STM32Java GUI Design

Technical insights & benefits

Régis Latawiec, IS2T, regis.latawiec@is2t.com

Graphical User Interfaces (GUIs) have become more important in the embedded world because they have a direct influence on end-customers. However GUIs are not easy to design and develop because of the inherent complexity related to human factor understanding and the involvement of many different stakeholders during design phases. Complexity comes from high variability of the specifications during development and maintenance, intrinsic dynamic behavior and specific embedded constraints such as memory footprint, CPU resources and cross-compiling issues.

Object oriented technologies

As a matter of facts, Object Oriented Programming (OOP) languages were immediately adopted in the area of GUI design because GUIs mainly deal with objects that rapidly need to be organized into lousy coupled entities.

Many GUI design foundations come from Smalltalk as it is described in "*Design Patterns – Elements of Reusable Object Oriented Software*", Erich Gamma , Addison Wesley Longman Inc. GUI design principles should rely on the Model-View-Controller (MVC) triad because this is the best known solution to designing flexible and easy to maintain GUIs. The model holds data that is updated by Controllers upon user interactions with graphical objects (the views). Once the model has changed its data, it tells "listeners" that is has changed, so that these listeners can do what they believe is appropriate. Views are listeners, so the Views refresh their graphical appearance. All entities are loosely coupled, so new views – using the existing Model data – can be added without impacting the other parts of the system.

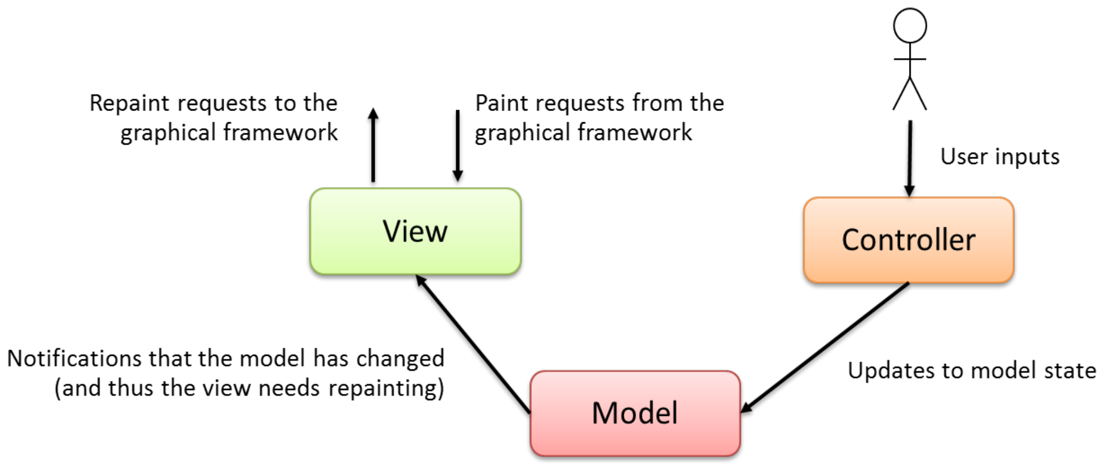
Repaint requests to the graphical framework
Paint requests from the graphical framework
User inputs
View
Controller
Notifications that the model has changed (and thus the view needs repainting)
Updates to model state
Model

Figure 1: MVC pattern

MVC intensively uses several Object Oriented Programming design patterns such as Observer, Composite, Strategy, Composite Method and Decorator; it can be designed in C++. However, C++ has many weaknesses that often prevent embedded developers to use it with significant gains compared to traditional C. Among these weaknesses, most common ones are a) fuzzy limit between structured C approach and OOP C++ approach; b) poor code density; c) still use pointers; d) no automatic memory management.

STM32Java key advantages

Virtualization

Java technology was invented in the 90's as an alternative technology easier to use compared to Smalltalk-80 and safer than C++. Java programming language therefore inherited its syntax from C/C++ that makes is so easy to learn for C developers and defined a safe execution environment that involves a well defined language semantic and a canonical execution runtime support thanks to a virtual processor featuring a safe instruction set architecture.

