Application Note:
TLT-0626-AN-MICROEJ-MicroUIButtons

Adding Support For Inputs

In relation to: MICROEJ products

Features

This Application Note explains how to add support for inputs to a Java Platform using 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 adding support for inputs to a Java Platform. It explains in detail all the steps required to enhance the platform and test it with a simple Java application that outputs text via the board's serial port when a hardware button is pressed.

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 input devices to a MicroEJ® Java Platform.

1.2 Scope

This Application Note describes the steps required to add support for hardware button inputs to an existing MicroEJ Java Platform that was built for an STM3220G-EVAL or STM3240G-EVAL board. Although this Application Note focuses on buttons, the same principles apply to other input devices associated with user interfaces, such as joysticks, touch screens and screen pointers.

1.3 Prerequisites

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

2 MicroUI input: principles of operation

MicroUI™ is the MicroEJ component that supports user-interface input and output devices. In this Application Note we examine how input actions, such as a button press, are communicated to a MicroEJ Java application. For full details about the operation of MicroUI please consult the *UI-PACK Reference Manual*[1].

2.1 Event flow

The path taken by an event is shown in the diagram below.

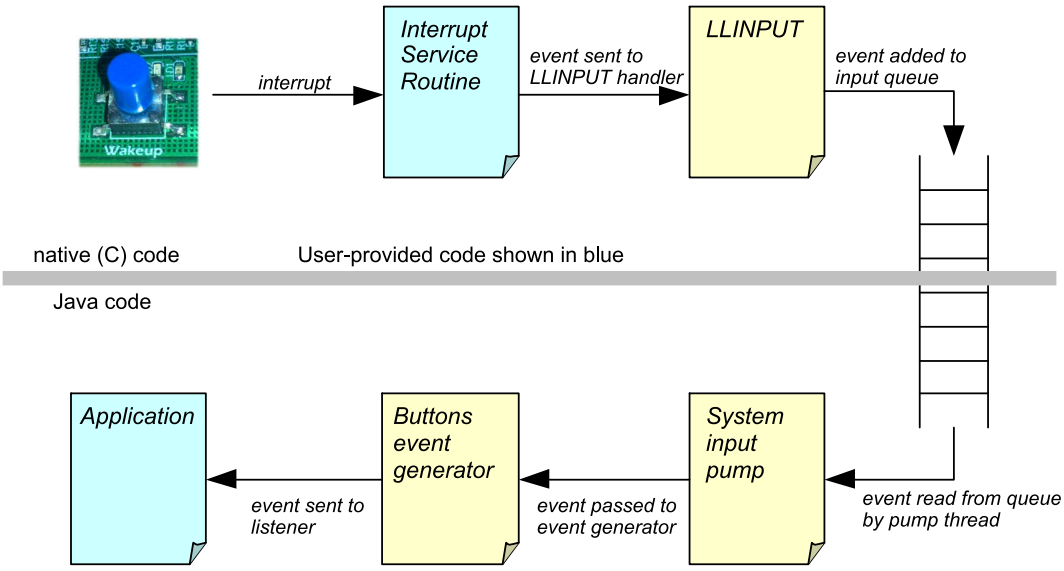

Figure 2.1. Event Flow

When the user presses a hardware button an interrupt is generated, which is handled by a interrupt service routine (ISR) that forms part of the user-supplied buttons driver. The ISR notifies MicroUI that the button has been pressed by calling a function in the `LLINPUT` ("low-level input") API and passing to it an integer that describes the event. In this case the event must specify:

- that the event is related to a button
- which button has been pressed
- that the button has been pressed (as opposed to released).

The event is added to the input queue. The fixed size of this queue is specified when the application is built.

A Java thread managed by MicroUI, called the *System Input Pump*, executes a loop that reads events from the input queue and passes them to the appropriate event generator. It determines the appropriate event generator by examining the contents of the event.

An event generator is a Java class that understands how to handle events sent by a particular kind of input device. In this case, the event goes to the `Buttons` event generator.

Each event generator can be associated with a listener object, which is notified of events as they arrive. In the example described in this Application Note, the Java application is the listener.

