Design and Implementation of a Java Interface for the National Instruments Driver – NI-DAQmx

Ítalo Macedo Laino
Diretório Acadêmico da Computação
Universidade Tecnológica Federal do Paraná
Piraju – São Paulo, Brasil
italomlaino@gmail.com

Elias Canhadas Genvigir
Diretório Acadêmico da Computação
Universidade Tecnológica Federal do Paraná
Londrina - Paraná, Brasil
elias@utfpr.edu.br

***Abstract*— We propose a design for a Java Interface for the newest National Instruments data acquisition driver, the NI-DAQmx. The connection between the proposed interface and the driver is done by a cross-language wrapper of the driver's C library using the Java Native Interface. We also provide an implementation with the most used features of the driver.**

Keywords—NI-DAQmx, Java Interface;

I. INTRODUÇÃO

A National Instruments (NI) não fornece uma interface Java para o seu driver de aquisição de dados, o NI-DAQmx. Assim há carência de um mecanismo que permita um aplicativo desenvolvido com a tecnologia Java acessar o driver.

Nesse artigo, nos propomos um design de uma interface Java para o driver. A conexão entre a interface proposta e o driver é feita por um *wrapper*[1] da biblioteca C do driver (NI-DAQmx C) utilizando a *framework* Java Native Interface (JNI).

Esse documento está estruturado da seguinte forma: na seção II é apresentado um resumo da literatura; III é apresentado a solução do problema; na IV é apresentado o resultado da implementação; na V é apresentado a análise dos resultados; na VI é apresentada a conclusão.

II. RESUMO DA LITERATURA

A. NI-DAQmx e sua Biblioteca C

Para o entendimento desse trabalho é necessário o conhecimento de alguns aspectos estruturas e técnicos da arquitetura do driver NI-DAQmx:

1) O driver NI-DAQmx fornece uma interface de programação única para centenas de dispositivos multifuncionais de aquisição de dados, assim hardwares diferentes com a mesma capacidade rodam o mesmo código sem qualquer modificação.

2) Como pode ser ver na Fig. 1, o NI-DAQmx funciona como o intermediário entre as aplicações externas e o hardware, logo aplicações renomadas, como: Measurement & Automation Explorer, MATLAB e LabView, utilizam a mesma interface que as que utilizam a biblioteca C para ter acesso ao driver.

3) Tarefa é uma coleção de um ou mais canais virtuais com propriedades de tempo, execução e outras. Conceitualmente representa uma medição ou geração que deseja executar. Todos os canais em uma tarefa devem ter a mesma direção (entrada ou saída), embora possa incluir canais de diferentes unidades de medida (voltagem, temperatura, corrente, etc.). Para a maioria dos dispositivos, só uma tarefa por sistema pode ser executada por vez, mas alguns podem executar múltiplas simultaneamente.

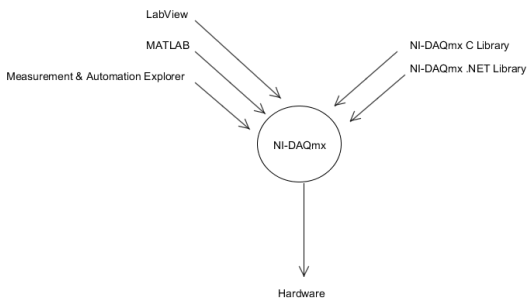

Fig. 1. Acesso ao *hardware* pelo NI-DAQmx.

4) Canal virtual, ou às vezes referido simplesmente por canal, são entidades que encapsulam o canal físico, juntamente com outras informações especificas do canal, como variação, configuração do terminal, e medida personalizada. Pode-se criar diferentes tipos de canais, de diferentes tipos de sinais – analógico, digital ou contador – e direções (entrada ou saída).

5) O relógio de amostra controla a taxa a qual as amostras são adquiridas e geradas. Esse define o intervalo de tempo entre as amostras. Cada turno do relógio inicializa a aquisição ou geração de amostra por canal. Já taxa de amostragem é a velocidade que o dispositivo adquire ou gera uma amostra em um canal.

6) Na biblioteca NI-DAQmx C, o sistema de tratamento de erros funciona da seguinte maneira: toda função retorna um

[1] Camada que traduz uma interface de uma biblioteca para uma outra interface compatível.

inteiro; se o valor desse inteiro for igual a zero então a operação foi bem sucedida; se o valor for maior que zero ocorreu um evento do tipo aviso; caso contrário, se o valor for menor que zero, ocorreu um evento do tipo erro. As informações sobre a classe do erro são obtidas através da função DAQmxGetErrorString. Já as informações especificas do erro ocorrido na ultima função que falhou são obtidas pela função DAQmxGetExtendedErrorInfo.

