1. Introduction. This is a socket library for Chez Scheme which attempts to remain as faithful as possible to BSD sockets while still maintaining a normal operation that will be familiar to Scheme programmers. Procedures are documented inline at the top of their main definitions. The current implementation uses records instead of magic numbers. Hopefully this keeps the system a little more portable. Only Internet and Local/Unix domain sockets are built-in, though this library supports additional types through its data hierarchy. The program itself is layed out like so:

⟨ Foreign code utilities 2 ⟩
⟨ Foreign constants 79 ⟩
⟨ Foreign functions 82 ⟩
⟨ Foreign code initialization 102 ⟩
⟨ Datatype definitions 5 ⟩
⟨ Socket constants 24 ⟩
⟨ Internal procedure definitions 18 ⟩
⟨ External procedure definitions 15 ⟩
⟨ Register pre-defined socket domains 34 ⟩

2. Because we are working at the system level, we need to know the operating system on which we are working. Practically speaking, when it comes to sockets, this really comes down to a question of whether we are working on Windows or now.

⟨ Foreign code utilities 2 ⟩ ≡
```
(meta define (windows?) (memq (machine-type) '(i3nt ti3nt)))
(define-syntax (if-windows? x)
  (syntax-case x ()
    [(_ c a) (if (windows?) #'c #'a)]))
```
This section exports `windows?andif-windows?`.

See also sections 4, 69, 70, 71, 72, 75, 76, 78, and87.

This code is used in section 1.

3. The following is an R6RS library that encapsulate this library for use by external R6RS programs.

⟨ *sockets.sls* 3 ⟩ ≡

```
(library (arcfide sockets)
  (export make-socket socket? socket-fd socket-domain socket-type socket-protocol
          socket-option? make-socket-option socket-option
          define-socket-option-type
          make-tcp-option make-udp-option make-raw-option make-ip-option
          tcp-option? udp-option? raw-option? ip-option?
          socket-address? socket-address
          unix-address? make-unix-address unix-address-path
          internet-address? make-internet-address internet-address-ip
          internet-address-port string->internet-address
          internet-address->string string->ipv4
          make-address-info address-info? address-info-domain
          address-info-type address-info-protocol address-info-address
          get-address-info
          address-info/canonical-name address-info/numeric-host
          address-info/passive
          create-socket make-socket-domain make-socket-type
          socket-domain/unix socket-domain/local
          socket-domain/internet socket-type/stream socket-type/datagram
          socket-type/sequence-packet socket-type/raw socket-type/random
          register-socket-domain!
          make-socket-protocol socket-protocol?
          protocol-entry-name protocol-entry-aliases protocol-entry-value
          socket-protocol/auto next-protocol-entry
          get-protocol-by-name get-protocol-by-constant
          open-protocol-database close-protocol-database
          bind-socket listen-socket accept-socket connect-socket
          close-socket shutdown-socket shutdown-method? make-shutdown-method
          shutdown-method/read shutdown-method/write shutdown-method/read&write
          send-to-socket send-to/dont-route send-to/out-of-band
          make-send-to-option
          receive-from-socket receive-from/out-of-band receive-from/peek
          receive-from/wait-all receive-from/dont-wait
          make-receive-from-option
          socket-maximum-connections
          get-socket-option set-socket-option! set-socket-nonblocking!
          socket-nonblocking?
          make-socket-condition socket-condition?
          socket-condition-who socket-condition-syscall socket-condition-type
          socket-condition-message socket-error socket-raise/unless)
  (import (chezscheme))
  (include "sockets.ss"))
```

4. Uncompleted/Planned Features. The following is a development to-do list of intended features and possible improvements.

1) Consider IP field name replaced with ADDRESS
2) Consider name change of RECEIVE-FROM-SOCKET
3) Consider name change of ACCEPT-SOCKET
4) Better handling of paths in UNIX sockets

Even though this library is mostly general, there are some exceptions where I have included some features by default that aren't really possible to access throughout the entire gambit of what machines and systems on which we expect this library to run. Most notably, Microsoft Windows does not support UNIX sockets.

To handle these situtations, the following procedure is run whenever there is a feature that is unsupported on a particular system.

