Version 1.0 September 30, 2013

Creating Native Libraries for ProcessJ Using C

by Matt B. Pedersen

1 Introduction

A *native* library in ProcessJ is a library that is written in C. There are two different kinds of native libraries in ProcessJ:

1. A library that maps **directly** to an existing C library like for example `math.h`.
2. A library that is implemented in C but does not map to any existing libraries.

All ProcessJ files that implement libraries must include the pragma `LIBRARY` as well as `FILE` and either `NATIVELIB` (if mapping to an existing library) or `NATIVE` if the implementation will be written in C.
All ProcessJ library files must declare a package name.

2 Mapping to and Existing Library

To define a native library mapping in ProcessJ follow the following example where we map some of the existing constants an procedures from C's `math.h` library to a library called `math` in the `std` package. Simply create this ProcessJ file:

```
#pragma LIBRARY;
#pragma NATIVELIB "math.h";
#pragma LANGUAGE "C";
#pragma FILE "math";

package std;

public native const double M_PI;        /* pi */
public native const double M_PI_2;      /* pi/2 */
```

```
public native const double M_PI_4;      /* pi/4 */
public native const double M_1_PI;      /* 1/pi */
public native const double M_2_PI;      /* 2/pi */
public native const double M_2_SQRTPI;  /* 2/sqrt(pi) */
public native const double M_SQRT2;     /* sqrt(2) */
public native const double M_SQRT1_2;   /* 1/sqrt(2) */

public native proc double acos(double x) ;
public native proc double asin(double x) ;
public native proc double atan(double x) ;
public native proc double atan2(double y, double x) ;
public native proc double cos(double x) ;
public native proc double cosh(double x) ;
public native proc double sin(double x) ;
public native proc double sinh(double x) ;
public native proc double tanh(double x) ;
public native proc double exp(double x) ;
public native proc double ldexp(double x, int exponent) ;
public native proc double log(double x) ;
public native proc double log10(double x) ;
public native proc double pow(double x, double y) ;
public native proc double sqrt(double x) ;
public native proc double ceil(double x) ;
public native proc double fabs(double x) ;
public native proc long abs(long x) ;
public native proc int abs(int x) ;
public native proc double floor(double x) ;
public native proc double fmod(double x, double y) ;
```

Both constants and procedures must be declared native and procedures cannot have a body and constants cannot be initialized.

The pragmas instruct the compiler to create a library with native C mappings to <`math.h`>.

The compiler will generate: `std_math.h` which will look like this:

```
#ifndef _LIB_STD_AMTH_
#define _LIB_STD_MATH_
#include <math.h>
```

```
#endif
```

which should be moved to `lib/C/include`

and `std_math.c` which will look like this:

```
#ifndef _STD_MATH_H
#define _STD_MATH_H
#include "std_math.h"
#endif
```

which should be moved to `lib/C/src`.

The object file `std_math.o` should be moved to `/lib/C/obj`.

The original ProcessJ file, `math.pj` should be moved to `include/C/std/`.

To avoid manually moving files around you can use the `pjc-install-c-library` script.

3 Self Written C libraries

To write a native library in C that does not map to an existing C library the `NATIVELIB` pragma should be replaced by the `NATIVE` pragma.

Procedures should use he `native` keyword and have no implementation in the ProcessJ file.

Constants should **not** be declared `native`. Non-native constants are declared without the `native` keyword and an initializer is required.

ProcessJ allows procedure overloading, C does not, we will return to this issue a little later.

Here is an example of a ProcesJ native library that does not map directly to an existing C library:

```
#pragma LIBRARY;
#pragma NATIVE;
#pragma FILE "file";
```

```
#pragma LANGUAGE "C";

package io;

public native const string READ;
public native const string WRITE;

public native proc int fileOpen(string fileName, string mode) ;
public native proc int fileClose(int file) ;

public native proc int fileWrite(int file, string data) ;
public native proc int fileWrite(int file, int data) ;
public native proc int fileWrite(int file, float data) ;
public native proc int fileWrite(int file, long data) ;
public native proc int fileWrite(int file, double data) ;
public native proc int fileWrite(int file, short data) ;
```

The ProcessJ compiler will generate a header (`.h`) file `io_file` which will look like this:

```
#ifndef _LIB_IO_FILE_
#define _LIB_IO_FILE_

// Add #include statements and constants here

int io_fileOpen_TT(char* fileName, char* Mode) ;
int io_fileClose_I(int file) ;
int io_fileWrite_IT(int file, char* data) ;
int io_fileWrite_II(int file, int data) ;
int io_fileWrite_IF(int file, float data) ;
int io_fileWrite_IJ(int file, long data) ;
int io_fileWrite_ID(int file, double data) ;
int io_fileWrite_IS(int file, short data) ;

#endif
```

Since the ProcessJ file contained multiple definitions (with different signatures) of the `fileWrite` procedure, these must be generated with different names for the C compiler to be happy. Since the signatures of the parameters will differ, we use these to distinguish the different versions of the procedure.

The generated C file looks like this:

```
#ifndef _IO_FILE_H
#define _IO_FILE_H
#include "io_file.h"
int io_fileOpen_TT(char* fileName, char* Mode) {
  // implementation code goes here.
}

int io_fileClose_I(int file) {
  // implementation code goes here.
}

int io_fileWrite_IT(int file, char* data) {
  // implementation code goes here.
}

int io_fileWrite_II(int file, int data) {
  // implementation code goes here.
}

int io_fileWrite_IF(int file, float data) {
  // implementation code goes here.
}

int io_fileWrite_IJ(int file, long data) {
  // implementation code goes here.
}

int io_fileWrite_ID(int file, double data) {
  // implementation code goes here.
}

int io_fileWrite_IS(int file, short data) {
  // implementation code goes here.
}

#endif
```

Also note, that the ProcessJ `string` type is converted to `char*` and the

`boolean` type is converted to `int`.

Implement the library in the C templates generated by the compiler; if additional internal functionality is required it can be added in this file as well, but be careful with naming these functions and "global" variables.

The location of the `.h`, `.c`, `.o` and `.pj` files go in the same locations as mentioned in the previous section.

The `pjc-install-c-library` can be used in the exact same way as in the previous section: if the third parameters is `test` the script will check for all the files and compile the library, but not move the files. To force a move of the files, replace `test` by `install`.