B. *Java Native Interface*

A *framework* JNI é um poderoso recurso que permite ter a vantagem da plataforma Java e ainda utilizar códigos escritos em outras linguagens de programação como C e C++. Como parte da implementação da máquina virtual Java (JVM), a JNI é uma interface bidirecional que permite aplicativos Java invocarem código nativo e vice-versa.

Aplicações Java que dependem da JNI não mais necessariamente rodam em múltiplos sistemas operacionais. Mesmo que a parte da aplicação escrita em Java seja portátil, é necessário recompilar a parte escrita em linguagem de programação nativa.

Enquanto a linguagem de programação Java é fortemente tipada e segura, as linguagens da programação nativa não são. Como resultado, deve se ter cuidado extra ao desenvolver aplicações usando JNI. Um comportamento não esperado do código nativo pode causar um erro na máquina virtual Java.

A JNI funciona da seguinte forma: quando JVM invocar a função, é passado um ponteiro *JNIEnv*, um tipo *jclass*, e todos argumentos declarados pelo método Java. O primeiro argumento aponta para outro ponteiro, esse último apontando para uma tabela de funções. Cada entrada nessa tabela de funções aponta para uma função JNI (vide Fig. 2). O segundo parâmetro difere dependendo se o método nativo é estático ou não. Se for estático, o segundo argumento aponta para a classe do qual o método é definido. Se não for estático, aponta para o objeto do qual o método é invocado.

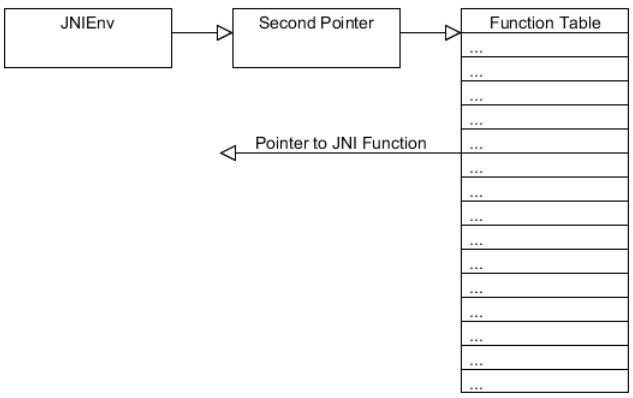

Fig. 2. JNIEnv.

Para a JVM conseguir achar a implementação do método nativo, é necessário que este esteja declarado conforme as normas da JNI. O nome do método nativo deve ser a concatenação dos seguintes componentes:

- Prefixo “Java_”
- Nome inteiro da classe, usando “_” ao invés de “.” como separador.
- “_” (*Underscore*)
- Nome do método

Os tipos dos argumentos na declaração do método nativo têm tipos correspondentes na língua de programação nativa. A JNI define um conjunto de tipos C e C++ que correspondem aos tipos da linguagem de programação Java.

Há duas categorias de tipos de dados em Java: tipos primitivos, iguais aos tipos *int*, *float* e *char*; e tipos referenciais, iguais aos tipos classe, instância e *array*.

A JNI trata tipos primitivos e referenciais de forma diferente. O mapeamento dos tipos primitivos é simples. Por exemplo, o tipo Java *int* é mapeado ao tipo C/C++ *jint*, enquanto o tipo Java *float* é mapeado para o tipo C/C++ *jfloat* (ambos *jint* e *jfloat* estão definidos na biblioteca “jni.h”).

Já o mapeamento de tipos referenciais é feito por meio de referências opacas. Essas que são ponteiros que referem a estruturas de dados internas na máquina virtual Java. O layout exato das estruturas de dados internas, no entanto, está escondido do programador. O código nativo deve manipular os objetos passados por meio das funções JNI apropriadas, essas que estão disponíveis através da interface *JNIEnv*. Por exemplo, o tipo JNI respondente ao *java.lang.String* é *jstring*. O valor exato da referencia *jstring* é irrelevante para o código nativo. Esse chama funções JNI iguais a função *GetStringUTFChars* para obter o conteúdo da *String*.

Todas as referências JNI tem o tipo *jobject*. Por conveniência e segurança, a JNI define um conjunto de tipos de referência que são subtipos de *jobject*. (*A* é um subtipo de *B* e toda instância de *A* é também uma instância de *B*).

```
1  ----------------------------------------------------------------
2  // HelloWorld.h
3
4  #include <jni.h>
5
6  JNIEXPORT void JNICALL
7  Java_com_HelloWorld_printHelloWorld(JNIEnv *env, jclass jc);
8
9  ----------------------------------------------------------------
10 // HelloWorld.c
11
12 #include "HelloWorld.h"
13
14 JNIEXPORT void JNICALL
15 Java_com_HelloWorld_printHelloWorld(JNIEnv *env, jclass jc){
16
17     printf("Hello World!");
18
19 }
20
21 ----------------------------------------------------------------
22 // HelloWorld.java
23
24 public class HelloWorld{
25
26     public native void printHelloWorld();
27
28     static{
29         System.loadLibrary("HelloWorld");
30     }
31 }
```

Fig. 3. *HelloWorld* usando JNI.

