

JAW User Guide

(Java API Wrapper)

Aegidius Plüss

www.aplu.ch/jaw

Content

1	Concepts	3
2	Installation of JAW and first sample application	5
2.1	Integration of third-party non-Java libraries	9
2.2	Using native windows	13
2.3	Callbacks using JNIMethod<type>	17
2.4	Transfer of string and array data	24
2.5	Using realtime measurement modules from National Instruments	29
2.5.1	Porting acquire1Scan	30
2.5.2	Porting contAcquireNScan.....	38
2.6	Data transfer using JNIBuffer	46
3	Apendix: Installation of Microsoft Visual Studio	55

1 Concepts

JAW is a frameworks with an underlying proxy pattern. In JAW a Java class and its methods have a close correspondence to the native C++ class and their methods. JAW uses but hides the JNI mechanism by defining two wrapper classes, `NativeHandler` for Java and `WindowHandler` for C++. A typical JAW application uses an instance of `NativeHandler` and derives from `WindowHandler`. No knowledge of JNI is required, but a basic knowledge of C++ is necessary, due to the fact that the user of JAW must edit and build the native DLL with a C++ IDE, typically with Microsoft's Visual Studio.

Features:

- Hides internal mechanism of JNI, no native Java methods and no JNI C-functions needed
- Native code runs in a separate native (high priority) thread that is transparent to the user
- Easy creation of visible or semi-transparent native windows with simple access to its event loop by registering specific messages
- Data transfer between primitive Java instance variables and C++ with templates eliminates type casts
- Data transfer between Java strings and array with C++ templates.
- Support for Unicode, MBCS and UTF-8 strings
- Callbacks of Java methods with C++ templates
- Native FIFO buffer for streaming data transfer from C++ to Java
- Designed to integrate third party DLL libraries into Java, e.g drivers for realtime measurement devices (National Instruments, Keithley, etc.)
- Error handling by callbacks or returning error codes rather than throwing exceptions
- Unnecessary to use *javah* to generate C header files

Compared to plain JNI, there are a few **restrictions**:

- No support of the Java invocation API (dynamic creation of a JVM, classes and methods)
- No support of data transfer via Java static fields
- Only available for the Windows operating system

On the Java side, all JAW functionalities are provided through the library `jaw.jar` that contains the wrapper class `NativeHandler`. While instantiating this class the link between Java and C++ is automatically establish by calling the C function

```
selectWindowHandler(JNIEnv * env, jobject obj, int select)
```

This is the only relict of JNI. In the body of this function a user provided C++ class (here called `MyWindowHandler`) must be instantiated that is derived from `WindowHandler`. The JNI parameters `env` and `obj` are used to initialize `WindowHandler`. They are never seen in the user code any more. `select` is used to distinguish more than one class in the same C++ project. This concept shields the user from most of the internal mechanism of JNI.

For simplicity the invocation of C++ methods is exclusively performed by `NativeHandler`'s method `int invoke(int tag)`, where the parameter `tag` is used to select one of several C++ methods. The data transfer between Java and C++ is mainly done through Java instance variables. In some simple cases, the return value of `invoke()` or callbacks may be used instead. Because the access to instance variables are based on C++ templates, this mechanism is particularly simple and omits ugly type casts between JNI and C++ data types.

Different overridden constructors of `NativeWindow` provide different native behavior. In the most simple situation, only the path of the native DLL is passed. In order to give C++ read and write access to instance variables of a Java class, its reference is passed to the constructor or `expose()` is called. As the word suggests, this operation exposes even private instance variables to the C++ code that may access them without restrictions.

The `NativeHandler`'s method `startThread()` creates a native high-priority thread. The corresponding `ThreadEventProc` calls the user provided `evThreadMsg()` in `MyWindowHandler` each time a Windows thread message is posted. In addition to the standard Win32-API messages, `WM_THREADSTART` and `WM_THREADSTOP` indicate the start and stop of the internal thread. Because `evThreadMsg()` runs in the native thread that is independent of the Java threads, asynchronous operations may be performed like natively clocked acquiring data from a measurement interface. A circular FIFO buffer is provided where C++ stores the acquired data while Java fetches them in its own thread, that may be somewhat delayed due to heavy CPU load.

A distinct constructor of `NativeWindow` creates a native window using the Win-32-API function `CreateWindow()` with selectable window style. Getting the windows messages from the corresponding `WindowEventProc` is very simple: Override `evMsg()` from `WindowHandler` in the user provided `MyWindowHandler`. To enhance performance only user selected messages defined with an OR-mask will call `evMsg()`. In order to make the native window independent of the JVM, `evMsg()` runs in a separate native thread, that is completely hidden from the user code.

2 Installation of JAW and first sample application

In all our examples we assume *Microsoft Visual Studio 2008 Express Edition* (English version) as IDE to develop the native DLL. A short installation guide can be found in the appendix. You download the latest JAW distribution from www.aplu.ch/jaw and unpack it in any folder. Copy `jaw.jar` in your favorite jar-folder, e.g. `c:\jars`, and include it as additional library to all Java projects. Some applications need classes from the package `ch.aplu.util`. For these samples copy `aplu5.jar` in `c:\jars` and include the jar file as additional library to the project.

Copy the static library `jaw.lib` in some other folder, e.g. `c:\myjni`, create a subdirectory `include` and put all include files from the JAW distribution in `c:\myjni\include`.

Our sample application shows how to pass the values of two instance variables to the native code where they are increased by one. The modified values are read back by Java and displayed in a modal dialog. We also demonstrate how to pass a value as method parameter to C++ and how to get the modified value back to Java by using the return value of the method call.

The native code is packed in the DLL `jawample.dll`, which will be created later on. The constructor of `NativeHandler` takes the fully qualified path to this DLL in order to load the native library. The `this`-reference passed to the constructor gives C++ read/write access to all instance variables, even if they are private.

Contrary to the JNI, the qualifier `native` is not necessary to share values with the native code and there is no need to create a C header file using `javah`. We proceed top-down and write the Java code first. (In this context we consider Java as main and C++ as helper language.)

```
// JawEx1.java

import javax.swing.JOptionPane;
import ch.aplu.jaw.*;

public class JniEx6
{
    // Native variables
    private int n = 0;
    private double x = 0;

    public JniEx6()
    {
        NativeHandler nh = new NativeHandler(
```

```

        "c:\\myjni\\jawsample\\release\\jawsample.dll",
        this); // Exposed object

int rc = JOptionPane.YES_OPTION;
int val = 0;
while (rc == JOptionPane.YES_OPTION)
{
    rc = JOptionPane.showConfirmDialog(null,
        "n = " + n + "; x = " + x + "; val = " + val,
        "JawInc",
        JOptionPane.OK_CANCEL_OPTION,
        JOptionPane.INFORMATION_MESSAGE);
    if (rc != JOptionPane.YES_OPTION)
    {
        nh.destroy();
        System.exit(0);
    }
    // Invoke native method
    val = nh.invoke(val);
}

public static void main(String args[])
{
    new JNIEx6();
}
}

```

As we will see, the invocation of `invoke()` calls the C++ method with the same name. Next you develop the native Code using Visual Studio. First create a new project with `File | New | Project`. In the New Project dialog select `Visual C++ | Win32` in the Project Types section and `Win32 Project` in the Templates section. Enter `jawsample` as project name and click `OK`. The `Win32 Application Wizard` will open, where you click `Next`. As `Application Settings` choose `DLL` as Application type and `Empty project` as Addition options and click `Finish`. The `Solution Explorer` pane will open. In the icon bar change `Debug` to `Release`. For the following steps we assume that the `JAW` includes are in `c:\myjni\include` and the `JAW` library `jaw.lib` is in `c:\myjni`.

Right-click on `Source Files` and select `Add | New Item`. The `Add New Item` dialog will open. Under `Categories` select `Code` and under `Templates` select `C++ File`. Enter `JawSampleHandler` as name and click `Add`. Right-click on `Header Files`, then select `Add | New Item`. Under `Categories` select `Code` and under `Templates` select `Header File`. Enter `JawSampleHandler` as name and click `Add`.

It is important to modify some of the project properties by right-clicking on the project name in the `Solution Explorer`:

Configuration Properties

- General | Character Set | Use Multi-Byte Character Set
- C/C++ | General | Additional Include Directories: c:\myjni\include
- Linker | Input | Additional Dependencies: c:\myjni\jaw.lib
- Linker | Debugging | General Debug Info: No

Now edit the header file `JawSampleHandler.h` in the usual C++ way of life.

```
// JawSampleHandler.h

#ifndef __JawSampleHandler_h__
#define __JawSampleHandler_h__

#include "jaw.h"

// ----- class JawSampleHandler -----
class JawSampleHandler : public WindowHandler
{
public:
    JawSampleHandler(JNIEnv * env, jobject obj)
        : WindowHandler(env, obj)
    {}

private:
    int invoke(int tag);
};

#endif
```

Remarks:

- The Class `JawSampleHandler` inherits from `WindowHandler` and has an empty constructor, but initializes the base class with `env` and `obj`. The only method is `invoke()` with given signature and return type
- We must include `jaw.h` to get the `JAW` class declarations

The implementation of `JawSampleHandler` defines the global function `selectWindowHandler()` that is called when Java instantiates `NativeWindow`.

```
// JawSampleHandler.cpp

#include "JawSampleHandler.h"

// ----- Selector -----
WindowHandler *
selectWindowHandler(JNIEnv * env, jobject obj, int select)
```

```

{
    return new JawSampleHandler(env, obj);
}

// ----- class JawSampleHandler -----
int JawSampleHandler::invoke(int val)
{
    // Read Java variables
    int n = JNIVar<int>(_exposedObj, "n").get();
    double x = JNIVar<double>(_exposedObj, "x").get();

    // Perform operations
    val++;
    n++;
    x++;

    // Write back Java variables
    JNIVar<int>(_exposedObj, "n").set(n);
    JNIVar<double>(_exposedObj, "x").set(x);

    return val;
}

```

Remarks:

- The global selector or dispatcher method `selectWindowHandler()` creates an instance of our own class while passing the initialization values `env` and `obj` to the base class `WindowHandler`. It establishes the link between the C based JNI and C++ and provides the information about the JVM and the Java object, that exposes its variables and methods. The parameter `select` can be used to make a selection between several user defined classes. It's not used in this example
- We access the Java instance variables through template classes `JNIVar<type>`. The constructor takes the inherited object reference and the name of the variable. We call its methods `get()` and `set()` of the anonymous temporary object. The following primitive types are supported: `bool`, `signed char`, `short`, `int`, `__int64`, `float`, `double`, `wchar_t`
- `invoke()` reads the instance variables, performs the increment operation and writes them back. It also reads its parameter `val` and returns it incremented by one. Since the signature of `invoke()` is predefined, only `int` parameters can be passed in this way

With menu option `Build | Build Solution` (or `F7`) you create `jawsample.dll` in the folder `c:\myjni\jawsample\Release`. No errors or warnings should occur. With satisfaction we run `JawEx1` with success.