⟨ Foreign code utilities 4 ⟩ +≡

```
(define (unsupported-feature feature)
  (error feature "this feature is not supported on this platform"))
```

This section exports `unsupported-feature`.

5. Datatypes. The next few sections define the various socket datatypes. Sockets are file descriptors which usually have integer representations. The `socket` datatype also defines fields for easily determining the domain, the type, and the protocol of the socket. The `fd` field should be a socket field descriptor, which is a positive integer. The `domain`, `type`, and `protocol` fields all contain constants which return true for `socket-domain?`, `socket-type?`, and `socket-protocol?`, respectively. This functionality could be introspected from the field descriptor, but it is more convenient to store this information directly.

⟨ Datatype definitions 5 ⟩ ≡

```
(define-record-type socket
  (fields fd domain type protocol (mutable nonblocking?))
  (protocol
    (lambda (n)
      (lambda (fd domain type protocol)
        (n fd domain type protocol #f)))))
```

This section exports `make-socket, socket?, socket, socket-nonblocking?, socket-nonblocking?-set!, socket-fd, socket-domain, socket-type, andsocket-protocol`.

See also sections 6, 7, 8, 9, 10, 11, 12, 13, 14, 20, 21, 25, 31, 36, 38, 40, 41, 51, 55, 61, 63, 64, 65, and66.

This code is used in section 1.

6. Socket Options. Socket options form a hierarchy of depth two that identify different settings for sockets. The `level` field should be a number identifying the highlevel setting group, and `id` should be a number identifying the setting itself, as described in `getsockopt(2)`. `valid?` is a predicate that returns true for valid setting values, and false otherwise. `make-socket-option` called without a level will default to a socket api level option, otherwise it expects a proper option level.

⟨ Datatype definitions 6 ⟩ +≡

```
(define-record-type socket-option
  (fields level id foreign-size foreign-maker foreign-converter)
  (protocol
    (lambda (p)
      (case-lambda
        [(id size maker converter)
         (p $sol-socket id size maker converter)]
        [(id size maker converter level)
         (p level id size maker converter)]))))
```

This section exports `socket-option?`, `make-socket-option`, `socket-option`, `socket-option-foreign-size`, `socket-option-foreign-maker`, `socket-option-foreign-converter`, `socket-option-id`, and`socket-option-level`.

7. To define some basic socket option levels, we have a form:

```
(define-socket-option-type <name> <level>)
```

Where `<level>` is an integer as detailed in `setsockopt(2)`. This will bind `name`, `make-name`, and `name?`.

⟨ Datatype definitions 7 ⟩ +≡

```
(define-syntax define-socket-option-type
  (syntax-rules ()
    [(_ name level)
     (define-record-type name (parent socket-option)
       (protocol
         (lambda (n)
           (lambda (id size maker converter)
             ((n id size maker converter level))))))]))
```

This section exports `define-socket-option-type`.

8. We currently define child option types for the protocols `tcp(7)`, `udp(7)`, and `raw(7)`. I have added the ip option as well because ip options are often applicable to the above.

⟨ Datatype definitions 8 ⟩ +≡

```
(define-socket-option-type tcp-option $ipproto-tcp)
(define-socket-option-type udp-option $ipproto-udp)
(define-socket-option-type raw-option $ipproto-raw)
(define-socket-option-type ip-option $ipproto-ip)
```

This section exports `make-tcp-option`, `make-udp-option`, `make-raw-option`, `make-ip-option`, `tcp-option?`, `udp-option?`, `raw-option?`, `ip-option?`, `define-socket-option-type`, `tcp-option`, `udp-option`, `raw-option`, and`ip-option`.

9. Socket Addresses. Other than sockets, one must also have a means by which to address other hosts. Socket Addresses represent the destinations or origins of transmissions. All socket address data types are subtypes of `socket-address`es.

⟨ Datatype definitions 9 ⟩ +≡
```
(define-record-type socket-address (fields converter))
```
This section exports `socket-address?, socket-address, andsocket-address-converter`.

