JNI (Java Native Interface)

und

JAW (Java API Wrapper)

Aegidius Plüss

Supplement zum Buch: Plüss, Java exemplarisch, ISBN 3-486-20040-2

Inhalt

1 Das Java Native Interface (JNI) 3

1.1 Einführung 3

1.2 Aufruf von Funktionen einer DLL 4

1.3 Entwicklungsschritte beim JNI 5

1.4 Ausschreiben von Debug-Informationen 10

1.5 Wertübergabe mit Funktionsparametern und Funktionsrückgabewerten 10

1.6 Wertübergabe mit Instanzvariablen 13

1.7 Aufruf von Java-Methoden (Callbacks) 16

2 Objektorientiertes JNI mit dem Proxy-Entwurfsmuster 21

3 JAW (Java API Wrapper) 25

3.1 Konzept des Frameworks 25

3.2 Installation von JAW und erstes Anwendungsbeispiel 26

3.3 Verwendung einer Third-Party-Bibliothek 30

3.4 Verwendung nativer Fenster 35

3.5 Callbacks mit JNIMethod<type> 39

3.6 Transfer von Strings und Arrays 46

3.7 Verwendung von Echtzeit-Messgeräte-Modulen von National Instruments 51

3.7.1 Portierung von acquire1Scan 52

3.7.2 Portierung von contAcquireNScan 60

3.8 Datentransfer mit JNIBuffer 69

4 Anhang: Installation von Microsoft Visual Studio 78

1 Das Java Native Interface (JNI)

1.1 Einführung

Das Java Native Interface (JNI) ist eine Schnittstelle, die Java mit plattformspezifischem (nativem) Code verbindet. JNI kann einerseits dazu benutzt werden, Funktionen aus einer vorhandenen nativen Funktionsbibliothek, beispielsweise aus dem umfangreichen API des Betriebssystems oder aus einer vorhandenen Software-Schnittstelle eines bestimmten Geräts oder Bus-Interfaces eines Dritthestellers aufzurufen. Nativer Code kann andererseits auch dazu benützt werden, eine Java Virtual Machine (JVM) zu starten, Java-Objekte zu erzeugen und deren Methoden aufzurufen. Als native Sprache ist C/C++ vorgesehen, da es aber eine Verbindung zwischen C und vielen anderen Programmiersprachen, insbesondere zu Assembler gibt, kann mit JNI auch eine Verbindung zu diesen Sprachen erstellt werden. Ein weiterer Grund für den Einsatz einer nativen Schnittstelle kann die höhere Ausführungsgeschwindigkeit des nativen Codes gegenüber Java-Code sein.

Es ist allgemein bekannt, dass das JNI eine low-level Implementierung besitzt und deshalb vom Java-Programmierer einen relativ hohen Einarbeitungsaufwand erfordert. Obschon Java und C++ objektorientierte Sprachen sind, ist das JNI als Verbindung der beiden nicht objektorientiert konzipiert. Es ist daher verständlich, dass es mehrere Lösungsansätze gibt, das JNI so zu erweitern, dass sich eine direkte Entsprechung von Java- und C++-Klassen ergibt.

Einige bekannte Produkte sind:

Jace *(Open source, http://sourceforge.net/projects/jace)*
Merkmale: Typensichere Datenübergabe mit automatischer Generierung der C++-Headerdateien und Proxy Klassen

JNIWrapper *(www.jniwrapper.com)*
Merkmale: Direkter Aufruf von vorhandenen exportierten DLL-Funktionen, keine C++-Compilation nötig

JunC++ion *(http://www.codemesh.com)*
Merkmale: Generierung von C++-Proxy-Klassen

JAW *(http://www.aplu.ch/jaw)*
Merkmale: Verwendung von Windows API Funktionen, leichte Übernahme von vorhandenen C/C++-Bibliotheken, Echtzeitunterstützung

Beim Einsatz von JNI verliert man die Plattformunabhängigkeit von Java, gewinnt aber an Flexibilität und Performanz durch den Einsatz von hardware- und betriebssystemnahem Programmcode. Im Bereich der Mess- und Steuerungstechnik werden die zu den Messgeräten und Echtzeitinterfaces gehörenden APIs fast ausschließlich für C/C++ und fast

nie für Java angeboten. Mit JNI ist es möglich, diese Bibliotheken auch von Java aus einzusetzen.

Es folgt eine exemplarische Einführung in das JNI unter Verwendung von C/C++ auf der Windows-Plattform. Zum Verständnis genügen Grundkenntnisse in Java und C++. Es werden nicht alle Möglichkeiten von JNI abgedeckt, insbesondere fehlt das Invocation API, mit dem man eine JVM starten, Java-Objekte erzeugen und ihre Methoden aufrufen kann. Mehr dazu findet man mit einer Suchmaschine mit den Stichworten *Java Native Interface Specification*.

1.2 Aufruf von Funktionen einer DLL

Die nativen Funktionen, welche von Java aus verwendet werden, müssen unter Windows in eine Dynamic Link Library (**DLL**) verpackt werden. Aus diesem Grund ist es wichtig, den Aufruf einer Funktion, die sich in einer DLL befindet, zu verstehen. Dies wird zuerst an einem in C++ geschriebenen Programm `prog.cpp` erläutert, das Funktionen in der DLL `mylib.dll` aufruft. Da die DLL erst zu Laufzeit, also dynamisch, in den Speicherbereich geladen wird, stellt sie ihre Funktionen über Funktionszeiger (**entry points**) zur Verfügung, deren Wert beim Laden gesetzt wird. Ein C/C++-Programm, welches diese Funktionen benützen möchte, wird mit einer zur DLL gehörenden **Import-Library** (LIB-Datei) gelinkt, das die Funktionszeiger kennt. Die Import-Library kann jederzeit mit einem speziellen Werkzeug (**implib**) aus der DLL gewonnen werden. Die C-Prototypen der Funktionen sind üblicherweise in einer Include-Datei (**Headerdatei**) enthalten.

Die verschiedenen Schritte zur Erstellung eines ausführbaren Programms, welches Funktionen aus einer DLL-Bibliothek verwendet, sind aus Abb. 1.1 ersichtlich. Sie zeigt die Situation, wo der Programmierer die Bibliotheksroutinen, die sich in `mylib.dll` befinden, aufruft. Die Prototypen der Funktionen befinden sich in der Headerdatei `mylib.h`. In der Quelldatei `prog.cpp` können diese als bekannt vorausgesetzt werden, da diese die Headerdatei einbindet. In der Import-Library `mylib.lib` ist lediglich ein kleines Codestück, damit das Programm korrekt gelinkt wird. Zu Laufzeit sorgt dieses Codestück dafür, dass die entsprechende Funktion, deren Code sich in der DLL `mylib.dll` befindet, ausgeführt wird.

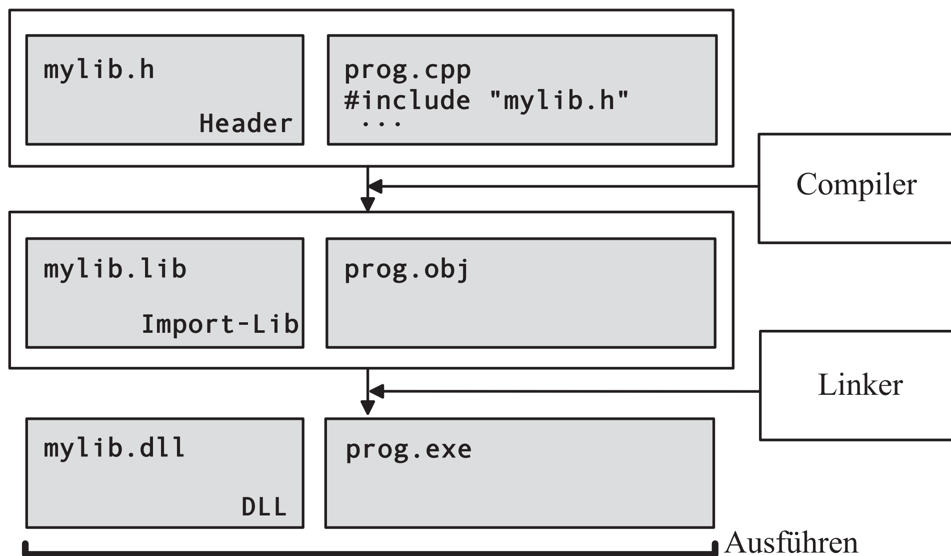

Abb. 1.1 Klassische Programm-Entwicklung eines ausführbaren Programms mit einer externen Bibliotheks-DLL

Das JNI verwendet den gleichen Mechanismus, wobei die JVM den Aufruf der nativen Funktion übernimmt.

1.3 Entwicklungsschritte beim JNI

Um die verschiedenen Entwicklungsschritte zu erläutern, gehen wir von folgendem exemplarischen Problem aus: Einer Java-Applikation soll eine native Methode `beep(int frequency)` mit einem int-Parameter zur Verfügung gestellt werden. Beim Aufruf soll die Windows-API-Funktion `Beep(int frequency, int duration)` (frequency in Hertz, duration in ms) mit duration = 500 und der übergebenen Frequenz im Bereich 100..10000 Hz ausgeführt werden. Die einzelnen Schritte sind aus Abb. 1.2 ersichtlich.

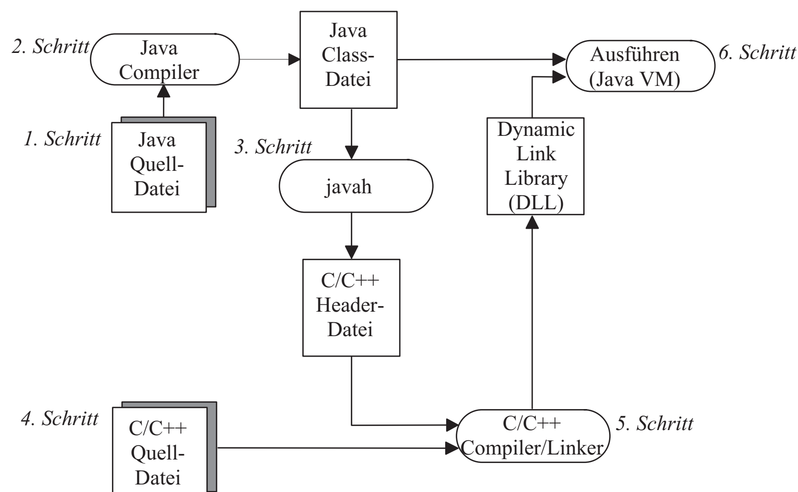

Abb. 1.2 Die Entwicklungsschritte zur Einbindung einer DLL mit dem JNI

Im ersten Schritt schreiben wir in einer Java-IDE den Quellcode von `JniEx1.java`. Die gewünschte Frequenz wird in einer `JOptionPane` eingelesen, wobei Falscheingaben (kein Integer, Wert außerhalb des Bereichs) auf typische Art mit einer Exception zurückgewiesen werden.

```
// JniEx1.java

import javax.swing.JOptionPane;

public class JniEx1
{
  private native void beep(int frequency);

  public JniEx1()
  {
    System.load("c:\\myjni\\beep\\release\\beep.dll");
    String valueStr = "";
    int value;

    while (valueStr != null)
    {
      valueStr = JOptionPane.showInputDialog(
        null, "Frequenz in Hertz? (100...10000)",
        "Windows System Beep",
        JOptionPane.QUESTION_MESSAGE);
      try
      {
        value = Integer.parseInt(valueStr);
```

```
        if (value < 100 || value > 10000)
          throw new NumberFormatException();
      }
      catch (NumberFormatException ex)
      {
        continue;
      }
      beep(value);
    }
    System.exit(0);
  }

  public static void main(String args[])
  {
    new JniEx1();
  }
}
```

Was besonders auffällt, ist das Schlüsselwort `native` in der Methodendeklaration der Methode `beep()`. Es fehlt, analog zu einer abstrakten Methode, der Körper, was verständlich ist, da `beep()` ja als nativer Code ausgeführt werden soll. Damit die JVM die native Methode `beep()` in der DLL findet, muss diese in den Hauptspeicher **geladen** werden. Dazu stehen uns in der Klasse `System` zwei statische Methoden `loadLibrary()` und `load()` zur Verfügung. `loadLibrary()` erhält als Parameter lediglich den Dateinamen der Bibliothek ohne Erweiterung. Beim Laden wird im Verzeichnis der Java-Classdatei und nachher im Pfad des Betriebssystems nach einer Bibliotheksdatei mit dem angegebenen Namen und (unter Windows) der Erweiterung `.dll` gesucht. (Üblicherweise werden DLLs in das Unterverzeichnis `system32` kopiert, das sich immer im Pfad befindet.) Der Methode `load()` übergibt man den vollständig qualifizierten Dateinamen inklusive Laufwerk. (Die Groß-Kleinschreibung ist unter Windows bei den Pfadangeben unwesentlich.) Wir werden nachfolgend dafür sorgen, dass sich `beep.dll` im Verzeichnis `c:\myjni\beep\release` befindet und schreiben daher zum Laden der DLL

```
System.load("c:\\myjni\\beep\\release\\beep.dll");
```

als erste Zeile in den Konstruktor.