2.1 Integration of third-party non-Java libraries

One of the main purpose for the development of JAW was the easy integration of non-Java libraries into a Java application. Sometimes the source code is available, but most of the time only a dynamic library (DLL) together with its static import library are distributed.

In our next example we show how easy it is to integrate a given DLL in a Java program. For the purpose of the demonstration we use the audio library **irrKlang**, that you can download without charge from the internet (using a search machine with the keyword *irrklang*). The library is fine to play sound files in several formats, such as wav, mp3, ogg, etc. Moreover you can enrich the sound with several effect, such as echo, reverb, distortion. The libraries is specially suitable to add a sound background to game applications.

After downloading and unpacking you should take a short view to the distributed sample programs found in the example folder. As you see there, it's very easy to play a sound file. Just create an instance of ISoundEngine

```
ISoundEngine * engine = createIrrKlangDevice();
```

and start playing with

```
engine->play2D("soundfile");
```

where `soundfile` is the fully qualified path to the sound file.

As usual we begin on the Java side and write a simple application that displays a modal dialog using a `JOptionPane` where the user clicks the OK button to start playing. Clicking the OK button in the next dialog, terminates the application. For the sake of simplicity the path to the sound file is hardwired in the Java code and read by the native code. `invoke(0)` initializes the native sound engine und returns an error code (0 for success, -1 for failure). The sound engine needs to run in a native thread, because the Java thread is blocked in the modal dialog. This native thread is started with `startThread()`.

```
// JawEx2.java
// Use free audio library irrKlang from www.ambiera.com
import javax.swing.JOptionPane;
import ch.aplu.jaw.*;

public class JawEx2
{
    // Native variable
    private String audiofile = "c:/scratch/bamboleo.mp3";

    public JawEx2()
    {
        NativeHandler nh = new NativeHandler(
            "c:\\myjni\\audio\\release\\audio.dll", this);
        if (nh.invoke(0) == -1)
    }
}
```

```

    {
        JOptionPane.
            showMessageDialog(null,
                "Could not start IrrKlang engine",
                "Fatal Error",
                JOptionPane.ERROR_MESSAGE);

        nh.destroy();
        System.exit(-1);
    }

    JOptionPane.
        showMessageDialog(null,
            "Press OK to start",
            "Trying to play " + audiofile + "...",
            JOptionPane.INFORMATION_MESSAGE);
    nh.startThread(); // Start engine in high priority thread
    JOptionPane.
        showMessageDialog(null,
            "Press OK to stop",
            "Playing...",
            JOptionPane.INFORMATION_MESSAGE);

    nh.destroy();
    System.exit(0);
}

public static void main(String args[])
{
    new JawEx2();
}
}

```

Like before, create a new project in Visual Studio named `audio` and add the new files `AudioHandler.cpp` and `AudioHandler.h`. In the project properties add the `irrKlang` include folder in addition to the `JAW` include folder. If you unpacked `irrKlang` in `c:\irrKlang`, the folder is named `c:\irrKlang\include`. Append to the linker additional dependencies the `irrKlang` import library `c:\irrKlang\lib\win32-visualstudio\irrKlang.lib`. Copy the `irrKlang` DLLs `irrKlang.dll` and `ikpMP3.dll` to the home directory of the Java project, in order to be loaded by `AudioHandler.dll`. (If you get an *RuntimeException: Can't load DLL*, these DLLs are not in the right folder.)

The interface `AudioHandler.h` declares the class `AudioHandler`. Override `WindowHandler`'s methods `wantThreadMsg()` and `evThreadMsg()` and declare the instance variables `engine` and `audiofile`.

```
// AudioHandler.h
```

```

#ifndef __AudioHandler_h__
#define __AudioHandler_h__

#include "jaw.h"
#include "irrklang.h"
using namespace irrklang;

// ----- class AudioHandler -----
class AudioHandler : public WindowHandler
{
public:
    AudioHandler(JNIEnv * env, jobject obj)
        : WindowHandler(env, obj)
    {}

protected:
    virtual bool wantThreadMsg(UINT uMsg);
    virtual void evThreadMsg(HWND hWnd,
                             UINT uMsg,
                             WPARAM wParam,
                             LPARAM lParam);

private:
    int invoke(int tag);
    ISoundEngine * engine;
    const char * audiofile;

};

#endif

```

We mention some particularities in the implementation of `AudioHandler` :

- `invoke()` initializes the sound engine. With a string template we get the path of the sound file from the Java instance variable
- An OR-mask in `ThreadMsg()` let you select which messages of the *ThreadEventProc* are really needed. In addition to the non-windows related WM_messages documented in the Win32-API documentation, two extra messages `WM_THREADSTART` and `WM_THREADSTOP` are sent when Java calls `startThread()` or `stopThread()` respectively
- `evThreadMsg()` is called each time a message is sent. All information about the message can be retrieved from its parameters. `evThreadMsg()` runs in a native thread, so we play the sound file within this thread by calling `play2D()` in the `WM_THREADSTART` section. In the `WM_THREADSTOP` section that also runs when `NativeHandler.destroy()` is called, the sound engine and the allocated string array are freed.

```
// AudioHandler.cpp
```

```

// Project properties: Add irrKlang include directory
//                               Add library irrKlang.lib
// irrKlang.dll and ikpMP3.dll must reside in the
// Java class home directory

#include "AudioHandler.h"

// ----- Selector -----
WindowHandler *
selectWindowHandler(JNIEnv * env, jobject obj, int select)
{
    return new AudioHandler(env, obj);
}

// ----- class AudioHandler -----
int AudioHandler::invoke(int val)
{
    // Get string from Java instance variable
    audiofile = JNIString(_exposedObj, "audiofile").getUTF8();

    // Start the sound engine with default parameters
    engine = createIrrKlangDevice();

    if (!engine)
        return -1; // Error starting the engine
    return 0;
}

bool AudioHandler::wantThreadMsg(UINT uMsg)
{
    return (uMsg == WM_THREADSTART || WM_THREADSTOP);
}

void AudioHandler::evThreadMsg(HWND /*hwnd*/,
                               UINT message,
                               WPARAM /* wParam */,
                               LPARAM /* lParam */)
// Runs in the high priority native JNIThread
{
    switch (message)
    {
        case WM_THREADSTART:
            engine->play2D(audiofile);
            break;

        case WM_THREADSTOP:
            engine->drop(); // Delete engine
            delete [] audiofile; // Release allocated memory
    }
}

```

```
        break;
    }
}
```

After building `audio.dll` run the Java application `JawEx2`. When you click the OK button on the modal dialog box the sound starts playing until you click the OK button again.

2.2 Using native windows

In some situations a native window that displays text, graphics or images and where the messages sent to its *WindowEventProc* are accessible is a better choice than Swing or AWT windows. Native windows may have interesting properties missing in Java like semi-transparency. In the next example you create a window with selectable transparency level. It contains an BMP image. As usual you start writing the Java application `JawEx3.java`.

```
// JawEx3.java

import ch.aplu.jaw.*;
import javax.swing.JOptionPane;

public class JawEx3
{
    public JawEx3()
    {
        NativeHandler nh = new NativeHandler(
            "c:\\myjni\\trans\\release\\trans.dll", // Path of DLL
            "Transparent window",                 // Window title
            10, 20,                               // Window position
            260, 200,                             // Window size
            NativeHandler.WS_EX_TRANSPARENT);    // Window style

        String valueStr = "";
        int value;

        while (valueStr != null)
        {
            valueStr = JOptionPane.showInputDialog(
                null, "Transparency (0..100)",
                "Native Window Transparency",
                JOptionPane.QUESTION_MESSAGE);
            try
            {
                value = Integer.parseInt(valueStr);
                if (value < 0 || value > 100)
                    throw new NumberFormatException();
            }
        }
    }
}
```

```

    }
    catch (NumberFormatException ex)
    {
        continue;
    }
    nh.showWindow(value);
}
nh.destroy();
System.exit(0);
}

public static void main(String args[])
{
    new JawEx3();
}
}

```

We use one of the overloaded constructors of `NativeWindow` that defines the most important window properties. When executing, JAW registers a window class using the Win32-API function `RegisterClassEx()` and creates a window with the given OR-combination of style flags with the API function `CreateWindow()`. These flags are declared and documented in `NativeWindow`.

The program is running in a loop and displays a modal dialog where the percentage of transparency may be entered. Next you edit the interface of the class `TransHandler`.

```

// TransHandler.h

#ifndef __TransHandler_h__
#define __TransHandler_h__

#include "jaw.h"

// ----- class TransHandler -----
class TransHandler: public WindowHandler
{
public:
    TransHandler(JNIEnv * env, jobject obj);

protected:
    virtual bool wantMsg(UINT uMsg);
    virtual bool evMsg(HWND hWnd,
                       UINT uMsg,
                       WPARAM wParam,
                       LPARAM lParam);
};

#endif

```

TransHandler overrides wantMsg() and evMsg() from WindowHandler. Similar to wantThreadMsg() and evThreadMsg() the OR-mask in wantMsg() determines which messages sent to the *WindowEventProc* will call evMsg(). In this example we need WM_CREATE and WM_DESTROY that occur when the window is created and destroyed respectively, and also WM_PAINT that occurs each time the window must be repainted. For details about the Win32-API you get many information from the internet using a search machine and the keywords *windows api programming tutorial*.

We should mention that evMsg() runs automatically in a native thread that is completely decoupled from the JVM what makes the native window completely independent of the Java threads.

```
// TransHandler.cpp

#include "TransHandler.h"
#include <windows.h>

// ----- Selector -----
WindowHandler *
selectWindowHandler(JNIEnv * env, jobject obj, int select)
{
    return new TransHandler(env, obj);
}

// ----- class TransHandler -----
// Constructor, initialize base class
TransHandler::TransHandler(JNIEnv * env, jobject obj)
    : WindowHandler(env, obj)
{}

// Select which window messages will call evMsg()
bool TransHandler::wantMsg(UINT uMsg)
{
    return (uMsg == WM_CREATE ||
            WM_PAINT ||
            WM_DESTROY);
}

// Event procedure
// Return false, if message is handled by this procedure,
// true to invoke DefWindowProc()
bool TransHandler::evMsg(HWND hWnd, UINT message,
                          WPARAM /* wParam */, LPARAM lParam)
{
    static HBITMAP hBm = NULL;
    HDC hDC;
    HDC hDCMem;

```

```

BITMAP bm;
PAINTSTRUCT ps;

switch (message)
{
    case WM_CREATE:
        hBm = LoadBitmap(getInstance(), TEXT("ROSE"));
        break;

    case WM_PAINT:
        hDC = BeginPaint(hWnd, &ps);
        hDCMem = CreateCompatibleDC(hDC);
        SelectObject(hDCMem, hBm);
        GetObject(hBm, sizeof(bm), &bm);
        BitBlt(hDC, 0, 0, bm.bmWidth, bm.bmHeight,
            hDCMem, 0, 0, SRCCOPY);
        DeleteDC(hDCMem);
        EndPaint(hWnd, &ps);
        return false;

    case WM_DESTROY:
        DeleteObject(hBm);
        break;
}
return true;
}

```

In order to use the image file `rose.bmp` as resource, the resource file `trans.rc` must be included in the project. Both files must reside in the same folder as `TransHandler.cpp` and `TransHandler.h`.

```

// trans.rc

ROSE                BITMAP    DISCARDABLE    "rose.bmp"

```

(Because the resource editor is not available in the current release of *Visual Studio Express Edition*, `trans.rc` must be created with the source editor or with any other text editor.)