Uma transição do código nativo para o código em Java é muito mais custosa computacionalmente que uma chamada de método normal. Assim, deve se estabelecer um limite adequado a fim de minimizar o máximo possível de transições entre as linguagens. Uma boa prática é evitar o uso de tipos não primitivos, por exemplo, ao invés de passar um *java.awt.Point* como parâmetro, é melhor refazer o design do método afim passar como parâmetro dois inteiros *X* e *Y*.

III. Solução do Problema

O design proposto está dividido em dois componentes: o *Wrapper*, responsável pela conexão entre as linguagens; e a *Interface*, responsável pela implementação de uma camada pública de programação amigável.

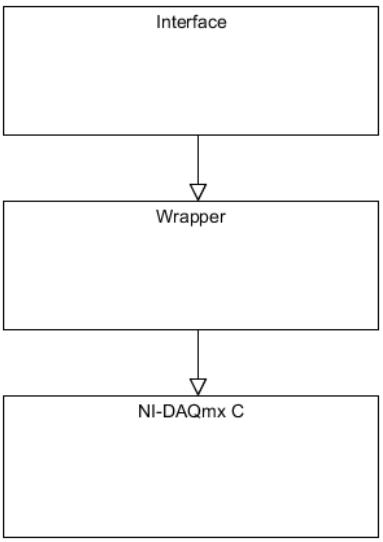

Fig. 4. Diagrama da Arquitetura.

A. Wrapper

O *wrapper* é desenvolvido em duas linguagens uma em C e outra em Java. Sua função é restrita a simplesmente repassar as funções já implementadas na NI-DAQmx C ao ambiente de desenvolvimento Java.

Devido à falta de mapeamento de alguns tipos correspondentes entre as linguagens Java e C por parte da JNI, relacionamos tipos similares entre as linguagens a fim de amenizar essa ausência (vide Tabela I). Embora essa solução em alguns casos peca no uso de memória.

TABELA I. Correspondência de Tipos Simplificada

NI-DAQmx C [a]	***JNI***	***Java***
char[]	jstring	String
int8	jbyte	byte
uInt8	jboolean	boolean
int16	jshort	short
uInt16	jchar	char
int32	jint	int
uInt32	jlong	long
float32	jfloat	float
float64	jdouble	double
int64	jlong	long
uInt64	jstring	String
bool32	jboolean	boolean
TaskHandle	jlong	long
CalHandle	jlong	long

a. A biblioteca NI-DAQmx C define novos tipos de dados.

Como não há como passar valores primitivos por referência na linguagem de programação Java, e em decorrência do uso constante desse método pela NI-DAQmx C, devido ao sistema de tratamento de erros, o *wrapper* irá repassar os códigos dos erros pelo lançamento de exceções.

B. Interface

A interface permitira a chamada de funções do *wrapper* por meio de objetos, tornando a programação mais amigável ao desenvolvedor de Java. Como a função da interface é basicamente organizar a lógica em objetos seu diagrama de classe é de baixa complexidade.

O NI-DAQmx C não possui nenhum sistema eventos ou *Callback*, e como a interface não é o único meio de acesso ao driver, temos que lidar com um possível problema de dados desatualizados. Para tentar minimizar esse problema não haverá um sistema de *caching* de informações, assim toda operação de obtenção de dados deverá recorrer ao driver.

IV. Resultado de Implementação

A. Wrapper

Na maioria das funções do *wrapper* a implementação se resume em passar os parâmetros da função nativa para a função correspondente da biblioteca NI-DAQmx C, e quando houver parâmetros não primitivos usar os próprios recursos da JNI para manipular esses objetos. Assim esse documento não irá cobrir todas as funções do NI-DAQmx C, mas cobrirá as funções que exijam uma implementação mais complexa.

Na Fig. 5 temos um pedaço da implementação da função *createTask* do *wrapper*, como podemos ver a implementação dessa função é basicamente passar os argumentos da função nativa para a função correspondente da biblioteca NI-DAQmx C, fazendo as conversões necessárias para manipular tipos não primitivos (nessa função especifica, o tipo String), e quando houver erro, disparar uma exceção. Essa implementação cobre a maioria das implementações das funções da biblioteca NI-DAQmx C.

```
-------------------------------------------------------------
// NIDAQmxJWrapper.java

public native static long createTask(String name)
        throws NIDAQmxJWrapperException;


-------------------------------------------------------------
// NIDAQmxJWrapper.h

JNIEXPORT jlong JNICALL
Java_com_wrapper_NIDAQmxJWrapper_createTask(
        JNIEnv *env, jclass jc, jstring taskName);


-------------------------------------------------------------
// NIDAQmxJWrapper.c

JNIEXPORT jlong JNICALL
Java_com_wrapper_NIDAQmxJWrapper_createTask(
        JNIEnv *env, jclass jc, jstring jtaskName) {
    int32 code;
    const char *taskName = NULL;
    TaskHandle taskHandle = NULL;

    taskName = getStringUTFChars(env, jtaskName, NULL );

    code = DAQmxCreateTask(taskName, &taskHandle);
    if (code < 0) {
        throwNIDAQmxJWrapperException(env, code);
        return 0;
    }

    (*env)->ReleaseStringUTFChars(env, jtaskName, taskName);

    return (jlong) (intptr_t) taskHandle;
}
```