2.2 Event format

Although an event is always represented by a 32-bit integer, the format of events sent by drivers is not the same as the format of events sent to Java listeners. Events sent by drivers have this general format:

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

31 24	23 0
Generator ID	Event data

MicroUI has several built-in event generator classes, of which `Buttons` is one. The `LLINPUT` implementation is aware of these built-in event generators, and formats the data part of the event to match with their expectations when it is asked to add an event to the queue. The exact format used is proprietary [2].

When the `Buttons` event generator notifies its listener, it uses the following event format, as defined in the MicroUI Specification (ESR-SPE-002-MicroUI):

31 24	23 16	15 8	7 0
Event type ID	Generator ID	Action	Button number

The event type ID for button-related events is the value 1.

[2]Users can create their own event generators, and define a suitable format for the data part of the event.

3 An outline of the required steps

To add support for button inputs:

1. Enhance the JPF

   MicroUI support is added to the JPF, and the required event generators configured.

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 binary file

   The Java project is built in the MicroEJ environment, targeting the embedded JPF (“embJPF”), 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

   The C project is built and deployed to the target board using the ST-LINK connection.

4 The steps in detail

4.1 Enhance the JPF

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 button inputs, it is necessary to install and configure two modules: `microui` and `input`. Microui module is a Java library that exposes to application features in order to manipulate input devices (ex: `Button.class`). Input module is a native library that provides facilities to create event from input devices drivers. To select them,

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

4.1.3 Configure selected modules

The next step is to configure these modules.

- Input module can be configured using an `input/input.properties` file (cf module details).

  Create an `input` folder on the JPF configuration project and copy the `input.properties` file provided by the Application Note on it. This configuration file tells builder to deploy the "default" input module implementation.

  The module configuration is marked as optional ; indeed, without configuration, "default" input module implementation is deployed. So, we could not configure this module and result should be the same.

- 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>
 <eventgenerators>
  <buttons name="HWBUTTONS"/>
 </eventgenerators>
</microui>
```

  This specifies that the MicroUI configuration will comprise a single event generator, whose type will be `Buttons` and whose name will be `HWBUTTONS`. A `Buttons` event generator can handle many buttons, but this example supports only a single button.

  The Build Platform task will generate some output files. In particular, the file `xxx-jpf/source/include/microui_constants.h` contains constants that will be used by the C code. The line:

```
#define MICROUI_EVENTGEN_HWBUTTONS 0
```

indicates that the `Buttons` event generator (called `HWBUTTONS`, as specified in the `microui.xml` file) has been allocated an ID of 0.

4.1.4 Update the JPF

In order to enhance the JPF with the display and MicroUI modules, 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. Import it using: **File** → **Import...** → **General** → **Existing Projects into Workspace**, press **Next**, select the `MicroUIButtonsApp.zip` archive file, press **Next**.

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

```
package com.is2t.example;

import ej.microui.EventGenerator;
import ej.microui.Listener;
import ej.microui.io.Buttons;

public class MicroUIButtons {
    public static void main(String[] args) throws Exception {

        EventGenerator[] eventGenerators = EventGenerator.get(Buttons.class);  ❶
        if (eventGenerators.length != 1) {
            System.out.println("Only expected one event generator but got " +
                    eventGenerators.length);
            return;
        }
        eventGenerators[0].setListener(new Listener() {  ❷
            public void performAction(int event) {
                int buttonID = Buttons.getButtonID(event);  ❸
                if (buttonID != 0) {
                    System.out.println("Unexpected event for button " + buttonID);
                    return;
                }
                if (Buttons.getAction(event) == Buttons.PRESSED) {  ❹
                    System.out.println("Wakeup button was pressed");
                } else if (Buttons.getAction(event) == Buttons.RELEASED) {
                    System.out.println("Wakeup button was released");
                }
            }

            public void performAction(int value, Object object) {
                // not used
            }