Fig. 2.1 shows the semi-transparent window dragged over some other application window.

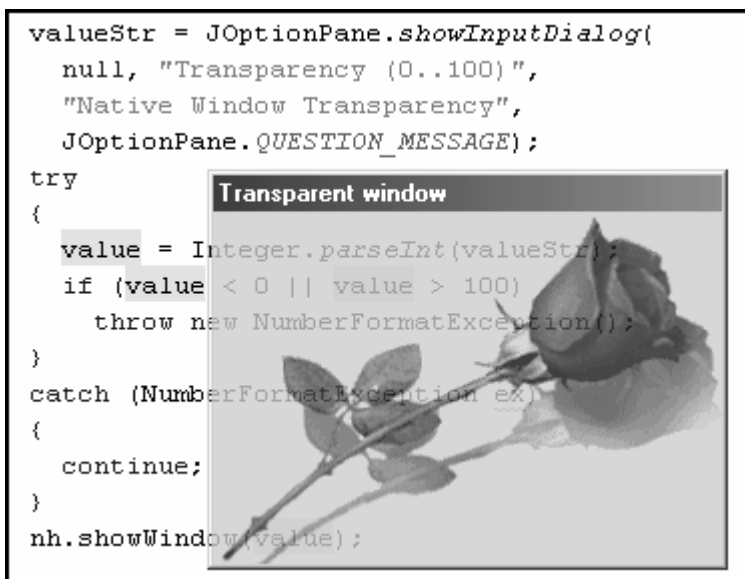


Fig 2.1: Semi-transparent window

2.3 Callbacks using JNIMethod<type>

One of the well-known features of JNI is calling Java methods from native code. Such methods are often named *callbacks*. With JAW creating callbacks is very simple because of the template JNIMethod<type> that is similar to JNIVar<type>. The parameterless Java method methodName is invoked with

```
JNIMethod<type>(_exposedObj, "methodName").call();
```

where type is the return type of methodName. As explained before, _exposedObj is inherited from WindowHandler.

When the Java method has parameters, the invocation from C++ is somewhat more complicated, because it is necessary to include the signature of the parameter types in the order they are specified in the method's declaration. As an example, if we want to call the Java method

```
void doIt(int i, double x, char c) {...}
```

from C++, we first declare doIt() as JNIMethod<void> using the signature sequence IDC (for int, double, char):

```
JNIMethod<void> doIt(_exposedObj, "doIt", "IDC");
```

Then we pack the three parameters in a `jvalue` union, which is declared as follows:

```
typedef union jvalue {
    jboolean z;
    jbyte    b;
    jchar    c;
    jshort   s;
    jint     i;
    jlong    j;
    jfloat   f;
    jdouble  d;
    jobject  l;
} jvalue;
```

In our example, where we have three parameters of type `int`, `double`, `char` this becomes

```
jvalue par[3];
par[0].i = 2;
par[1].d = 3.1415;
par[2].c = 'A';
```

The invocation is done with the method `call()` that takes the parameter `par`:

```
doIt.call(par);
```

In our sample application, some of the mouse events from the native window are reported back to Java and displayed in a console redirected to a window from the package `ch.aplu.util`.

```
// JawEx4.java

import ch.aplu.util.*;
import ch.aplu.jaw.*;

public class JawEx4 implements ExitListener
{
    private int evt;
    private int x;
    private int y;
    private NativeHandler nh;

    public JawEx4()
    {
        Console c = new Console();
```

```

c.addExitListener(this);
nh = new NativeHandler(
    "c:\\myjni\\mouseevent\\release\\mouseevent.dll",
    this,
    "Move Window",
    10, 20,
    258, 184,
    NativeHandler.WS_POPUP | NativeHandler.WS_VISIBLE);

System.out.println(
    "Click or drag image with left mouse button");
while (true)
{
    Monitor.putSleep();
    displayEvent();
}

// Native call
private void mouseEvent(int evt, int x, int y)
{
    this.evt = evt;
    this.x = x;
    this.y = y;
    Monitor.wakeUp();
}

// Implement ExitListener
public void notifyExit()
{
    nh.destroy();
    System.exit(0);
}

private void displayEvent()
{
    switch (evt)
    {
        case NativeMouse.lPress:
            report("LeftButtonDown", x, y);
            break;

        case NativeMouse.lRelease:
            report("LeftButtonUp", x, y);
            break;

        case NativeMouse.lDClick:
            report("LeftButtonDoubleClick", x, y);
    }
}

```

```

        break;
    }
}

private void report(String msg, int x, int y)
{
    System.out.println(
        msg + " at x: " + x + " y: " + y);
}

public static void main(String args[])
{
    new JawEx4();
}
}

```

The callback method `mouseEvent` has 3 integer parameters, the first is a code for the event type, the other are the coordinates `x` and `y`, where the event occurred. We strongly advise against long lasting code in any callback, because they may happen in a rapid succession. It is much better to copy the values to instance variables and to wake up a sleeping Java thread that performs the more lengthy operation of displaying the values in the console window.

In the C++ project named `mouseevent`, we declare the class `MouseEventHandler`. Like before, in order the image file `rose.bmp` to be loaded, the resource file `MouseEvent.rc` must be part of the project.

```

// MouseEvent.rc

ROSE                BITMAP  DISCARDABLE    "rose.bmp"

```

The interface `MouseEventHandler.h` is very simple.

```

// MouseEventHandler.h

#ifndef __MouseEventHandler_h__
#define __MouseEventHandler_h__

#include "jaw.h"

// ----- class MouseEventHandler -----
class MouseEventHandler: public WindowHandler
{
public:
    MouseEventHandler(JNIEnv * env, jobject obj);

protected:
    virtual bool wantMsg(UINT uMsg);
    virtual bool evMsg(HWND hWnd,

```

```

        UINT uMsg,
        WPARAM wParam,
        LPARAM lParam);
private:
    void sendMouseMessage(int message, int x, int y);
};
#endif

```

In addition to reporting the mouse events in the implementation of `MouseEventHandler`, the native window can be dragged by pressing the left mouse button. To realize this feature we register `WM_MOVE` and move the window to its new position with `SetWindowPos()`.

```

// MouseEventHandler.cpp

#include "MouseEventHandler.h"

// ----- Selector -----
WindowHandler *
selectWindowHandler(JNIEnv * env, jobject obj, int select)
{
    return new MouseEventHandler(env, obj);
}

// ----- class MouseEventHandler -----
// Constructor, initialize base class
MouseEventHandler::MouseEventHandler(JNIEnv * env,
                                     jobject obj)
    : WindowHandler(env, obj)
{}

// Select which windows message will call evMsg()
bool MouseEventHandler::wantMsg(UINT uMsg)
{
    return (uMsg == WM_CREATE ||
            WM_LBUTTONDOWN ||
            WM_LBUTTONUP ||
            WM_LBUTTONDBLCLK ||
            WM_MOVE ||
            WM_PAINT ||
            WM_DESTROY);
}

// Event procedure
// Return false, if message is handled by this procedure,
// true to invoke DefWindowProc()
bool MouseEventHandler::evMsg(HWND hWnd, UINT message,
                              WPARAM wParam, LPARAM lParam)

```

```

{
    static HBITMAP hBm = NULL;
    static int xPos;
    static int yPos;
    HDC hDC;
    HDC hDCMem;
    BITMAP bm;
    PAINTSTRUCT ps;

    switch (message)
    {
        case WM_CREATE:
            hBm = LoadBitmap(getInstance(), TEXT("ROSE"));
            return false;

        case WM_LBUTTONDOWN:
            xPos = LOWORD(lParam); // Relative coordinates
            yPos = HIWORD(lParam);
            sendMouseMessage(LPRESS, xPos, yPos);
            return false;

        case WM_LBUTTONUP:
            sendMouseMessage(LRELEASE, LOWORD(lParam),
                             HIWORD(lParam));

            return false;

        case WM_LBUTTONDBLCLK:
            sendMouseMessage(LDCLICK, LOWORD(lParam),
                             HIWORD(lParam));

            return false;

        case WM_MOUSEMOVE:
            if (wParam == MK_LBUTTON)
            {
                int xNew = LOWORD(lParam); // Relative coordinates
                int yNew = HIWORD(lParam);
                int xDelta = xNew - xPos; // Displacement
                int yDelta = yNew - yPos;
                _xPos += xDelta; // Absolute coordinates
                _yPos += yDelta; // of upper left corner
                SetWindowPos(hWnd, HWND_TOPMOST, _xPos, _yPos, 0, 0,
                             SWP_NOSIZE | SWP_SHOWWINDOW);
            }
            return false;

        case WM_PAINT:
            hDC = BeginPaint(hWnd, &ps);
            hDCMem = CreateCompatibleDC(hDC);
    }
}

```

```

        SelectObject(hDCMem, hBm);
        GetObject(hBm, sizeof(bm), &bm);
        BitBlt(hDC, 0, 0, bm.bmWidth, bm.bmHeight,
              hDCMem, 0, 0, SRCCOPY);
        DeleteDC(hDCMem);
        EndPaint(hWnd, &ps);
        return false;

    case WM_DESTROY:
        DeleteObject(hBm);
        break;
    }
    return true;
}

void MouseEventHandler::sendMouseEvent(int message,
                                       int x, int y)
{
    JNIMethod<void>
        mouseEvent(_exposedObj, "mouseEvent", "III");
    jvalue par[3];
    par[0].i = message;
    par[1].i = x;
    par[2].i = y;
    mouseEvent.call(par);
}

```

For the callback we declare `mouseEvent` with the parameter signature `III` and invoke `call()` with the three integers packed into `par`. Fig. 2.2 shows the native window dragged over the console window.

