*Application Note:*
*TLT-0633-AN-MICROEJ-JavaCSync*

# Java / C Synchronization

*In relation to: MICROEJ products*

## *Features*

This Application Note explains how to synchronize a Java thread with one or more C tasks. The example shown uses the Keil RTX Kernel.

## *Description*

This Application Note assumes the reader wishes to understand the steps involved in synchronizing a Java thread with one or more C tasks.

# Table of Contents

## List of Figures

# 1 Introduction

## 1.1 Intended audience

The intended audience for this Application Note are developers who wish to synchronize a Java thread with one or more C tasks.

## 1.2 Scope

This Application Note shows a simple implementation of a mutual-exclusion monitor that can be used to:

- ensure that only one Java thread or C task can access a resource at any time, or
- allow a Java thread to temporarily block execution of a C task, or
- allow a C task to temporarily block execution of a Java thread.

The example assumes the use of the Keil RTX Kernel ; but the design can be easily ported to other real-time operating systems.

## 1.3 Prerequisites

This document assumes the reader is familiar with the process of creating a Java Platform (JPF), and with the creation of native methods using SNI.

## 1.4 Files supplied with this Application Note

This application note is packaged with an archive of an Eclipse project in the file `JavaCSync-example.zip`, which can be imported in the normal way.

The project contains four files of particular interest:

- The file `JavaCSyncExample.java`, in the `src` folder, is a Java application that demonstrates Java / C synchronization.
- The file `java_c_sync_example.c`, in the `c-src` folder, is the C part of the application that demonstrates Java / C synchronization.
- The file `NativeMonitor.java`, in the `src` folder, is a Java class that provides part of the implementation of a synchronization monitor.
- The file `native_monitor.c`, in the `c-src` folder, is a the C part of the implementation of a synchronization monitor.

## 2 Design

The synchronization is provided by a `NativeMonitor` object. The implementation of this object is split between a Java part and a C part, although most of the actual synchronization is done in the C part.

The API of a `NativeMonitor` object consists of two functions, which can be called from either a Java thread or a C task. These are:

- `enter` - enter (acquire) the monitor. Only a single Java thread or C task can enter the monitor. If the monitor has already been acquired this function blocks execution of the Java thread or C task until the monitor is exited.
- `exit` - exit (release) the monitor. The monitor becomes available[1].

This Application Note is supplied with an implementation of `NativeMonitor` and an example showing its use. The example comprises a Java application (the Java class `JavaCSyncExample`) and a C task (in the file `java_c_sync_example.c`). The behavior of the example is shown in the UML sequence diagram below.

