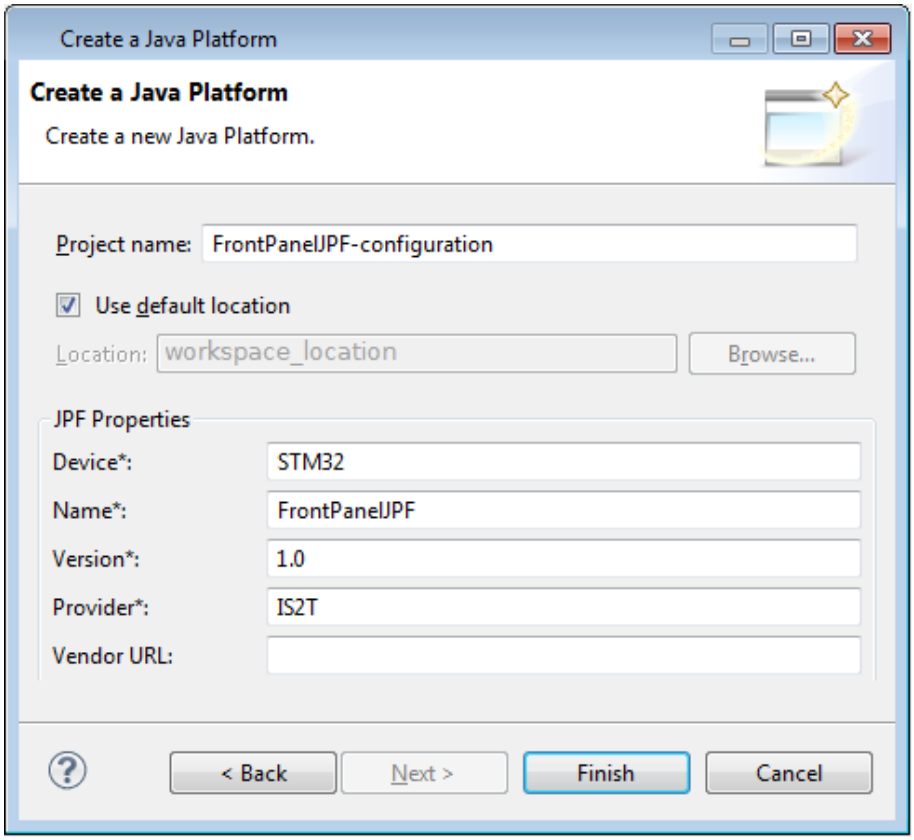

Figure 3.2. Java Platform Configuration (properties set)

The provider can be any name you wish. On pressing **Finish**, a new project is created containing the JPF configuration.

3.2 Configure and build the JPF

The next step consists to select necessary modules, configure them and then build the JPF regarding the configuration.

3.2.1 Select the necessary modules

In order to simulate an application that reacts from user inputs and displays information on a screen, the JPF needs to be populated by the **FrontPanel** module. This module depends on **MicroUI** and **Display** ones. These dependencies are listed by the details panel when you click on FrontPanel module.

FrontPanel depends on MicroUI in order to be able to get same `EventGenerators` indexes than the ones used by embedded platform (cf. TLT-0626-AN-MICROEJ_MicroUIButtons). The Section 3.3 will explain how to use them.

FrontPanel depends on Display in order to simulate the bits per pixels value used by the device screen. Display module configuration needs also some other properties but they are not relevant with simulation.

The Content tab of the **FrontPanelJPF.platform** file editor allows you to select the modules ; check the required modules like on following figure :