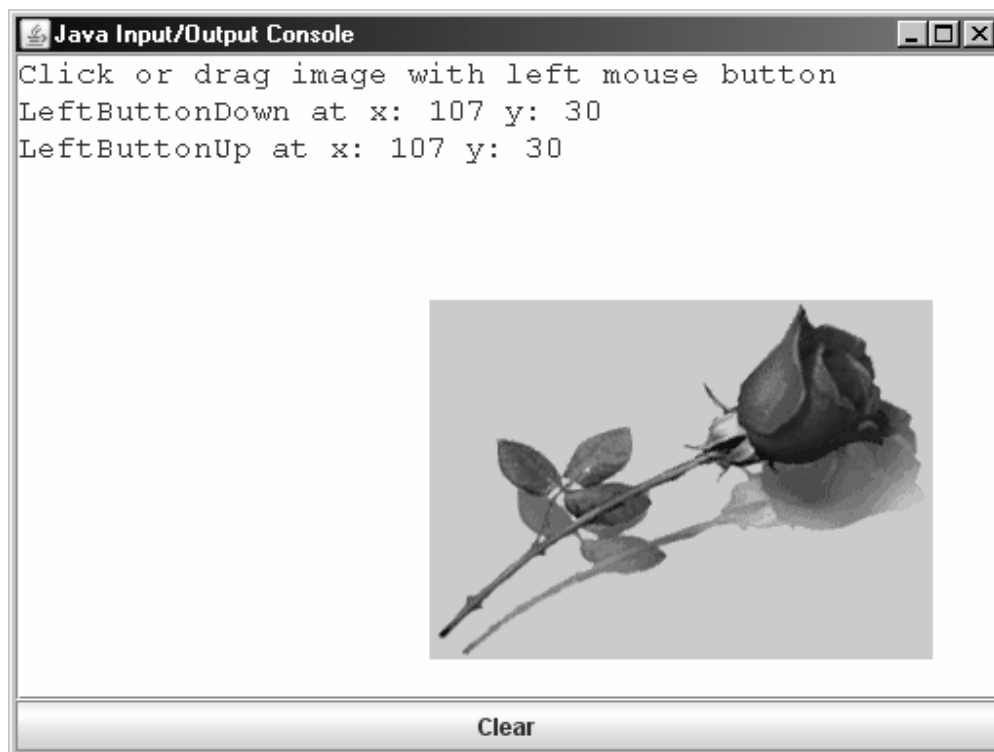


Fig. 2.2: Callback with mouse events

2.4 Transfer of string and array data

In JAW we use the class `JNIString` to transfer strings between Java and the native code. In C++ a string can be represented by a character array, where the bytes are interpreted differently depending on the character set. In Java strings are always coded in the Unicode character set. When a Java string is transferred to C++, different `get()` methods let you decide if characters are stored in the Multibyte Character Set (MBCS), the UTF-8 or the Unicode character encoding. When calling `get()` a character array of sufficient length is created that holds the transferred characters. Do not forget to free the allocated memory with `delete[]` before the program ends.

The mechanism for transferring arrays of primitive types remains the same. Template classes `JNIArray<type>` are used, where `type` is one of `wchar_t`, `bool`, `signed char`, `short`, `int`, `__int64`, `float`, `double`. Depending on the JVM, an internal buffer is created, that will be automatically deallocated by JAW when the destructor of `JNIArray<>` runs. Array elements are read or set by using an array pointer that is returned for

all array types by the same method `get()`. Be aware that this pointer becomes invalid after the destructor is executed.

For the sake of a simple demonstration we transfer string and array data from Java to C++ and then back to Java.

```
// JawEx5.java

import ch.aplu.jaw.*;

public class JawEx5
{
    // Native transfer data block
    private String str = "abc-äöü";
    private int[] ary = {1, 2, 3};

    public JawEx5()
    {
        NativeHandler nh =
            new NativeHandler("c:\\myjni\\data\\release\\data.dll",
                this);

        System.out.println("Sending string to C++...");
        nh.invoke(0);

        nh.invoke(1);
        System.out.println("Got back from C++ (UTF8):");
        System.out.println(str);

        nh.invoke(2);
        System.out.println("Got back from C++ (MBCS):");
        System.out.println(str);

        nh.invoke(3);
        System.out.println("Got back from C++ (Unicode):");
        System.out.println(str);

        System.out.println("Sending array to C++...");
        nh.invoke(4);
        System.out.println("Got back from C++:");
        for (int i = 0; i < 3; i++)
            System.out.print(ary[i] + ", ");

        nh.destroy();
        System.exit(0);
    }

    public static void main(String args[])

```

```

{
    new JawEx5();
}
}

```

The C++ project named `data` declares a class `DataDemo` that contains the only method `invoke()`. The interface `DataHandler.h` is very simple:

```

// DataHandler.h

#ifndef __DataHandler_h__
#define __DataHandler_h__

#include "jaw.h"

// ----- class DataDemo -----
class DataDemo : public WindowHandler
{
public:
    DataDemo(JNIEnv * env, jobject obj)
        : WindowHandler(env, obj)
    {}

private:
    int invoke(int tag);
};

#endif

```

As you see in the following implementation, `invoke(0)` displays the transferred strings in a native console using `cout`. `invoke(1)`, `invoke(2)` and `invoke(3)` returns the strings using different character sets. Because we selected the Multi-Byte character set in the project's properties, `setMBCS()` returns accentuated characters correctly. Since we used `wchar_t`, `setUnicode()` works fine too, however `setUTF8()` fails.

With `invoke(4)` the Java integer array is transferred to C++, displayed in a console, modified and returned back to Java, where it is displayed again.

```

// DataHandler.cpp

#include "DataHandler.h"
#include <iostream>

using namespace std;

// ----- Selector -----

```

```

WindowHandler *
selectWindowHandler(JNIEnv * env, jobject obj, int select)
{
    return new DataDemo(env, obj);
}

// ----- class DataDemo -----
int DataDemo::invoke(int val)
{
    switch (val)
    {
        case 0:
        {
            const char * charstr =
                JNIString(_exposedObj, "str").getMBCS();
            cout << "C++ got MBCS string: " << charstr << endl;
            delete[] charstr; // Release memory

            const char * utf8str =
                JNIString(_exposedObj, "str").getUTF8();
            cout << "C++ got UTF8 string: " << utf8str << endl;
            delete[] utf8str; // Release memory

            const wchar_t * unicodestr =
                JNIString(_exposedObj, "str").getUnicode();
            int len = int(wcslen(unicodestr));
            char * asciistr = new char[len + 1];
            WideCharToMultiByte(CP_ACP, 0, unicodestr, -1,
                               asciistr, len+1, NULL, NULL);
            cout << "C++ got unicode string: " << asciistr << endl;
            delete[] unicodestr; // Release memory
            delete[] asciistr; // ditto
        }
        break;

        case 1:
        {
            const char * cstr = "ABC-ÄÖÜ";
            JNIString(_exposedObj, "str").setUTF8(cstr);
        }
        break;

        case 2:
        {
            const char * cstr = "ABC-ÄÖÜ";
            JNIString(_exposedObj, "str").setMBCS(cstr);
        }
        break;
    }
}

```

```

case 3:
{
    // Don't forget prefix L
    const wchar_t * unicodeStr = L"ABC-ÄÖÜ";
    JNIString(_exposedObj, "str").setUnicode(unicodeStr);
}
break;

case 4:
{
    JNIArray<int> jary(_exposedObj, "ary");
    int * ary = jary.get();
    cout << "C++ got array:" << endl;
    for (int i = 0; i < 3; i++)
        cout << (int)ary[i] << ", ";
    cout << endl;

    // Write new values back to Java
    for (int i = 0; i < 3; i++)
        ary[i] = 10*(i+1);

    // JNIArray<> destructor releases array here
    break;
}
}
return 0;
}

```

The execution shows:

```

Sending string to C++...
C++ got UTF8 string: abc-ÃÄÅ¼
C++ got MBCS string: abc-äöü
C++ got unicode string: abc-äöü
Got back from C++ (UTF8):
ABC-ÄÖÜ
Got back from C++ (MBCS):
ABC-ÄÖÜ
Got back from C++ (Unicode):
ABC-ÄÖÜ
Sending array to C++...
C++ got array:
1, 2, 3,
Got back from C++:
10, 20, 30,

```

2.5 Using realtime measurement modules from National Instruments

National Instruments (NI) does not provide any support for Java programmers. In their Knowledge Base NI suggests to use a JNI wrapper:

How can I access National Instruments drivers from Java?

Solution:

National Instruments does not currently provide language interfaces for Java. It is possible, however, to access National Instruments drivers by making calls to the driver DLL.

Sun Java includes the Java Native Interface (JNI) which allows calls to be made to C DLLs from Java. To utilize JNI with National Instruments drivers, it is necessary to create a wrapper DLL for your driver DLL. This wrapper DLL will comply with JNI naming requirements, and perform the conversion from Java to C data types.

Consulting the sample programs written in C that are part of the distribution of *NI-DAQmx* and the *NI-DAQmx Function Reference*, JAW gives you the means to integrate the Nidaq C library smoothly into you Java code.

First you need to download and install the distribution of *NI-DAQmx Base*. You find it with a internet search machine and the keywords *nidaqmx download*. During the installation some DLLs are copied to the windows\system32 folder, amongst others `nidaqmxbase1v.dll`.

On the native side we continue to use the development environment *Microsoft Visual Studio 2008 Express Edition (English version)*. For *NI-DAQmx Base* add the following items to the project properties:

- Configuration Properties | C/C++ | General | Additional Include Directories: directory where is `NIDAQmxBase.h`, e.g. `c:\nidaq\include`
- Configuration Properties | Linker | Input | Additional Dependencies: fully qualified path of `nidaqmxbase.lib`, e.g. `c:\nidaq\nidaqmxbase.lib`

For our example we use the inexpensive NI-module *USB-6008* with the following features:

- 8 analog input channels (12 bits), max sample rate 10 kHz (for 1 channel)
- 2 analog output channels (12 bits)
- 12 digital input/output channels
- 32 bit counter

After installing the NI-DAQmx Base software, the module is automatically recognized as new USB device.

2.5.1 Porting acquire1Scan

In our next example we port the functionalities of the sample program `acquire1Scan.c` that is included in the NI-DAQmx distribution from C to Java. The application reads a single value from the analog input and displays it in a console window. The original source is well documented, if further information is needed consult the *NI-DAQmx Base C Function Reference* found with an internet search machine. Here is the code in the original formatting:

```

/*****
 *
 * ANSI C Example program:
 *   acquire1Scan.c
 *
 * Example Category:
 *   AI
 *
 * Description:
 *   This example demonstrates how to take a single Measurement
 *   (Sample) from an Analog Input Channel.
 *
 * Instructions for Running:
 *   1. Select the Physical Channel to correspond to where your
 *      signal is input on the DAQ device.
 *   2. Enter the Minimum and Maximum Voltage Ranges.
 *   Note: For better accuracy try to match the Input Ranges to the
 *         expected voltage level of the measured signal.
 *   Note: Use the genVoltage example to provide a signal on
 *         your DAQ device to measure.
 *
 * Steps:
 *   1. Create a task.
 *   2. Create an Analog Input Voltage Channel.
 *   3. Use the Read function to Measure 1 Sample from 1 Channel on
 *      the Data Acquisition Card. Set a timeout so an error is
 *      returned if the sample is not returned in the specified time
 *      limit.
 *   4. Display an error if any.
 *
 * I/O Connections Overview:
 *   Make sure your signal input terminal matches the Physical
 *   Channel I/O Control.
 *
 * Recommended Use:
 *   Call the Read function.
 *
 *****/

#include "NIDAQmxBase.h"
#include <stdio.h>

#define DAQmxErrChk(functionCall) \
    { if( DAQmxFailed(error=(functionCall)) ) { goto Error; } }

int main(int argc, char *argv[])
{
    // Task parameters
    int32    error = 0;
    TaskHandle taskHandle = 0;