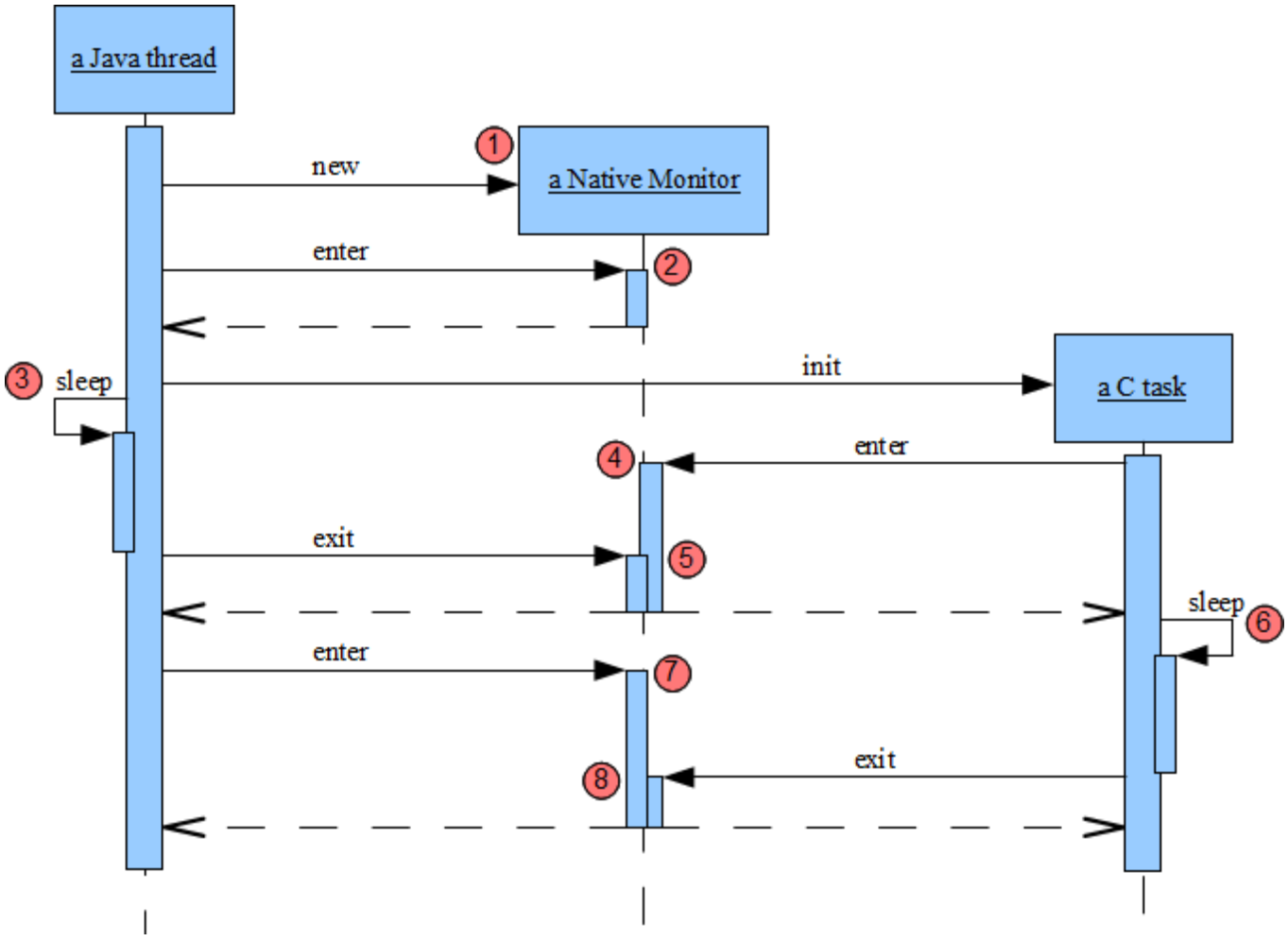


*Figure 2.1. Example Application*

Execution of the example starts with the creation of the Java thread. The key points of execution, shown by numbers on the diagram, are:

1. The Java thread instantiates a `NativeMonitor`, making it available for use.

[1]The supplied implementation is based on a counting mutex (because that is what the Keil RTX Kernel provides), so in fact the monitor is only released if the holder makes as many `exit` calls as `enter` calls.

2. The Java thread enters the `NativeMonitor`. The Java thread is now holding the monitor.

3. The Java thread invokes the native method `startCTask`, which creates the C task. The id of the `NativeMonitor` is passed to the C task as a parameter of the `startCTask` method. The Java thread then sleeps.

4. The C task tries to enter the `NativeMonitor`, but it cannot because the monitor is held by the Java thread. Execution of the C task is blocked - it must wait until the monitor is exited.

5. When it awakes from its sleep the Java code exits the `NativeMonitor`. The monitor can now be given to the C task, which continues its execution.

6. The C task now sleeps, holding the monitor.

7. The Java thread tries to enter the `NativeMonitor`, but it cannot because the monitor is held by the C task. Execution of the Java thread is blocked - it must wait until the monitor is exited.

8. When it awakes from its sleep the C task exits the `NativeMonitor`. The monitor can now be given to the Java thread, which continues its execution.