`JniEx1.java` lässt sich im zweiten Schritt problemlos compilieren, hingegen ergibt die Ausführung die Exception

```
Exception in thread "main" java.lang.UnsatisfiedLinkError:
Can't load library: e:\myjni\beep\release\beep.dll
```

was uns wenig erstaunt, da die DLL noch nicht vorhanden ist.

Im dritten Schritt erzeugen wir mit dem Kommandozeilen-Programm `javah`, das im JDK enthalten ist, aus der Classdatei `JniEx1.class` die C/C++-Headerdatei `JniEx1.h`. Dabei setzen wir voraus, dass sich das bin-Verzeichnis der verwendeten JDK im Pfad von Windows befindet. Wir wechseln in einer Command-Shell in das Verzeichnis, das `JniEx1.class` enthält, und schreiben

```
javah JniEx1
```

Mit einem Texteditor, z.B. dem Editor der verwendeten Java-IDE öffnen wir anschließend das dabei erzeugte `JniEx1.h`.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JniEx1 */

#ifndef _Included_JniEx1
#define _Included_JniEx1
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     JniEx1
 * Method:    beep
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_JniEx1_beep
  (JNIEnv *, jobject, jint);

#ifdef __cplusplus
}
#endif
#endif
```

Den maschinenerzeugten Code brauchen wir glücklicherweise nicht in allen Einzelheiten zu verstehen. Wir erkennen aber, dass aus der Java-Methode `beep()` ein C-Prototyp wie folgt erzeugt wurde:

```
JNIEXPORT void JNICALL
Java_JniEx1_beep(JNIEnv *, jobject, jint);
```

Aus dem einzigen int-Parameter der Java-Methode `beep()` sind also in C/C++ drei Parameter geworden: Der Zeiger mit dem Typ `JNIEnv *` dient als Interface zwischen der JVM und C/C++. Der zweite mit dem Typ `jobject` liefert eine Objektreferenz der Java-Klasse, in der der Aufruf erfolgt. Beide werden gebraucht, um von C/C++ auf Instanzvariablen und Methoden dieser Klasse zuzugreifen, was in unserem Beispiel unnötig ist. Der dritte Parameter mit dem Typ `jint` liefert uns den aktuellen Aufrufparameter. Tab. 1 zeigt die Entsprechung von Java- und C/C++-Datentypen. Die Unterscheidung zwischen

nativem Typ und C/C++-Typ ist deswegen nötig, da einige Basistypen in C/C++ plattform- bzw. compilerabhängig sind, insbesondere der Character- und der 64-bit-lange Integertyp.

Java Typ	**Nativer Typ**	**C/C++ Typ (32 bit Windows)**
void	void	void
boolean	jboolean	bool
char	jchar	wchart_t
byte	jbyte	signed char
short	jshort	short
int	jint	int
long	jlong	__int64
float	jfloat	float
double	jdouble	double

Tab. 1: Entsprechung der primitiven Datentypen (Basistypen)

Im vierten Schritt erstellen wir die in C++ geschriebene DLL. Wir gehen davon aus, dass *Microsoft Visual Studio 2008 Express Edition* (deutsche Version) gemäß den Angaben im Anhang installiert wurde. Wir erzeugen zuerst mit dem Projekttyp *Win32* und der Vorlage *Win32-Projekt* ein neues Projekt in `c:\myjni` mit dem Namen `beep`. Da wir eine DLL erstellen wollen, müssen wir im Dialog *Anwendungseinstellungen* als *Anwendungstyp | DLL* und unter *Zusätzliche Optionen | Leeres Projekt* auswählen. In der Listbox der Werkzeug-Leiste stellen wir *Release* ein.

Im Projektmappen-Explorer wählen wir mit einem rechten Mausklick unter *Quelldateien | Hinzufügen | Neues Element | C++-Datei* und nennen die neue Datei `beep.cpp`. Im Editor schreiben wir folgenden Quellcode:

```
// beep.cpp

#include <windows.h>
#include <jni.h>
#include "JniEx1.h"

JNIEXPORT void JNICALL
Java_JniEx1_beep(JNIEnv *, jobject, jint frequency)
{
  Beep(frequency, 500);
}
```

Vor der Compilation müssen wir noch die Headerdatei `JniEx1.h` in das Verzeichnis, in dem sich mit `beep.cpp` befindet, kopieren.

Im fünften Schritt compilieren und linken wir mit der Menuoption *Erstellen*, wobei `beep.dll` im Verzeichnis `c:\myjni\beep\Release` erzeugt wird.

Schließlich führen wir im sechsten und letzten Schritt `JniEx1` aus und freuen uns, dass das Programm wunschgemäß funktioniert.

1.4 Ausschreiben von Debug-Informationen

Bei der Suche nach Laufzeitfehlern ist es oft hilfreich, Parameter- oder Variablenwerte auf einer Console auszuschreiben. Dazu eignet sich der C++-IOStream `cout` besonders gut. Zu seiner Verwendung ist es nötig, `iostream` zu includen. Statt immer `std::cout` zu schreiben, kann `using namespace std` verwendet werden. Soll beispielsweise der übergebene Parameter `frequency` auf der Console erscheinen, so schreiben wir

```
// beep.cpp

#include <windows.h>
#include <jni.h>
#include "JniEx1.h"
#include <iostream>

using namespace std;

JNIEXPORT void JNICALL
Java_JniEx1_beep(JNIEnv *, jobject, jint frequency)
{
  cout << "frequency = " << frequency << endl;
  Beep(frequency, 500);
}
```

1.5 Wertübergabe mit Funktionsparametern und Funktionsrückgabewerten

Am vorhergehenden Beispiel habe wir bereits gelernt, wie man Werte von Java an C++ mit Hilfe von Methodenparameter übergibt. Die Rückgabe von Werten von C++ an Java erfolgt am einfachsten über den Rückgabewert von aufgerufenen nativen Methoden. Wir zeigen dies an einem Beispiel, bei dem die Windows-Registry ausgelesen wird, was direkt mit Java nicht möglich ist. Wir deklarieren drei native Methoden:

- `openKey()` erhält als Parameter den Registryschlüssel als String, z.B. "HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0" und sucht nach diesem Schlüssel. Falls es ihn gibt, liefert die Methode den booleschen Wert `true`, im Fehlerfall `false` zurück

- `queryValue()` erhält als Parameter den Namen des Registry-Eintrags als String, z.B. "Path" und gibt den Wert eines Eintrags vom Typ `REG_SZ` (null terminierter String) zurück. Falls der Eintrag nicht existiert, wird `null` zurückgegeben
- `closeKey()` schließt den vorher geöffneten Registerschlüssel wieder (dabei werden interne Ressourcen freigegeben).

Wie es sich gehört, fangen wir die Fehler mit einer Fehlermeldung ab und beenden das Programm.

```
// JniEx2.java

import javax.swing.JOptionPane;

public class JniEx2
{
  private native boolean openKey(String key);
  private native String queryValue(String valueName);
  private native void closeKey();

  public JniEx2()
  {
    System.load("s:\\myjni\\reg\\release\\reg.dll");
    String key =
         "HARDWARE\\DESCRIPTION\\System\\CentralProcessor\\0";
    String valueName = "ProcessorNameString";

    if (!openKey(key))
    {
      JOptionPane.
        showMessageDialog(null,
                          "Can't open registry key:\n" + key,
                          "Fatal Error",
                          JOptionPane.ERROR_MESSAGE);
      System.exit(0);
    }
    String value = queryValue(valueName);
    if (value == null)
    {
      JOptionPane.
        showMessageDialog(null,
                          "Can't read registry entry:\n" +
                           valueName,
                          "Fatal Error",
                          JOptionPane.ERROR_MESSAGE);
      System.exit(0);
    }
    closeKey();
    JOptionPane.
```

```
      showMessageDialog(null,
                          "Processor Name: " + value,
                          "Registry Read",
                          JOptionPane.INFORMATION_MESSAGE);
  }

  public static void main(String args[])
  {
    new JniEx2();
  }
}
```

Nach der Java-Compilation erzeugen wir wiederum mit `javah` die Headerdatei `JNIEx2.h` und entnehmen daraus die Prototypen der nativen Funktionen. Wie vorhin erzeugen wir ein neues C++-Projekt und kopieren die Prototypen mit *Kopieren&Einfügen* in die Quelldatei `reg.cpp`. Nachher ergänzen wir den C++-Code unter Beizug einer Win32-Api-Dokumentation.

(*Informationen über das Win32-API findet man im MSDN, das zusammen mit Visual Studio installiert werden kann. Eine gute Übersicht findet man in der Hilfedatei win32.hlp, die man von www.aplu.ch/links downloaden kann*.)

```
// reg.cpp

#include <windows.h>
#include <jni.h>
#include "JniEx2.h"

HKEY keyHandle;

JNIEXPORT jboolean JNICALL
Java_JniEx2_openKey(JNIEnv * env, jobject, jstring key)
{
  // Get Java string in C char array
  const char * c_key = env->GetStringUTFChars(key, 0);

  long rc =
    RegOpenKeyA(HKEY_LOCAL_MACHINE, c_key, &keyHandle);

  // Release memory used to hold ASCII representation
  env->ReleaseStringUTFChars(key, c_key);

  // Return true if success, false if error
  return (rc == ERROR_SUCCESS) ? true : false;
}

JNIEXPORT jstring JNICALL
Java_JniEx2_queryValue(JNIEnv * env, jobject,
```

```
                       jstring valueName)
{
  unsigned char buf[200];
  DWORD bufsize = sizeof(buf);
  DWORD type;

  // Convert Java string to C string
  const char * c_valueName =
    env->GetStringUTFChars(valueName, 0);

  long rc =
    RegQueryValueExA(keyHandle, c_valueName, NULL, &type,
                     buf, &bufsize);

  // Release memory used to hold ASCII representation
  env->ReleaseStringUTFChars(valueName, c_valueName);

  // Return converted C string or null, if error
  return (rc == ERROR_SUCCESS) ?
    env->NewStringUTF((const char *)buf) : 0;
}

JNIEXPORT void JNICALL
Java_JniEx2_closeKey(JNIEnv *, jobject)
{
  RegCloseKey(keyHandle);
}
```

Mit der JNI-Funktion `GetStringUTFChars()` holt man sich den Java-String aus der Variable `reg`. Der dabei allozierte Buffer muss mit `ReleaseStringUTFChars()` wieder freigegeben werden. An Stelle der API-Funktionen mit der Endung `A` können auch die Versionen ohne diese Endung verwendet werden. In diesem Fall müssen aber in Visual Studio die Projekteigenschaften wie folgt verändert werden: *Konfigurationseigenschaften | Allgemein | Zeichensatz | Multi-Byte-Zeichensatz verwenden*. (Die JNI-Funktionen sind in den *"Java Native Interface Specifications"* und den nachfolgend erschienenen *"JNI Enhancements"* beschrieben, die man mit einer Suchmaschine findet.)

1.6 Wertübergabe mit Instanzvariablen

Der native Code kann auch direkt lesend und schreibend auf Java-Instanzvariablen zugreifen. Dies ist dann besonders wichtig, wenn mehrere Werte mit unterschiedlichen Datentypen ausgetauscht werden. Im folgenden Beispiel `JniEx3.java` sollen eine int- und eine double-Instanzvariable bei jedem Aufruf von `increase()` um 1 erhöht werden. In einer `JOptionPane` wird der aktuelle Wert der Instanzvariablen ausgeschrieben. Man beachte,

dass die Instanzvariablen sogar `private` deklariert werden können und das JNI trotzdem lesenden und schreibenden Zugriff darauf hat.

```
// JniEx3.java

import javax.swing.JOptionPane;

public class JniEx3
{
  private int n = 0;
  private double x = 0;

  private native void increase();

  public JniEx3()
  {
    System.load("s:\\myjni\\inc\\release\\inc.dll");

    int rc = JOptionPane.YES_OPTION;
    while (rc == JOptionPane.YES_OPTION)
    {
      rc = JOptionPane.showConfirmDialog(null,
        "n = " + n + ";   x = " + x,
        "Inc",
        JOptionPane.OK_CANCEL_OPTION,
        JOptionPane.INFORMATION_MESSAGE);
      if (rc != JOptionPane.YES_OPTION)
        System.exit(0);
      increase();
    }
  }

  public static void main(String args[])
  {
    new JniEx3();
  }
}
```

Der zugehörige C++-Code verwendet sowohl den `JNIEnv *` - wie den `jobject`-Parameter.