```

```

char      errBuff[2048]={'\0'};

// Channel parameters
char      chan[] = "Dev1/ai0";
float64   min = -10.0;
float64   max = 10.0;

// Timing parameters
uInt64    samplesPerChan = 1;

// Data read parameters
float64   data;
int32     pointsToRead = 1;
int32     pointsRead;
float64   timeout = 10.0;

DAQmxErrChk (DAQmxBaseCreateTask("",&taskHandle));
DAQmxErrChk (DAQmxBaseCreateAIVoltageChan(
    taskHandle,chan,"",DAQmx_Val_Cfg_Default,min,
    max,DAQmx_Val_Volts,NULL));
DAQmxErrChk (DAQmxBaseStartTask(taskHandle));
DAQmxErrChk (DAQmxBaseReadAnalogF64(
    taskHandle,pointsToRead,timeout,DAQmx_Val_GroupByChannel,
    &data,samplesPerChan,&pointsRead,NULL));

printf ("Acquired reading: %f\n", data);

Error:
if( DAQmxFailed(error) )
    DAQmxBaseGetExtendedErrorInfo(errBuff,2048);
if( taskHandle!=0 ) {
    DAQmxBaseStopTask(taskHandle);
    DAQmxBaseClearTask(taskHandle);
}
if( DAQmxFailed(error) )
    printf("DAQmxBase Error: %s\n",errBuff);
return 0;
}

```

As you see, the initialization is done step-by-step. We will maintain this process in the Java code by calling `invoke()` with several different parameters. The return value is used as error code: 0 means success, -1 means failure. Java can retrieve and display the description of the error from the String `errorDescription`. The Java application uses a console window from the package `ch.aplu.util`.

```

// JawEx6.java
// Prototype is Acquire1Scan from NIDAQmx distribution
// Acquire analog data and display in modeless dialog
// For demonstration with NI USB-6008 interface
// Apply voltage +-5V max at terminals #2 and #1 (GND) or
// attach 10 kOhm potentiometer at terminals #31/#32
// with potentiometer tap at terminal #2
// Timed by API Sleep

```

```

import ch.aplu.jaw.*;
import ch.aplu.util.*;

public class JawEx6 implements ExitListener
{
    // Native transfer data block
    private String channel = "Dev1/ai0"; // DeviceID/channel
    private double min = -10; // Input range min
    private double max = 10; // Input range max
    private double value; // Acquired data
    private String errorDescription = ""; // Error report buffer
    private int period = 1000; // Measurement interval (ms)
    // End of native transfer data block

    private NativeHandler nh =
        new NativeHandler("c:\\myjni\\nidaq\\release\\nidaq.dll",
            this); // Provide access by exposing
    private enum State{failed, initializing, running, quitting};
    private volatile State state = State.initializing;

    public JawEx6()
    {
        Console c = Console.init();
        c.addExitListener(this);
        System.out.println("Initializing...");
        System.out.print("  Creating task...");
        if (nh.invoke(0) == -1)
        {
            showError();
            return;
        }
        System.out.println("OK");

        System.out.print("  Open channel...");
        if (nh.invoke(1) == -1)
        {
            showError();
            return;
        }
        System.out.println("OK");

        System.out.print("  Start task...");
        if (nh.invoke(2) == -1)
        {
            showError();
            return;
        }
        System.out.println("OK");
    }
}

```



```

System.out.println("\nGetting data...");
int nb = 1;
state = State.running;
while (state == State.running)
{
    if (nh.invoke(3) == -1)
        break;
    else
    {
        System.out.
            println("Data #" + nb + ": " + value + " V");
        nh.invoke(4); // Sleep a while
        nb++;
    }
}
if (state == State.running)
{
    state = State.failed;
    System.out.println("\nError while acquiring data");
}
else
    System.out.println("\nStopped by user");
}

private void showError()
{
    state = State.failed;
    System.out.println("\nError: " + errorDescription);
}

public void notifyExit()
{
    switch (state)
    {
        case failed:
            nh.destroy();
            System.exit(1);
            break;

        case initializing:
            // Don't interrupt
            break;

        case running:
            state = State.quitting;
            break;
    }
}

```

```

        case quitting:
            nh.destroy();
            System.exit(0);
            break;
    }
}

public static void main(String args[])
{
    new JawEx6();
}
}

```

Once again, the data transfer between Java and C++ is done with Java instance variables that can be accessed from both side. We are careful to free all allocated resources by calling `destroy()` when the program ends. A state variable implemented as `enum` makes the program design simple.

Because we initialize step-by-step, some variables must survive from one call of `invoke()` to another. Therefore we copy the passed values into C++ instance variables (member variables). You create a new C++ project named `nidaq` and code the header file `NIDAQHandler.h` first, where the original naming and formatting conventions are maintained.

```

// NIDAQHandler.h

#ifndef __NIDAQHandler_h__
#define __NIDAQHandler_h__

#include "jaw.h"
#include "NIDAQmxBase.h"

// ----- class NIDAQAcquire1Scan -----
class NIDAQAcquire1Scan : public WindowHandler
{
public:
    NIDAQAcquire1Scan(JNIEnv * env, jobject obj);
    ~NIDAQAcquire1Scan();

private:
    int invoke(int tag);
    int period;

    // Task parameters
    int32      error;
    TaskHandle taskHandle;
    char       errBuff[2048];
}

```

```

// Channel parameters
const char* chan;
float64    min;
float64    max;

// Timing parameters
uInt32     samplesPerChan;

// Data read parameters
float64    data;
int32      pointsToRead;
int32      pointsRead;
float64    timeout;
};

#endif

```

In implementation `NIDAQHandler.cpp` most of the original C program is copied. There is no need to understand each detail. We maintain the function call and error recovery using the macro `DAQmxErrChk`. Instead of displaying the error message with `printf()`, we copy it into the string `errorDescription`, where it can be retrieved by Java.

```

// NIDAQHandler.cpp
// Must link with nidaqmxbase.lib

#include "NIDAQHandler.h"

// ----- Selector -----
WindowHandler *
selectWindowHandler(JNIEnv * env, jobject obj, int select)
{
    return new NIDAQAcquire1Scan(env, obj);
}

#define DAQmxErrChk(functionCall) \
    { if (DAQmxFailed(error = (functionCall))) { goto Error; } }

// ----- class NIDAQAcquire1Scan -----
// National Instruments NIDAQmx
// Prototype "acquire1Scan" from distribution of NI-DAQmx

NIDAQAcquire1Scan::NIDAQAcquire1Scan(JNIEnv * env,
                                      jobject obj)
    : WindowHandler(env, obj),
      samplesPerChan(1),
      timeout(10.0),
      pointsToRead(1)
{

```

```

period = JNIVar<int>(_exposedObj, "period").get();
chan = JNIString(_exposedObj, "channel").getUTF8();
min = JNIVar<double>(_exposedObj, "min").get();
max = JNIVar<double>(_exposedObj, "max").get();
}

NIDAQAcquire1Scan::~NIDAQAcquire1Scan()
{
    delete [] chan;
    if(taskHandle != 0)
    {
        DAQmxBaseStopTask(taskHandle);
        DAQmxBaseClearTask(taskHandle);
    }
}

int NIDAQAcquire1Scan::invoke(int val)
// Returns 0, if successful, -1, if error
{
    switch (val)
    {
        case 0: // Create task
            DAQmxErrChk(DAQmxBaseCreateTask("", &taskHandle));
            break;

        case 1: // Open channel
            DAQmxErrChk(
                DAQmxBaseCreateAIVoltageChan(taskHandle,
                    chan, "",
                    DAQmx_Val_RSE,
                    min,
                    max,
                    DAQmx_Val_Volts,
                    NULL));

            break;

        case 2: // Start task
            DAQmxErrChk(DAQmxBaseStartTask(taskHandle));
            break;

        case 3: // Get data
            DAQmxErrChk(
                DAQmxBaseReadAnalogF64(taskHandle,
                    pointsToRead,
                    timeout,
                    DAQmx_Val_GroupByChannel,
                    &data,
                    samplesPerChan,

```

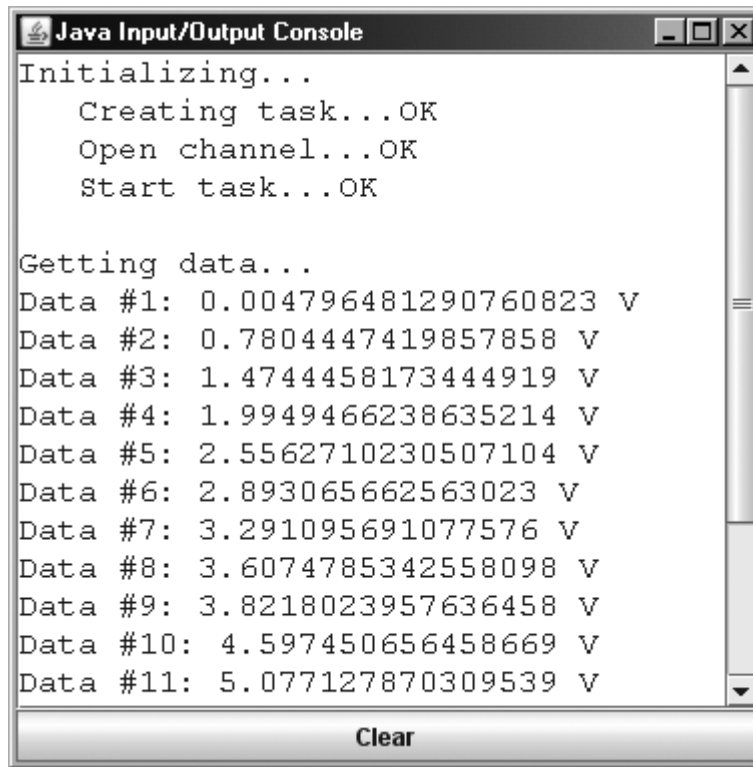
```

                                &pointsRead,
                                NULL));
    JNIVar<double>(_exposedObj, "value").set(data);
    break;

    case 4: // Sleep
        Sleep(period);
        break;
    }
    return 0;
Error:
    if (DAQmxFailed(error))
        DAQmxBaseGetExtendedErrorInfo(errBuff, 2048);
    if (taskHandle!=0)
    {
        DAQmxBaseStopTask(taskHandle);
        DAQmxBaseClearTask(taskHandle);
    }
    if (DAQmxFailed(error))
    {
        JNIString(_exposedObj, "errorDescription").setMBCS(errBuff);
        return -1;
    }
    return 0;
}

```

If everything goes well, the values of a slowly varying voltage from the ADC is displayed every second (Fig. 2.3).



```
Java Input/Output Console
Initializing...
  Creating task...OK
  Open channel...OK
  Start task...OK

Getting data...
Data #1: 0.004796481290760823 V
Data #2: 0.7804447419857858 V
Data #3: 1.4744458173444919 V
Data #4: 1.9949466238635214 V
Data #5: 2.5562710230507104 V
Data #6: 2.893065662563023 V
Data #7: 3.291095691077576 V
Data #8: 3.6074785342558098 V
Data #9: 3.8218023957636458 V
Data #10: 4.597450656458669 V
Data #11: 5.077127870309539 V

Clear
```

Fig. 2.3 Output from JawEx6 when the input voltage varies slowly

2.5.2 Porting contAcquireNScan

When the sample rate exceeds 1000 Hz, reading and processing each single data with a timer is not suitable. Entire blocks of data should be transferred from a acquisition module to a fast buffer (often implemented with DMA). The module hardware and the driver software must be designed accordingly, as it is the case with NI modules and the NI-DAQmx software. The sample program contAcquireNScan from the NI-DAQmx distribution shows how to proceed. With JAW it is rather simple to port the functionalities to Java. (For lack of space, the original code is not show here.)