Fig. 5. Algoritmo da função *createTask.*

A função *readAnalogF64* tem uma implementação um pouco mais complexa devido a necessidade de retornar um vetor de *double* (vide Fig. 6). Como esse não é um tipo primitivo é necessário usar métodos apropriados para criar uma instancia e escrever dados, nesse caso como é um vetor de *double* utiliza-se as funções já fornecidas pela JNI para esse tipo, a função *NewDoubleArray* para criação e *SetDoubleArrayRegion* para a escrita de dados.

Se fosse um vetor de *int* seria utilizado as funções *NewIntArray* e *SetIntArrayRegion*, se fosse de *byte* seria as funções *NewByteArray* e *SetByteArrayRegion.* Para mais funções ver TABELA II.

```
-------------------------------------------------------------
// NIDAQmxJWrapper.java

public native static double[] readAnalogF64(
        long pTask, int numSampsPerChan,
        double timeout, long fillMode)
        throws NIDAQmxJWrapperException;


-------------------------------------------------------------
// NIDAQmxJWrapper.h

JNIEXPORT jdoubleArray JNICALL
Java_com_wrapper_NIDAQmxJWrapper_readAnalogF64(
        JNIEnv *env, jclass jc, jlong ptaskHandle,
        jint numSampsPerChan, jdouble timeout,
        jboolean fillMode, jlong arraySizeInSamps);


-------------------------------------------------------------
// NIDAQmxJWrapper.c

JNIEXPORT jdoubleArray JNICALL
Java_com_wrapper_NIDAQmxJWrapper_readAnalogF64(
        JNIEnv *env, jclass jc, jlong ptaskHandle,
        jint numSampsPerChan, jdouble timeout,
        jboolean fillMode, jlong arraySizeInSamps) {

    int32 code;
    float64 *data = NULL;
    jdoubleArray array = NULL;
    TaskHandle taskHandle = (TaskHandle) (intptr_t) ptaskHandle;

    data = (float64 *) malloc(
            (size_t) (
            sizeof(float64) * arraySizeInSamps));

    code = DAQmxReadAnalogF64(
            taskHandle, numSampsPerChan,
            timeout, (bool32) fillMode,
            data, arraySizeInSamps, NULL, NULL );
    if (code < 0) {
        throwNIDAQmxJWrapperException(env, code);
        return NULL ;
    }

    array = (*env)->NewDoubleArray(env, arraySizeInSamps);
    (*env)->SetDoubleArrayRegion(
            env, array, 0, arraySizeInSamps, data);

    return array;
}
```

Fig. 6. Algoritmo da função *readAnalogF64.*

TABELA II. FUNÇÕES PARA CRIAÇÃO, LEITURA, ESCRITA E LIBERAÇÃO DE TIPOS NÃO PRIMITIVOS

Tipo	***Criação***	***Leitura***	***Escrita***	***Liberação***
Vetor de *boolean*	*NewBooleanArray*	*GetBooleanArrayRegion*	*SetBooleanArrayRegion*	*ReleaseBooleanArrayElements*
Vetor de *byte*	*NewByteArray*	*GetByteArrayRegion*	*SetByteArrayRegion*	*ReleaseByteArrayElements*
Vetor de *char*	*NewCharArray*	*GetCharArrayRegion*	*SetCharArrayRegion*	*ReleaseCharArrayElements*
Vetor de *int*	*NewIntArray*	*GetIntArrayRegion*	*SetIntArrayRegion*	*ReleaseIntArrayElements*
Vetor de *long*	*NewLongArray*	*GetLongArrayRegion*	*SetLongArrayRegion*	*ReleaseLongArrayElements*
Vetor de *float*	*NewFloatArray*	*GetFloatArrayRegion*	*SetFloatArrayRegion*	*ReleaseFloatArrayElements*
Vetor de *double*	*NewDoubleArray*	*GetDoubleArrayRegion*	*SetDoubleArrayRegion*	*ReleaseDoubleArrayElements*
String	*NewStringUTF*	*GetStringUTFChars*	Não se aplica [a]	*ReleaseStringUTFChars*