```
// inc.cpp

#include <jni.h>
#include "JniEx3.h"

JNIEXPORT void JNICALL
Java_JniEx3_increase(JNIEnv * env, jobject obj)
```

```
{
  int _n;
  double _x;

  // Get class instance
  jclass clazz = env->GetObjectClass(obj);

  // Get field IDs
  jfieldID n_ID = env->GetFieldID(clazz, "n", "I");
  jfieldID x_ID = env->GetFieldID(clazz, "x", "D");

  // Read variables and cast to c-type
  _n = (int)env->GetIntField(obj, n_ID);
  _x = (double)env->GetDoubleField(obj, x_ID);

  // Perform operation
  _n++;
  _x++;

  // Cast to jni-type and write back to Java
  env->SetIntField(obj, n_ID, (jint)_n);
  env->SetDoubleField(obj, x_ID, (jdouble)_x);
}
```

Die einzelnen Schritte sind im Code als Kommentar beschrieben: Zuerst wird mit `GetObjectClass()` eine Instanz der zum Java-Objekt gehörenden Klasse geholt. Dann werden mit `GetFieldID()` Identifiers der Instanzvariablen bestimmt. Dabei muss der Name der Instanzvariablen (englisch auch `field` genannt) und eine Signatur für den Variablentyp angegeben werden, die man aus Tab. 2 entnimmt.

Signatur	**Beschreibung**
B	byte
C	char
D	double
F	float
I	int
J	long
S	short
V	void
Z	boolean

L<voll qualifizierte Klasse>;	voll qualifizierte Klasse
[<Signatur>	Array vom Typ mit Signatur
(<Parametersignatur-Liste>)<Signatur-Rückgabetyp)	Methodensignatur

Tab. 2: Die JNI-Signaturen (man beachte den Strichpunkt bei der Klassensignatur)

Mit dem erhaltenen Identifier kann anschließend mit Get<type>Field() lesend und nachher mit Set<type>Field() schreibend auf die Instanzvariablen zugegriffen werden. Der Funktionsname entspricht den jni-Typen gemäß Tab.3.

jni-Typ	**Get/Set Funktionsname**
jobject	GetObjectField/SetObjectField
jboolean	GetBooleanField/SetBooleanField
jbyte	GetByteField/SetByteField
jchar	GetCharField/SetCharField
jshort	GetShortField/SetShortField
jint	GetIntField/SetIntField
jlong	GetLongField/SetLongField
jfloat	GetFloatField/SetFloatField
jdouble	GetDoubleField/SetDoubleField

Tab. 3: Methodennamen von Get<type>Field() und Set<typeField()

Zur Sicherheit sollte man die jni-Typen auf die entsprechenden C-Typen casten und dabei auf die Plattformabhängigkeit der C-Typen achten, insbesondere bei jbyte, jchar, jshort und dem 64-bit langen jlong.

1.7 Aufruf von Java-Methoden (Callbacks)

Nativer Code kann Java-Methoden aufrufen, die sogar private deklariert sind. Der Mechanismus ist ähnlich wie beim Zugriff auf Instanzvariablen. Zuerst wird mit GetObjectClass() eine Klasseninstanz der Klasse geholt, in der die Methode enthalten ist. Einen Methoden-Identifier erhält man mit GetMethodID(). Schließlich erfolgt der Aufruf mit Call<type>Method(), wobei der Funktionsname dem Rückgabetyp gemäß Tab.4 entspricht.

jni-Typ	Funktionsname
void	CallVoidMethod
jobject	CallObjectMethod
jboolean	CallBooleanMethod
jbyte	CallByteMethod
jchar	CallCharMethod
jshort	CallShortMethod
jint	CallIntMethod
jlong	CallLongMethod
jfloat	CallFloatMethod
jdouble	CallDoubleMethod

Tab. 4: Methodennamen von Call<type>Method()

Eventuell vorhandenen aktuelle Methodenparameterwerte werden in der Parameterklammer angefügt. (Es handelt sich um eine C-Ellipse, d.h. eine nicht spezifizierte Zahl von Parametern.) Sie müssen den richtigen jni-Datentyp besitzen.

Ruft nativer Code eine Java-Methode auf, so spricht man auch oft von Callback. Dies ist besonders interessant bei asynchronen Notifikationen, d.h. Rückmeldungen vom nativen Code, währendem das Java-Programm weiter läuft. Grundsätzlich wird der native Code immer im Java-Thread ausgeführt, in dem der native Aufruf erfolgt (also nicht etwa in einem nativen Thread). Dies führt mitunter zu Schwierigkeiten, da das Programm solange blockiert, bis der native Code zu Ende gelaufen ist. Will man dies vermeiden, so muss der native Code entweder in einem eigenen nativen Thread ablaufen, was fortgeschrittene API-Programmierung oder ein Framework wie JAW voraussetzt, oder man erzeugt zusätzliche Java-Threads. Dies wird im folgenden Beispiel `JniEx4.java` illustriert. Eine länger dauernde Berechnung (Bestimmung der Anzahl Primzahlen in einem Bereich mit großen Zahlen) soll als C++-Programm ausgeführt werden. Während der Berechnung, die mehrere Sekunden dauern kann, soll fortlaufend die Rechenzeit ausgeschrieben werden. Am Ende der Berechnung ruft der native Code als Callback eine Notifikationsmethode auf, welche den Thread beendet und das Resultat ausschreibt.

```
// JniEx4.java

import java.util.*;
import ch.aplu.util.*;

public class JniEx4
{
```

```
  // Inner class
  private class StatusThread extends Thread
  {
    String msg = "Suche Anzahl Primzahlen im Bereich\n" +
                 start + " und " + end +
                 "\nBitte warten...";

    public void run()
    {
      mop.setText(msg);
      long startTime = new Date().getTime();
      while (isRunning)
      {
        int elapsed = (int)((new Date().getTime() -
                             startTime)/1000.0);
        mop.showTitle("Rechenzeit: " + elapsed + " s");
        Console.delay(1000);
      }
    }
  }

  // Native method
  private native void getNbPrimes();

  // Instance variables used by native code
  private long start = 1000000000000000L; // 10^15
  private long end   = 1000000000000200L;

  // Instance variables
  private volatile boolean isRunning = true;
  private ModelessOptionPane mop = new ModelessOptionPane("");
  private StatusThread statusThread;

  // Constructor
  public JniEx4()
  {
    System.load("s:\\myjni\\callback\\release\\callback.dll");
    statusThread = new StatusThread();
    statusThread.start();
    getNbPrimes();
  }

  // Callback method invoked by native code
  private void notify(int nbPrimes)
  {
    isRunning = false;
    mop.setText("Anzahl gefundener Primzahlen: " + nbPrimes);
  }
```

```
  public static void main(String args[])
  {
    new JniEx4();
  }
}
```

Sobald im Konstruktor die native Methode `getNbPrimes()` aufgerufen wird, blockiert der Java-Mainthread. Der Status wird daher mit einem eigenen Thread, dem `StatusThread`, in eine nicht-modale `ModelessOptionPane` aus der Klasse `ch.aplu.util` ausgeschrieben.

Der Primzahlsuchbereich wird in diesem Beispiel über zwei Instanzvariablen `start` und `end` an das native Programm übergeben. Die Rückgabe der Anzahl gefundener Primzahlen erfolgt über einen int-Parameter der Callbackmethode `notify()`. In `notify()` wird zuerst der `StatusThread` beendet, indem das boolesche Flag `isRunning` auf `false` gesetzt wird. Dies ist wie üblich `volatile` deklariert, da aus von zwei Threads verwendet wird.

Nachdem man mit `javah` die Headerdatei `JniEx4.h` erzeugt hat, schreibt man den nativen Code in `callback.cpp`.

```
// callback.cpp

#include <jni.h>
#include <iostream>
#include <windows.h>
#include "JniEx4.h"

using namespace std;

// Prototype
bool isPrime(__int64 n);

JNIEXPORT void JNICALL
Java_JniEx4_getNbPrimes(JNIEnv * env, jobject obj)
{
  __int64 start;
  __int64 end;

  // Get class instance
  jclass clazz = env->GetObjectClass(obj);

  // Get field IDs
  jfieldID startID = env->GetFieldID(clazz, "start", "J");
  jfieldID endID = env->GetFieldID(clazz, "end", "J");

  // Read variables and cast to c-type
```

```
  start = (__int64)env->GetLongField(obj, startID);
  end = (__int64)env->GetLongField(obj, endID);

//  cout << "start: " << start << endl;
//  cout << "end: " << end << endl;
  int nbPrimes = 0;
  for (__int64 i = start; i < end; i++)
  {
   if (isPrime(i))
    nbPrimes++;
   Sleep(1);  // Improve performance for other threads
  }
//  cout << "nb of primes: " << nbPrimes << endl;

  // Get method ID
  jmethodID notifyID =
    env->GetMethodID(clazz, "notify", "(I)V");

  // Invoke Java method
  env->CallVoidMethod(obj, notifyID, nbPrimes);
}

bool isPrime(__int64 n)
{
  for (__int64 i = 2; i * i <= n; i++)
  {
    if (n % i == 0)
      return false;
  }
  return true;
}
```

Auf folgende Codeteile soll besonders hingewiesen werden:

- Der jni-Typ `jlong` muss auf den nativen 64-bit signed Integertyp gecastet werden. Auf der Window-Plattform ist dies `__int64`
- Während der Entwicklung werden wichtige Werte mit `cout` auf die Konsole ausgeschrieben. Im produktiven Code sind diese Teile auskommentiert. (Es gibt auch elegantere Debug-Verfahren)
- Die numerische Berechnung führt auf den meisten Systemen zu einer so hohen CPU-Belastung, dass andere Prozesse stark verlangsamt ablaufen. Um sie weniger stark zu bremsen, wird nach jedem Durchlauf der For-Schleife `Sleep(1)` aufgerufen, was die Dauer der Berechnung nicht wesentlich beeinflusst
- `GetMethodID()` wird als letzter Parameter die Signatur der Methode gemäß Tab. 2 als String übergeben. Da `void notify(int nbPrimes)` einen int-Parameter und den Rückgabetyp `void` besitzt, lautet die Signatur `(I)V`. Die Signaturen können auch mit

einem Kommandozeilen-Werkzeug angezeigt werden. Geht man in das Verzeichnis mit `JniEx4.class` und tippt

```
javap -s -private JniEx4
```

so ergibt sich unter anderem

```
private void notify(int);
  Signature: (I)V
```

- Da `notify()` den Rückgabetyp `void` besitzt, muss der Aufruf mit `CallVoidMethod()` erfolgen. Der letzte Parameterwert ist der aktuelle Parameterwert, den die Java-Methode `notify()` beim Aufruf zurück erhält.

Callbacks sind systemkritische Aufrufe und es gilt Folgendes zu beachten:

- Werden Callbackmethoden in schneller Folge aufgerufen, so muss der Code zu Ende laufen, bevor ein neuer Aufruf erfolgt. Callbackmethoden dürfen daher keinen komplexen, länger dauernden Code enthalten (Erzeugung von Objektinstanzen, Speicherallozierungen, u.ä.)
- Callbackmethoden dürfen in der Regel nicht von einem eigenen nativen Thread aufgerufen werden

Bei Missachtung dieser Grundsätze kann die Applikation sporadisch und schwierig reproduzierbar abstürzen.

2 Objektorientiertes JNI mit dem Proxy-Entwurfsmuster

Obwohl Java eine vollwertige objektorientierte Programmiersprache ist, unterstützt das JNI vom Konzept her die objektorientierte Programmierung nicht. Es ist darum naheliegend, JNI so zu erweitern, dass es eine direkte Entsprechung von Java- und C++-Klassen derart gibt, dass beim Aufruf der Java-Methoden gleichlautende Methoden der C++-Klassen ausgeführt werden. Wir orientieren uns dabei am bekannten Entwurfsmuster `Proxy`, wie es von der GoF [*Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software*] beschrieben wurde. Der Name stammt daher, dass einer gegebenen Klasse `A` eine Stellvertreterklasse, eben ein Proxy `P`, vorgelagert wird, wobei `P` alle oder

auch nur einzelne public Methoden der ursprünglichen Klasse `A` enthält, die sie vertritt. Bei der Ausführung einer solchen Methode von `P` wird aber eigentlich die entsprechende Methode von `A` ausgeführt. Dies erreicht man dadurch, dass der Konstruktor von `P` eine Instanz von `A` erzeugt und ihre Referenz als Instanzvariable in `P` kopiert, anders gesagt, `A` ist eine Komponente von `P` (*P has-an A*). Wird bei der Konstruktion von `A` komplizierter Code ausgeführt, so kann es günstig sein, einen einfacheren Konstruktor zu schreiben, und aufwändige Operationen, sowie umfangreiche Speicherallozierungen, auf später zu verschieben.

Im einfachsten Fall deklariert man in `P` die Methoden so, dass über die gespeicherte Referenz die entsprechende Methode von `A` aufgerufen wird. Es ist auch möglich, dass man gewisse Methoden von `A` in `P` gar nicht implementiert oder mit zusätzlichen Bedingungen, beispielsweise einer Sicherheitsüberprüfung, ausstattet.

Wir beschreiben das Prinzip exemplarisch für eine in Java geschriebene Klasse `P`, welche die Methode `f()` aufweist. Die zugehörige Klasse `A` soll in C++ implementiert sein. Offensichtlich handelt es sich bei `f()` um eine native Methode. Mit dem JNI müssen wir nun einen kleinen Umweg beschreiten, um auf die entsprechenden C++-Methode zuzugreifen. Dazu schreiben wir die JNI-Bibliothek als Dispatcher oder Handler, der als Interface zwischen den C-Aufrufen des JNI und den Methodenaufrufen der C++-Methoden konzipiert ist. Der Dispatcher hat zwei Aufgaben: Zum einen muss er beim Erzeugen einer Instanz von `P` eine Instanz von A erzeugen und den dabei erhaltenen Objektzeiger als Instanzvariable in die `P` zurückspeichern, zum anderen ruft er beim Aufruf von `f()` mit dem gespeicherten Objektzeiger die entsprechende Methode der C++-Instanz auf. Abb. 2.1 stellt diesen Mechanismus schematisch dar.

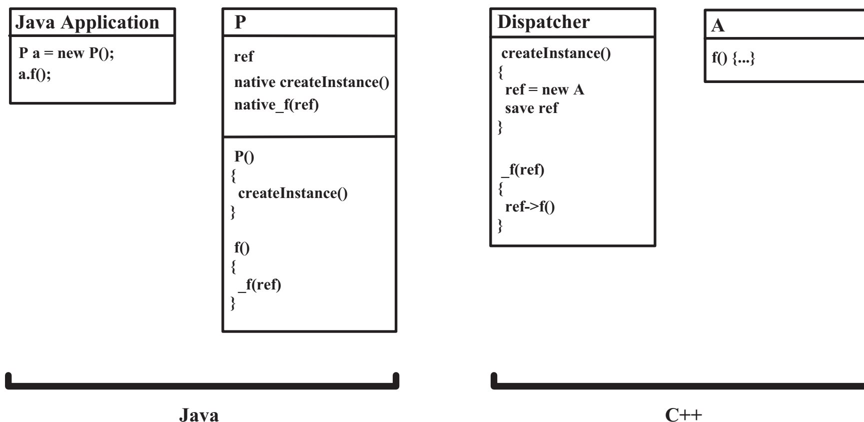

Abb 2.1: Aufruf mit Proxy

Die in Java geschriebene Proxy-Klasse `P` ist ein Stellvertreter für die native Klasse `A`, die in C++ implementiert ist und in der die Methode `f()` tatsächlich codiert ist . Beim Instanzieren von `P` wird der Zusammenhang der Klassen aufgebaut: Der Konstruktor von `P` ruft über das JNI die Funktion `createInstance()` des Dispatchers auf. Dieser erstellt eine Instanz

von A und kopiert den Wert des Instanzzeigers ref, als Instanzvariable in die Proxy-Klasse P zurück.

Damit beim Aufruf der Methode f() der Java-Klasse P tatsächlich die gleichlautende Methode der C++-Klasse A ausgeführt wird, ruft f() seine native Methode _f() auf. Diese enthält als Parameter den gespeicherten Instanzzeiger ref. Der Dispatcher ist nun in der Lage, mit ref->f() die entsprechende Methode in A aufzurufen.

Wir führen den Entwicklung "Top-Down" durch und beginnen mit der Java-Applikationsklasse JniEx5.java, wo wir lediglich f() aufrufen.