            public void performAction() {
                // not used
            });
    }
}
```

The key lines in the application are marked with numbers:

❶ The static method get is called to get references to all the event generators of a specified type – `Buttons`, in this case. There should be only one such event generator, because the MicroUI configuration file specified only one.

❷ The listener for the event generator is set to be the instance of the anonymous class created on this line. The event generator will notify this object of all events by calling its `performAction(int)` method.

❸ The event carries with it the ID of the button. The static `getButtonID(int)` method extracts the ID from the event. The application only expects events for the button whose ID is 0.
❹ The event also carries with it the action performed. The static `getAction(int)` method extracts the action from the event.

The button ID and action are specified by the buttons driver when the event is created and added to the input queue.

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 `MicroUIButtons` 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:

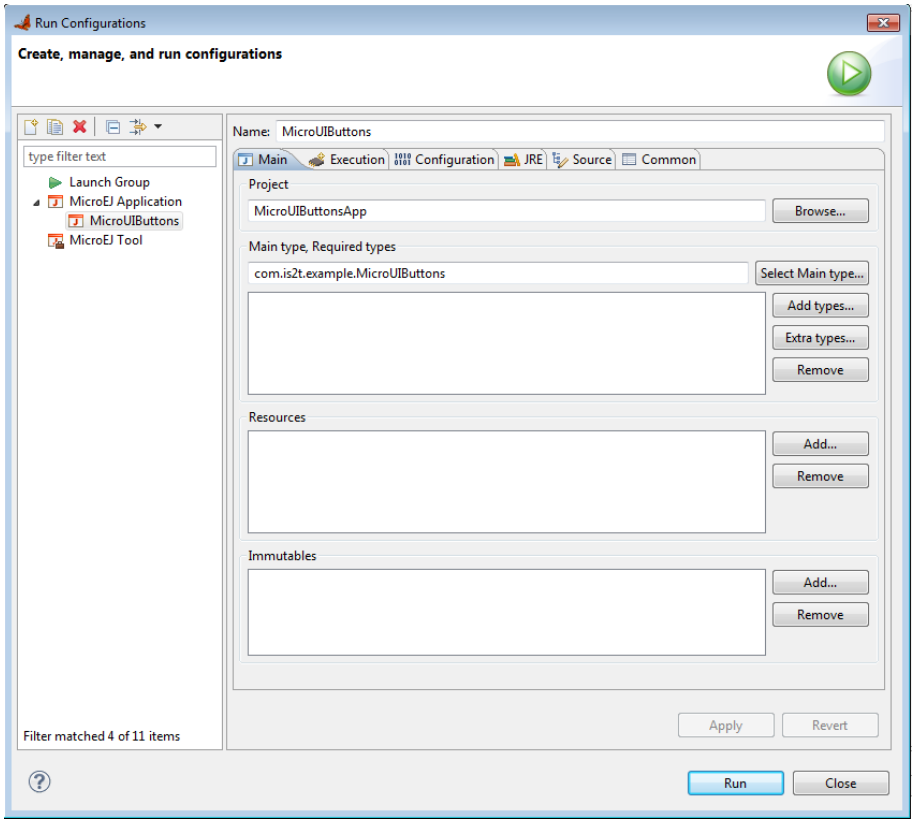

Figure 4.1. Launch Configuration - Main Tab

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

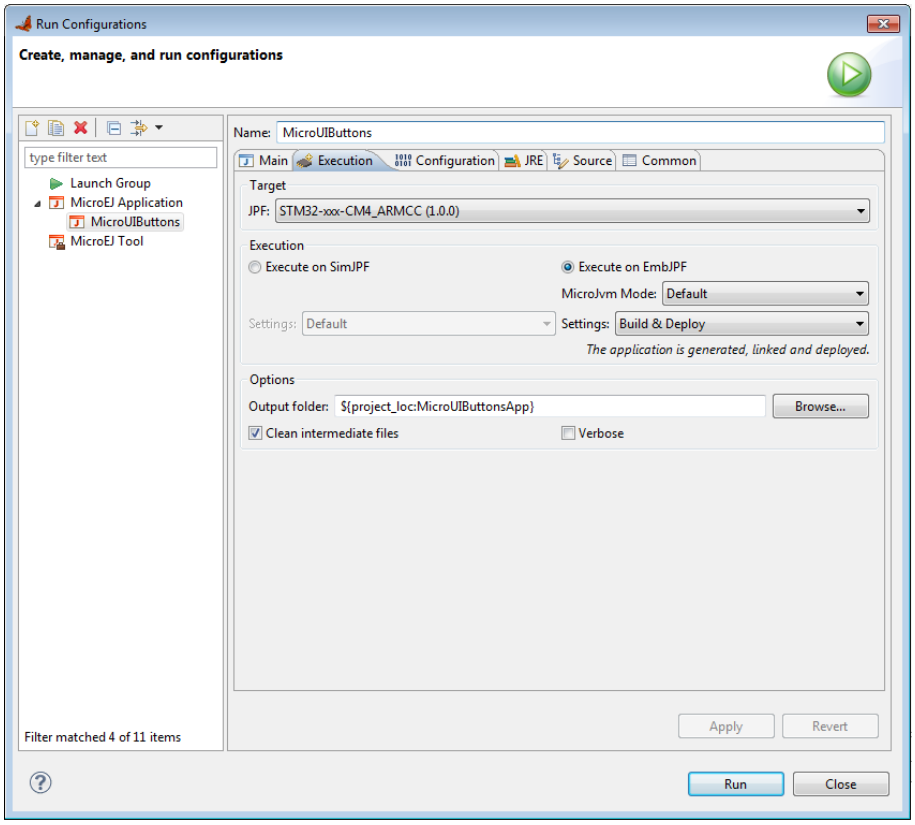

Figure 4.2. Launch Configuration - Execution Tab

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

4.4 Write the required C files

The JPF 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 `LLINPUT` API, as defined in the `LLINPUT_impl.h` include file. This API defines a number of functions that are called by the MicroUI input stack.
- Detects button presses and requests that events be added to the input queue, using functions defined in the `LLINPUT.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-MICROEJ-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, in the extra-files folder, are:

- `LLINPUT.c` – implements the `LLINPUT` API.
- `buttons.c` – responsible for connecting the low-level driver to the input stack. Adds events to the input queue.
- `buttons.h` – header file for `buttons.c`

The contents of the first two of these files are examined in detail below.

Install the supplied files into a new C project:

1. Create a new folder: `STM32Fxxx_StdPeriph_Lib_Vxxx/Project/MicroUIButtons`
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 LLINPUT.c

The source file `LLINPUT.c` contains an implementation of the `LLINPUT` API, as defined in the `LLINPUT_impl.h` include file. Four functions are implemented, as shown below.