a. O tipo String em Java é imutável.

```
------------------------------------------------------------
// NIDAQmxJWrapper.java

public native static int writeAnalogF64(
        long ptaskHandle, int numSampsPerChan,
        boolean autoStart, double timeout,
        boolean dataLayout, double writeArray[])
        throws NIDAQmxJWrapperException;


------------------------------------------------------------
// NIDAQmxJWrapper.h

JNIEXPORT jint JNICALL
Java_com_wrapper_NIDAQmxJWrapper_writeAnalogF64(
        JNIEnv *env, jclass jc, jlong ptaskHandle,
        jint numSampsPerChan, jboolean autoStart,
        jdouble timeout, jboolean dataLayout,
        jdoubleArray writeArray);


------------------------------------------------------------
// NIDAQmxJWrapper.c

JNIEXPORT jint JNICALL
Java_com_wrapper_NIDAQmxJWrapper_writeAnalogF64(
        JNIEnv *env, jclass jc, jlong ptaskHandle,
        jint numSampsPerChan, jboolean autoStart,
        jdouble timeout, jboolean dataLayout,
        jdoubleArray writeArray){
    int32 code;
    int32 sampsPerChanWritten;
    float64 *data = NULL;
    TaskHandle taskHandle = (TaskHandle) (intptr_t) ptaskHandle;

    data = env->GetDoubleArrayElements(writeArray, NULL);

    code = DAQmxWriteAnalogF64(
        taskHandle, numSampsPerChan, autoStart, timeout,
        dataLayout, data, &sampsPerChanWritten, NULL);

    if (code < 0){
        throwNIDAQmxJWrapperException(env, code);
        return 0;
    }

    return (jint) sampsPerChanWritten;
}
```

Fig. 7. Algoritmo da função *writeAnalogF64*.

A função *registerDoneEvent* (ver Fig. 9) registra uma função para ser chamada quando uma tarefa for terminada por erro, ou quando a escrita ou leitura estiver completa. Assim para ser implementada no *wrapper*, a função nativa deve registrar outra função para ser chamada, e essa ultima deve invocar a função Java que atuará após o evento ser disparado (quando a tarefa estiver terminada).

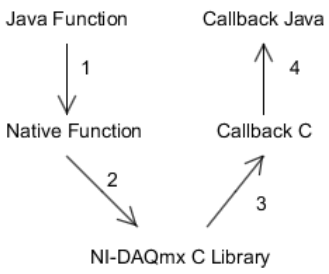

Fig. 8. Diagrama de chamada da função *registerDoneEvent*.

```
------------------------------------------------------------
// NIDAQmxJWrapper.java

public native static void registerDoneEvent()
            throws NIDAQmxJWrapperException;


------------------------------------------------------------
// NIDAQmxJWrapper.h

JNIEXPORT void JNICALL
Java_com_wrapper_NIDAQmxJWrapper_registerDoneEvent(
        JNIEnv *env, jclass jc,
        jlong ptaskHandle, jlong options,
        DAQmxDoneEventCallbackPtr callbackFunction,
        void *callbackData);


------------------------------------------------------------
// NIDAQmxJWrapper.c

JNIEXPORT void JNICALL
Java_com_wrapper_NIDAQmxJWrapper_registerDoneEvent(
        JNIEnv *env, jclass jc, jlong ptaskHandle,
        jlong options,
        DAQmxDoneEventCallbackPtr callbackFunction,
        void *callbackData) {

        int32 code;
        TaskHandle taskHandle = (TaskHandle)(intptr_t)ptaskHandle;

        code = DAQmxRegisterDoneEvent(
            taskHandle, options,
            RegisterDoneEventCallback, NULL);
        if (code < 0){
            throwNIDAQmxJWrapperException(env, code);
            return;
        }

}

------------------------------------------------------------
// NIDAQmxJWrapper.c

int32 CVICALLBACK RegisterDoneEventCallback (
    TaskHandle taskHandle, int32 status, void *callbackData){

        class = (*env)->FindClass(
            env, "com/wrapper/NIDAQmxJWrapper");

        jmethodID method = env->GetMethodID(
                    class, "RegisterDoneEventCallback",
                    NULL);
        if (method == 0){
            throwNIDAQmxJWrapperException(
            env, "Error calling Java CallBack function!");

            return;
        }

        (*env)->CallVoidMethod(env,obj,aMethodID, NULL);
}

------------------------------------------------------------
// NIDAQmxJWrapper.java

public void registerDoneEventCallback(){
    // ...
}
```

Fig. 9. Algoritmo da função *registerDoneEvent*.

B. *Interface*

Uma tarefa no NI-DAQmx tem um conjunto de propriedades (nome, *hardware* que será usado, *timing*, etc..) e canais, assim é modelada como uma classe, essa que será responsável pela criação e gerenciamento de canais.