# 3 Implementation

## 3.1 Java part

We will start by looking at the Java part of the implementation of `NativeMonitor`, as implemented in the Java class of the same name. For the most part, the implementation merely calls native methods - these are discussed below.

```
public NativeMonitor() {
    id = newNativeMonitor();
    if (id < 0) {
        throw new IllegalMonitorStateException("Unable to create Native Monitor");
    }
}
```

*Figure 3.1. Java - creating a new monitor*

To create the C part of the monitor the constructor invokes a native method called `newNativeMonitor`. We will see the implementation of that method later. The native method returns the id used by the C code to identify the monitor. This id is held by the Java object. A negative id indicates a problem, which in this implementation can only be that no more monitors are available.

Note that in the supplied implementation monitors are immortal - no facility is provided for their disposal.

```
public void enter() throws IllegalMonitorStateException {
    synchronized (this) {
        if (owner != null && owner != Thread.currentThread()) {
            throw new IllegalMonitorStateException("Monitor " + id + " is already
 taken by another thread");
        } else {
            owner = Thread.currentThread();
        }
    }
    while (!tryEnter(id));
}
```

*Figure 3.2. Java - entering a monitor*

It is a restriction of this implementation that the monitor can be used by only a single Java thread. Therefore an exception is thrown if a different thread attempts to enter the monitor. The monitor holds the owning thread in a field.

The intention of this method is to block the caller until the monitor is available, and the native method `tryEnter` will suspend the calling thread if the monitor is unavailable. However, if the C task exits the monitor while this is taking place the Java thread may not suspend, so it is essential to retry the native call until its return code indicates that the monitor was entered.

```
public void exit() throws IllegalMonitorStateException {
    synchronized (this) {
        if (owner != Thread.currentThread()) {
            throw new IllegalMonitorStateException("Monitor " + id + " is held by a
 different thread");
        }
        exit(id);
        owner = null;
    }
}
```

*Figure 3.3. Java - exiting a monitor*

Except for checking that the Java thread exiting the monitor is the same thread as the one that entered it, this method relies on the native method `exit`.

## 3.2 C part

We will now examine the C part of the implementation of `NativeMonitor`. It is important when reading this code to bear in mind that it relies on specific features of the real-time operating system, which in this example is the Keil RTX Kernel. The parts that are OS-specific are highlighted.

The implementation is based on use of a RTX "mutex" - a resource that can be acquired by only one RTX task. At any moment the mutex is either:

- available, or
- owned by the RTX task that is executing the MicroJvm® virtual machine, or
- owned by some other RTX task.

Each Java instance of `NativeMonitor` is matched in the C code with a simple data structure comprising two fields, as shown in the figure below.

```
typedef struct {
    int32_t javaRequester;
    OS_MUT osMonitor;
} NativeMonitor;

#define NUMBER_OF_MONITORS 3
static int32_t next_monitorIndex = 0;
static NativeMonitor monitors[NUMBER_OF_MONITORS];
```

*Figure 3.4. Monitor data structures*

The `javaRequester` field holds the id of the Java thread waiting for this monitor, if any. The `osMonitor` field holds an OS-specific data structure used to manage the mutex.

An array of these data structures is declared, and the monitor's id is the array index. Although the supplied implementation supports 3 monitors the example uses only one of them.

### 3.2.1 Java native implementations

```
int32_t Java_com_is2t_examples_NativeMonitor_newNativeMonitor(void) {
    return new_native_monitor();
}
```

*Figure 3.5. C - Java native for creating a monitor*

The behavior for creation of a monitor by the Java thread is the same as for a C task, hence the `newNativeMonitor` native merely delegates to the function used by C tasks, as shown later.

```
int32_t Java_com_is2t_examples_NativeMonitor_tryEnter(int32_t monitorID) {
    NativeMonitor* monitor = &monitors[monitorID];
    monitor->javaRequester = SNI_getCurrentJavaThreadID();
    OS_RESULT result = os_mut_wait(&monitor->osMonitor, 0);
    if (result == OS_R_TMO) {
        SNI_suspendCurrentJavaThread(0);
        return JFALSE;
    } else {
        monitor->javaRequester = JNULL;
        return JTRUE;
    }
}
```

