

## RxTx 2010 Rewrite

### The Goal:

To provide a reference implementation of the `javax.comm` package using open source code. The existing version 2.x source code will be completely re-written.

### Design Objectives:

- Reference implementation – strictly adheres to the `javax.comm` API.
- Small footprint - Java and C code should strive to be as small as possible.
- Good performance – Java code should provide the shortest path possible to the native code. Native code should provide the shortest path possible to the underlying system.
- Fault tolerant - Java and C code should handle I/O errors gracefully, providing good error recovery.
- Proven – Java unit tests should achieve 100% code coverage.
- Compatible – the rewrite should be a drop-in replacement for the `javax.comm` library.

## Architecture

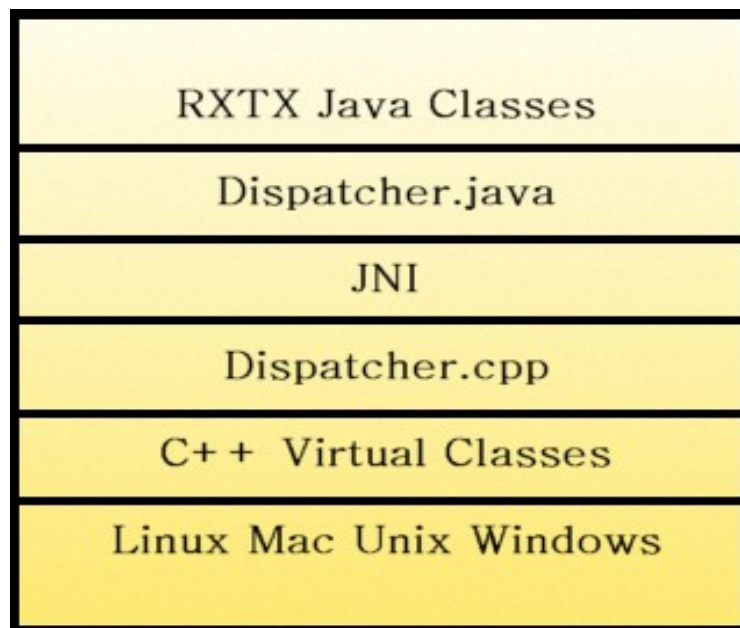
The Java classes have very little included behavior. All I/O related methods are delegated to native code. Java code is responsible for starting threads and thread synchronization. In general, Java code is “in charge.”

Native code has very little included behavior, and it does not start any threads. Native functions have only two results: success or exception thrown. Exceptions will be propagated back to the Java code.

Java classes delegate calls to native code through the `Dispatcher` class. This is for convenience – the Java->JNI->C connection is kept in a single place. The `Dispatcher` class defines a contract between Java code and native code – both sides must obey the contract for the library to operate correctly.

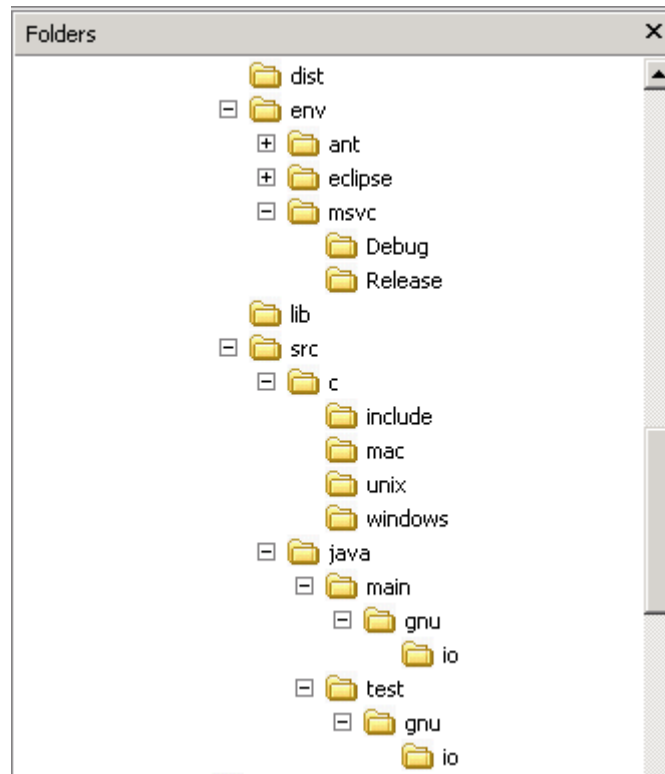
All of the native methods defined in the Java `Dispatcher` class are implemented in native code in the `Dispatcher.cpp` file. The native `Dispatcher` code is responsible for JNI data type marshalling/unmarshalling, argument validation, and exception handling.