10. When passing socket addresses to foreign procedures, we must first convert these Scheme datatypes to a proper foreign socket address structure. To do this, every child of the `socket-address` type must provide a converter. This should be defined by default, and it should not be necessary for the user to specify the converter. The converter should accept the socket address type for which it is defined as its sole argument. It should then return a bytevector that represents the foreign structure.

Internally, when we need to communicate with a foreign procedure that expects a socket address, we use the following wrapper `socket-address->foreign` and if we need to convert an unknown address type into a Scheme data structure, we use `foreign->socket-address`.

When constructing a bytevector container for foreign code, it is useful to know the size a structure should be before creating it. This is defined as `foreign-address-size`, but it is defined as part of the record definition for socket domains below."

⟨ Datatype definitions 10 ⟩ +≡
```
(define (socket-address->foreign sock-addr)
  ((socket-address-converter sock-addr) sock-addr))
(define (foreign->socket-address domain addr addr-len)
  ((socket-domain-extractor domain) addr addr-len))
```
This section exports `socket-address->foreignandforeign->socket-address`.

11. UNIX Socket Addresses. UNIX domain sockets have addresses that are just paths, which in turn are simply strings.

⟨ Datatype definitions 11 ⟩ +≡
```
(define-record-type unix-address
  (parent socket-address)
  (protocol
    (lambda (n) (lambda (path) ((n unix-address->foreign) path))))
  (fields path))
```
This section exports `unix-address?, make-unix-address, unix-address-path, andunix-address`.

12. The protocol uses `unix-address->foreign` as the converter for a unix address. It returns a bytevector that is the equivalent layout of the `sockaddr_un` structure. Converting the foreign address back to a UNIX address can be done by grabbing the vector elements of the range from the start of the path to the first null.

⟨ Datatype definitions 12 ⟩ +≡
```
(define (unix-address->foreign addr)
  (if-windows?
    (unsupported-feature 'unix-sockets)
    (values (make-foreign-unix-address (unix-address-path addr))
            size-of/sockaddr-un)))
(define (foreign->unix-address addr addr-len)
  (if-windows?
    (unsupported-feature 'unix-sockets)
    (make-unix-address (foreign-unix-address-path addr))))
```
This section exports `unix-address->foreignandforeign->unix-address`.

13. IPV4 Internet Socket Addresses. Internet addresses are represented by an IP address and a port number. The highest eight bits of the ip address should be the first octet of the ip address, and so forth. The port value is a 16-bit unsigned integer.

⟨ Datatype definitions 13 ⟩ +≡

```
(define-record-type internet-address
  (parent socket-address)
  (protocol
    (lambda (n)
      (lambda (i p) ((n internet-address->foreign) i p))))
  (fields ip port))
```

This section exports `internet-address`, `internet-address?`, `make-internet-address`, `internet-address-ip`, `andinternet-address-port`.

14. ⟨ Datatype definitions 14 ⟩ +≡

```
(define (internet-address->foreign addr)
  (values
    (make-foreign-ipv4-address
      (internet-address-port addr)
      (internet-address-ip addr))
    size-of/sockaddr-in))
(define (foreign->internet-address addr addr-len)
  (make-internet-address
    (foreign-ipv4-address-ip addr)
    (foreign-ipv4-address-port addr)))
```

This section exports `internet-address->foreignandforeign->internet-address`.

15. IP addresses often come in the form of strings. So, let's define a few procedures for handling strings as IPs. Usually internet addresses are given as a colon delimited ip address string and a port number. `string->internet-address` converts this to a proper internet address structure.

⟨ External procedure definitions 15 ⟩ ≡

```
(define (string->internet-address s)
  (let-values ([(ip-string port-string) ⟨ Split IPV4 address 16 ⟩])
    (let ([ip (and ip-string (string->ipv4 ip-string))]
          [port (and port-string (string->number port-string))])
      (assert (or (not ip) (>= 32 (bitwise-length ip))))
      (assert (or (not port) (< 0 port 65536)))
      (make-internet-address ip port))))
```

This section exports `string->internet-address`.

See also sections 17, 19, 22, 29, 33, 35, 42, 43, 44, 45, 48, 49, 50, 53, 57, and62.

This code is used in section 1.