*Figure 3.6. C - Java native for entering a monitor*

The essence of this native method is that it must attempt to acquire the mutex, and if that succeeds (`result != OS_R_TMO`) return a success code to the Java thread. The second parameter to the `os_mut_wait`

is `0`, indicating that the caller should not be blocked if the mutex is unavailable. If the monitor cannot be acquired the calling Java thread is suspended, and a failure code returned. In this case the Java thread must try to enter the monitor again once it has been resumed.

```
void Java_com_is2t_examples_NativeMonitor_exit__I(int32_t monitorID) {
    NativeMonitor* monitor = &monitors[monitorID];
    os_mut_release(&monitor->osMonitor);
}
```

*Figure 3.7. C - Java native for exiting a monitor*

The `exit` native method simply releases the mutex.

### 3.2.2 Functions for use by C tasks

```
int32_t new_native_monitor(void) {
    if (next_monitorIndex == NUMBER_OF_MONITORS) {
        return -1;
    }
    int32_t id = next_monitorIndex++;
    NativeMonitor* monitor = &monitors[id];
    os_mut_init(&monitor->osMonitor);
    return id;
}
```

*Figure 3.8. C - creating a monitor*

This function allocates a monitor data structure and initializes the mutex in an OS-specific way. The id of the monitor (which is the index into the array of structures) is returned. Note that this function is not thread-safe - a full implementation should use another mutex to put the incrementing and testing of `next_monitorIndex` in a critical section.

```
void native_monitor_enter(int32_t monitorID) {
    NativeMonitor* monitor = &monitors[monitorID];
    OS_RESULT result = os_mut_wait(&monitor->osMonitor, 0xFFFF);
}
```

*Figure 3.9. C - entering a monitor*

The `enter` function called by C tasks just acquires the mutex in an OS-dependent manner. The second parameter to the `os_mut_wait` is `0xFFFF`, indicating that the caller should be blocked indefinitely if the mutex is unavailable.

```
void native_monitor_exit(int32_t monitorID) {
    NativeMonitor* monitor = &monitors[monitorID];
    os_mut_release(&monitor->osMonitor);
    int32_t javaRequester = monitor->javaRequester;
    if (javaRequester != JNULL) {
        int32_t result = SNI_resumeJavaThread(javaRequester);
    }
}
```

*Figure 3.10. C - exiting a monitor*

When a C task exits the monitor the mutex is released and if a Java thread is waiting it is resumed.

# 4 Running the example

To run the example you must already have a suitable Java Platform (JPF), possibly a "Basic" JPF created using the Java Platorm Example feature of the MicroEJ workbench.

1. Import the Eclipse project provided with this Application note. The project is in the file `JavaCSync-example.zip`.

2. Create a suitable "EmbJPF" launch configuration to build the `JavaCSyncExample` application, and run it.

3. Copy the files `java_c_sync_example.c` and `native_monitor.c` from the `c-src` folder to the `src` folder of your BSP project.

4. Copy the file `native_monitor.h` from the `c-src` folder to the `inc` folder of your BSP project.

5. In Keil µVision, add the files `java_c_sync_example.c` and `native_monitor.c` to your µVision project.

6. Ensure that the object file built by running the "EmbJPF" launch in the earlier step is available to your µVision project - the projects created using the Java Platorm Example feature assume that the Java object file is in the `xxxJPF/source/lib` folder, so either your launch should copy it there or you should reconfigure the µVision project to access it from wherever the launch puts it.

7. Build, deploy and run your Keil µVision project.

# 5 Document History

| Date | Revision | Description |
|---|---|---|
| March 20th 2013 | A | First release |
| November 12th 2013 | B | MicroEJ CM_ARMCC 2.0.0 compatibility |

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