```
// JniEx5.java

public class JniEx5
{
  public JniEx5()
  {
    P a = new P();
    a.f();
  }

  public static void main(String args[])
  {
    new JniEx5();
  }
}
```

Als nächstes schreiben wir die Proxy-Klasse P gemäß unserem Konzept.

```
// P.java

public class P
{
  long ref;
  native void createInstance();
  native void _f(long ref);

  P()
  {
   System.load("s:\\myjni\\proxy\\release\\proxy.dll");
   createInstance();
  }

  void f()
  {
   _f(ref);
  }
}
```

Ein C++-Zeiger ist unter Win32 64-bit lang. Daher wählen wir als `ref` einen Long. Grundsätzlich hätten wir die Proxy-Klasse ebenfalls mit `A` bezeichnen können. Auch könnten wir die native Methode ohne den Underline versehen, wobei sie dabei überladen würde. Damit wäre die Entsprechung zwischen Java- und C++-Klassen noch offensichtlicher.

Nach der Compilation der beiden Klassen erstellen wir mit `javah` die Headerdatei `A.h.` In Visual Studio erstellen wir in einem Projekt mit dem Namen `proxy` die native Klassen `A`. Wie in C++ üblich, verwenden wir dazu als Interface die Headerdatei `aif.h` und implementieren in `a.cpp`. Dabei begnügen wir uns damit, einer Messagebox den erfolgreichen Aufruf anzuschreiben.

```
// aif.h

class A
{
public:
  void f();
};
```

```
// a.cpp

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

void A::f()
{
  MessageBoxA(0, "Hello, native f() executing...",
                 "Windows Message Box", MB_OK);
}
```

Es bleibt uns nur noch die Entwicklung des Dispatchers oder Handlers. Dazu kopieren wir wie üblich die Funktionsdeklarationen aus den Headerdateien `P.h` in die Quelldatei `handler.cpp`. Wir includen neben `jni.h` und `P.h` auch das Klasseninterface `aif.h`, weil wir die Klassendeklaration von `A` benötigen.

Zuerst implementieren wir die native Funktionen `Java_P_createInstance()`. Wir erzeugen darin eine Instanz der Klasse `A` und kopieren den Wert des Objektzeigers als Java-Instanzvariable zurück in die Proxy-Klasse.

Schließlich implementieren wir die native Funktion `Java_P__1f()`. Dabei rufen wir mit dem zurück erhaltenen Objektzeiger die entsprechende Methode auf. Der nötige Cast führt zu einem etwas unschönen Klammerwald.

```
// handler.cpp

#include <jni.h>
#include "P.h"
#include "aif.h"
```

```
JNIEXPORT void JNICALL
Java_P_createInstance(JNIEnv * env, jobject obj)
{
   A * ref = new A;
   jfieldID id =
      env->GetFieldID(env->GetObjectClass(obj), "ref", "J");
   env->SetLongField(obj, id, (jlong)ref);
}

JNIEXPORT void JNICALL
Java_P__1f(JNIEnv *, jobject, jlong ref)
{
   ((A *)ref)->f();
}
```

Nach der Erstellung von `proxy.dll` freuen wir uns am bei der Ausführung von `JniEx5` an der erfolgreichen Implementierung des Proxy-Entwurfsmusters.

3 JAW (Java API Wrapper)

3.1 Konzept des Frameworks

JAW ist ein Framework, das auf dem oben beschriebenen Proxy-Entwurfsmuster aufbaut. Bei der Verwendung von JAW sind elementare Kenntnisse von C++ notwendig, da in einer Entwicklungsumgebung C++-Klassen geschrieben und compiliert werden. Der Zugriff auf Java-Instanzvariablen und Java-Methoden erfolgt aber über C++-Templates, so dass die Verwendung der JNI-Funktionen entfällt. Daher ist es auch nicht nötig, mit `javah` eine Headerdatei zu erzeugen.

Die Java-Bibliothek `jaw.jar` enthält eine Wrapper-Klasse `NativeWindow`, welche die JNI-Aufrufe ausführt, Bei der Instanzierung eines `NativeWindow` wird automatisch die Verbindung zu C/C++ erstellt und die vordefinierte native Funktion `selectWindowHandler(JNIEnv * env, jobject obj, int select)` aufgerufen. Diese muss eine vom Benützer geschriebene Klasse, beispielsweise `MyWindowHandler`, erzeugen, die von der Klasse `WindowHandler` abgeleitet ist und welche mit den Parametern `env` und `obj` initialisiert wird. (Mit dem Parameter `select`

kann man eine Klasse aus mehreren Möglichkeiten auswählen.) Da sowohl `NativeWindow` auf der Seite von Java wie `WindoHandler` auf der Seite von C/C++ Teile der Bibliothek sind, wird der Benützer von JAW weitgehend vom internen Mechanismus des JNI abgeschirmt.

C++-Methoden werden von Java über die vordefinierte Methode `int invoke(int tag)` aufgerufen, wobei der Parameter `tag` die Möglichkeit gibt, zwischen mehreren Operationen auszuwählen. Der Datentransfer zwischen Java und C++ erfolgt, abgesehen vom Rückgabewert von `invoke()`, ausschließlich über Java-Instanzvariablen, auf die mit C++-Templates zugegriffen wird. Diese verbinden die Java- und C-Datentypen direkt ohne die Verwendung der JNI-Datentypen. Damit entfällt auch das unschöne und unsichere Casten von C und JNI-Datentypen.

Der Konstruktor von `NativeWindow` ist mehrfach überladen. Im einfachsten Fall übergibt man im lediglich den Pfad der DLL, die von der C++-IDE erzeugt wurde. Damit `MyWindowHandler` auf (auch private) Instanzvariablen einer Java-Klasse lesend oder schreibend zugreifen können, übergibt man dem Konstruktor eine Referenz auf diese Klasse oder ruft die Methode `expose()` auf. Man kann sich anschaulich vorstellen, dass man durch Übergabe der Java-Klassenreferenz dem nativen Code den Zugriff auf Instanzvariablen und Methoden **freigibt** bzw. **erlaubt**.

Mit der Methode `startThread()` kann ein nativer Thread mit hoher Priorität gestartet werden. Überschreibt man in `MyWindowHandler` die Methode `evThreadMsg()`, so wird diese von der *ThreadEventProc* bei jedem Thread-Event aufgerufen, beispielsweise bei WM_THREADSTART. Nativer Code, der dabei ausgeführt wird, läuft im nativen Thread und ist damit vom Prozess der JVM entkoppelt. Dadurch ist es möglich, im nativen Teil asynchrone Operationen auszuführen, wie beispielsweise das getaktete Auslesen eines Messinterfaces. Im nativen Teil steht zudem ein zirkulärer FIFO-Buffer zur Verfügung, um einen unterbrechungsfreien gepufferten Datentransfer zurück zum Java-Code zu gewährleisten.

Mit einem weiteren überladenen Konstruktor von `NativeWindow` wird im nativen Teil mit dem API-Call `CreateWindow()` ein Fenster mit wählbaren Window-Styles erzeugt. Die dazugehörige *WindowEventProc* leitet wählbare Messages an die in `MyWindowHandler` überschriebene Methode `evMsg()` weiter. Diese Methode läuft automatisch in einem separaten nativen Thread, was zu einer Entkopplung der Window-Messageloop vom JVM-Prozess führt und damit das native Fenster unabhängig von der JVM macht..

3.2 Installation von JAW und erstes Anwendungsbeispiel

Im Folgenden wird weiterhin auf der nativen Seite als IDE *Microsoft Visual Studio 2008 Express Edition* (deutsche Version) vorausgesetzt, deren Installation im Anhang beschrieben ist. Zur Verwendung von JAW wird zuerst die aktuellste Distribution von *www.aplu.ch/jaw* heruntergeladen, ausgepackt und `jaw.jar` in irgend ein Verzeichnis, z.B. `c:\jars`

kopiert. Diese Datei muss in den Library- oder Classpath der Java-IDE aufgenommen werden. Die statische Bibliothek `jaw.lib` kopiert man in irgend ein Verzeichnis, z.B. `c:\myjni` und alle Includedateien in `c:\myjni\include`.

Das nachfolgende Beispiel zeigt wie bei `JniEx3.java`, wie der Wert von zwei Instanzvariablen dem nativen Code übergeben wird, wo er um 1 erhöht wird. Nachfolgend wird der modifizierte Wert von Java zurückgeholt und in einem modalen Dialog dargestellt. Ein weiterer Wert wird als Methodenparameter an C++ übergeben und modifiziert als Rückgabewert zurückgeholt.

Der native Code ist in die DLL `jawsample.dll` verpackt, die später entwickelt wird. Der Konstruktor von `NativeHandler` erhält den voll qualifizierten Pfad auf diese DLL, um sie zu laden. Die `this`-Referenz, welche dem Konstruktor übergeben wird, gibt C++ Lese- und Schreibzugriff auf alle Instanzvariablen, sogar wenn diese `private` deklariert sind.

Im Gegensatz zu JNI ist der Bezeichner `native` für Variablen, auf die vom nativen Code zugegriffen wird, nicht mehr nötig. Darum entfällt auch die Erzeugung der C Headerdatei mit `javah`. Wir schreiben zuerst den Java-Code in `JawEx1.java`, der sich in der Programmlogik nur unwesentlich von `JniEx3.java` unterscheidet.