After incoming arguments are unmarshalled and validated, the function calls are delegated to a set of C++ virtual classes. Various platforms implement the virtual classes. In effect, conditional compilation is replaced by polymorphism. When the function returns, `Dispatcher.cpp` marshalls the returned data and then returns control to Java code. If the native function throws an exception, `Dispatcher.cpp` catches the C++ exception, converts it to a Java exception, and passes it back to Java code.



## Project Layout

New folders have been created and source files have been reorganized.



The `dist` folder is the final target folder for distribution files.

The `env` folder contains subfolders for various development environments. In those folders the development environments keep their make files, settings, intermediate files, etc.

The `lib` folder is for third-party Java libraries – like Junit and Cobertura.

The `src` folder is the root folder for the source code.

The `src/c/include` folder contains platform-independent C code that is shared by all platforms, and the other folders in `c` are for platform-specific code.

The main Java source code is in `src/java/main` and the Java unit test code is in `src/java/test`.

Not pictured is the `doc` folder – which is created when the `doc` (JavaDoc) `ant` task is run.

## **Code Style**

In general, code is organized in a modular fashion and it uses object-oriented design patterns. The source code is written in a way that is self-documenting - method and function names describe what they do, and parameter and variable names describe what they contain. There are very few explanatory comments in the code.

The source code does not contain debugging code. Debugging code should be removed once the bug has been found and corrected.

## **Author and License**

The 2010 Rewrite Java code and Windows native code were created by Adrian Crum and they are essentially “clean room” implementations. The Java package name, the constants, and the RXTX identifier were kept from version 2.x - everything else is an original work or it was derived from readily available code samples. In those places where new code looks similar to old code, it is because both versions were derived from the same source.

The author would prefer to attribute authorship to the RXTX developer community rather than to an individual, and the source code comments reflect that.

The original RXTX license has been preserved for legal reasons.

## **Rewrite versus Version 2.x**

The rewrite does not contain RXTX-specific extensions to the original `javax.comm` API. Those extensions are best left to application programmers, and the API provides a means to implement those extensions outside of this library.

The rewrite creates a single native code library file instead of two. The native code library file name is `RXTXnative.*`.

The rewrite should work in existing 2.x applications. Any differences encountered are due to the rewrite's strict conformance to the original API – something 2.x didn't achieve. In other words, if the rewrite does not work properly in an existing application, it is because version 2.x allowed the application to do something it wasn't supposed to do.

Port discovery and enumeration have been pushed down to native code. It didn't make sense to have platform-specific code in Java.

## Implementing Native Code

Porting the rewrite to various platforms is fairly straightforward – simply implement two C++ virtual classes and two C++ class functions.

Native code implementations must implement the virtual classes `ParallelPort` and `SerialPort`.

Native code implementations must implement `PortInfoList::getValidPorts()` and `CommPortFactory::getInstance(const char *portName, int portType)`.

## Implementing Custom Port Types

Custom port types can be implemented in two ways: create a separate native code library containing the custom port types and load it with a custom `CommDriver` implementation, or add the custom port types to the RXTX native code.

An example of loading a custom `CommDriver` implementation can be found in the `CommDriver.java` [JavaDocs](#).

To add a custom port type to the RXTX native code, simply have it implement one of the C++ virtual classes (`ParallelPort` or `SerialPort`), and then override `CommPortFactory::getInstance(const char *portName, int portType)` to return an instance of the custom port type. To include the custom port type in the port enumeration logic, override the `PortInfoList::getValidPorts()` function.