```
uint8_t LLINPUT_IMPL_initialize(void){
    buttonsInitialize();
    return MICROEJ_TRUE;
}

int32_t LLINPUT_IMPL_getInitialStateValue(int32_t stateMachinesID, int32_t stateID){
    // no state on this BSP
    return 0;
}

void LLINPUT_IMPL_enterCriticalSection() {
    OS_SUPPORT_disable_context_switching();
    disableButtonsInterrupts();
}

void LLINPUT_IMPL_leaveCriticalSection() {
    enableButtonsInterrupts();
    OS_SUPPORT_enable_context_switching();
}
```

The function `LLINPUT_IMPL_initialize` is called by the input stack when it starts up to request initialization of all input devices. In this case, only buttons need to be initialized, so the `buttonsInitialize` function, implemented in buttons.c, is called.

The function `LLINPUT_IMPL_getInitialStateValue` is concerned with support for state machines. We have not declared any state machines in the MicroUI configuration (in `config.microui`), so this function will not be called – however an implementation is still required.

The other two functions provide support for critical sections. Whenever the input stack wants to ensure that it has exclusive access to the stack and its data structures it calls `LLINPUT_IMPL_enterCriticalSection`. When it no longer requires exclusive access it calls `LLINPUT_IMPL_leaveCriticalSection`.

Calls to these functions are always strictly paired and not nested. There will never be two calls to `LLINPUT_IMPL_enterCriticalSection` without an interleaving call to `LLINPUT_IMPL_leaveCriticalSection`.

In the example created in this Application Note the only asynchronous access to the input stack comes from the interrupt service routine (ISR) that runs when a button is pressed or released. So for this example a satisfactory implementation of `LLINPUT_IMPL_enterCriticalSection` would be to disable the button interrupt. However, the implementation shown here copes with the more general case where asynchronous access to the input stack can come from an ISR or from another operating system task. The call to `OS_SUPPORT_disable_context_switching` ensures that the current task will not be preempted by another task.

4.4.2 buttons.c

The source file `buttons.c` contains functions that connect the low-level driver to the input stack.

Most of the functions in `buttons.c` require no explanation, but two are worthy of some discussion.