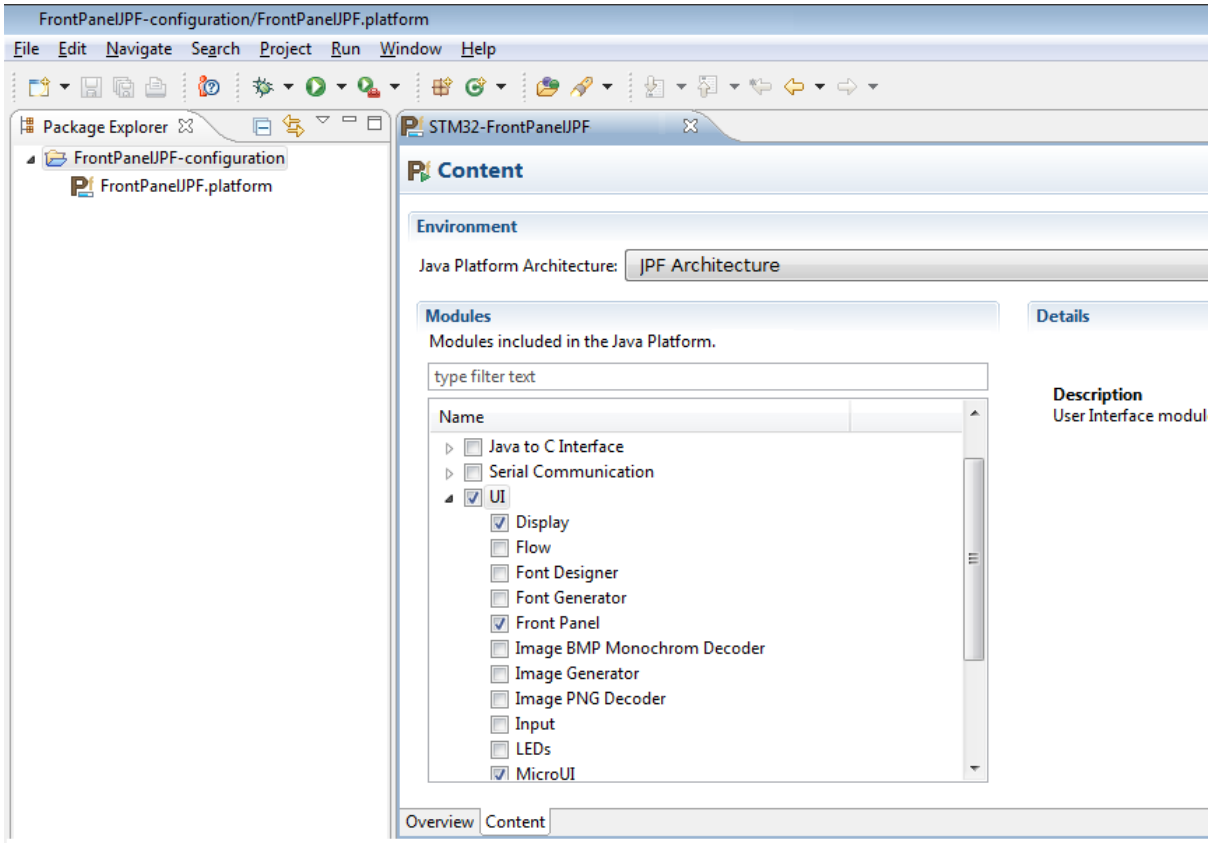

Figure 3.3. Java Platform Configuration (modules selection)

3.2.2 Configure the modules

For each module, JPF configuration editor shows module's details like file for module configuration. Thus, and with the help of User's Manuals, it is possible to configure wanted modules. The following explains how to configure selected modules to be able to execute an application that handles `Buttons` and `Pointer` events and displays information on a screen :

- FrontPanel : it requires "frontpanel/frontpanel.properties" file. A folder named "frontpanel" needs to be created on JPF Configuration project, on which we have to create the "frontpanel.properties" file. This file is provided by the application note ; it defines the mandatory FrontPanel project name. This project doesn't exist for now, it will be created during the JPF building.

- MicroUI : it requires "microui/microui.xml" file. To understand how to populate this XML file, please refer to ARM Cortex-Mx - UI User Manual at chapter "Static Initialization". For the application note, this file is provided. It defines two MicroUI elements : a `Display` and two `EventGenerators` (one for buttons and the other for touch screen).

- Display : it requires "display/display.properties". Please refer to ARM Cortex-Mx - UI User Manual at chapter "Display" to understand provided property file. This latter defines the display module as the default one. For this kind of display module, we have to configure the others properties (bpp, layout and mode). For the application note example, we will simulate a display of 16 bpp.

The JPF Configuration project content should now look like the following figure :