Like a two channel oscilloscope, the application shows two input signals from the ADC at a sampling rate of 5 kHz (sampling period 0.2 ms) in a simple graphics window (using GPanel from the package ch.aplu.util). Most of the ideas comes from the previous example and are not explained again. We still initialize step-by-step and display the results in the title bar of the graphics window. A single data block (burst) contains 200 values of each channel and is retrieved from the native buffer by calling `invoke(5)`. We must ensure that this call is repeated every $200 * 0.2 \text{ ms} = 40 \text{ ms}$ on average, in order to avoid any buffer

overflow. When the system is heavily loaded by other processes, this value can be exceeded for a while without losing any data or getting incorrect values.

In order to terminate the application properly, an `ExitListener` is implemented, so that `notifyExit()` is called, when the close button of the graphics window is hit. In `notifyExit()` all necessary cleanup operations are done.

```
// JawEx7.java
// ContAcquireNScan from NIDAQmx distribution
// Acquire faster varying analog data and display on GPanel
// For demonstration with NI USB-6008 interface
// connect a first signal at terminals #2 and #32(GND)
// and a second signal at terminals #5 and #32(GND)
// both in range -10..10V

import java.awt.Color;
import ch.aplu.util.*;
import ch.aplu.jaw.*;

public class JawEx7 implements ExitListener
{
    // Transfer variables
    private String channel = "Dev1/ai0, Dev1/ai1"; // AI0 & AI1
    private double min = -10; // Measuring range -10..10V
    private double max = 10;
    private double sampleRate = 5000.0; // Samples per seconds
    private long samplesPerChannel = 200; // Samples per burst
    private String errorDescription =
        new String(); // Verbose error report
    private double[] data =
        new double[2*(int)samplesPerChannel]; // Data per burst
    // End of native variables

    private NativeHandler nh;
    private int time = 0;
    private int maxTime = 2000; // ms
    private double dt = 1000.0 / sampleRate; // ms
    private GPanel p = new GPanel(0, maxTime, min, max);
    private volatile boolean isRunning = false;
    private double offset0 = 4; // y-offset of AI0
    private double offset1 = -4; // y-offset of AI1
    private String title = "NI ADC - x: " + (maxTime / 10) +
        " ms/div; y: " + (max - min) / 10 + " V/div";

    public JawEx7()
    {
        p.title("Initializing...");
        drawCoordinateSystem();
        p.addExitListener(this);
    }
}
```

```

nh = new NativeHandler(
    "c:\\myjni\\nidaqc\\release\\nidaqc.dll",
    this);

if (isError(nh.invoke(0)))
    return;
if (isError(nh.invoke(1)))
    return;
if (isError(nh.invoke(2)))
    return;
if (isError(nh.invoke(3)))
    return;
if (isError(nh.invoke(4)))
    return;

p.title(title);
int pointsRead; // Number of samples in one burst
int oldTime = 0;
double oldEndData0 = 0;
double oldEndData1 = 0;
isRunning = true;
while (isRunning)
{
    while (time < maxTime)
    {
        // Get data block by block
        pointsRead = nh.invoke(5);
        if (pointsRead == 0)
        {
            p.title("Error while reclaiming data");
            isRunning = false;
            break;
        }

        // Draw signal AI0
        p.color(Color.red);
        p.move(oldTime, oldEndData0);
        for (int i = 0; i < pointsRead; i++)
            p.draw(time + i * dt, data[2*i] + offset0);
        oldEndData0 = data[2*(pointsRead-1)] + offset0;

        // Draw signal AI1
        p.color(Color.green);
        p.move(oldTime, oldEndData1);
        for (int i = 0; i < pointsRead; i++)
            p.draw(time + i * dt, data[2*i+1] + offset1);
        oldEndData1 = data[2*(pointsRead-1) + 1] + offset1;
    }
}

```



```

        time += pointsRead * dt;
        oldTime = time;
    }
    if (!isRunning)
        break;
    p.clear();
    drawCoordinateSystem();
    oldTime = 0;
    oldEndData0 = 0;
    oldEndData1 = 0;
    time = 0;
}
}

public void notifyExit()
{
    if (isRunning)
    {
        isRunning = false;
        p.title(title + " - stopped");
    }
    else
    {
        nh.destroy();
        System.exit(0);
    }
}

private void drawCoordinateSystem()
{
    p.color(Color.black);
    double xspan = maxTime / 10;
    for (int i = 0; i <= 10; i++)
        p.line(i * xspan, -10, i * xspan, 10);
    double yspan = (max - min) / 10;
    for (int i = 0; i <= 10; i++)
        p.line(0, min + i * yspan, maxTime, min + i * yspan);
}

private boolean isError(int rc)
{
    if (rc == -1)
    {
        p.title(errorDescription);
        nh.destroy();
        return true;
    }
    return false;
}

```

```

}

public static void main(String args[])
{
    new JawEx7();
}
}

```

As usual create a new C++ project named nidaqc and write the code with the original C code at hand. First write NIDAQHandler.h which is very similar to the previous example.

```

// NIDAQHandler.h

#ifndef __NIDAQHandler_h__
#define __NIDAQHandler_h__

#include "NIDAQmxBase.h"
#include "jaw.h"

// ----- class NIDAQContAcquireNScan -----
class NIDAQContAcquireNScan : public WindowHandler
{
public:
    NIDAQContAcquireNScan(JNIEnv * env, jobject obj);
    ~NIDAQContAcquireNScan();

private:
    int invoke(int tag);

    // Task parameters
    int32      error;
    TaskHandle taskHandle;
    char      errBuff[2048];
    int32      i;

    // Channel parameters
    const char* chan;
    float64    min;
    float64    max;

    // Timing parameters
    char      clockSource[128];
    uint64    samplesPerChan;
    float64    sampleRate;

    // Data read parameters
    int32      pointsToRead;
    int32      pointsRead;
}

```

```

float64    timeout;
};

#endif

```

The implementation is scarcely different from the previous example and should be self-explanatory by consulting the original code of `AcquireNScan.c`. We just point out how easy it is to fill the Java array by calling `invoke(5)`. You simply get the array pointer with

```

JNIArray<double> jary(_exposedObj, "data");
double * data = jary.get();

```

and pass it to `DAQmxBaseReadAnalogF64()` for filling the Java array.

```

// NIDAQHandler.cpp
// National Instruments NI-DAQmx
// Prototype contAcquireNScan.c from distribution of NI-DAQmx

#include "NIDAQHandler.h"

#define DAQmxErrChk(functionCall) \
    { if (DAQmxFailed(error = (functionCall))) { goto Error; } }

// ----- Selector -----
WindowHandler *
selectWindowHandler(JNIEnv * env, jobject obj, int select)
{
    return new NIDAQContAcquireNScan(env, obj);
}

// ----- class NIDAQContAcquireNScan -----
NIDAQContAcquireNScan::NIDAQContAcquireNScan(JNIEnv * env,
                                              jobject obj)
    : WindowHandler(env, obj),
      samplesPerChan(1),
      timeout(10.0),
      pointsToRead(1)
{
    chan = (_exposedObj, "channel").getUTF8();
    min = JNIVar<double>(_exposedObj, "min").get();
    max = JNIVar<double>(_exposedObj, "max").get();
    samplesPerChan =
        JNIVar<__int64>(_exposedObj, "samplesPerChannel").get();
    sampleRate =
        JNIVar<double>(_exposedObj, "sampleRate").get();
}

```

```

// Timing parameters
strcpy(clockSource, "OnboardClock");

// Data read parameters
pointsToRead = (int32)samplesPerChan;
}

NIDAQContAcquireNScan::~NIDAQContAcquireNScan()
{
delete [] chan;
if(taskHandle != 0)
{
DAQmxBaseStopTask(taskHandle);
DAQmxBaseClearTask(taskHandle);
}
}

int NIDAQContAcquireNScan::invoke(int val)
// Returns 0, if successful, -1, if error
{
switch (val)
{
case 0: // Create task
DAQmxErrChk(DAQmxBaseCreateTask("", &taskHandle));
break;

case 1: // Open channel
DAQmxErrChk(
DAQmxBaseCreateAIVoltageChan(taskHandle,
chan, "",
DAQmx_Val_RSE,
min, max,
DAQmx_Val_Volts,
NULL));

break;

case 2: // Configure internal clock
DAQmxErrChk(
DAQmxBaseCfgSampClkTiming(taskHandle,
clockSource,
sampleRate,
DAQmx_Val_Rising,
DAQmx_Val_ContSamps,
samplesPerChan));

break;

case 3: // Allocate buffer

```

```

    DAQmxErrChk(
        DAQmxBaseCfgInputBuffer(taskHandle, 200000));
    break;

case 4: // Start task
    DAQmxErrChk(DAQmxBaseStartTask(taskHandle));
    break;

case 5: // Acquire data
    JNIArray<double> jary(_exposedObj, "data");
    double * data = jary.get();

    DAQmxErrChk(
        DAQmxBaseReadAnalogF64(taskHandle,
                                pointsToRead,
                                timeout,
                                DAQmx_Val_GroupByScanNumber,
                                data,
                                2*(int)samplesPerChan,
                                &pointsRead,
                                NULL));

    return pointsRead;
}
return 0;

Error:
if (DAQmxFailed(error))
    DAQmxBaseGetExtendedErrorInfo(errBuff, 2048);
if (taskHandle!=0)
{
    DAQmxBaseStopTask(taskHandle);
    DAQmxBaseClearTask(taskHandle);
}
if (DAQmxFailed(error))
{
    JNIString(_exposedObj, "errorDescription").
        setMBCS(errBuff);
    return -1;
}
return 0;
}

```

When connecting two low-frequency signal generators to the ADC module, a graphics like fig. 2.4 is displayed.

