QSocket: System Additions

This thread has been locked by a moderator.

13

IMPORTANT If you haven't yet read Calling BSD Sockets from Swift, do that first.

With addressing sorted out, there's the question of how you call the BSD Sockets API itself. The Swift System framework has wrappers for some BSD calls, like `open` and `close`, but not for BSD Sockets. However, using `QSockAddr` it's relatively easy to create these wrappers yourself.

The full set of wrappers is rather large, so I'm just going to post the most critical stuff. Let's start with opening a socket:

```
extension FileDescriptor {

    /// Creates a socket.
    ///
    /// Equivalent to the `socket` BSD Sockets call.

    public static func socket(_ domain: CInt, _ type: CInt, _ proto: CInt, retryOnInterrupt: Bool = true) throws -> FileDescriptor
{
        let socket = try errnoQ(retryOnInterrupt: retryOnInterrupt) {
            Foundation.socket(domain, type, proto)
        }
        return FileDescriptor(rawValue: socket)
    }
}
```

This extends the Swift System `FileDescriptor` type, which is a common feature of all these wrappers. To close the socket, call the file descriptor's `close()` method.

This wrapper, and all the wrappers below, rely on the following helper:

```
/// Calls a closure that might fail with `EINTR`.
///
/// This calls the supplied closure and, if it returns a negative value,
/// extracts the error from `errno`.  If `retryOnInterrupt` on interrupt is set
/// and the error is `EINTR`, it repeats the call.  Otherwise it throws that
/// error.
///
/// This is marked with `@discardableResult` because in many cases, like
/// `setsockopt`, the result isn't relevant.
///
/// - Parameters:
///   - retryOnInterrupt: If true, check for `EINTR` and call the closure again.
///   - body: The closure to call.
/// - Returns: The closure result. This will not be negative.

@discardableResult
public func errnoQ<Result: SignedInteger>(retryOnInterrupt: Bool, _ body: () -> Result) throws -> Result {
    repeat {
        let result = body()
        let e = Foundation.errno
        if result >= 0 {
            return result
        }
        if retryOnInterrupt && e == Foundation.EINTR {
            continue
        }
        throw Errno(rawValue: e)
    } while true
}
```

The next step typically involves connecting or binding the socket. Here's how you'd call `connect` using `QSockAddr.withSockAddr(…)` to convert the address:

```
extension FileDescriptor {

    /// Connects a socket to an address.
    ///
    /// Equivalent to the `connect` BSD Sockets call.
    ///
    /// The `ignoreInProgressError` parameter defaults to false.  If you set it,
    /// the call treats an `EINPROGRESS` error as success.  Do this for a
    /// non-blocking connect, where you monitor the connection status using
    /// `select` or one of its friends.

    public func connect(_ address: String, _ port: UInt16, ignoreInProgressError: Bool = false, retryOnInterrupt: Bool = true)
throws {
        _ = try QSockAddr.withSockAddr(address: address, port: port) { sa, saLen in
            try errnoQ(retryOnInterrupt: retryOnInterrupt) {
                var err = Foundation.connect(self.rawValue, sa, saLen)
                if err < 0 && errno == EINPROGRESS {
                    err = 0
                }
                return err
            }
        }
    }
}
```

The wrapper for `bind` is very similar (sans the special case for `EINPROGRESS`).

The listen and accept operations are super easy:

```
extension FileDescriptor {

    /// Configures a socket for listening.
    ///
    /// Equivalent to the `listen` BSD Sockets call.

    public func listen(_ backlog: CInt, retryOnInterrupt: Bool = true) throws {
        try errnoQ(retryOnInterrupt: retryOnInterrupt) {
            Foundation.listen(self.rawValue, backlog)
        }
    }

    /// Accepts an incoming connection
    ///
    /// Equivalent to the `accept` BSD Sockets call when you pass `NULL` to the
    /// `address` and `address_len` parameters.  If you need the connection's
    /// remote address, call ``getPeerName(retryOnInterrupt:)``.

    public func accept(retryOnInterrupt: Bool = true) throws -> FileDescriptor {
        let newSocket = try errnoQ(retryOnInterrupt: retryOnInterrupt) {
            Foundation.accept(self.rawValue, nil, nil)
        }
        return FileDescriptor(rawValue: newSocket)
    }
}
```

As is the get-local-address operation:

```
extension FileDescriptor {

    /// Gets the socket's local address.
    ///
    /// Equivalent to the `getsockname` BSD Sockets call.

    public func getSockName(retryOnInterrupt: Bool = true) throws -> (address: String, port: UInt16) {
        let result = try QSockAddr.fromSockAddr() { sa, saLen in
            try errnoQ(retryOnInterrupt: retryOnInterrupt) {
                Foundation.getsockname(self.rawValue, sa, &saLen)
            }
        }
        return (result.address, result.port)
    }
}
```

To get the remote address, wrap `getpeername` in the same way.

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

Network

Reply

Posted 5 days ago by eskimo

Add a Comment