Um canal só pode ser de um único tipo de sinal, e de entrada ou saída. Essa regra de negocio é obedecida da seguinte forma: a classe abstrata *Channel* é o pai de todos os canais, os seus primeiros herdeiros são as classes abstratas que representam os canais de diferentes tipos de sinais: para

analógico, *AChannel*; para digital, *DChannel*; e para contador, *CChannel*. Esses últimos são pais dos canais que os representam com diferentes tipos de direções; para analógico, entrada e saída respectivamente, *AIChannel* e *AOChannel*; para digital, entrada e saída respectivamente, *DIChannel* e *DOChannel*; e para contador, entrada e saída respectivamente, *CIChannel* e *COChannel*. As classes concretas dos canais implementam a interface que corresponde a direção do canal: para entrada, implementam a interface *IChannel*; e para saída, implementam a interface *OChannel*.

Para ler os canais de diferentes tipos de sinais temos as classes: para analógico, *AChannelReader*; para digital, *DChannelReader*; para contador, *CChannelReader*. Essas herdeiras da classe *ChannelReader*. Similarmente, para escrever temos as classes: para analógico, *AChannelWriter*; para digital, *DChannelWriter*, para contador, *CChannelWriter*.

Para as funções do sistema, como listar todas as tarefas ou todos os aparelhos disponíveis, a classe *System* é a responsável.

V. ANÁLISE DOS RESULTADOS

Diante dos resultados obtidos pela implementação levando em consideração objetivo inicial traçado para o trabalho, criar um mecanismo de acesso de aplicativos desenvolvidos em Java ao driver NI-DAQmx, pode ser constatado que:

1) A utilização da *framework* JNI para ter acesso e ser acessado apartir do código nativo, escrito na linguagem de programação C, funcionou como esperado.

2) Apesar de alguns empecilhos, como algumas faltas de correspondências exatadas entre tipos de dados da linguagem Java e C, ao primeiro momento, não houve problemas decorrentes e gerou-se resultados concretos como se pode ver na Fig. 11, Fig. 12 e Fig. 13.