Designing embedded GUIs using Java technology is a perfect match because its Object Oriented characteristics allows designing architectures with loose coupling between entities. Java technology offers few other key features that off-load developers from software development complexity related to GUIs often considered as highly dynamic and event driven systems. The three most important aspects deals with automatic data memory management thanks to the virtual processor and its automatic garbage collector, runtime exception checks such as out of bound array accesses and hardware resource availability and portability at binary level of compiled Java code to capitalize software investments across a wide range of projects.

STM32Java software simulation

Software simulation has always been a difficult task because simulation can embrace several aspects such as cycle true execution, functional behavior and hardware in the loop testing.

If expensive simulation system may be able to bring all these features at the same time by mixing Verilog simulation models and software tool chain integration often boosted by hardware acceleration means, simple simulators often remain disappointing solutions because they lack of well defined simulation models.

The STM32Java technology is a new opportunity to redefine the goal of a simulation system. Since a Java platform is standardized by a well defined execution model and software APIs, this clearly separates the execution platform (hardware and virtual processor) from the application itself. Since the execution platform can be fully tested and characterized independently, simulation can only focus on the application itself.

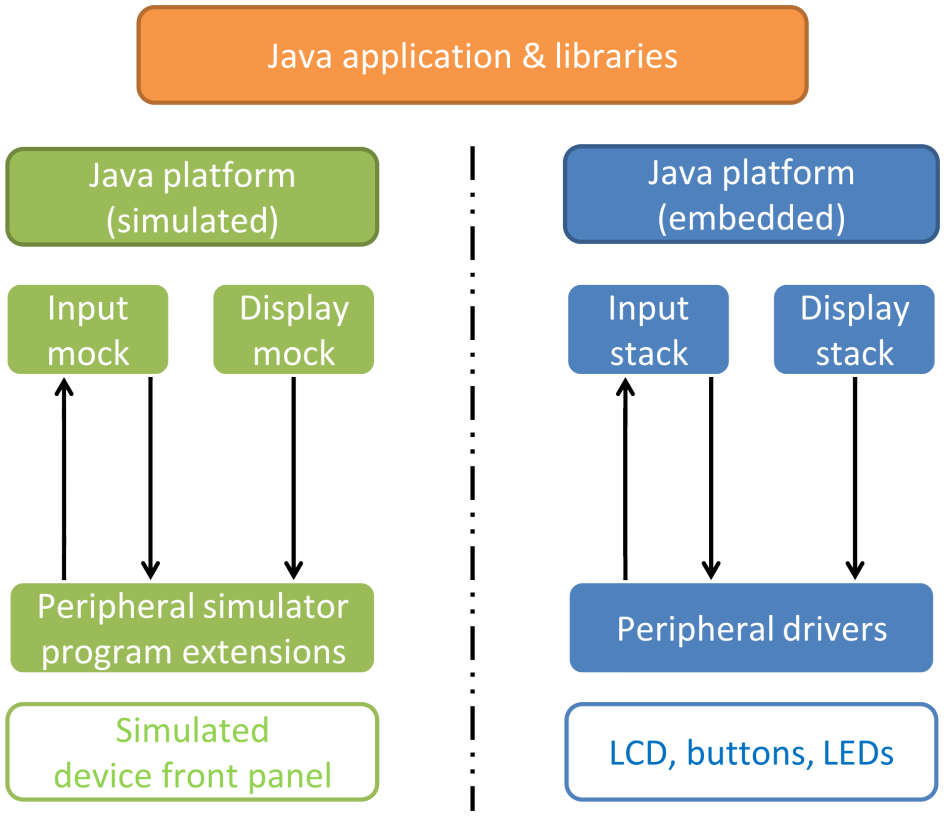
Java application & libraries
Java platform (simulated)
Input mock
Display mock
Peripheral simulator program extensions
Simulated device front panel
Java platform (embedded)
Input stack
Display stack
Peripheral drivers
LCD, buttons, LEDs

Figure 2: STM32Java Embedded & simulated Java platforms

From the Java virtual platform concept, it is easy and convenient to build a sibling embedded platform that runs on a workstation to offer simulation capability. The simulated platforms simulates properties such as embedded RTOS scheduling and provide mockups that extend the simulated platforms with specific capabilities not originally available from the workstation such as a device front panel (LCD display, buttons, LEDs) sensors, actuators, special communication protocols, etc.