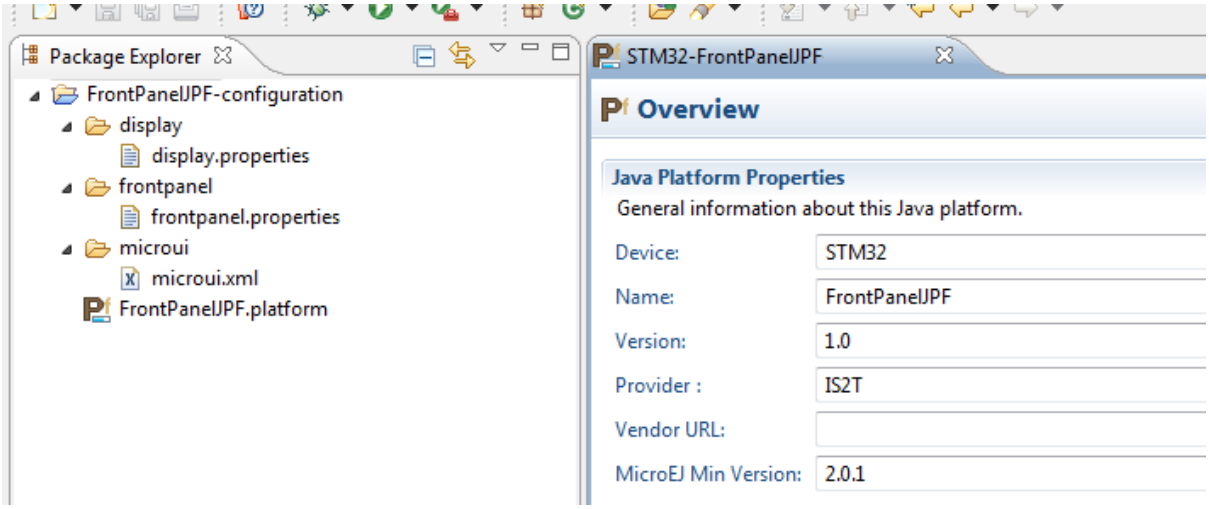

Figure 3.4. Java Platform Configuration (project content)

3.2.3 Build the JPF

The **FrontPanelJPF.platform** file editor provides a facility to build the JPF thanks to the **Build Platform** link on the Overview tab. By clicking on it, some projects are generated within the workspace after a process time. Are generated :

- a JPF project, notable by a JFP icon, which name is the concatenation of the JPF name and its version. It contains JPF source files.
- A FrontPanel project, notable by a FP icon, which name has been defined on properties file. A properties file named `microui.properties` is generated by JPF builder from MicroUI configuration. It contains the `EventGenerators` names and internal indexes.

The next step consists to define and implement the front panel content. That is the subject of the next section.

3.3 Define the Front Panel content and behavior

In this section we will describe the use of the Front Panel Designer tool. For further discussion about the Front Panel Designer, see ARM Cortex-Mx - UI User Manual.

3.3.1 Add Resources

To create a Front Panel Mock, you need to provide an image to use as the visual representation of the hardware. We have provided an image `stm32_skin.png` which you can use. Copy this into the resources folder of the front panel project.

In order to animate button presses, you need to provide an image of a button when pressed and when not pressed. We have provided images `Button_down.png` and `Button_up.png` which you can use. Copy these into the resources folder too.

3.3.2 Add a Button Listener

In order to simulate a hardware button press, the Front Panel Mock needs to know how to handle button input events. For the purposes of this Application Note, we will use the simplest possible event handling. Create a class in the `src` folder of the `FrontPanelJPF-fp` project, by selecting the `src` folder, and then selecting **File** → **New** → **Class**. Enter the package name `com.is2t.example` and class name `MyButtonListener`. Then click the **Add...** button and select the `com.is2t.microej.frontpanel.input.listener.PushButtonListener` interface.

The implementation of `MyButtonListener` needs to do the same as whatever the hardware button will do. In this case, we will make it send button pressed and released events :

```
package com.is2t.example;

import com.is2t.microej.frontpanel.input.generator.EventGenerator;
import com.is2t.microej.frontpanel.input.listener.PushButtonListener;
import com.is2t.microej.microui.Constants;

public class MyButtonListener implements PushButtonListener {

 /*
  * Name of buttons EventGenerator
  */
 private String eventName = "HWBUTTONS";

 @Override
 public void press(int buttonId) {
  EventGenerator.sendButtonPressedEvent(eventName, buttonId);
 }

 @Override
 public void release(int buttonId) {
  EventGenerator.sendButtonReleasedEvent(eventName, buttonId);
 }

}
```

The interface `PushButtonListener` defines two methods that must be implemented that are called when the virtual button displayed on the Front Panel Mock is pressed and released. The implementation above sends events to the `Buttons` event generator by making calls to the static methods of the `EventGenerator` class. The methods provided by this class mirror exactly the functions provided by the C equivalent, as defined in the `LLINPUT` API (defined in the `LLINPUT.h` file).

The first parameter of the two `EventGenerator` methods called is the ID of the event generator that should process these events, which in this case is the `Buttons` event generator called `HWBUTTONS`. The Constants interface was generated by the JPF builder.

This code is available in the App Note in the file `MyButtonListener.java`.

3.3.3 Add a Pointer Listener

Like the adding of Button Listener, it is necessary to create a Pointer Listener to simulate a touch hardware press. Create a class named `MyPointerListener` near `MyButtonListener`, which implements the `com.is2t.microej.frontpanel.input.listener.PointerListener` interface.