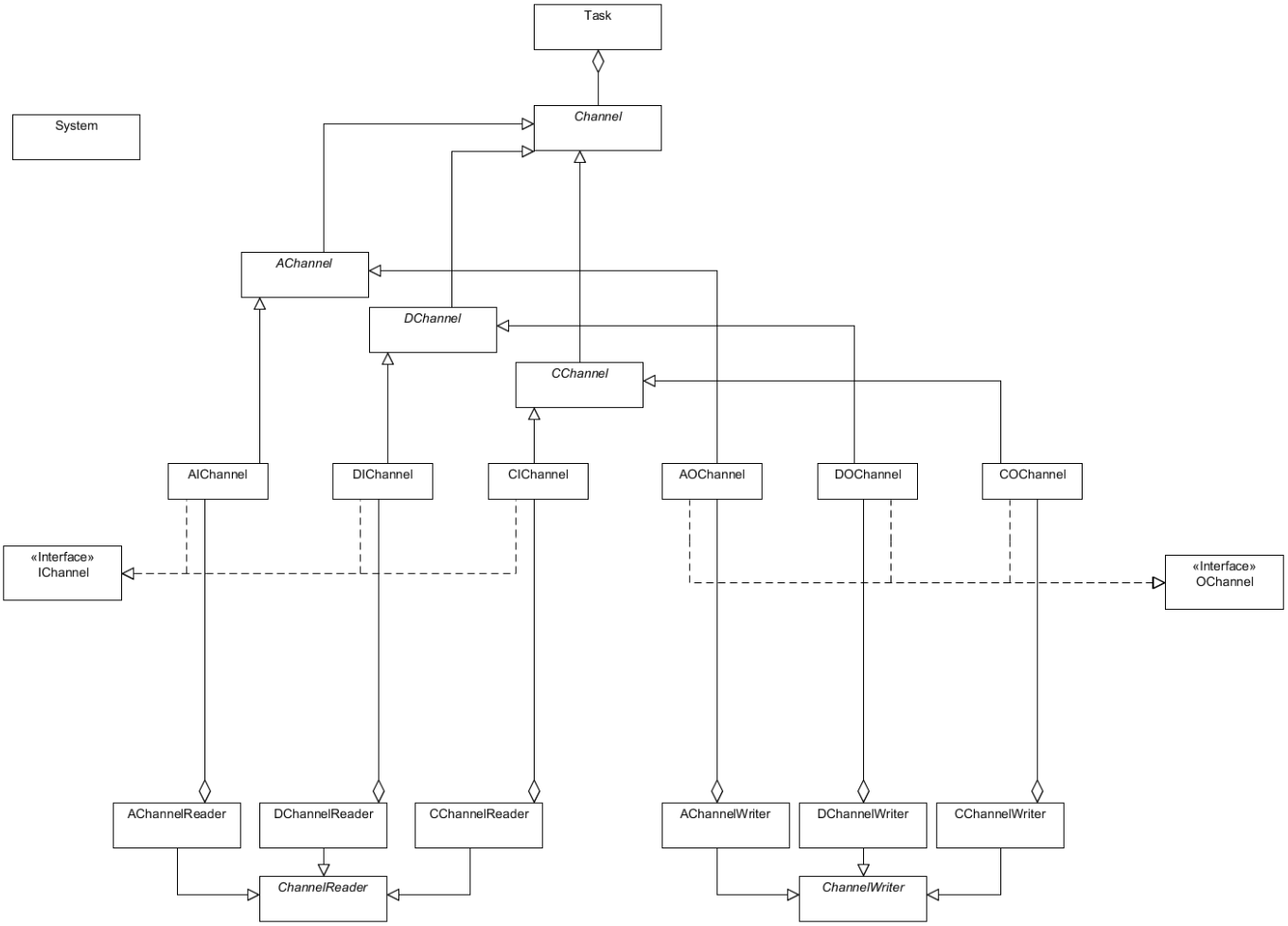

Fig. 10. Diagrama de classe simplificado da interface.

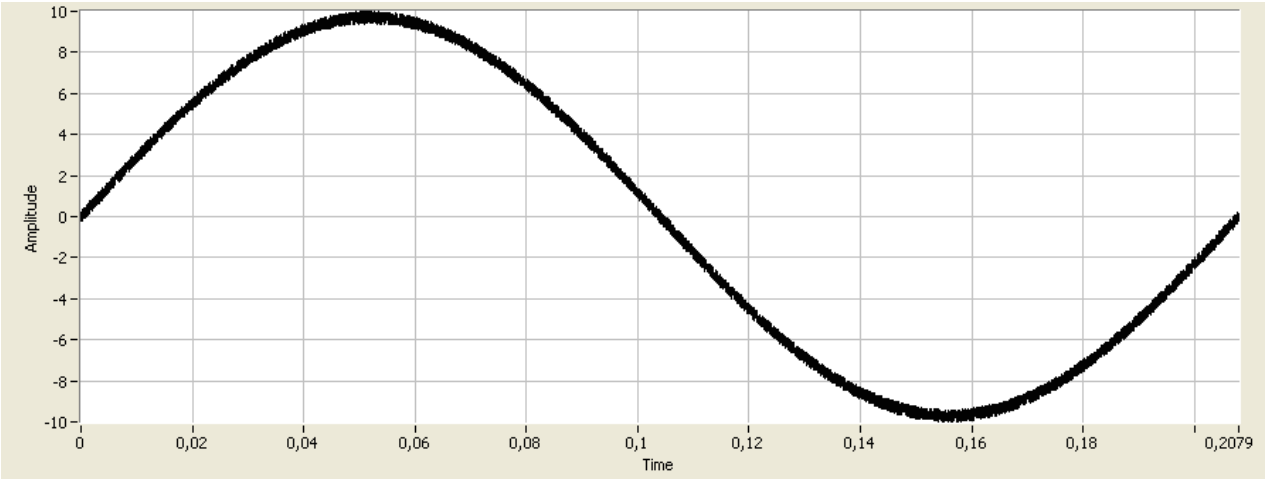

Fig. 11. Saída da função *readAnalogF64* com os parâmetros: amplitude máxima 10 e minima -10, número de amostrar para ler 10.000, frequência 48.000Hz.

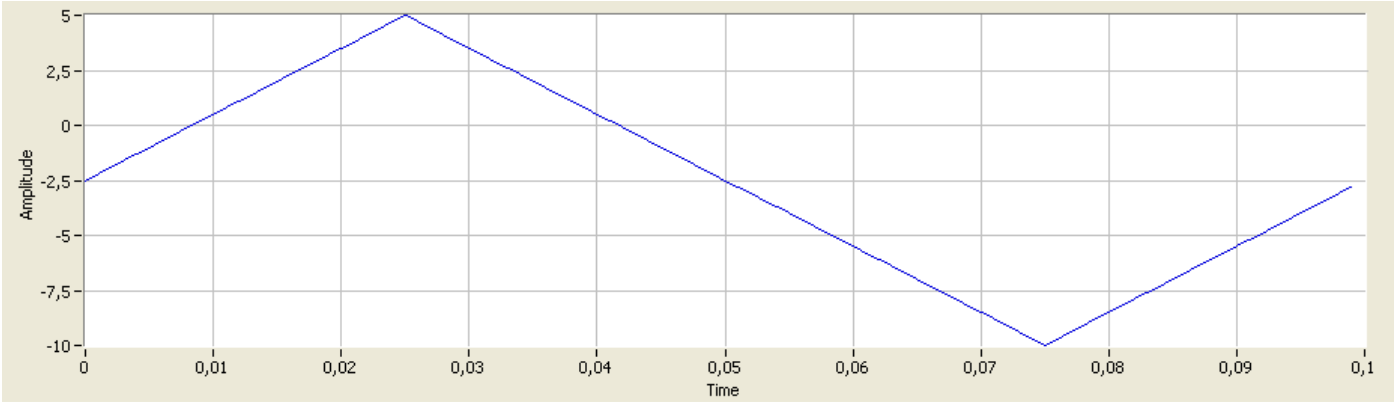

Fig. 12. Sinal gerado pela função *writeAnalogF64* com os parâmetros: amplitude máxima 5 e minima -10, número de amostrar para ler 10.000, frequência 48.000Hz, sinal do tipo triangular.