```
// JawEx1.java

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

public class JawEx1
{
  // Native variables
  private int n = 0;
  private double x = 0;

  public JawEx1()
  {
    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 JawEx1();
  }
}
```

Wie wir sehen werden, führt der Aufruf von `invoke()` zum Aufruf der C++-Methode mit dem gleichem Namen.

Als nächstes entwickeln wir den nativen Code. Dazu erzeugen wir in Visual Studio ein neues Projekt mit dem Namen `jawsample` und wählen wie üblich als Projekttyp *Win32* und als Vorlage *Win32-Projekt*. In den Anwendungseinstellungen ist als *Anwendungstyp DLL* und unter *Zusätzliche Optionen: Leeres Projekt* auszuwählen. In der Kopfzeile stellen wir *Release* ein. Im Folgenden gehen wir davon aus, dass das JAW-Include-Verzeichnis `c:\myjni\include` heißt und sich die JAW-Bibliothek `jaw.lib` in `c:\myjni` befindet.

Mit rechtem Mausklick auf *Quelldateien* wählen wir *Hinzufügen | Neues Element | C++-Datei* mit dem Namen `JawSampleHandler` und auf *Headerdateien* unter *Hinzufügen | Neues Element | Headerdatei* `JawSamleHandler`.

In den Projekteigenschaften (in der Projektmappe rechter Mausklick auf dem Projektnamen) stellen wir Folgendes ein:

Konfigurationseigenschaften

- Allgemein | Zeichensatz | Multi-Byte-Zeichensatz verwenden
- C/C++ | Allgemein | Zusätzliche Includeverzeichnisse: `c:\myjni\include`
- Linker | Eingabe | Zusätzliche Abhängigkeiten: `c:\myjni\jaw.lib`
- Linker | Debugging | Debuginfo generieren: Nein

Wir schreiben zuerst das Interface, d.h. die Headerdatei `JawSampleHandler.h`:

```
// 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
```

Bemerkungen:

- Die Klasse `JawSampleHandler` wird von `WindowHandler` abgeleitet und besitzt einen leeren Konstruktor, initialisiert aber die Basisklasse mit `env` und `obj`. Sie besitzt lediglich die Methode `invoke()` mit vorgegebener Signatur und Rückgabetyp
- Es wird die Headerdatei `jaw.h` included.

Die zughörige Implementierung nehmen wir in `JawSampleHandler.cpp` vor.

```
// 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;
}
```

Bemerkungen:

- Die globale Selector- (oder Dispatcher-) Methode `selectWindowHandler()` erzeugt eine Instanz der Klasse mit den Übergabewerten `env` und `obj`. Sie stellt die Verbindung zwischen JNI und C++ her und stellt `WindowHandler` das Java-Environment und die Referenz der Klasseninstanz zur Verfügung, auf dessen Instanzvariablen und Methoden zugegriffen werden kann. Der Parameter `select` steht zur Verfügung, um eine aus mehreren Klassen auszuwählen. Er wird in diesem Beispiel nicht verwendet
- Der Zugriff auf Java-Instanzvariablen erfolgt über die Klassen-Templates `JNIVar<type>`, welche als Parameter die geerbte Referenz auf das freigegebene Objekt und den Variablennamen als String aufweisen. Die Objektreferenz ist aus der public Variable `_exposedObj` der Basisklasse `WindowHander` zu beziehen. Wir rufen die Methoden `get()` und `set()` des anonymen temporären Objekts auf. Als Variablentypen stehen folgende C++ Datentypen zur Verfügung: `bool, signed char, short, int, __int64, float, double, wchar_t`
- Mit den Methoden `get()` und `set()` wird lesend bzw. schreibend auf die Instanzvariablen zugegriffen
- In `invoke()` werden die Instanzvariablen gelesen, die gewünschte Operation durchgeführt und die Werte zurück geschrieben. Der Methodenparameter wird ebenfalls inkrementiert und als Rückgabewert an Java zurückgegeben. Da die Signatur von `invoke()` vorgegeben ist, können dabei nur ints verwendet werden.

Nach der Projekt-Erstellung finden wir `jawsample.dll` im Verzeichnis `c:\myjni\jawsample\Release`. Wir starten `JawEx1` und freuen uns am Erfolg.

3.3 Verwendung einer Third-Party-Bibliothek

Ein wichtiges Entwicklungsziel von JAW bestand darin, bestehende Programmbibliotheken, die in einer anderen Programmiersprache als in Java geschrieben wurden, mit Java zu verwenden. Manchmal ist der Quellcode vorhanden, meist werden aber nur eine dynamische Library (DLL), zusammen mit ihrer statischen Import-Library vertrieben.

Wir zeigen hier exemplarisch, wie einfach es ist, eine DLL mit JAW in ein Java-Programm einzubinden. Dazu verwenden wir die Audio-Library **irrKlang**, welche kostenlos über das Internet angeboten wird (Suchmaschine mit Stichwort *irrklang*). Mit ihr können Sounddateien mit mehreren bekannten Audioformaten (wav, mp3, ogg, usw.) gespielt werden. Zudem ist es möglich, den Sound mit verschiedenen Effekten (echo, reverb,

distortion, usw.) zu verändern. Die Bibliothek ist vor allem zur akustischen Bereicherung von Spielprogrammen geschrieben worden.

Nach dem Download und dem Auspacken der Distribution orientiert man sich am besten an den beigefügten Beispielprogrammen, die sich im Verzeichnis *examples* befinden. Das Abspielen einer Sounddatei ist denkbar einfach: Mit

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

erzeugt man eine *Soundengine*, und startet das Abspielen der Sounddatei mit

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

wobei `soundfile` der voll qualifizierte Pfad der Sounddatei ist.

Zuerst schreiben wir die Java-Applikation `JawEx2.java`, wo in einem modalen Dialog als `JOptionPane` der OK-Button geklickt wird, um das Abspielen zu starten. Klickt man im nächsten Dialog den OK-Button, so wird die Applikation beendet. Der Einfachheit ist der Pfad der Sounddatei als Instanzvariable fest verdrahtet und wird vom nativen Code gelesen. `invoke(0)` soll die native Soundengine initialisieren und einen Fehlercode zurückliefern (bei Erfolg `0`, im Fehlerfall `-1`). Die Soundengine muss in einem nativen Thread laufen, da der Java-Thread während dieser Zeit in einem modalen Dialog blockiert ist. Dieser native Thread wird mit `startThread()` gestartet.

```
// 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(
      "s:\\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();
  }
}
```

Wie vorher, erstellen wir mit Visual Studio ein Projekt mit dem Namen `audio` und den Dateien `AudioHandler.cpp` und `AudioHandler.h`. In den Projekteigenschaften müssen wir als zusätzliches Includeverzeichnis neben dem JAW-Includeverzeichnis noch dasjenige der irrKlang-Distribution angeben. Haben wir irrKlang in `c:\irrKlang` ausgepackt, so heißt dieses `c:\irrKlang\include`. Zudem ist beim Linker die zusätzliche Abhängigkeit `c:\irrKlang\lib\win32-visualstudio\irrKlang.lib` anzugeben. Dies ist die statische Import-Bibliothek zur DLL `irrKlang.dll`, die sich im Verzeichnis `c:\irrKlang\bin\win32-visualstudio` befindet. Wir kopieren `irrKlang.dll` und das MP3-plugin `ikpMP3.dll` bereits in das Homeverzeichnis des Java-Projekts bzw. in das Verzeichnis, in dem sich die Java-Klassendateien befinden, damit die DLLs gefunden werden. (Falls eine *RuntimeException: Can't load DLL* auftritt, befinden sich diese DLLs nicht im richtigen Verzeichnis.)

Im Interface `AudioHandler.h` deklarieren wir die Klasse `AudioHandler`. Dabei überschreiben wir die Methoden `wantThreadMsg()` und `evThreadMsg()` der Klasse `WindowHandler` und deklarieren die Instanzvariablen `engine` und `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
```

In AudioHandler.cpp gilt es, einige Besonderheiten gegenüber den vorausgehenden Beispielen zu beachten:

- In invoke() führen wir die Initialisierung der Soundengine durch. Dabei holen wir mit einem String-Template den Pfad der Sounddatei aus der Java-Instanzvariable
- In der Methode wantThreadMsg() bestimmen wir mit einer OR-Maske, welche der Events der *ThreadEventProc* wir tatsächlich benötigen. Dabei stehen zusätzlich zu den nicht fensterbezogenen WM_messages, welche man aus der Win32-API-Dokumentation entnimmt, noch WM_THREADSTART und WM_THREADSTOP zur Verfügung, die beim Aufruf von startThread() bzw. stopThread() oder destroy() erzeugt werden
- Die Methode evThreadMsg() wird jedesmal aufgerufen, wenn eine der in wantThreadMsg() angegebenen Messages auftritt. Alle Informationen über die Message kann von den Parametern bezogen werden. Sie läuft im nativen Thread, und wir spielen bei WM_THREADSTART mit play2D() die Sounddatei ab. Bei der Message WM_THREADSTOP, die beim Aufruf von NativeHandler.destroy() auftritt, geben wir die Soundengine und den allozierten String-Array wieder frei.

```
// 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;
  }
}
```

Nach dem Erzeugen von `audio.dll` können wir das Java-Programm `JawEx2` starten. Mit einem Klick auf den OK-Button der modalen Messagebox wird das Abspielen gestartet, bis der OK-Button wieder gedrückt wird.

3.4 Verwendung nativer Fenster

Bei gewissen Aufgabenstellungen möchte man von Java aus ein natives Fenster erzeugen und über seine *WindowEventProc* Manipulationen darin vornehmen, beispielsweise Text, Grafiken oder Bilder anzeigen. Native Fenster können zudem interessante Eigenschaften besitzen, die Java-Fenster fehlen, beispielsweise Teiltransparenz. Im folgenden Beispiel erzeugen wir ein Fenster mit wählbarem Transparenzgrad, das ein Bild enthält. Wir beginnen wie üblich mit der Java-Applikation `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();
  }
}
```

Wir verwenden einen der überladenen Konstruktoren von `NativeWindow`, der die wichtigsten Fenstereigenschaften festlegt. Bei seiner Ausführung registriert JAW mit den Windows-API-Funktionen `RegisterClassEx()` eine Window-Klasse und erzeugt mit `CreateWindow()` ein Fenster mit der angegebenen OR-Kombination von Styleflags. Diese sind in `NativeWindow` als Konstanten deklariert und dokumentiert.

In der Abfrageschleife kann der Prozentsatz der Transparenz in einer modalen Dialogbox eingegeben werden. Nach der Eingabeprüfung wird mit `showWindow()` das Fenster angezeigt. Wichtig ist es, bei Programmende mit `destroy()` alle allozierten Ressourcen des nativen Fensters wieder freizugeben.

Das C++-Projekt mit dem Namen `trans` deklariert eine Klasse `TransHandler` mit folgendem Interface:

```
// 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
```

Dabei werden die Methoden `wantMsg()` und `evMsg()` aus der Klasse `WindowHandler` überschrieben. Mit `wantMsg()` wir mit einer OR-Maske angegeben, welche Window-Messages der *WindowEventProc* an `evMsg()` weiter geleitet werden. Wir benötigen die Messages WM_CREATE und WM_DESTROY, die beim Erzeugen, bzw. Vernichten des Fensters gesendet werden und die Message WM_PAINT, die immer dann gesendet wird, wenn Windows das Fenster neu zeichnen muss. Über die Grundlagen der Win32-API-Programmierung informiert man sich beispielsweise auf dem Internet mit einer Suchmaschine und den Stichwörtern *windows api programming tutorial*.

Es ist wichtig zu wissen, dass `evMsg()` automatisch in einem nativen Thread läuft, der unabhängig von der JVM ist, wodurch das native Fenster seine volle Eigenständigkeit von den Java-Threads erhält.

```
// 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;
}
```

Damit man die Bilddatei `rose.bmp` als Ressource verwenden kann, muss die Ressourcedatei `trans.rc` ins Projekt aufgenommen werden. Diese und die Bilddatei `rose.bmp` müssen sich im gleichen Verzeichnis wie `TransHandler.cpp` und `TransHandler.h` befinden.

```
// trans.rc

ROSE                 BITMAP  DISCARDABLE     "rose.bmp"
```

(Da in der aktuellen Version von *Visual Studio Express Edition* der Ressource-Editor nicht zur Verfügung steht, muss `trans.rc` mit dem Quelltext-Editor oder irgend einem anderen Texteditor erzeugt werden.)

Abb 3.1 zeigt das semi-transparente Fenster, das über ein anderen Applikationsfenster gezogen wurde.

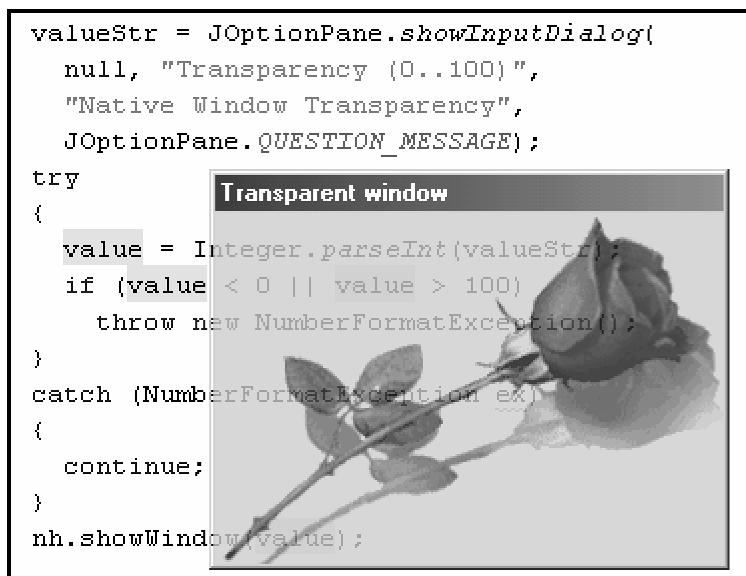

Abb. 3.1: Semi-transparentes Fenster

3.5 Callbacks mit JNIMethod<type>

Wie mit JNI können auch mit JAW vom nativem Code aus Java-Methoden aufgerufen werden. Wie üblich nennen wir einen solchen Aufruf einen Callback. Mit der Template-Klasse JNIMethod<type>, die analog zu JNIVar<type> aufgebaut ist, vereinfacht sich das Vorgehen wesentlich. Der Aufruf ist für Methoden ohne Parameter besonders einfach:

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

wobei type der Rückgabetyp der Java-Methode und methodName ihr Name sind. _exposedObj wird, wie bekannt, von WindowHandler geerbt.

Für Methoden mit Parametern, muss die Signatur der Parameter in der Reihenfolge ihrer Aufzählung angegeben werden. Will man beispielsweise die Java-Methode

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

von C++ aus aufrufen, so wird zuerst doIt() als JNIMethod<void> unter Angabe der Signatursequenz IDC deklariert:

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

Anschließend packt man die drei Parameter in eine `jvalue`-union. Diese ist wie folgt deklariert:

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

Für einen `int`, `double` und `char` also

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

Den Aufruf führt man mit `call()` aus, dem man `par` übergibt:

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

Als Beispiel sollen die Mausevents des nativen Fensters an Java zurückgegeben und dort in einer Console aus dem Package `ch.aplu.util` ausgeschrieben werden.