Since the application binary code is executed by both platforms, the simulated platform becomes a powerful tool for debugging and testing the GUI application from the highest level (functional test coverage) to the lowest one (binary test coverage) and allows the analysis of the application memory footprint requirements during the development phase.

Adding a Graphical User Interface to a STM32 C application

To enhance an existing C application with a GUI part written in Java, three tasks need to be considered:

1) embed a STM32Java virtual machine into the exiting software system;

2) interface the graphical hardware resources to the graphical Java libraries
3) interface the native (C/asm) business code to the Java GUI part using the MVC principle.

Java integration with native code (C/asm) has to be very simple to guarantee developers with an easy and safe path to mix new Java code and their well known and stable native code. Existing drivers and application tasks must be accessible to the new Java code keeping in mind that call overheads, memory and CPU resource allocations need to remain efficient.

No RTOS - Basic cooperative integration

Although the use of a RTOS become more and more common in the STM32 arena, many applications still use a basic cooperative scheduling scheme.

Simple cooperative task organization often looks like a state machine execution process where each task is called within a defined sequence and has to be executed within a given time to preserve the system coherence. In such a case, the STM32Java virtual machine plays the role of the basic cooperative scheduling scheme. C native tasks are called as usual and hand back after a known period of time. Note that the Java world dedicated to the GUI part can leverage the possibilities offered by a multi-tasking virtual processor while keeping the rest of the legacy application scheduled as in the past before the addition of the GUI part.

RTOS integration

Adding a GUI to an existing application often requires the addition of a simple RTOS, usually pre-emptive one, because the application needs to be responsive upon user’s interactions, even if there are some background tasks to be processed.

STM32Java uses the “Green threads” integration while this is the most reliable and common way to add the virtual machine to an existing software system. The STM32Java platform defines a multi-threaded environment for the whole Java world which runs within one single RTOS task. The Java CPU consumption is fully controlled by configuring the RTOS task, allowing easy arbitration between the different RTOS tasks of the entire application. Indeed, the maximum amount of CPU power given to the Java world is fixed, whatever the number and/or the activities of the (Java) green threads.

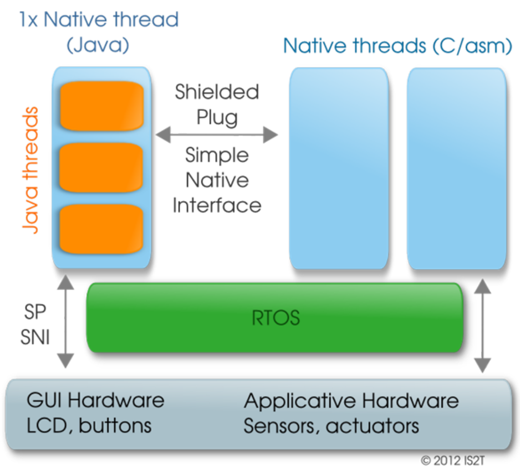

Figure 3: Green thread RTOS integration

Native Interface
STM32Java allows to interface Java to C or ASM in order to access drivers or legacy firmware: SNI, ESR012[1], Simple Native Interface. The SNI APIs are straightforward based on a simple name convention: they allow direct calls of C functions from Java and of Java methods from C. Arguments of types such as integer or float and can be passed to transfer data.