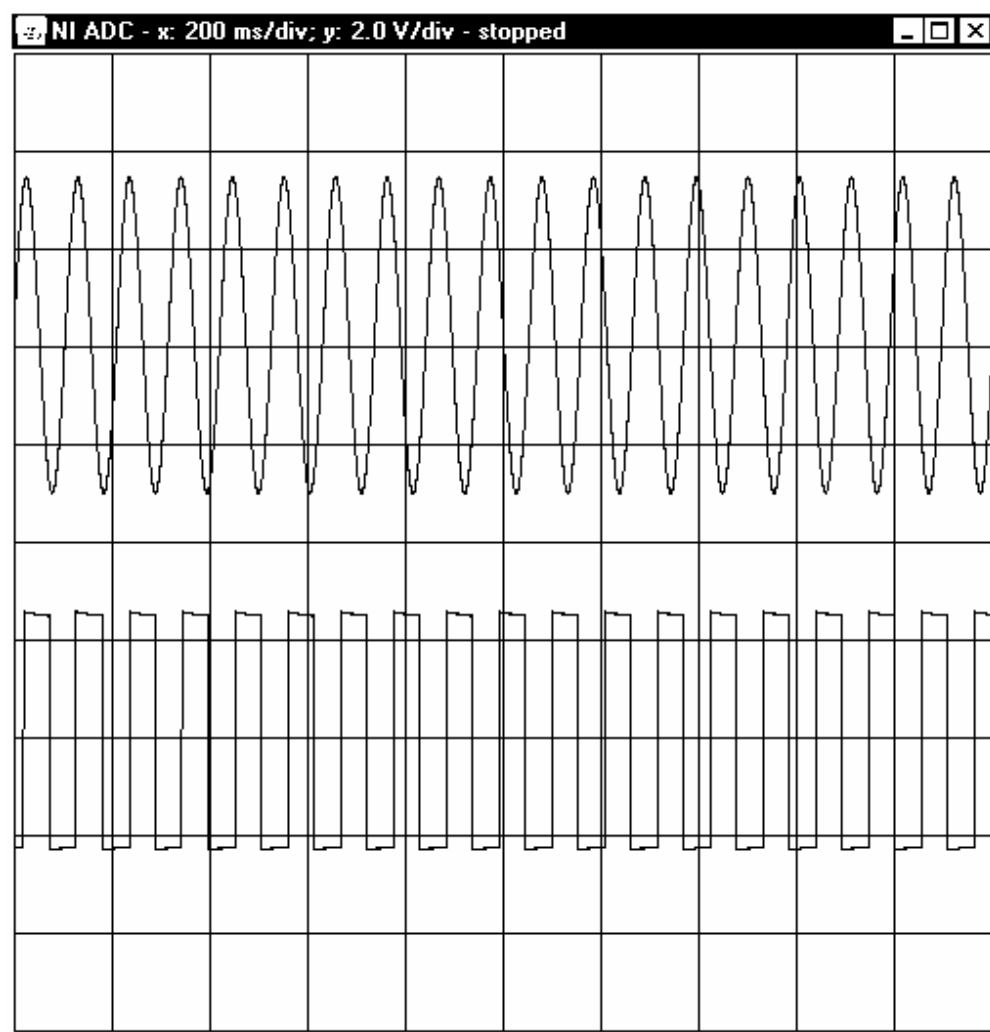


Abb. 2.4 Two channel oscilloscope with JawEx7

2.6 Data transfer using JNIBuffer

In analogy to streams, most drivers for typical measurement interfaces have three fundamental operations: initializing, (*open operation*), reading data from input ports/writing data to output ports (*read/write operations*) and releasing resources (*close operation*). If the distribution contains a DLL and its accompanying import library

(.lib) and a documentation of the function calls is available, with JAW you easily integrate these interfaces into Java without any knowledge of the cumbersome JNI.

If you do not have the import library, you may create it from the DLL with a utility normally called implib. A version of this tool is found in the distribution of the free Borland C++ compiler (found with a internet search machine and the keywords "free borland compiler")

In the next example we show the typical procedure to develop a Java application to acquire data from a measurement device. Similar to the previous examples a class derived from `WindowHandler` is declared. The constructor initializes (opens) the driver, the destructor closes it. For slow acquisition rates (1 Hz or lower) we acquire single values by calling `invoke()` and read the data back to Java via instance variables. For the clocking a Java timer is used. For faster acquisition rates (to a maximum of 1000 Hz) it is preferable to use a native timer, that runs in a high priority native thread independent (asynchronous) of any Java thread. In order to avoid overflows when the CPU is heavily charged, the acquired data should be buffered in a native FIFO buffer (called `JNIBuffer`) and not passed directly to Java. Because the implementation of circular FIFO buffers is not straightforward, JAW offers its own implementation as follows:

In your Java application create an array of the desired data type, e.g. for characters

```
char[] values = new char[5];
```

The size of this buffer is determined by the number of data that you want to transfer from C++ to Java in one junk, but has nothing to do with the size of the native `JNIBuffer`. You create the `JNIBuffer`, that contains 18 elements with an instance `nh` of `NativeHandler`

```
nh.createBuf(values, 18);
```

If you omit the size parameter, a default of 1000 applies. In C++ you fill the buffer with `write<type>` methods, e.g. for characters

```
writeChar('A');  
writeChar('B');  
writeChar('C');  
writeChar('D');  
writeChar('E');
```

`readBuf()` transfers the given number of data (or all available data) from the `JNIBuffer` to the Java array, e.g.

```
readBuf(2);
```

requests 2 elements from the `JNIBuffer`. If less than 2 elements are available, only these are transferred to the Java array. `readBuf()` returns the number of elements actually transferred. Since we have 5 characters in the `JNIBuffer`, we get 2 and the "oldest" two characters, 'A' and 'B', are transferred.

Since the `JNIBuffer` is implemented as circular FIFO buffer, its head pointer is advanced for two elements. Java accesses the transferred elements through `value[0]` and

value [1] . Fig. 1.5 illustrates this process (of course the empty fields are not really empty but contains invalid data).

Since JNIBuf is a circular buffer, there is never a memory overflow. When the buffers fills up, an overflow flag is set and subsequent data are lost. The state of this flag may be requested (and reset) by isBufOverflow().

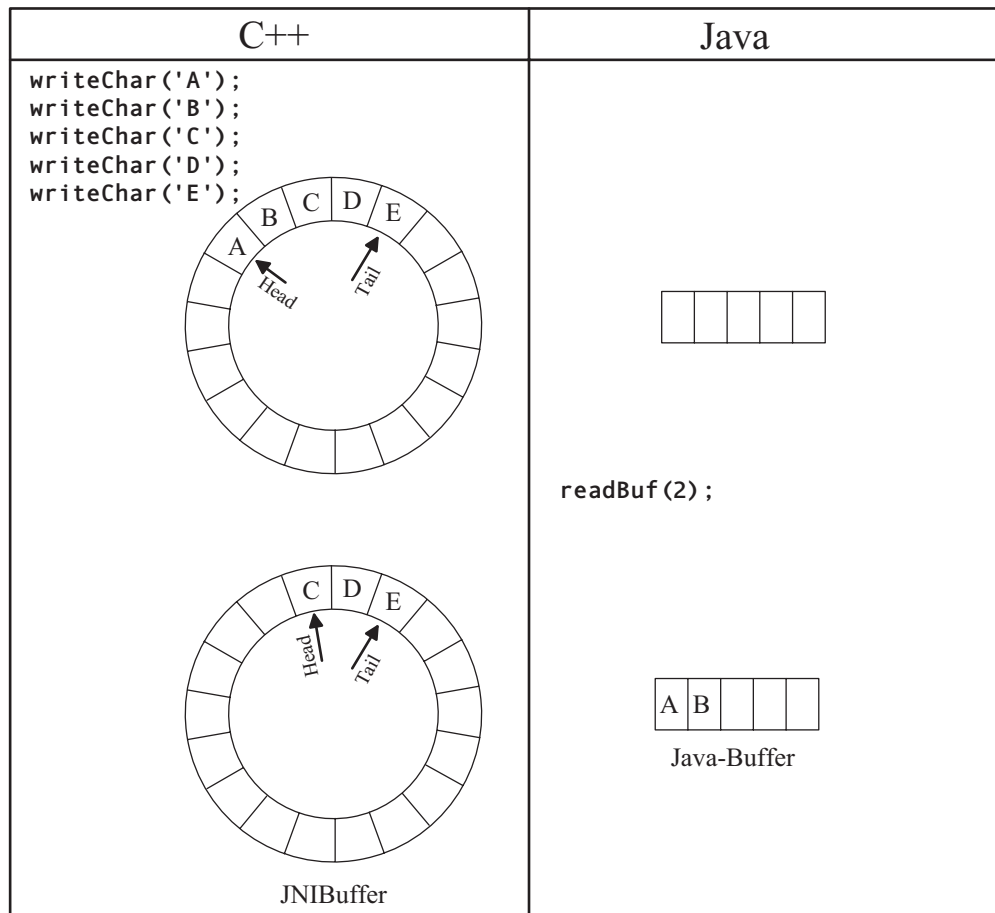


Fig. 2.5 JAW FIFO buffer (JNIBuffer) of size 18

In the next example you learn how this works. We use a low-cost interface USB-AD12 from BMC (<http://www.bmc.de>), but if you do not have this module, you can define

```
#define simulation
```

and the programs runs in a simulation mode.

The module *USB-AD12* has 16 analog input channels (12 bits, +- 5V, single ended), one output channel (12 bits) and 4 digital I/O channels. The function library is freely available from the manufacture's homepage (search machine with keywords *libad4 sdk*). It contains the DLL *libad4.dll* and its import library *libad4.lib*. The USB driver called *install-drivers-<version>.exe* must be installed separately.

Like in the previous example, the acquired data are displayed in a simple graphics window (GPanel from package *ch.aplu.util*). A native timer that runs in a native high-priority thread, requests a new voltage value every 20 ms. In the title bar of the graphics window we show the number of remaining data elements in the *JNIBuffer*. Because this number can exceed 100 when the system is heavily loaded, you realize the need of a native FIFO buffer. In the reading loop we call *readBuf(1)* and transfer only one element, if one is available.

```
// JawEx8.java

import ch.aplu.jaw.*;
import ch.aplu.util.*;

public class JawEx8 implements ExitListener
{
    // Native variables
    private String device = "usb-ad";
    private int period = 20; // ms
    private int maxTime = 10000; // ms
    private double range = 5; // +- 5V
    // End of native variables

    private NativeHandler nh;
    private GPanel p = new GPanel(0, maxTime, -range, range);

    public JawEx8()
    {
        p.addExitListener(this);
        nh = new NativeHandler(
            "c:\\myjni\\bmc\\release\\bmc.dll", this);
        double[] values = new double[1]; // Array of 1 double
        nh.createBuf(values); // JNIBuffer with default bufSize
        nh.startThread(); // Start high priority thread

        int time = 0;
        while (true)
        {
            int nbPending = nh.countBuf();
            // Show number of pending data on heavily loaded systems
            if (time > 0 && time % 200 == 0)
                p.title("Buffer Count: " + nbPending);

            // Read exactly one element from buffer
            if (nh.readBuf(1) == 1)
```

```

    {
        if (time == 0)
        {
            p.clear();
            drawCoordinateSystem();
            p.move(time, values[0]);
        }
        else
            p.draw(time, values[0]);
        time += period;
        if (time > maxTime)
            time = 0;
    }
    // Sleep 1 ms if few pending data
    // in order to decrease CPU load substantially
    if (nbPending < 2)
        Console.delay(1);
}

private void drawCoordinateSystem()
{
    double xspan = maxTime / 10;
    for (int i = 0; i <= 10; i++)
        p.line(i * xspan, -10, i * xspan, 10);
    double yspan = 2*range / 10;
    for (int i = 0; i <= 10; i++)
        p.line(0, -range + i * yspan,
            maxTime, -range + i * yspan);
}

public void notifyExit()
{
    nh.destroy();
    System.exit(0);
}

public static void main(String args[])
{
    new JawEx8();
}
}

```

We will not waste CPU time when no elements are ready to be transferred. Therefore we put the Java thread to sleep for 1 ms, if the number of available elements is less than 2. Because we read a new value every 20 ms and the JNIBuffer may contain up to 1000 elements, there is little risk for a buffer overflow when the system is busy.