```
void EXTI0_IRQHandler(void) {
    enterInterrupt();
    BUTTONS_interrupt(BUTTON_WAKEUP);
    leaveInterrupt();
}
```

The function `EXTI0_IRQHandler` is called when an interrupt occurs on the interrupt line to which the **Wakeup** button is connected. The function uses the `enterInterrupt` and `leaveInterrupt` functions, implemented in `interrupts.c`, to keep track of the current execution state. It checks that the button really does have a pending interrupt, and if it does it calls the `BUTTONS_event` function, implemented in the same file and shown below.

```
// table that records the last state of the button
static uint8_t BUTTON_PRESSED[BUTTONn] = {MICROEJ_FALSE, MICROEJ_FALSE,
 MICROEJ_FALSE};

static void BUTTONS_event(Button_TypeDef Button) {
 uint8_t pin_is_one;
 uint8_t button_pressed;

 pin_is_one = GPIO_ReadInputDataBit(BUTTON_PORT[Button], BUTTON_PIN[Button]) ==
 Bit_SET;
 button_pressed = (pin_is_one && !BUTTON_REVERSE[Button]) || (!pin_is_one &&
 BUTTON_REVERSE[Button]);

 if (button_pressed) {
  if (!BUTTON_PRESSED[Button]) {
   // button was previously released, so this is a press event
   if (LLINPUT_sendButtonPressedEvent(MICROUI_EVENTGEN_HWBUTTONS, Button)) { ❶
    // the event has been queued: we can store the new button state
    BUTTON_PRESSED[Button] = MICROEJ_TRUE;
   } else {
    // the event has not been queued. We must not change the button state to prevent
    // sending a future release event when the press event has never been sent!
   }
  }
 } else {
  // button is released
  if (BUTTON_PRESSED[Button]) {
   // button was previously pressed, so this is a release event
   if (LLINPUT_sendButtonReleasedEvent(MICROUI_EVENTGEN_HWBUTTONS, Button)) { ❷
    // the event has been queued: we can store the new button state
    BUTTON_PRESSED[Button] = MICROEJ_FALSE;
   } else {
    // the event has not been queued. We must not change the button state to prevent
    // sending a future press event when the release event has never been sent!
   }
  }
 }
}
```

It is this function that requests the creation of events. It remembers the previous state of the button so that it can avoid sending confusing events if the previous attempt to send the event failed (as it will do if the input queue becomes full).