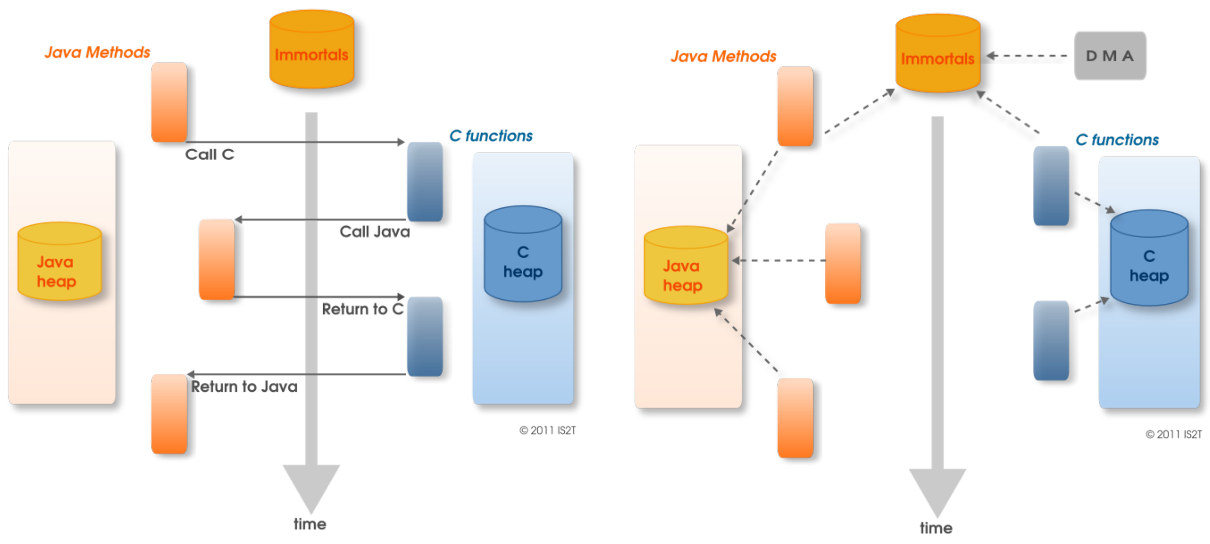

Figure 4: Calling C function from Java (& vice versa), and sharing "immortal" data.

1 www.e-s-r.net

Here is an example how Java code can call C code:

```
package com.corp.examples;
public class Hello{
public static void main(String[] args){
int i = printHelloNbTimes(3);
   }
   public static native int printHelloNbTimes(int times);
}
```

```
#include <sni.h>
#include <stdio.h>
jint Java_com_corp_examples_Hello_printHelloNbTimes(jint times){
    while (--times){
        printf("Hello world!\n") ;
    }
    return 0 ;
}
```

Figure 5: The “native” Java function `printHelloNbTimes` is written in C.

STM32Java - an Eclipse based IDE

STM32Java SDK is based on MicroEJ® SDK from IS2T and is built on the popular Eclipse™ IDE. It provides a complete software development environment for designing embedded Java applications: developers can prototype, develop and test their Java applications with simulated Java platforms before programming them to embedded devices with the embedded Java platform runtime.

Java embedded platform design for STM32F microcontrollers

Main criteria for choosing an embedded platform for designing GUIs on Cortex M series such as STM32F microcontrollers are:

a) low requirements in Flash and RAM consumption;

b) high speed execution engine;

c) seamless integration with legacy code (RTOS and firmware);

d) API and tools dedicated embedded GUI applications

MicroUI and MWT (ESR Libraries)

STM32Java SDK provides two graphical Java libraries: MicroUI® (Micro User Interface) and MWT (Micro Widget Toolkit). MicroUI deals with one or several character oriented displays and/or dot matrix LCDs (black & white, common devices such as pointers (touch, multi-touch, and mouse), buttons, LEDs and audio. MicroUI

CPU usage is highly optimized: display updates are clipped whenever possible, duplicated events are skipped and LCD refresh activity can be tuned to balance graphic activity against the overall application activity.

MWT provides a higher level framework than MicroUI to manage graphical widgets, containers, layouts, navigation, look & feel, etc. It provides the basis for the construction of specific GUI widget libraries.

Shielded PLug

The STM32Java JPF implements the Shielded Plug specification, which specifies well-defined segregation between producers and consumers of data. Moreover, producers and consumers do not need to be from the "same" world. Indeed, it provides a full segregation of the processes (producers versus consumers), which can be written in either C or Java. A typical use of a Shielded Plug is related to Graphical User Interface integration with legacy application logic. Both (i) application logic and (ii) the whole GUI can communicate only by exchanging data through a shielded plug.

