Application Note:
TLT-0709-AN-MICROEJ-DataSharing

Java/C Data Sharing Using SNI

In relation to: MICROEJ products

Features

This Application Note explains how to share data between Java and C tasks.

Description

This Application Note assumes the reader wishes to understand the steps involved in sharing data between Java and native processes, using SNI library and the Immortal heap.

Table of Contents

List of Figures

1 Introduction

1.1 Intended audience

The intended audience for this Application Note are developers who wish to share data between Java thread and C task.

1.2 Scope

This Application Note shows a simple implementation of data sharing through a buffer that can be accessed from both Java and C sides. Data sharing can be used to :

- send data from Java to C task, or vice versa (ex: usage of a communication stack),
- implement a ring buffer recording history of values (ex: sensor driver).

This application note deals with data sharing between Java and C tasks, using SNI library. Another library named ShieldedPlug provides also mechanisms to share data between two processes. Differences between those two libraries will not be discussed on this application note.

This Application Note mainly highlights data sharing on simulated platform. Indeed, memory sharing is more obvious using embedded platform since Immortal heap can be accessed on native side. On contrary, mocks don't share same memory space than application since they are executed on their own process (see reference manual about Simulation).

1.3 Prerequisites

This Application Note assumes that the reader has already created a Java Platform, as described in the "Building a platform from scratch" Application Note. All the prerequisites specified in that Application Note apply equally to this Note.

1.4 Files supplied with this Application Note

This application note is packaged with archives of Eclipse projects, which can be imported in the normal way. The application note contains :

- Archive file `DataSharingApp.zip`, that is a MicroEJ® Java application that demonstrates data sharing between processes.
- Archive file `DataSharing-mock.zip`, that is the simulated part of platform that shows mechanisms of data sharing.
- Folder `c-src` containing native implementation for embedded platform.

2 Design

2.1 How to share data ?

MicroEJ® brings Java technology into embedded devices. An application built on this technology sees only libraries contained into the platform that executes it. As platform doesn't manage every board specificities, sometimes you will need to create native API to expose features that will be executed on another language (most of time in C).

This application note explains how to share data between those two "worlds".

The solution is to create a memory space that can be accessed by Java and C tasks. For this, MicroEJ® provides, thanks to SNI library, mechanism to share reference of an array. This array can only be constituted by Java based types (byte, int, float, double). This API also limits reference sharing to Immortal arrays. Indeed, as Immortal objects are not processed by the garbage collector (see garbage collector as memory defragmentation), those objects keep same memory location along the execution. In that way, Immortal arrays can be shared between Java and C tasks.

2.2 Example design

The following sequence diagram explains application note example, highlighting the initialization sequence and execution steps :

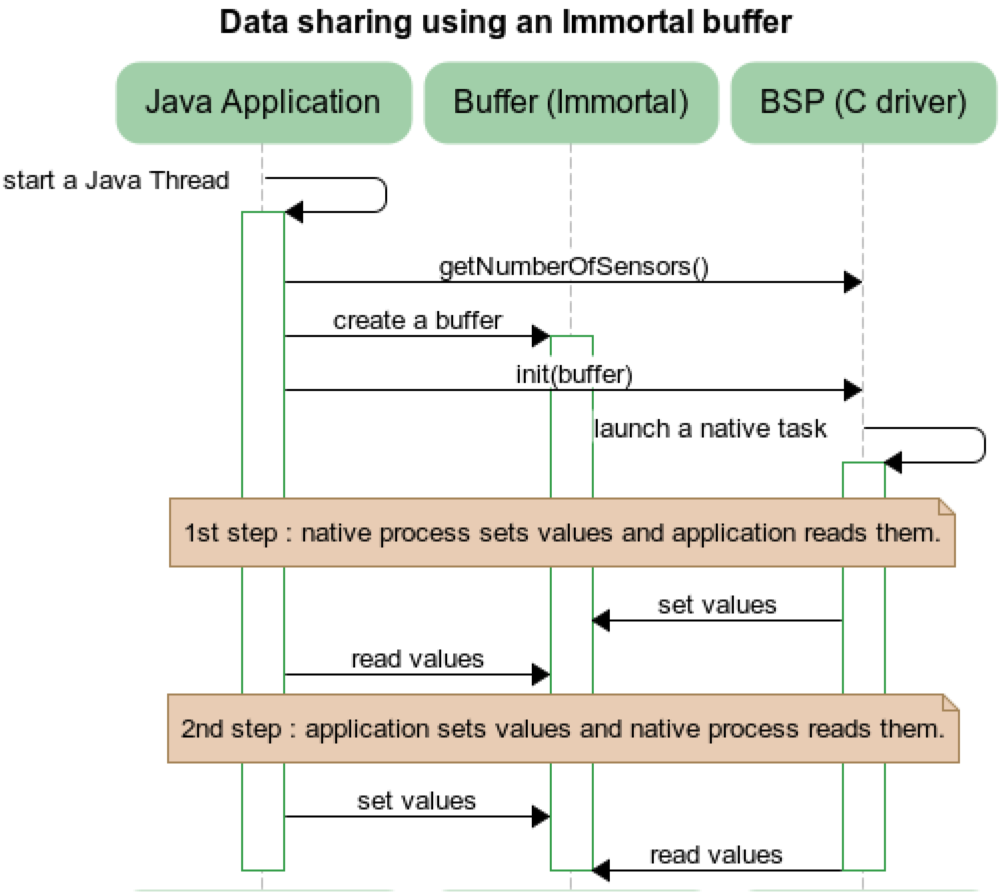

Figure 2.1. Example application sequence diagram

Data sharing needs an initialization phase. Example execution starts with the creation of an Immortal buffer. then the application shares buffer renferences with the platform by calling a native method with a buffer as parameter.