16. To parse the string, we assume that any valid internet address string must have a single colon in it separating the two parts. If this is the case, then we can take this and use the parts individually. On the other hand, if we do not, then we consider it an invalid internet address string.

⟨ Split IPV4 address 16 ⟩ ≡
```
(let ([val (split-string s #\:)])
  (if (pair? val)
      (values
        (car val)
        (and (pair? (cdr val)) (cadr val)))
      (values #f #f)))
```
This section captures `s`.

This code is used in section 15.

17. Another helper splits the ip address and converts it to a bytevector in big endian or network byte order.

⟨ External procedure definitions 17 ⟩ +≡
```
(define (string->ipv4 s)
  (let ([bytes (map string->number (split-string s #\.))])
    (assert (= 4 (length bytes)))
    (fold-left
      (lambda (s x)
        (assert (<= 0 x 255))
        (+ (bitwise-arithmetic-shift s 8) x))
      0
      bytes)))
```
This section exports `string->ipv4`.

18. Both of the above utilities rely on a string splitting function. We'll define that here.

⟨ Internal procedure definitions 18 ⟩ ≡
```
(define (split-string s c)
  (define (debuf buf) (list->string (reverse buf)))
  (define (split lst res buf)
    (cond
      [(null? lst) (reverse (cons (debuf buf) res))]
      [(char=? c (car lst)) (split (cdr lst) (cons (debuf buf) res) '())]
      [else (split (cdr lst) res (cons (car lst) buf))]))
  (if (fxzero? (string-length s))
      '()
      (split (string->list s) '() '())))
```
This section exports `split-string`.

See also sections 67and68.

This code is used in section 1.

19. The reverse procedure `internet-address->string` is more straightforward.

⟨ External procedure definitions 19 ⟩ +≡

```
(define (internet-address->string addr)
  (let ([ip (or (internet-address-ip addr) 0)]
        [port (internet-address-port addr)])
    (assert (or (not port) (< 0 port 65536)))
    (assert (>= 32 (bitwise-length ip)))
    (do ([ip ip (bitwise-arithmetic-shift ip -8)]
         [i 0 (+ i 1)]
         [res '() (cons (mod ip 256) res)])
      [(= 4 i)
       (fold-right
         (lambda (x s)
           (cond
             [(string? s)
              (string-append (number->string x) "." s)]
             [(number? s)
              (string-append (number->string x) ":" (number->string s))]
             [else (number->string x)]))
         port
         res)])))
```

This section exports `internet-address->string`.

20. Socket Constants. Procedures such as `create-socket` accept records which wrap numeric constant values for passing into the FFI. These constants are limited with what is built in to have the widest acceptance and portability, but if the user wishes to use more values, he can do so by using the appropriate make form. This requires that the user know the Operating specific constant value that should be used in the FFI.

Every constant type is a child of the `socket-constant` type.

⟨ Datatype definitions 20 ⟩ +≡

```
(define-record-type socket-constant (fields (immutable value)))
```

This section exports `socket-constant, make-socket-constant, socket-constant?, andsocket-constant-value`.

21. Address Information. The `get-address-info` procedure returns a list of `address-info` structures that specify different means by which a host may be contacted and a string representing the canonical name of the host if this information was requested. Otherwise, the second returned value is false. It corresponds to the `getaddrinfo(3)` UNIX system call, with a few things modified to be more Schemely.

```
(get-address-info node service [domain type protocol])
  => addresses canonical-name
```

Each `address-info` structure identifies or associates an address with a given domain, type, and protocol, which is enough to create a socket and connect to that address using the right types.

⟨ Datatype definitions 21 ⟩ +≡

```
(define-record-type address-info (fields domain type protocol address))
```

This section exports `make-address-info, address-info?, address-info, address-info-domain, address-info-type, address-info-protocol, andaddress-info-address`.

22. The `address` field of an `address-info` record should be an `internet-socket-address`. Also note that in the normal C equivalent of this record type, `struct addrinfo`, there is also a field for the canonical name. Because this is given only once, it does not make sense to have a field for this in every structure that is returned, so I have decided to place this as an additional value that is returned by `get-address-info` instead, which makes more sense, since it really is a separate thing to be returned.