The calls that send the events are marked. Call (1) sends the “pressed” event; call (2) sends the “released” event. Note the parameters to these calls:

- The first parameter is the ID of the event generator that should handle the event. The symbol `MICROUI_EVENTGEN_HWBUTTONS` was generated when the `config.microui` XML file was processed, as discussed earlier in this document. The `HWBUTTONS` part of the symbol name corresponds to the name given to the event generator in the `microui.xml` file.

- The second parameter is the ID of the button. These IDs start at 0, but do not have to correspond with the number used in the hardware layer. These calls could have used a different value for the **Wakeup** button, but, of course, whatever value is used must match with the expectations of the Java applications.

4.5 Configure the Keil µVision project

The Keil µVision project file `Project.uvproj` in the `MicroUIButtons/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:

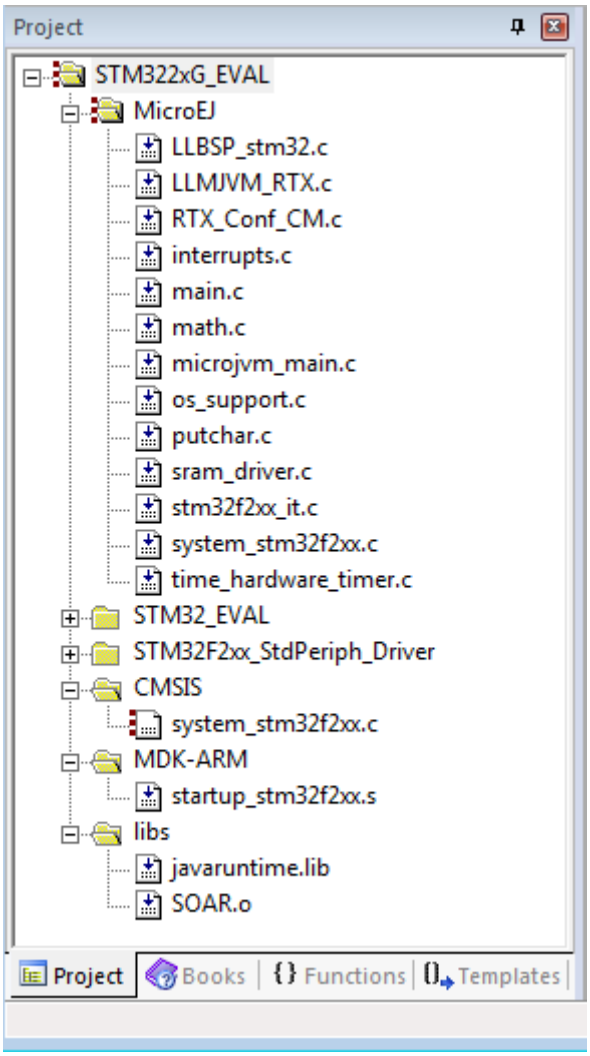

Figure 4.3. µVision Project Content Before Configuration

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

Several different configuration actions are required:

- Add the extra C files

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

- Remove the `FromScratch` application.

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

- And replace it by the right application.

  Right-click on the **MicroEJ** group and select **Add Files to Group 'MicroEJ'...**. Navigate to the `com.is2t.example.MicroUIButtons` folder of the `MicroUIButtonsApp` 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**.

The project should now look like this:

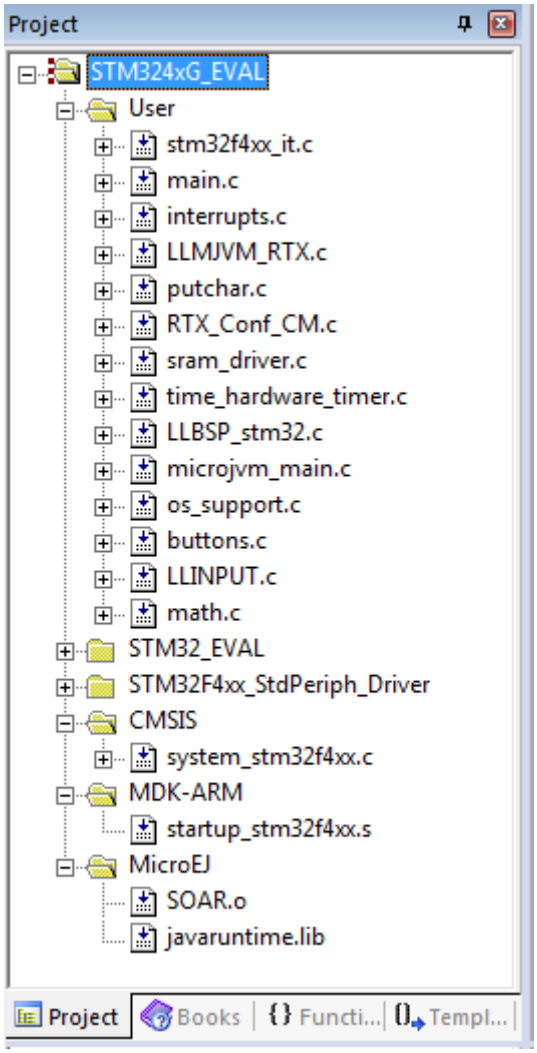

Figure 4.4. µVision Project Content After Configuration

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, press the **Wakeup** button, and the output will appear in the terminal emulator connected to the serial port.

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 05th, 2014	C	MicroEJ ARM Cortex-M (Keil MDK-ARM) 3.0.0 compatibility
October 02nd, 2014	D	MicroEJ 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