```
// 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();
  }
}
```

Die Callbackmethode `mouseEvent()` erhält als Parameter einen int, der die Art des Mausereignisses beschreibt und die beiden Mauskoordinaten `x` und `y`. Es ist wie in jeder Callbackmethode davon abzuraten, im Callback länger dauernden Code auszuführen, da die Events in schneller Folge auftreten können. Vielmehr werden die Werte in Instanzvariablen kopiert und der schlafende main-Thread aufgeweckt, der das Ereignis auf der Console ausschreibt.

Das C++-Projekt mit dem Namen `mouseevent` deklariert eine Klasse `MouseEventHandler`. Damit die Bilddatei `rose.bmp` als Ressource geladen werden kann, muss analog zum vorhergehenden Beispiel die Ressourcedatei `MouseEvent.rc` ins Projekt aufgenommen werden.

```
// MouseEvent.rc

ROSE                 BITMAP  DISCARDABLE     "rose.bmp"
```

Das Interface `MouseEventHandler.h` ist denkbar einfach.

```
// 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 der Implementierung wollen wir noch vorsehen, dass man das native Fenster mit gedrückter Maustaste verschieben kann. Dazu registrieren wir den Event `WM_MOVE` und verschieben das Fenster mit `SetWindowPos()` an die neue Position.

```
// 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::sendMouseMessage(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);
}
```

Für den Callback deklarieren wir die Instanz `mouseEvent` mit der Parameter-Signatur `III` und rufen `call()` mit den drei in `par` verpackten ints auf. Ziehen wir die Rose auf das Console-Fenster, so ergibt sich die Abb. 3.2.

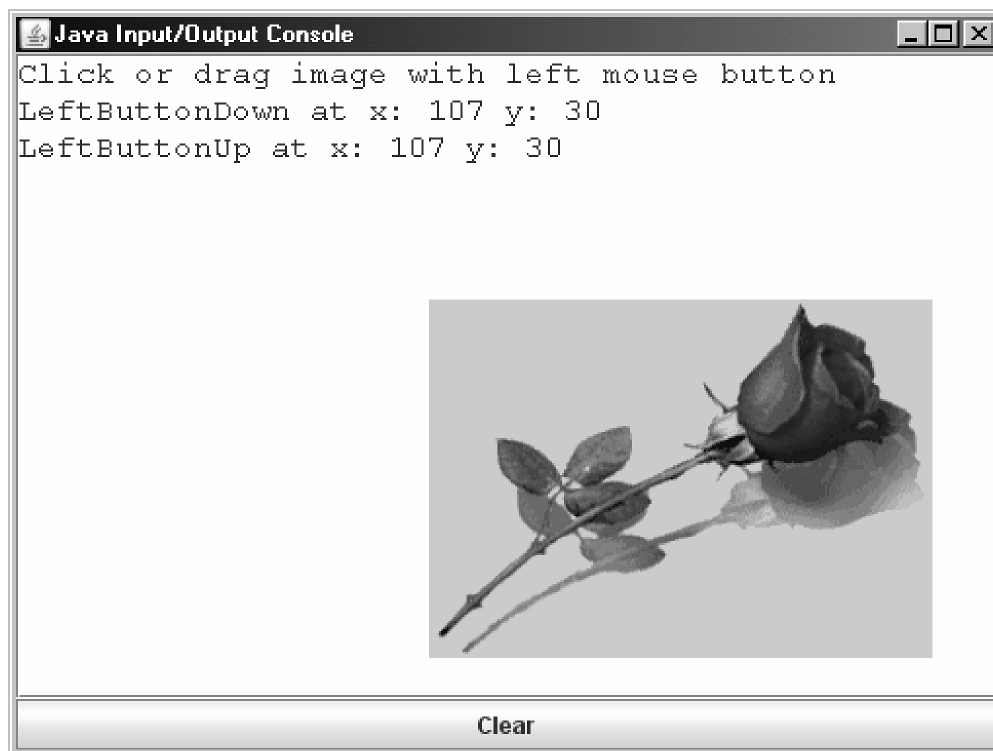

Abb 3.2: Callback von Mausevents

3.6 Transfer von Strings und Arrays

Für den Transfer von Strings wird in JAW die Klasse `JNIString` verwendet. In C++ kann ein String immer als Character-Array dargestellt werden, dessen einzelne Bytes aber je nach Zeichensatz anders interpretiert werden. Java verwendet für Strings durchwegs den Unicode-Zeichensatz. Transferiert man einen Java-String nach C++. so kann mit verschiedenen `get()`-Methoden unterschieden werden, ob im Character-Array die einzelnen Java-Zeichen im Multibyte Character Set (MBCS), als UTF-8 oder Unicode abgespeichert sind. Beim Aufruf von `get()` wird intern ein Character-Array mit der nötigen Länge erzeugt. Dieser muss mit `delete[]` wieder freigegeben werden.

Bei Arrays von Basistypen ist das Verfahren analog. Es werden Template-Klassen `JNIArray<type>` verwendet, wo type einer der Typen `wchar_t, bool, signed char, short, int, __int64, float, double` sein kann. Je nach JVM wird für den Transfer intern ein Zwischenspeicher angelegt, der von JAW durch den Destruktor von `JNIArray<>` automatisch wieder freigegeben wird. Das Lesen und Schreiben erfolgt über

einen Array-Zeiger, den man für alle Typen mit der gleichen Methode `get()` erhält. Es ist darauf zu achten, dass nach Ausführung des Destruktors dieser Zeiger nicht mehr verwendet werden darf, da er ungültig wird.

Im folgenden Beispiel werden als Illustration einige String- und Arraydaten von Java nach C++ und wieder zurück transferiert. Sie werden sowohl von C++ und Java in einer Konsole ausgeschrieben.

```
// 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("s:\\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();
  }
}
```

Das C++-Projekt mit dem Namen `data` deklariert in der Klasse `DataDemo` lediglich die Methode `invoke()`. Das Interface ist also sehr einfach:

```
// 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
```

In der Implementierung wollen wir mit `invoke(0)` die erhaltenen Strings in einer Konsole mit `cout` ausschreiben. `invoke(1)`, `invoke(2)` und `invoke(3)` sollen den String in verschiedenen Zeichensätzen interpretiert wieder an Java zurückgeben. Da wir in den Projekteigenschaften den Multi-Byte-Zeichensatz gewählt haben, wird `setMBCS()` akzentuierte Zeichen richtig zurückgeben. Da wir bei `setUnicode()` den String als `wchar_t` deklarieren, ist die Rückgabe auch hier richtig, die Übergabe als UTF-8 aber falsch.

Beim Aufruf von `invoke(4)` soll der Integer-Array an C++ übergeben, dort ausgeschrieben und modifiziert wird an Java zurückgegeben werden, wo er wieder ausgeschrieben wird.

```
// 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;
}
```

Die Ausführung ergibt:

```
Sending string to C++...
C++ got UTF8 string: abc-Ã¤Ã¶Ã¼
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,
```

3.7 Verwendung von Echtzeit-Messgeräte-Modulen von National Instruments

National Instruments (NI) bietet keinen Support zur Verwendung von Java zur Steuerung ihrer Messgeräte an. Gemäß der Knowledge Base von NIs wird empfohlen, einen JNI-Wrapper einzusetzen.

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.

Ausgehend von den in C vorliegenden Anwendungsbeispielen aus der NI-DAQmx-Installation und der *NI-DAQmx Function Reference*, ist es mit JAW einfach, einen JNI-Wrapper zu schreiben.

Als Vorbereitung muss die Distribution von *NI-DAQmx Base* heruntergeladen und installiert werden. Man findet sie mit einer Internetsuchmaschine und den Stichworten *nidaqmx download*. Bei der Installation werden auch die benötigen DLLs in das windows\system32-Verzeichnis kopiert, insbesondere `nidaqmxbaselv.dll`.

Im Folgenden wird weiterhin auf der nativen Seite als IDE Microsoft Visual Studio 2008 Express Edition (deutsche Version) vorausgesetzt. Zur Verwendung von *NI-DAQmx Base* sind die folgenden zusätzlichen Projekteigenschaften nötig:

- Konfigurationseigenschaften | C/C++ | Allgemein | Zusätzliche Includeverzeichnisse: Verzeichnispfad, in dem sich `NIDAQmxBase.h` befindet, z.B. c:\nidaq\include

- Konfigurationseigenschaften | Linker | Eingabe | Zusätzliche Abhängigkeiten: vollständig qualifizierter Pfad zu `nidaqmxbase.lib`, z.B. c:\nidaq\.nidaqmxbase.lib

Wir verwenden hier das kostengünstige NI-Modul *USB-6008*, das folgende Eigenschaften aufweist:

- 8 analoge Input-Kanäle (12 bit), max. Abtastrate 10 kHz (1 Kanal)
- 2 analoge Output-Kanäle (12 bit)
- 12 digitale Input-/Output-Kanäle
- 32 bit Zähler

Nach der Installation von NI-DAQmx Base wird das Interface beim Anschließen an die USB-Schnittstelle automatisch erkannt.

3.7.1 Portierung von acquire1Scan

Wir gehen in diesem Anwendungsbeispiel von `acquire1Scan.c` aus, wo gezeigt wird, wie man einzelne Analog-Werte aus einem Interface ausliest. Die Vorlage ist gut dokumentiert, bei Bedarf zieht man noch die *NI-DAQmx Base C Function Reference* heran, die man ebenfalls mit einer Internet-Suchmaschine findet. Nachfolgend das Programm in Originalformatierung:

```
/*********************************************************************
*
* 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 Acquistion 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;
}
```

Wie sichtbar, wird die Initialisierung in einzelne Schritte zerlegt, die wir in Java beibehalten wollen. Wir führen sie durch den Aufruf von `invoke()` mit verschiedenen Parameterwerten aus. Läuft der Schritt fehlerfrei ab, so wird 0, im Fehlerfall -1 zurückgegeben und wir können eine Fehlerbeschreibung aus dem String `errorDescription` herausholen und anzeigen. Das Java-Programm `JawEx6.java` schreibt in ein Console-Fenster aus dem 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();
  }
}
```

Den Datentransfer zwischen Java und C++ führen wir wieder mit Instanzvariablen durch, die beiden Sprachen zugänglich sind. Wir achten auch sorgfältig darauf, dass beim Abbruch des Programms in jedem Fall `destroy()` aufgerufen wird, damit alle allozierten Ressourcen freigegeben werden. Als Strukturelement eignet sich eine Zustandsvariable, die als `enum` implementiert ist.

Da wir die einzelnen Schritte durch einzelne Aufrufe ausführen, müssen wir die Variablen des C-Programms, welche die Aufrufe überleben, als Instanzvariablen (member variables) der C++-Klasse implementieren. Wir erstellen ein neues Projekt mit dem Namen `nidaq` und schreiben wie gewohnt zuerst die Headerdatei `NIDAQHandler.h`. Dabei verwenden wir die gleichen Bezeichner wie in der Vorlage und behalten auch mehrheitlich die Formatierung bei.

```
// 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
```

Auch in der Implementierung `NIDAQHandler.cpp` übernehmen wir den größten Teil der Vorlage, ohne dass wir die Bedeutung in allen Einzelheiten verstehen müssen. Wir belassen auch den etwas altertümlichen Funktionsaufruf und Fehlerabfang mit dem Macro `DAQmxErrChk`. Statt wie ursprünglich den Fehlertext mit `printf()` auszugeben, schreiben wir den Text in einem Java-String `errorDescription` zurück.