The downside to this approach is that it requires two values to be accepted when calling `get-address-info`, even if one does not care about the canonical name. I am open to better approaches, but this does not seem to be too inconvenient in practice.

`get-address-info` takes an optional set of hints, such as the domain, type, protocol, or a set of flags that can be used to filter out the results obtained from the call.

To grab the address we make our foreign buffers, call `$getaddrinfo`, check for errors, and then convert the foreign address foreign-integer to a Scheme structure.

⟨ External procedure definitions 22 ⟩ +≡

```
(define get-address-info
  (case-lambda
    [(node service) (%get-address-info node service #f #f #f '())]
    [(node service dom type proto . flags)
     (%get-address-info node service dom type proto flags)]))
(define (%get-address-info node service domain type protocol flags)
  ⟨ Check get-address-info argument types 23 ⟩
  (let ([alp (make-foreign-pointer)]
        [hints ⟨ Build address information hints 26 ⟩]
        [service (or (and (string? service) service)
                     (number->string service 10))])
    (let ([res ($getaddrinfo node service hints alp)])
      (if (zero? res)
          (values ⟨ Convert foreign address information list 27 ⟩
                  (foreign-address-info-canonical-name
                   (foreign-pointer-value alp)))
          (error 'get-address-info
            "getaddrinfo() failed with code"
            `(code ,res)
            ($gai_strerror res))))))
```

This section exports `get-address-info`.

23. The optional domain, type, and protocol may be false or may correspond to some form of socket options. They are used as hints for `get-address-info` in the same manner as the corresponding hints structure for `getaddrinfo(2)`. The flags also work in the same way, but they must all be values for which `address-info-option?` returns true when applied to them. `node` should be a hostname string, and `service` should be either a service name string, integer number string identifying a valid port, or a positive integer representing a valid port.

⟨ Check get-address-info argument types 23 ⟩ ≡
```
(assert (or (not domain) (socket-domain? domain)))
(assert (or (not type) (socket-type? type)))
(assert (or (not protocol) (socket-protocol? protocol)))
(assert (for-all address-info-option? flags))
(assert (string? node))
(assert
  (or
    (string? service)
    (and
      (integer? service)
      (positive? service)
      (< 0 service 65536))))
```
This section captures `node`, `service`, `domain`, `type`, `protocol`, `andflags`.

This code is used in section 22.

24. There are a few built in address info options.

⟨ Socket constants 24 ⟩ ≡
```
(define address-info/canonical-name
  (make-address-info-option %ai/canonname))
(define address-info/numeric-host
  (make-address-info-option %ai/numerichost))
(define address-info/passive
  (make-address-info-option %ai/passive))
```
This section exports `address-info/canonical-name`, `address-info/numeric-host`, `andaddress-info/passive`.

See also sections 32, 37, 39, 52, 56, and60.

This code is used in section 1.

25. Each of the above constants is an `address-info-option`. This is a datatype to encapsulate the standard socket constants.

⟨ Datatype definitions 25 ⟩ +≡
```
(define-record-type address-info-option (parent socket-constant))
```
This section exports `make-address-info-option`, `address-info-option`, `andaddress-info-option?`.

26. We need to convert the hints given to use in Scheme terms and convert them to foreign hints, which is its own structure.

⟨ Build address information hints 26 ⟩ ≡
```
(make-foreign-address-info
  (fold-left
    (lambda (s v) (fxior s (socket-constant-value v)))
    0 flags)
  (or (and domain (socket-constant-value domain)) 0)
  (or (and type (socket-constant-value type)) 0)
  (or (and protocol (socket-constant-value protocol)) 0)
  0 0 0 0)
```

This section captures `domain, type, protocol, andflags`.

This code is used in section 22.

27. The foreign `struct addrinfo` is a linked list of information records. To convert these to real lists of `address-info` records, I use the foreign accessors and loop over the linked list.

