SPARK

Software Modulator for Digital Radio

Native Device Interface (NDI)

Version: 1.1

Author: Michael Feilen
Birkerstraße 34
D-80636 Munich (Germany)
mail@drm-sender.de

Last changed: August, 6th 2011

First published: September, 9th 2009

1 What is the Spark NDI?

The NDI is a set of C++ functions which allow for using proprietary input and output devices together with Spark. Figure 1 depicts the role of the NDI in the Spark I/O environment. The NDI is provided as a shared library to Spark, i.e. as a Dynamic Linked Library (DLL) under Windows or as a Shared Object File (SO) under Linux.

In short, the NDI enables you to:

- Interface your output devices with Spark for **RF output**.
- Interface your input devices with Spark for **audio input**.

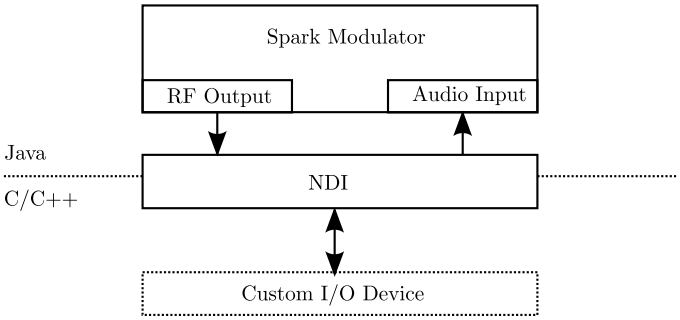

Figure 1: Role of the native device interface in the Spark environment.

2 Architecture

The interface to Spark is realized by communicating to the static functions as stated in the header file `NativePCMDevice.h`. This file contains several functions to read or write to the device and to control the I/O activity. Using a direct implementation of the functions defined in `NativePCMDevice.h` is not recommended. In the following paragraphs describe the recommended way to connect your I/O devices to Spark.

An NDI device may consist of multiple input lines, i.e. **source lines**, and multiple output lines, i.e. **target lines**.

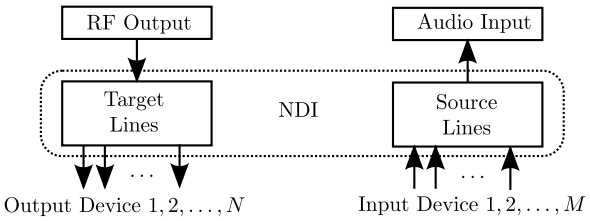

Figure 2: Schematic of Spark native PCM device interface.

Figure 1 shows a schematic overview of the Spark NDI architecture. Spark forwards the RF data to the NDI target lines or retrieves the audio data from the NDI source lines.

The classes `SourceLine` and `TargetLine` classes inherit from the superclass `Line` as shown in Figure 3. All member functions of these classes are pure virtual.

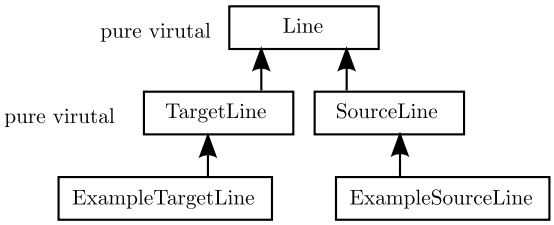

Figure 3: Inheritance Model

3 Line Names and UIDs

Each source line and target line is uniquely identified by an **unique line id** (UID), reflected by a 32 bit integer number. The UIDs are necessary for line identification and line management by Spark. Source and target lines must not have the same line UID.

A native PCM device must return an ASCII-coded name by the to the application by overwriting the function:

```
void GetName(char name[], uint32 numChars, uint32 *numCharsRequired);
```

The example device implementation `ExampleDevice.cpp` is defined to return the name *Native Example Device* to Spark as shown in Figure 4.