```
// 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;
}
```

Wenn wir alles richtig gemacht haben, so können wir jede Sekunde eine langsam veränderliche Spannung am ADC einlesen (Abb. 3.3).

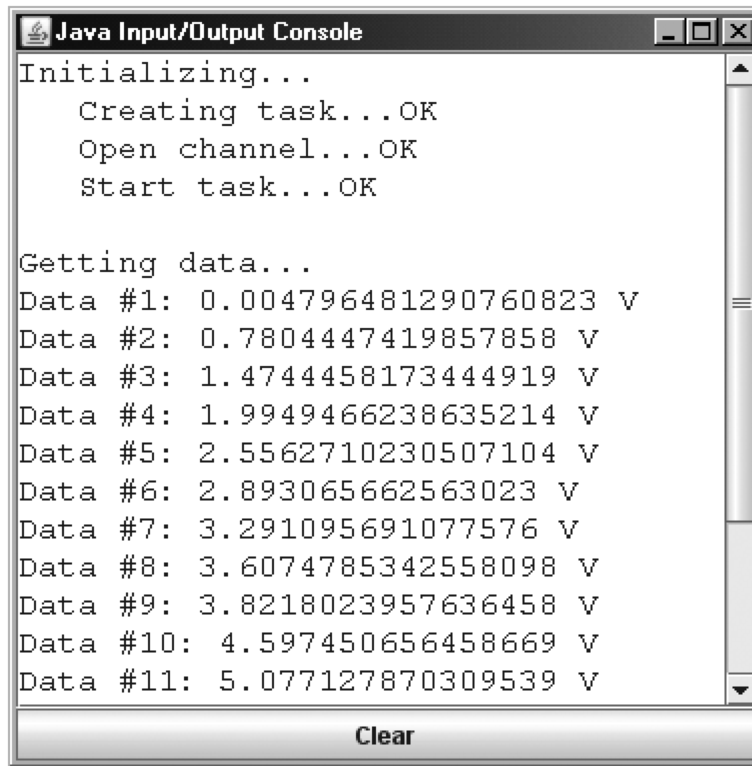

Abb. 3.3 Ausgabe von JawEx6 beim langsamen Verändern der Eingangsspannung

3.7.2 Portierung von contAcquireNScan

Liegen die Abtastraten über 1000 Hz, so eignet sich das Verfahren, mit einem Timer Einzelmessungen auszulösen und auszuwerten, nicht mehr. Vielmehr sollten ganze Datenblöcke aufgenommen werden, die in einen schnellen Buffer (meist als Direct Memory Access (DMA) implementiert) gespeichert werden. Die Hardware und Software-Treiber des Interface-Herstellers müssen die entsprechende Unterstützung anbieten, wie dies ist bei National Instruments der Fall ist und im Beispielsprogramm `contAcquireNScan` der NI-DAQmx-Distribution gezeigt wird. Wiederum ist einfach, unter Beizug von JAW eine Portierung nach Java vorzunehmen. (Aus Platzgründen wird die Originalversion des Examples nicht abgedruckt.)

Das Programm soll in einem einfachen Grafikfenster (`GPanel` aus dem Package `ch.aplu.util`) zwei Signale, die mit einer Abtastrate von 5 kHz, bzw. einer Periode von 0.2 ms aufgenommen werden, in der Art eines Zweikanal-KOs dargestellt werden. Viele Ideen können vom vorhergehenden Beispiel übernommen werden. Die Initialisierung soll immer noch schrittweise durchgeführt werden, wobei eventuelle Fehler auf der Titelzeile des Grafikfenster erscheinen. Ein Datenblock (burst) umfasst 200 Werte der beiden Kanäle und

wird durch `invoke(5)` vom nativen Datenbuffer abgeholt und dargestellt. Um keine Buffer-Überlauf zu erhalten wird vorausgesetzt, dass dieser Prozess im Mittel nicht länger als 200 * 0.2 ms = 40 ms dauert. Kurzzeitig, beispielsweise bei großer Systembelastung durch andere Windows-Prozesse, kann diese Wert größer sein, ohne dass Daten verloren gehen oder falsch angezeigt werden, da die Daten im nativen Buffer gespeichert bleiben.

Um das Programm ordnungsgemäß zu beenden, wird ein `ExitListener` implementiert, so dass in `notifyExit()`, das beim Klicken auf den Close-Button des Grafikfensters aufgerufen wird, die Aufräumarbeiten ausgeführt werden können.

```
// 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();
  }
}
```

Wir erstellen ein neues Projekt mit dem Namen `nidaqc` und schreiben wiederum unter Beizug des Originalbeispiels zuerst die Headerdatei `NIDAQHandler.h`. Bis auf wenige Details ist der Code identisch mit dem vorhergehenden Beispiel.

```
// 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
```

Die Implementierung weist gegenüber dem vorhergehenden Beispiel nur wenige Änderungen auf, die unter Beizug des Originalbeispiels selbsterklärend sind. Besonders hingewiesen wird aber auf die elegante Art, beim Abruf eines Datenbursts mit invoke(5) einen ganzen Java-Array zu füllen. Dazu holt man sich mit den beiden Zeilen

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

den Array-Zeiger auf den Java-Array und kann diesen `DAQmxBaseReadAnalogF64()` zum Füllen des Arrays übergeben.

```
// 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;
}
```

Schließt man zwei niederfrequente Signalgeneratoren an, so ergibt sich ein Bild gemäß Abb. 3.4.

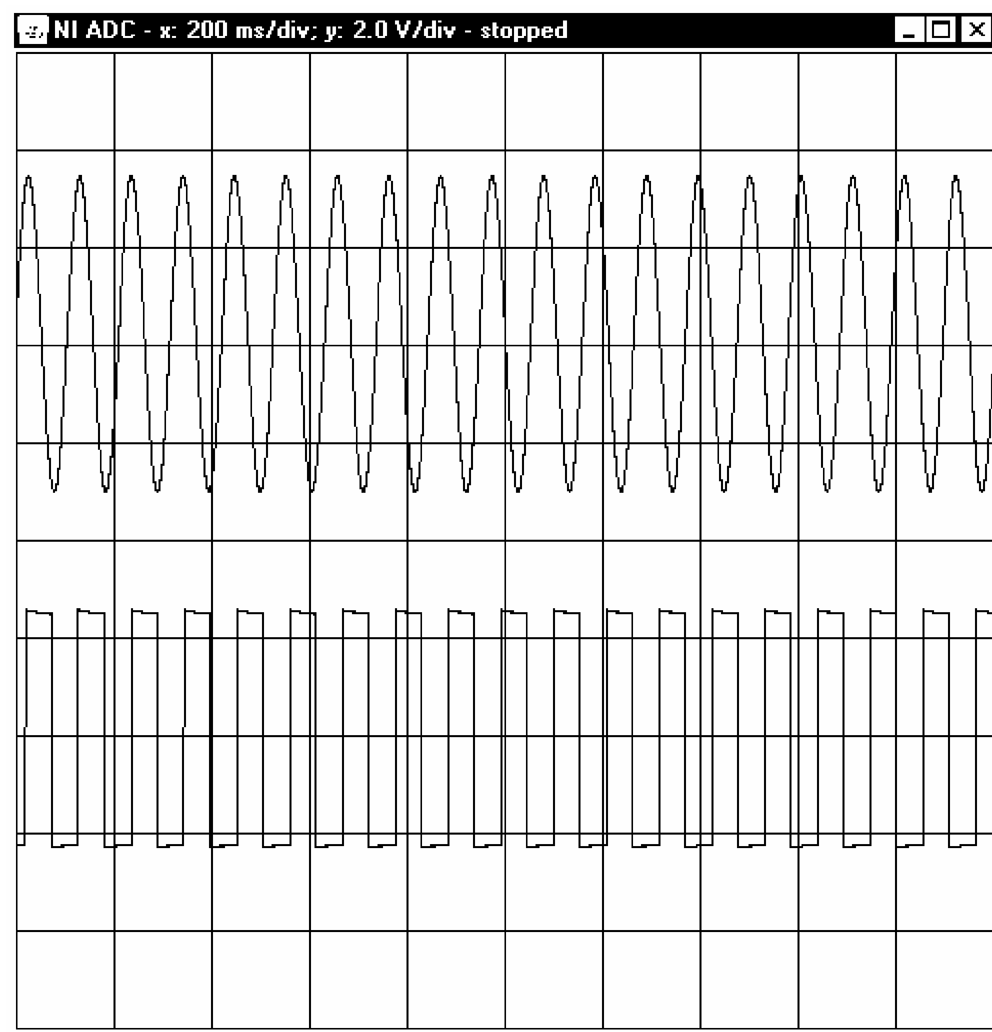

Abb. 3.4 Simulierter Zweikanal-KO von JawEx7

3.8 Datentransfer mit JNIBuffer

Die Treiber von vielen typischen Mess-Interfaces kennen in Analogie zu Streams drei grundlegende Operationen: *Initialisierung*, oft *Öffnen* (*open*) genannt, *Lesen* bzw. *Schreiben* von Daten und *Freigabe* der Ressourcen, oft *Schließen* (*close*) genannt. Liegt der Treiber als DLL vor und gibt es dazu eine Import-Library (.lib) und eine Dokumentation der Funktionsaufrufe, so ist es leicht, solche Interfaces unter Verwendung von JAW von Java aus anzusprechen.

Aus einer vorliegenden DLL kann mit einem Utility, das normalerweise implib heißt, die Import-Library erzeugt werden. Eine Version von implib ist in der Distribution des freien Borland C++ Compilers (Suchmaschine mit diesen Stichworten) enthalten.

Im nächsten Beispiel zeigen wir das typische Vorgehen, um eine Java-Applikation zu entwickeln, die in Echtzeit Daten eines Messinterfaces einliest und grafisch darstellt. Man deklariert in C++ wie in allen vorhergehenden Beispielen eine Klasse, die von `WindowHandler` abgleitet ist, und initialisiert (`open`) im Konstruktor den Treiber. Im Destruktor wird er wieder geschlossen (`close`). Für langsame Datenakquisitionen (Abtastrate < 1 Hz) verwendet man einen Aufruf `invoke()`, um einzelne Werte einzulesen, wobei die Daten über Instanzvariablen in Java zurückgeholt werden. Das Takten kann durch einen in Java implementierten Timer erfolgen. Für schnellere Abtastraten (bis maximal 1000 Hz) ist es besser, das Takten durch einen nativen Timer auszuführen, der in einem eigenen nativen Thread hoher Priorität und unabhängig (asynchron) vom Java Thread läuft. Die eingelesenen Daten sollten im nativen Teil in einen nativen zirkulären FIFO-Buffer (*JNIBuffer* genannt) eingelesen und nicht direkt an Java übergeben werden, damit kein Datenüberlauf bei starker Belastung der CPU auftritt. Da die Implementierung dieses Buffermechanismus nicht ganz unproblematisch ist, bietet JAW folgende Unterstützung:

Im Java-Code erstellt man einen Java-Buffer als Array des gewünschten Datentyps, beispielsweise für chars mit

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

Die Länge ist dadurch bestimmt, wie viele Datenelemente als Block miteinander vom nativen JNIBuffer zum Java-Buffer transferiert werden sollen, hat aber nichts mit der Länge des JNIBuffers zu tun. Vielmehr wird dieser mit einer Instanz `nh` von `NativeHandler` erzeugt, wobei die Bufferlänge (hier 18) angegeben werden kann.

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

Fehlt der Größenparameter so wird standardmäßig eine Bufferlänge von 1000 angenommen. In C++ füllt man den JNIBuffer mit

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

Java holt die Werte mit `readBuf()` im JNIBuffer ab, wobei die maximale Anzahl von Datenelementen angeben wird. Mit