⟨ Convert foreign address information list 27 ⟩ ≡
```
(define (get-address-info-entry alp)
  (let ([dom (lookup-domain (foreign-address-info-domain alp))])
    (if dom
        (make-address-info
          dom
          (make-socket-type (foreign-address-info-type alp))
          (make-socket-protocol (foreign-address-info-protocol alp))
          (foreign->socket-address
            dom
            (foreign-address-info-address alp)
            (foreign-address-info-address-length alp)))
        #f)))
(do ([ptr (foreign-pointer-value alp) (foreign-address-info-next ptr)]
     [res '()
       (let ([entry (get-address-info-entry ptr)])
         (if entry (cons entry res) res))])
  [(zero? ptr) (reverse res)])
```

This section captures `alp`.

This code is used in section 22.

28. Socket Procedures. There are a number of socket programming functions that we bind and wrap here. The following table matches the foreign system call to the Scheme procedure defined in this library.

Scheme Procedures and System Call Equivalents

`create-socket`	`socket(2)`
`next-protocol-entry`	`getprotoent(2)`
`get-protocol-by-name`	`getprotobyname(2)`
`get-protocol-by-constant`	`getprotobynumber(2)`
`open-protocol-database`	`setprotoent(2)`
`close-protocol-database`	`endprotoent(2)`
`bind-socket`	`bind(2)`
`listen-socket`	`listen(2)`
`accept-socket`	`accept(2)`
`connect-socket`	`connect(2)`
`close-socket`	`close(2)`
`shutdown-socket`	`shutdown(2)`
`send-to-socket`	`sendto(2)`
`receive-from-socket`	`recvfrom(2)`
`socket-maximum-connections`	`SOMAXCONN`
`get-socket-option`	`getsockopt(2)`
`set-socket-option!`	`setsockopt(2)`

29. Creating Sockets. Creating sockets is achieved through the `create-socket` procedure. The datatypes for its arguments are described further down.

```
(create-socket domain type protocol) => socket
```

After calling the foreign `socket(2)` call, we need to error out if its a bad socket, but otherwise, we need to build the appropriate struture and set any options. In this case, we default to nonblocking sockets, while most BSD sockets systems start in blocking mode.

⟨ External procedure definitions 29 ⟩ +≡

```
(define (create-socket domain type protocol)
  ⟨ Check create-socket arguments 30 ⟩
  (call-with-errno
   (lambda ()
     ($socket (socket-constant-value domain)
              (socket-constant-value type)
              (socket-constant-value protocol)))
   (lambda (ret err)
     (if (= ret invalid-socket)
         (socket-error 'create-socket 'socket err)
         (let ([sock (make-socket ret domain type protocol)])
           (set-socket-nonblocking! sock #t)
           sock)))))
```

This section exports `create-socket`.

30. I do very simple argument checking of the `create-socket` arguments.

⟨ Check `create-socket` arguments 30 ⟩ ≡

```
(define who 'create-socket)
(unless (socket-domain? domain)
  (error who "invalid socket domain" domain))
(unless (socket-type? type)
  (error who "invalid socket type" type))
(unless (socket-protocol? protocol)
  (error who "invalid socket protocol" protocol))
```

This section captures `domain, type, andprotocol`.

This code is used in section 29.

31. `create-socket` uses three different constant types for the domain, type, and protocol of the socket. Socket domains determine the family to which the socket belongs. They also must embed an extracter and a size value so that converting to and from foreign values can be done without explicitly knowing the type of the domain beforehand.

⟨ Datatype definitions 31 ⟩ +≡

```
(define-record-type (%socket-domain make-socket-domain socket-domain?)
  (parent socket-constant)
  (fields
    (immutable extractor socket-domain-extractor)
    (immutable addr-size foreign-address-size)))
```

This section exports `%socket-domain, make-socket-domain, socket-domain?, socket-domain-extractor, andforeign-address-size`.

32. We predefine UNIX/Local and Internet IPV4 domain types.

⟨ Socket constants 32 ⟩ +≡
```
(define socket-domain/unix
  (make-socket-domain
   %socket-domain/unix
   foreign->unix-address
   size-of/addr-un))
(define socket-domain/local
  (make-socket-domain
   %socket-domain/local
   foreign->unix-address
   size-of/addr-un))
(define socket-domain/internet
  (make-socket-domain
   %socket-domain/internet
   foreign->internet-address
   size-of/addr-in))
```
This section exports `socket-domain/unix, socket-domain/local, andsocket-domain/internet`.