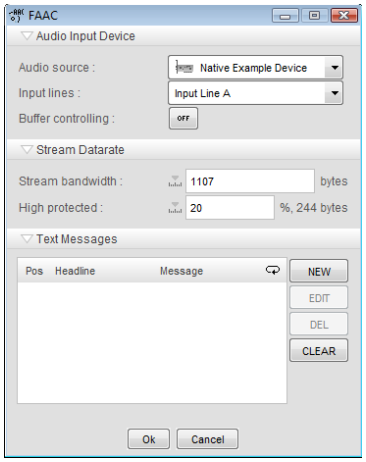

Figure 4: The *Native Example Device* in the FAAC configuration dialog

The example device implementation contains two source line instances with the names **Input Line A** and **Input Line B**. Furthermore, the example device contains two target line instances **Output Line A** and **Output Line B** as shown in Figure 5.

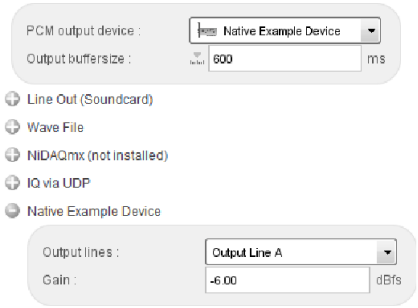

Figure 5: The *Native Example Device* in the Spark output device dialog.

The line UIDs in the example device have been chosen arbitrarily and do not appear in the Spark UI.

4 Example Implementation

An example may say more than thousand words. The following files contain a simple implementation example for the source and target line interfaces:

```
source/ExampleDevice.cpp
source/ExampleTargetLine.cpp
source/ExampleSourceLine.cpp
```

It is **highly recommended** to use the existing implementation as defined in the file `ExampleDevice.cpp` and create your own source and target lines. As shown in the last section, your source line implementations must implement the pure virtual functions of the class `SourceLine` and your target line implementation must implement the pure virtual functions of the class `TargetLine`.

For a deeper understanding of the I/O control functionality it is strongly recommended to have a look at the example implementations `ExampleSourceLine.cpp` and `ExampleTargetLine.cpp`.

After successful compilation of the source files, the created dynamic link library `pcmout.dll` under Windows or `libpcmout.lo` under Linux must be copied into the folder where the Spark executable is located.

5 Reading and Writing of Data

In this section the data format of the read and write functions of the source and target lines will be explained.

The source line read function passes a quantized pulse-code-modulated (PCM) frame as a byte-array (`char pBytes[]`) which is to be filled by the respective input device. Similarly, the target line write function passes a PCM frame as a byte-array (`char pBytes[]`) from which the data can be read to be written to the respective output device.

A PCM frame is partitioned into a certain number of bytes depending on the number of bytes per sample. The byte-order is little-endian. Hence, the number of samples per function-call can be calculated as:

```
numSamples = numBytes / numBytesSample
```

The stream of samples for each channel is interleaved depending on the number of channels. An interleaved block of samples containing one sample for each channel is called a sample block. Thus, the number of sample blocks per function-call is calculated as follows:

```
numSampleBlocks = numSamples / numChannels
```

The PCM frame structure and an example of a sample block for a signal with 2 channels (I and Q) and a resolution of 16 bits per sample is shown in Figure 6.

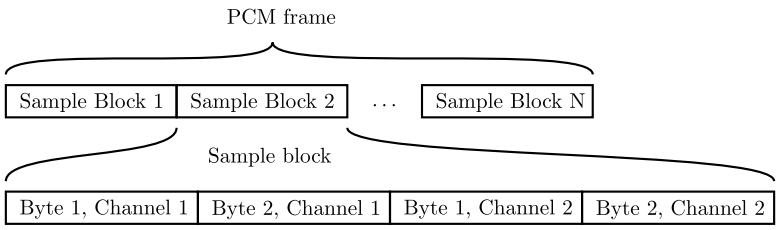

Figure 6: Example of a sample block for a complex 16 bit IQ signal (2 channels) and a PCM frame with N sample blocks.

The following source code shows an example function that decodes a sample block comprising of two 16 bit samples `i` and `q` in C.

```
function process_iq_16bit(unsigned char* pBytes, unsigned int numBytes)
{
        int n;
        short i, q;

        for (n = 0; n < numBytes; n += 4)
        {
                /* Decode 16 bit signed integer (I/Q) samples */
                i = (pBytes[n+1] << 8) | pBytes[n];
                q = (pBytes[n+3] << 8) | pBytes[n+2];

                /* Do something with i and q */
        }
}
```