```
readBuf(2);
```

werden also maximal 2 Elemente vom JNIBuffer in den Java-Buffer zurückgeholt. Sind im JNIBuffer weniger als die angegebene Anzahl von Datenelementen vorhanden, so werden natürlich nur diese transferiert. Darum gibt `readBuf()` als Rückgabewert die Anzahl der tatsächlich transferierten Elemente. Da im Beispiel 5 vorhanden sind, wird hier 2 zurückgegeben und die zwei "ältesten" Zeichen, 'A' und 'B' in den Java-Array kopiert. Im JNIBuffer, der als FIFO-Ringbuffer organisiert ist, wird dabei der Head-Zeiger um 2 Elemente vorgerückt. Dem Java-Code stehen die Daten nachfolgend in `value[0]` und `value[1]` zur Verfügung. Abb. 3.5 zeigt den Vorgang, wobei die leer gezeichneten Bufferfelder natürlich nicht leer, sondern ungültig sind.

Da `JNIBuf` als Ringbuffer implementiert ist, gibt bei einem Bufferüberlauf keinen Speicherfehler, vielmehr wird bei einem vollen Buffer ein Overflow-Flag gesetzt und nachfolgend hineingeschriebene Elemente gehen verloren. Das Flag kann mit `isBufOverflow()` abgefragt (und zurückgesetzt) werden.

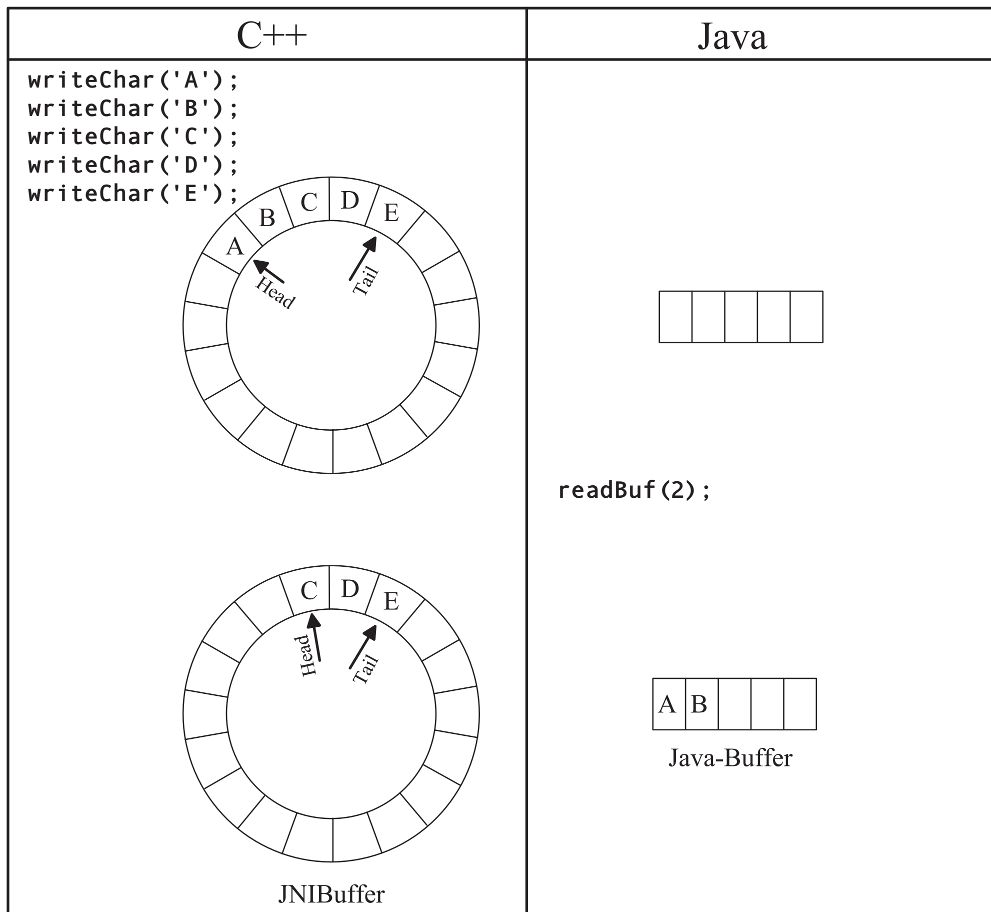

Abb. 3.5 JAW FIFO-Buffer(JNIBuffer) der Länge 18

Im nächsten Beispiel wird die Verwendung des JNIBuffers im Zusammenhang mit einem kostengünstigen Interface USB-AD12 der Firma BMC (`http://www.bmcm.de`) gezeigt. Damit das Programm aber zu Demonstrationszwecken auch ohne vorhandene Hardware ausgeführt werden kann, sehen wir durch Setzen von

```
#define simulation
```

einen Simulationsmodus vor.

Das Interface USB-AD12 besitzt 16 analoge Eingangskanäle (+- 5V, single ended) mit 12-bit Auflösung, 1 analogen 12-bit Ausgangskanal, sowie 4 digitale Ein-/Ausgangskanäle. Die Modulbibliothek ist auf der Website des Herstellers frei verfügbar (Suchmaschine mit Stichworten libad4 sdk) und enthält die DLL libad4.dll, sowie die zughörige Import-Library libad4.lib. Der USB-Treiber muss separat installiert werden (install-drivers-<version>.exe).

Wie im vorhergehenden Beispiel soll die Java-Applikation die eingelesen Daten in einem einfachen Grafikfenster (`GPanel` aus dem Package `ch.aplu.util`) darstellen. Das Interface wird mit einem nativen Timer alle 20 ms ausgelesen, der in einem nativen Thread läuft. In der Titelzeile wird angezeigt, wie viele Daten im Buffer zum Abholen bereit stehen. Wir erkennen die Notwendigkeit einer FIFO-Zwischenbufferung, da bei hoher Systembelastung dieser Wert durchaus auf einige Hundert ansteigen kann. In der Leseschleife holt Java mit `readBuf(1)` jeweils einen einzigen neuen Wert, falls es diesen überhaupt gibt, und stellt ihn in der Grafik dar.

```
// 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();
  }
}
```

Damit die Leseschleife das System nicht unnötig belastet, legt sich der Java-Thread 1 ms zum Schlafen, wenn die Anzahl der abzuholenden Elemente kleiner als 2 ist. Da ein neuer Wert alle 20 ms eingelesen wird und der JNIBuffer 1000 Elemente aufnehmen kann, ist auch bei hoher Systembelastung nicht mit einem Bufferüberlauf zu rechnen

Wir erstellen ein neues Projekt mit dem Namen `bmc` und schreiben unter Beizug der Beispielprogramme oder der Dokumentation der Distribution von `libad4` zuerst das

Interface BMC_Hander.h. (Falls das Interface nicht vorhanden ist, kann man sich auf die Codeteile beschränken, die compiliert werden, falls simulation definiert ist.)

```
// 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
```

Die Implementierung holt im Konstruktor die Variablenwerte der Java-Applikation und initialisiert mit ad_open() das Interface. Im Destruktor werden mit ad_close() alle Ressourcen freigegeben. Der Konstruktor wird beim Instanzieren von NativeHandler ausgeführt, der Destruktor beim Aufruf seiner Methode destroy().

Aus Kapitel 3.3 kennen wir die Rollen von wantThreadMsg() und evThreadMsg(). Wir verwenden die Message WM_THREADSTART um mit der API-Funktion SetTimer() einen nativen Timer mit vorgegebener Periode einzurichten. Bei WM_THREADSTOP geben wir mit KillTimer() dessen Ressourcen wieder frei. Bei jedem Timerevent, den wir mit WM_TIMER erhalten, lesen wir einen neuen Analogwert mit ad_discrete_in() ein bzw. erzeugen im Simulationsmodus einen Wert einer exponentiell abfallenden Sinuskurve. Mit writeDouble() schreiben wir den erhaltenen Wert in den JNIBuffer. Da dieser asynchron von Java ausgelesen wird, entfällt eine Notifikation, dass ein neuer Wert vorliegt.

In evThreadMsg() dürfen weder Java-Variablen noch Callbacks verwendet werden, da der native Thread nicht mit der JVM synchronisiert ist.

```
// 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();
#ifndef simulation
  adHandle = ad_open(driver);
#endif
  delete [] driver;
}

 AnalogIn::~AnalogIn()
 {
#ifndef 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);
      // Interprete 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;
  }
}
```

Im Simulationsmodus wird die gedämpfte Sinusschwingung aufgezeichnet. Im Titelbalken kann die Anzahl der Werte abgelesen werden, die im JNIBuffer zur Abholung bereit stehen. Normalerweise ist diese Zahl 0 oder 1, kann aber je nach Rechnertyp bei großer Systembelastung, beispielsweise beim Start von großen Windows-Applikationen, durchaus

kurzzeitig auf über 100 ansteigen. Da alle Werte zwar etwas verspätet, aber trotzdem abgeholt werden, zeigt die Grafik auch bei großer Systembelastung keine Fehler (Abb. 3.5).

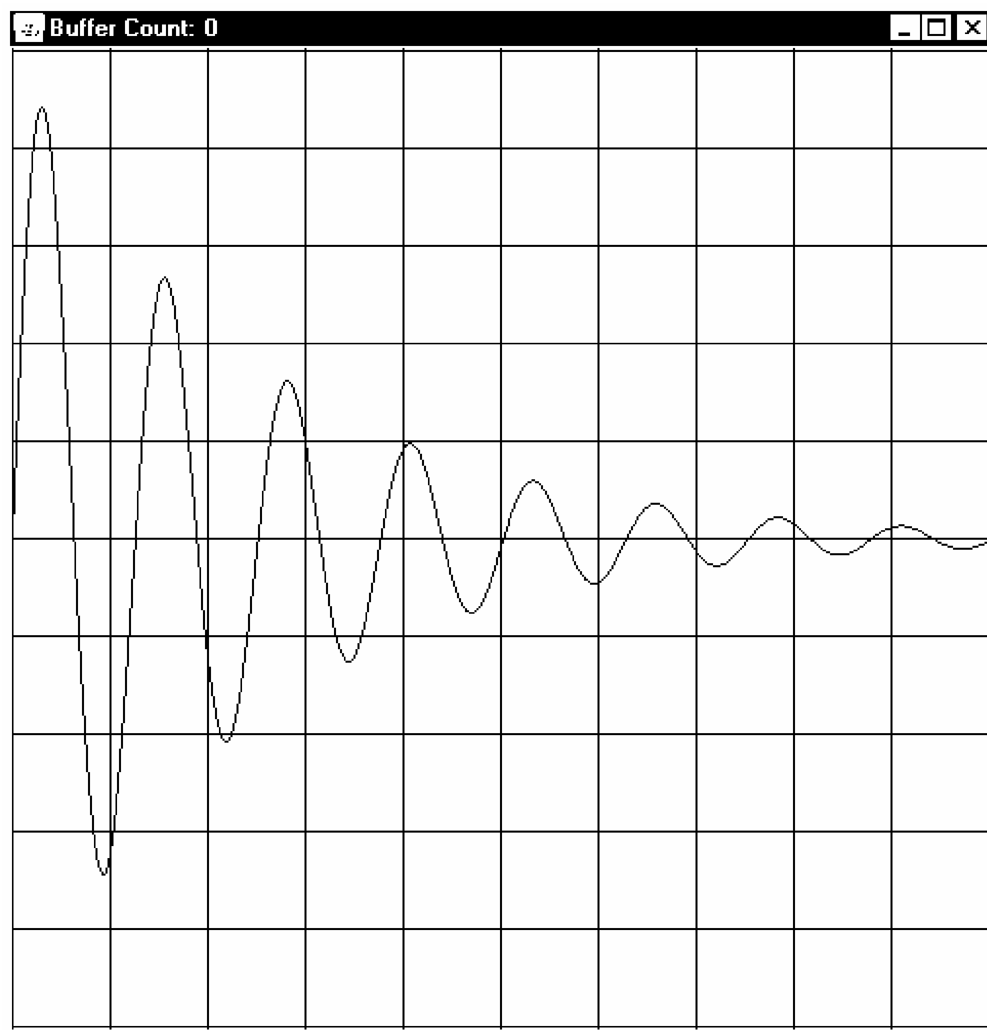

Abb. 3.5 JawEx8 im Simulationsmodus

4 Anhang: Installation von Microsoft Visual Studio

Microsoft Visual Studio ist eine Entwicklungs-Suite zur Erstellung von Programmen, die unter Microsoft Windows laufen. Verschiedene Programmiersprachen werden unterstützt, im Wesentlichen *Visual Basic*, *Visual C++* und *Visual C#*. Alle haben eine gemeinsame Entwicklungsumgebung (IDE).

Seit einiger Zeit ist auf dem Internet eine kostenlose Version unter dem Namen *Visual Studio Express Edition* erhältlich, welche man mit einer Suchmaschine und den Stichwörter *Visual Studio Express Edition download* findet. Obschon diese Version über einen reduzierten Funktionsumfang gegenüber der kommerziellen Version besitzt, ist sie für die Entwicklung von Applikationen im Zusammenhang mit dem JAW Framework völlig ausreichend. (Es fehlen mehrere Klassen und der Ressource-Editor.) Für JAW genügt die Installation von Visual C++.

Nach dem Download und der Installation von Visual C++ entwickeln wir als Test wie üblich ein Windows-Konsolenprogramm, das lediglich `Hello World` ausschreibt. Hier das Rezept:

- Wähle *Datei | Neu | Projekt*.
- Im erscheinenden Dialog wähle als Projekttyp *Win32* und als Vorlage *Win32-Konsolenanwendung*. Unter *Name* gebe beispielsweise *HelloWorld* ein und klicke *OK*
- Es erscheint der Win32-Anwendungsassistent, wird *Weiter* geklickt. Unter *Anwendungstyp* wähle *Konsolenanwendung* und unter *Zusätzliche Optionen Leeres Projekt* und klicke *Fertig stellen*
- Im Projektmappen-Explorer klicke mit der rechten Maustaste im Projekt *HelloWorld* auf *Quelldateien* und wähle *Hinzufügen | Neues Element*
- Um Dialog wähle als Vorlage *C++-Datei* und gebe den Namen *HelloWorld* ein. Nach dem Klicken auf *Hinzufügen* öffnet sich der Editor. Schreibe folgendes Programm:

```
// HelloWorld.cpp

#include <iostream>

using namespace std;

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

- In der Listbox der Toolbar, welche den Eintrag *Debug* enthält, wähle *Release* und compiliere/linke unter der Menüoption *Erstellen | Projektmappe erstellen*
- Im Unterverzeichnis `Release` des Projektordners entsteht die Datei `HelloWorld.exe`, die mit der Menüoption *Debuggen | Starten ohne Debugging* (oder mit *Ctrl-F5*) ausgeführt wird.

In der Einführung in das JNI und mit JAW wird ausschließlich das Win32-API verwendet und auf das .NET-Frameworks verzichtet. Mit nur wenigen Modifikationen können daher die Beispiele auch mit anderen Compilern (Borland C++, gcc, usw.) erzeugt werden. Für JAW muss eventuell eine angepasste Library verwendet werden. Informationen über deren Verfügbarkeit erhält man auf `www.aplu.ch/jaw`.