On the C++ side create a new project name `bmc`. Consult the sample programs and the documentation distributed with `libad4`. This gives you enough information how to write the code. If you don't possess the ADC module, just look at the parts that compile when `simulation` is defined. First write `BMCHandler.h` and declare the class `AudioIn`:

```
// BMCHandler.h

#ifndef __BMCHandler_h__
#define __BMCHandler_h__

#include "jaw.h"
#include "libad.h"

// ----- class AnalogIn -----
class AnalogIn : public WindowHandler
{
public:
    AnalogIn(JNIEnv * env, jobject obj);
    ~AnalogIn();

protected:
    virtual bool wantThreadMsg(UINT uMsg);
    virtual void evThreadMsg(HWND hWnd,
                             UINT uMsg,
                             WPARAM wParam,
                             LPARAM lParam);

private:
    int period;
    int maxTime;
    double range;
    int32_t adHandle;
};

#endif
```

The constructor of `AnalogIn` fetches the Java variables and initializes the module with `ad_open()`. On the other hand we release all resources in the destructor by calling `ad_close()`. Remember that the constructor is executed when `NativeHandler` is instantiated and the destructor runs when `destroy()` is called.

From chapter 2 we know the role of `wantThreadMsg()` and `evThreadMsg()`. We use the occurrence of `WM_THREADSTART` to call the API function `SetTimer()` to setup a native timer with the given period. We release its resources with `KillTimer()` when `WM_TREADSTOP` occurs. On each timer event we get a `WM_TIMER` message. Then we read a new value from the interface by calling `ad_discrete_in()` or generate a value from a damped oscillation when running in `simulation` mode. `writelnDouble()` writes this

value into the JNIBuffer. There is no need of a notification to the Java program because Java fetches the values in the JNIBuffer asynchronously in its own thread.

Accessing Java variables or invoking callbacks in evThreadMsg() is not allowed, because the native thread and the JVM are not synchronized.

```
// BMCHandler.cpp
// Must link with libad4.lib, unless using simulation mode

#include "BMCHandler.h"
#include <math.h>

#define simulation

// ----- Selector -----
WindowHandler *
selectWindowHandler(JNIEnv * env, jobject obj, int select)
{
    return new AnalogIn(env, obj);
}

// ----- class AnalogIn -----
AnalogIn::AnalogIn(JNIEnv * env, jobject obj)
    : WindowHandler(env, obj)
{
    const char * driver =
        JNIString(_exposedObj, "device").getMBCS();
    period = JNIVar<int>(_exposedObj, "period").get();
    maxTime = JNIVar<int>(_exposedObj, "maxTime").get();
    range = JNIVar<double>(_exposedObj, "range").get();
#ifdef simulation
    adHandle = ad_open(driver);
#endif
    delete [] driver;
}

AnalogIn::~AnalogIn()
{
#ifdef simulation
    if (adHandle >= 0)
        ad_close(adHandle);
#endif
}

bool AnalogIn::wantThreadMsg(UINT uMsg)
{
    return (uMsg == WM_THREADSTART ||
            WM_TIMER ||
            WM_THREADSTOP);
}
```

```

}

void AnalogIn::evThreadMsg(HWND /*hwnd*/,
                           UINT message,
                           WPARAM /* wParam */,
                           LPARAM /* lParam */)
// Runs in JNIThread and not in the normal application thread
{
    #define ID_TIMER 1
    double value;
    static double time = 0;

    switch (message)
    {
        case WM_THREADSTART:
            SetTimer(0, ID_TIMER, period, (TIMERPROC)NULL);
            break;

        case WM_TIMER:
#ifdef simulation
            value = range*exp(-0.0004*time)*sin(0.005*time);
            time += period;
            if (time > maxTime)
                time = 0;
#else
            int32_t channel = 1;
            uint32_t data;
            ad_discrete_in(adHandle,
                          AD_CHA_TYPE_ANALOG_IN | channel,
                          0, &data);
            // Interpret data as 32-bit 2's complement
            // and convert to volts
            value = range * (int)data / (double)0x80000000;
#endif
            // Copy to JNIBuffer
            writeDouble(value);
            break;

        case WM_THREADSTOP:
            KillTimer(0, ID_TIMER);
            break;
    }
}

```

When running the simulation, the damped oscillation is shown. In the title bar you find the number of available values in the JNIBuf waiting to be fetched by Java. Normally this number is 0 or 1, but when the system is heavily loaded (to do so, start a big application),

this number may increase over 100. Because all the stored values are fetched later on, the curve is somewhat delayed but displayed without any error (Fig. 2.5).

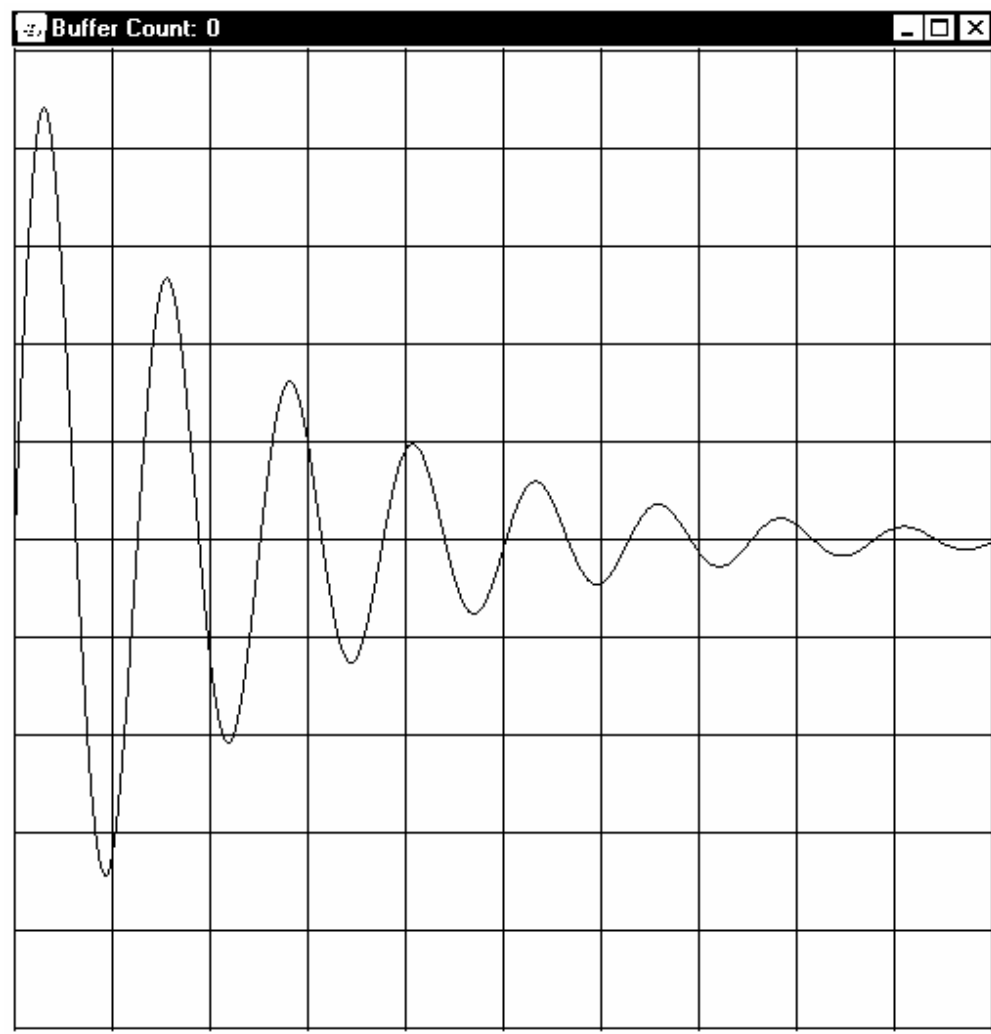


Fig. 2.5 *JawEx8 in simulation mode*

3 Apendix: Installation of Microsoft Visual Studio

Microsoft Visual Studio is a development suite for creating programs running under Microsoft Windows. Several languages are supported, essentially *Visual Basic*, *Visual C++* and *Visual C#*. All of them uses the same Integrated Development Environment (IDE).

A promotion version *Visual Studio Express Edition* is available free of charge. (You find it with a internet search machine and the keywords *Visual Studio Express Edition download*.) Despite of the somewhat reduced functionality compared to the commercial versions (lack of classes, no resource editor, etc.), it is fully adequate for developing applications with the JAW framework. For JAW only Visual C++ is needed.

After downloading and installing the Visual C++ part of Visual Studio you better test the installation by creating a sample application. As everywhere on the world, it is the *Hello World* that writes a greeting message to a console window. Proceed as follows:

- Select *File / New / Project*. A dialog appears
- Select in pane *Project types: Win32* and in pane *Templates: Win32 Console Application*. Type in text field *Name* whatever you like, e.g. *HelloWorld* and click *OK*
- In the first dialog of the Win32 Application Wizard click *Next*. Under Application Type select *Console application* and under *Additional options Empty project*. Click *Finish*
- In the Solution Explorer pane right-click on *Source Files* and select *Add / New Item*
- In the dialog select in pane *Templates: C++ File* und enter a name, e.g. *HelloWorld*. Click *Add* and the program editor will open. Enter the following code:

```
// HelloWorld.cpp
#include <iostream>

using namespace std;

void main()
{
    cout << "Hello World" << endl;
}
```

- In the toolbar's list box that contains *Debug*, select *Release* and compile/link using the menu option *Build / Build solution*
- *HelloWorld.exe* will be created in the subdirectory *Release* of the project folder. Run it with the menu option *Debug / Start Without Debugging* (or *Ctrl-F5*)

JAW uses the Win32-API and does not support the .NET framework. This makes it easy to create the DLLs with any other C++ compiler/linker such as Borland C++, gcc etc. If another compiler/linker or an old version of VC++ is used, it may be necessary to link with a slightly adapted version of `jaw.lib`. More information about availability of older or non VC-versions of the JAW library is obtained from www.aplu.ch/jaw.