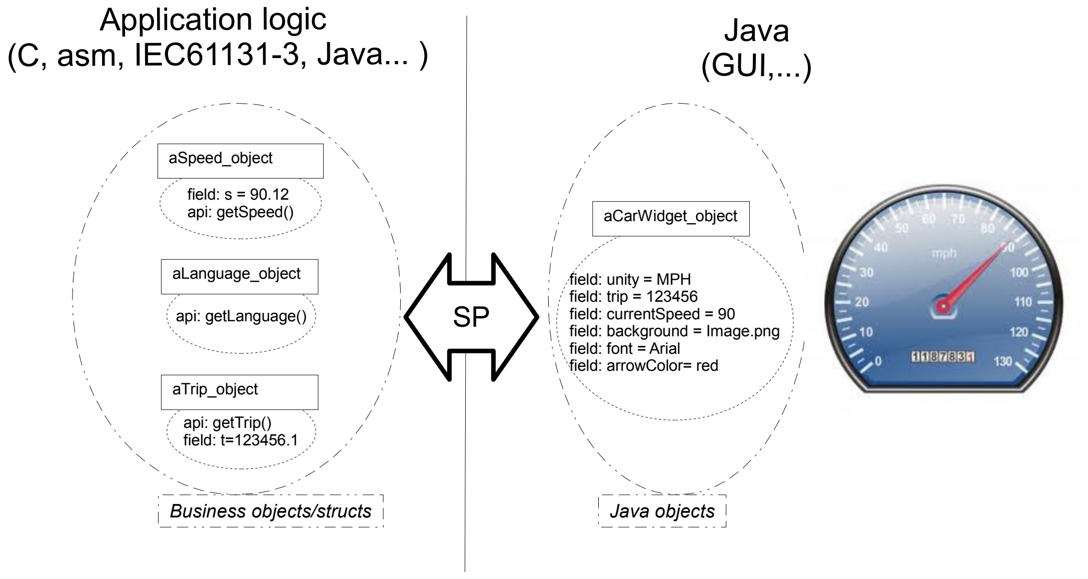

Figure 6: Shielded Plug : no-coupling between software components.

Specific tools for GUI application design

The Front Panel Designer offers a means to extend the Java platform simulator with custom mocks that simulate the appearance of a hardware device. The Front Panel mock can feature a LCD display including touch capabilities, LEDs, buttons, etc. It becomes easy to simulate the GUI application within its hardware environment and to check - early in the project specification phases - human factor issues, graphical rendering, etc.

The Storyboard Designer allows to quickly specify the possible human-machine interactions with the application. A graphical artist can design screens using bitmap graphical tools and can import them into the storyboard as a sequence of images. Then the storyboard can be built by defining interaction areas within screens (click, double click, swipe, etc.) and linked actions (jumps to another screen with possible effects).

The resulting storyboard can be played on the simulator (using a Front Panel for a realistic simulation) and can be also exported to the real device.

The Font Designer allows designing font characters in many ways: characters can be designed from scratch or can be imported from standard font files (bitmap, true-type). Once characters have be designed at pixel level, they can be gathered in scripts compliant to the Unicode standard and several utilities are provided to optimize character usage according to actual text strings that need to embedded in the application, hence optimizing the memory space usage allocated to the fonts.

Conclusion

STM32Java technology is known to be an appropriate solution for designing GUIs as it features both an Object Oriented language and a software processor that offers advanced mechanisms for dealing with highly variable software programs and event driven architectures.

SMT32Java platforms optimized for STM32J microcontrollers (see www.stm32java.com for full update of the J partNumbers) have low memory footprints and high speed execution thanks to strong architectural choices in the Java platform design. These platforms can provide simulation capabilities for prototyping and testing GUI designs and are easy to integrate to legacy C software (RTOS and libraries) to offer seamless migration to higher level languages for GUIs.

Providing a high level of portability across a wide range of hardware / software platforms, developers can easily capitalize on their investments and deploy their GUI software to virtually any device such as low-end devices (simple RTOS, Linux, etc...) and smart 3rd party devices (IOS, Android).