33. Socket domains sometimes need to be grabbed by just their internal value. We set up a database to hold the registered domains and allow for additional domains to be registered.

⟨ External procedure definitions 33 ⟩ +≡
```
(define socket-domain-db
  (make-parameter '()))
(define (register-socket-domain! domain)
  (assert (socket-domain? domain))
  (let* ([val (socket-constant-value domain)]
         [res (assv val (socket-domain-db))])
    (if res
        (set-cdr! res domain)
        (socket-domain-db
          (cons (cons val domain)
                (socket-domain-db))))))
```
This section exports `register-socket-domain!andsocket-domain-db`.

34. We register only the two necessary ones right now.

⟨ Register pre-defined socket domains 34 ⟩ ≡
```
(register-socket-domain! socket-domain/unix)
(register-socket-domain! socket-domain/internet)
```
This code is used in section 1.

35. We'll want to be able to look these domains up by number.

⟨ External procedure definitions 35 ⟩ +≡
```
(define (lookup-domain val)
  (let ([res (assv val (socket-domain-db))])
    (and res (cdr res))))
```
This section exports `lookup-domain`.

36. Socket types determine the nature of the data stream that transmits over the socket. See the `socket(2)` man page for more details.

⟨ Datatype definitions 36 ⟩ +≡
```
(define-record-type (%socket-type make-socket-type socket-type?)
  (parent socket-constant))
```
This section exports `%socket-type, make-socket-type, andsocket-type?`.

37. ⟨ Socket constants 37 ⟩ +≡
```
(define socket-type/stream
  (make-socket-type %socket-type/stream))
(define socket-type/datagram
  (make-socket-type %socket-type/datagram))
(define socket-type/sequence-packet
  (make-socket-type %socket-type/sequence-packet))
(define socket-type/raw
  (make-socket-type %socket-type/raw))
(define socket-type/random
  (make-socket-type %socket-type/random))
```
This section exports `socket-type/stream, socket-type/datagram, socket-type/sequence-packet, socket-type/raw, andsocket-type/random`.

38. Dealing with protocol numbers is slightly different, since these entries are found in a database that can change, rather than in some header file. The datatype declaration is still the same, though. Generally, it is fine to use an automatically chosen protocol number, so the user will not usually need to use the more complicated database searching tools in the next sections. Instead, we define a default protocol here for automatic protocol selection.

⟨ Datatype definitions 38 ⟩ +≡
```
(define-record-type
  (%socket-protocol make-socket-protocol socket-protocol?)
  (parent socket-constant))
```
This section exports `make-socket-protocolandsocket-protocol?`.

39. ⟨ Socket constants 39 ⟩ +≡
```
(define socket-protocol/auto (make-socket-protocol 0))
```
This section exports `socket-protocol/auto`.

40. Protocols can be retreived by the `getproto*` family of functions. These functions return `protocol-entry` structures.

For protocol entries, we expect the value to be a protocol constant. Each of the general protocol retreival functions that utilize `foreign->protocol-entry` will return false when they are at the end of the protocols database or if there was an error.

The `protocol-entry` structure encapsulates the important elements of each record in the protocol database.

⟨ Datatype definitions 40 ⟩ +≡
```
(define-record-type protocol-entry (fields name aliases value))
```
This section exports `make-protocol-entry, protocol-entry?, protocol-entry, protocol-entry-name, protocol-entry-aliases, andprotocol-entry-value`.

41. We use a simple convert that allows us to take a foreign entry and turn it into a `protocol-entry`. This is used in all of the protocol accessor functions that need to return some protocol.

⟨ Datatype definitions 41 ⟩ +≡
```
(define (foreign->protocol-entry x)
  (make-protocol-entry
    (foreign-protocol-entry-name x)
    (foreign-protocol-entry-aliases x)
    (foreign-protocol-entry-protocol x)))
```
This section exports `foreign->protocol-entry`.

42. We follow the standard procedure layout for navigating through the protocol database. There is an iterator that allows us to traverse through the protocol database, as well as procedures for closing and opening the database. These are rarely used, though, and most of the time a protocol will be retrieved by name or by constant, and there are two procedures that specifically enable this in the library.