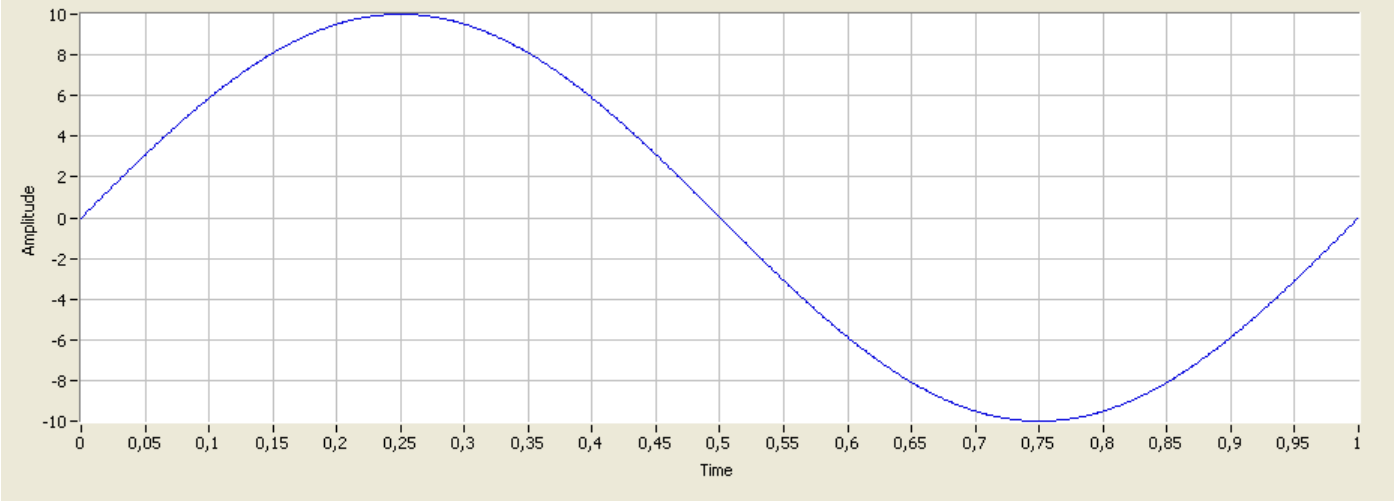

Fig. 13. Sinal gerado pela função *writeAnalogF64* com os parâmetros: amplitude máxima 10 e minima -10, número de amostrar para ler 10.000, frequência 48.000Hz, sinal do tipo senoidal.

VI. CONCLUSÕES

Este trabalho apresentou um mecanismo de acesso de aplicativos desenvolvidos com a tecnologia Java ao driver da NI-DAQmx. O mecanismo proposto foi divido em duas partes: o wrapper, camada responsável por traduzir a biblioteca NI-DAQmx C para o ambiente de desenvolvimento Java; e a interface, responsável por organizar as funções em objetos, tornando o ambiente mais amigável ao desenvolvedor. Durante o processo de desenvolvimento foi possivel perceber que:

1) Apesar de falta de matérias didáticos, a *framework* Java Native Interface (JNI), se for implementada corretamente, se apresenta viável para permitir que o Java chame e seja chamado por programas nativos e bibliotecas escritas em outras linguagens de programação como C, C++ e assembly. **2)** O processo de desenvolvimento de um *wrapper*, se resume, predominantemente, a repetição de técnicas de programação.

REFERÊNCIAS

[1] S. Liang, The Java native interface: programmer's guide and specification. Addison-Wesley, vol. 1, 1999.

[2] R. Gordon, Essential JNI: Java Native Interface. Prentice Hall, vol. 1, 1998.

[3] C. S. Horstmann and G. Cornell, Core Java, Vol. 2: Advanced Features. Prentice Hall, vol. 8, 2008.

[4] M. Dawson, G. Johnson and A. Low, "Best practices for using the Java Native Interface".

Internet: www.ibm.com/developerworks/java/library/j-jni, Jul. 7, 2009 [Jan. 15, 2013]

[5] S. Stricker, "Java programming with JNI". Internet: www.ibm.com/developerworks/java/tutorials/j-jni/, Mar. 26, 2002 [Jan. 13, 2013]