The platform stores the buffer reference and lets the hand at the application by creating a native task.

To finish the demonstration, this application note example divides its execution in 2 steps : first one is focused on a native C process writing data and a Java application reading them on application side ; second step is doing the opposite.

3 Implementation

This chapter explains data sharing implemented by application note example.

3.1 Application

Import project from `DataSharingApp.zip` into MicroEJ SDK. This project contains several classes :

- The `EntryPoint` class. It implements the main method by creating and launching a Java task.
- The `SensorManager` class. It shares a buffer with a native task.
- And the `NativeInterface` class. This one defines native API to communicate with feature implemented in C language.

This application can be executed on any platform as we provide embedded and simulated implementations of native methods.

Following illustration shows the application code :

```
    ...

    /**
     * Sensors values buffer.
     */
    private int[] sensorsValues;

    /**
     * Constructor.
     */
    public SensorManager() {
        nbSensors = NativesInterface.getNumberOfSensors();
        sensorsValues = new int[nbSensors];
        // Before sharing buffer, we have to set it as Immortal.
        sensorsValues = (int[]) Immortals.setImmortal(sensorsValues);
    }

    ...

        /*
     * (non-Javadoc)
     * @see java.lang.Runnable#run()
     */
    public void run() {
        System.out.println("--- 1st step : application reads buffer filled by a
native process ---");
        NativesInterface.init(sensorsValues);

    ...
```

Figure 3.1. Application code

The code shows how to set an array as Immortal (see last line of constructor method).

3.2 Embedded side

Folder `c-src` contains native implementation for embedded platform. This implementation uses the Keil RTX Kernel.

Following illustration shows the C implementation of `init()` native method :

```
    ...

int32_t* SENSORS_VALUES;

    ...

void Java_com_is2t_appnote_NativesInterface_init(int32_t* buffer)
{
    // Store buffer address
    SENSORS_VALUES = buffer;
    // Create and launch a task
    os_tsk_create_user(sensorsTask, 10, &sensorsTaskStack, sizeof(sensorsTaskStack));
}
```

Figure 3.2. Buffer reference sharing (C side)

We can see that `int[]` array parameter in Java side becomes a memory pointer (`int*`) on the C side. According to this principle, array values can be read / modified using this pointer.

To be able to execute the provided application on your board, we will need to add `sensorsmanager.c` as source file set of the C project. Without this, 3rdd party linker will failed to find `init()` and `getNumberOfSensors()` symbols.

3.3 Simulated side

As previously said on this document, the simulated part of data sharing uses particular mechanisms to reproduce the behavior on board. On board execution shares the same memory space for both Java and C tasks. Thus, memory references in Java can be read from C (and inversely). On contrary, a simulated native task does not share same memory space since they are processed through the HIL engine[1].

To simulate shared memory, HIL engine provides API to refresh or flush content of an array coming from application. As previously said, this array has to be Immortal. Following illustration shows native method signature on simulated platform :

```
package com.is2t.appnote;

public class NativesInterface {

    /**
     * Native method used to retrieve number of sensors.
     *
     * @return Number of plugged sensors.
     */
    public static int getNumberOfSensors() {
        return SensorsDriver.NB_SENSORS;
    }

    /**
     * Native method used to share buffer address.
     *
     * @param buffer
     *            Shared buffer.
     */
    public static void init(int[] buffer) {
        SensorsDriver driver = new SensorsDriver(buffer);
        new Thread(driver).start();
    }

}
```

Figure 3.3. Natives simulated implementation

Contrary to embedded implementation, the application buffer is seen as an array. A mock can store an array reference in order to use it after.

---

[1] During simulation, native methods are executed by the HIL engine, which provide also an API to manage JVM.

To use this buffer between the application and the simulated platform, HIL API provides two methods :

- The `flush()` method. It modifies application buffer values by replacing them with mock buffer ones.
- And the `refreshContent()` method. It does the opposite of flush method. It replaces mock buffer values by application ones.

Following illustration shows how to use the API :

```
    ...


    /*
     * (non-Javadoc)
     *
     * @see java.lang.Runnable#run()
     */
    @Override
    public void run() {
        System.out.println("[SimJPF] - update sensors values");
        for (int i = 0; i < NB_SENSORS; i++) {
            this.buffer[i] = i * 10;
        }
        HIL.getInstance().flushContent(this.buffer);
        try {
            // Wait 750 ms so that application can read new buffer values
            // and modify them
            Thread.sleep(750);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("[SimJPF] - Read sensors values : ");
        HIL.getInstance().refreshContent(this.buffer);
        for (int i = 0; i < NB_SENSORS; i++) {
            System.out.println("[SimJPF] - Sensor_" + i + " = "
                    + this.buffer[i]);
        }
    }
```

Figure 3.4. Buffer sharing (mock side)

To be able to execute the provided application on your workstation, we will need to export `DataSharing-mock` project as a jar file in your platform (like others mocks). Without this, the execution will end with timeout error since HIL engine will not find native methods implementations on platform.

4 Document History

Date	Revision	Description
December 17th 2013	A	First release

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