The implementation of `MyPointerListener` needs to do the same as whatever the hardware touch will do. In this case, we will make it send pointer pressed, moved and released events :

```
package com.is2t.example;

import com.is2t.microej.frontpanel.input.generator.EventGenerator;
import com.is2t.microej.frontpanel.input.listener.PointerListener;
import com.is2t.microej.microui.Constants;

public class MyPointerListener implements PointerListener {

 /*
  * Name of touch EventGenerator
  */
 private String eventName = "TOUCH";

 @Override
 public void move(int x, int y) {
  EventGenerator.sendPointerMovedEvent(eventName, x, y, true);
 }

 @Override
 public void press(int x, int y, int pointerId) {
  EventGenerator.sendPointerPressedEvent(eventName, pointerId, x, y, true);
 }

 @Override
 public void release(int x, int y, int pointerId) {
  EventGenerator.sendPointerReleasedEvent(eventName, pointerId);
 }

}
```

This code is available in the App Note in the file `MyPointerListener.java`.

3.3.4 Add a Display Extension

In order to simulate a hardware display, the Front Panel Mock needs to know how to convert the MicroUI representation of colors to and from the display representation of colors. These conversions are coded in a display extension.

Create a class in the `src` folder of the `FrontPanelJPF-fp` project by selecting the `src` folder and then selecting **File** → **New** → **Class**. Enter the package name `com.is2t.example` and class name `MyDisplayExtension`. Then specify the superclass : `com.is2t.microej.frontpanel.display.GenericDisplayExtension`.

The display extension needs to override 6 methods. The implementation of these methods will be discussed one at a time :

```
@Override
public boolean isColor() {
 return true;
}
```

It is a color display.

```
/**
 * rgb color: xxxx xxxx RRRR Rxxx GGGG GGxx BBBB Bxxx
 * lcd color:                   RRRR RGGG GGGB BBBB
 * we keep 5 msbits R & B and 6 mbmits for G
 */
@Override
public int convertRGBColorToDisplayColor(int rgbColor) {
 /* 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);
}
```

The Front Panel Mock needs to know how to convert colors from the MicroUI representation to the display representation. The equivalent function is shown in the *TLT-0627-AN-MICROEJ_MicroUIDisplay* Application Note in the implementation of `display.c` for adding support for a display on an actual target.

```
/**
 * lcd color:                   RRRR RGGG GGGB BBBB
 * rgb color: 0000 0000 RRRR R000 GGGG GG00 BBBB B000
 * we keep 5 msbits R & B and 6 mbmits for G
 */
@Override
public int convertDisplayColorToRGBColor(int 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);
}
```

Similarly to `convertRGBColorToDisplayColor`, the Front Panel Mock needs to know how to convert colors from the display representation to the MicroUI representation. The equivalent function is shown in the *TLT-0627-AN-MICROEJ_MicroUIDisplay* Application Note in the implementation of `display.c`.

This code is available in the App Note in the file `MyDisplayExtension.java`.

3.3.5 Configure fp File

The Front Panel Mock is configured using a .fp file. Create a `device.fp` file within the `definitions` folder of the `FrontPanelJPF-fp` project. Right-click and select **Open With → XML Editor**.

Then specify the skin – this is the image that is used to visually represent the hardware. In this case, we have provided a photograph of part of the STM3220G-EVAL board which was previously copied to the `resources` folder. To do so, copy the following code :

```
<?xml version="1.0"?>
<frontpanel
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://xml.is2t.com/ns/1.0/frontpanel"
 xsi:schemaLocation="http://xml.is2t.com/ns/1.0/frontpanel .fp1.0.xsd">
 <description file="widgets.desc"/>
 <device name="STM3220G-EVAL" skin="stm32_skin.png">
  <body />
 </device>
</frontpanel>
```

Save the file and open the Front Panel Preview view, if it isn't already open, by selecting **Window** → **Show View** → **Other...** → **MicroEJ** → **Front Panel Preview**. Your MicroEJ workbench should now look something like this:

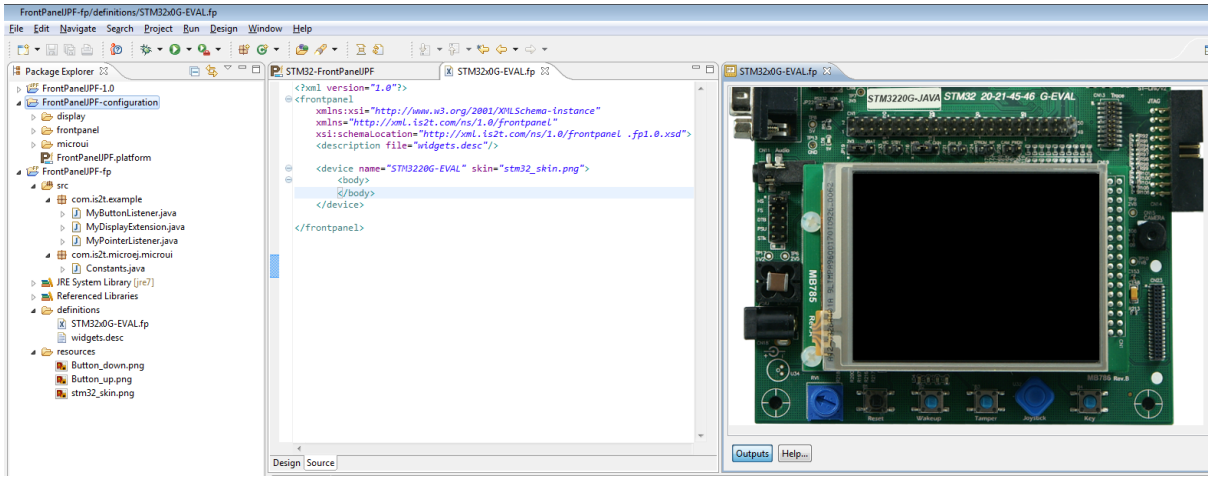

Figure 3.5. MicroEJ workbench

Of course, the mock won't respond to input events yet. We need to configure the front panel to identify where the hardware button is that we want to simulate.

Simulated hardware elements are defined within the `<body>` tag of the `.fp` file. We will add a `push`, a `pointer` and a `pixelatedDisplay` element to represent device hardwares. To know how define these elements, you can refer to the `widgets.desc` file on the `definitions` folder.

- `push` tag definition defines which images represent button pressed and released states. The `Button_up.png` image is a photograph of a hardware button when not pressed, and `Button_down.png` is a version of that image made to look like the button is down. They have to be precisely located to appear in the correct place over the hardware button on the `stm32_skin.png` image. In this case, we have located this button in the “wakeup” button location. The button ID is the value passed to the listener methods. Each button must have a unique ID.

- `pointer` tag defines the active area, represented by its top-left corner location on skin, its width and height. Like `push` element, each touch must have a unique ID.

- `pixelatedDisplay` tag defines the simulated screen, also represented by its top-left corner location on skin, its width and height.

The `device.fp` file should be like the one provided by the Application Note. You can now test the input elements by clicking on them in the Front Panel Preview using the mouse. The FrontPanelPreviewConsole should show some traces. Output elements are visible by clicking on FrontPanel Preview **Outputs** button.

3.3.6 Export the Front Panel Mock

In order to be available in the JPF, the Front Panel Mock must now be exported to it. Select the `FrontPanelJPF-fp` project, right click and select **Export...** → **MicroEJ** → **Front Panel**. Press **Next** followed by **Finish** and the Front Panel Mock will be exported into the JPF.

3.4 Import the Java application that will test the JPF

Import the Java Project "FrontPanelApp" delivered with the application note. This operation can be done using : **File** → **Import...** → **Existing Projects into Workspace** → **Select archive file**. The project is copied within the workspace.

The class `DisplayAndInputsApp` contains the main method. Execution begins by checking if platform has well been configured ; it stops if there is no `EventGenerator` of type `Buttons` and if no `Display` is reachable. Once execution has passed these tests, the application implements a MVC pattern :

- `MyModel` represents the Model and contains a boolean value.
- `MyEventListener` plays the Controller role. It is notified of input events, and in case of `Buttons` ones, it modified the Model switching the state of its boolean.
- `MyView` class is the View part of the design pattern. Associated to a Model, a View is requested by MicroUI to be repainted each time the Model changes.

3.5 Run the application in the Simulator

The application can now be executed using the simulator. Select the class `DisplayAndInputsApp`, right-click and select **Run As** → **MicroEJ Application**. A dialog appears to choose the JPF that will execute the application ; select the **FrontPanelJPF** one.

When the application starts, it opens the Front Panel Mock into a workstation frame. Press any buttons or touch the screen of the Front Panel Mock by using the mouse ; the application will react to the press and release events by refreshing the screen and outputting to the console :

```
button was pressed
button was released
```

4 Document History

Date	Revision	Description
February 19th 2013	A	First release
June 20th, 2013	B	Removed unnecessary fragment install steps
November 12th 2013	C	Product version 2.0.0 compatibility
July 4th 2014	D	Product version 3.0.0 compatibility
October 16th 2014	E	Product version 3.1.0 compatibility

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