⟨ External procedure definitions 42 ⟩ +≡
```
(define (next-protocol-entry)
  (if-windows?
    (unsupported-feature 'next-protocol-entry)
    (foreign->protocol-entry ($getprotoent))))
(define (get-protocol-by-name name)
  (foreign->protocol-entry ($getprotobyname name)))
(define (get-protocol-by-constant proto)
  (foreign->protocol-entry
    ($getprotobynumber (socket-constant-value proto))))
(define (open-protocol-database keep-alive?)
  (if-windows?
    (unsupported-feature 'open-protocol-database)
    ($setprotoent keep-alive?)))
(define (close-protocol-database)
  (if-windows?
    (unsupported-feature 'close-protocol-database)
    ($endprotoent)))
```
This section exports `next-protocol-entry`, `get-protocol-by-name`, `get-protocol-by-constant`, `open-protocol-database`, and`close-protocol-database`.

43. Binding and listening to sockets. Binding sockets works with a fairly direct mapping from the traditional BSD sockets interface, so there isn't much to say here. You bind a given socket to a given address. 'Nuff said.

```
(bind-socket socket address)
```

Unless there has been some tragic error, the return value of this function is unspecified.

⟨ External procedure definitions 43 ⟩ +≡

```
(define (bind-socket sock addr)
  (let-values ([(foreign-addr foreign-size)
                (socket-address->foreign addr)])
    (call-with-errno
      (lambda () ($bind (socket-fd sock) foreign-addr foreign-size))
      (lambda (ret err)
        (foreign-free foreign-addr)
        (when (= ret $socket-error)
          (socket-error 'bind-socket 'bind err))))))
```

This section exports `bind-socket`.

44. Listening to sockets corresponds directly to the `listen(2)` system call. It's behavior is the same. The queue length should be a positive integer not greater than the maximum number of allowed connections.

```
(listen-socket socket queue-length)
```

`listen-socket` does not return a value.

⟨ External procedure definitions 44 ⟩ +≡

```
(define (listen-socket sock queue-length)
  (call-with-errno (lambda () ($listen (socket-fd sock) queue-length))
    (lambda (ret err)
      (when (= ret $socket-error)
        (socket-error 'listen-socket 'listen err)))))
```

This section exports `listen-socket`.

45. Accepting Connections to Sockets. Because of the interesting interface of `accept(2)` we can't directly map the interface to Scheme without making a lot of people twitch. Instead, I take advantage of multiple return values and have `accept-socket` return two values. The first value is a socket suitable for talking with the connecting client. The second value returned is the connecting client's address record.

```
(accept-socket socket) => socket address
```

Accept also behaves slightly differently depending on whether the listening socket is blocking or non-blocking. For a blocking socket, this function will block operation until it receives some connection. In this case, the only thing you should receive in the normal case is a socket in the first return value, and a proper address record in the second. If, however, the listening socket is set to non-blocking, then accept will return immediately even if there is no existing connection. If `accept-socket` returns without having a connection to hand over, the first return value will be false, and the second will be a condition record indicating the type of error that was returned, such as the EAGAIN or EWOULDBLOCK error conditions. This will give you some more information about how to proceed, but not much. It will not return a condition if the condition would be a true error condition. In this case, it will raise the error and not return. Because Chez Scheme will block the GC whenever a foreign function is running, we have to do some special work to disable the foreign thread before running a block IO operation like `accept(2)`. However, we don't need to do that overhead if we are dealing with purely nonblocking sockets. So, before going to the foreign side, check and call the appropriate function accordingly.

⟨ External procedure definitions 45 ⟩ +≡

```
(define (accept-socket sock)
  (let ([size (foreign-address-size (socket-domain sock))])
    (let ([addr (foreign-alloc size)]
          [addr-len (make-foreign-size-buffer size)])
      (call-with-errno
        (lambda ()
          ((if (socket-nonblocking? sock) $accept $accept-blocking)
           (socket-fd sock) addr addr-len))
        (lambda (ret err)
          (if (= ret invalid-socket)
              ⟨ Return intelligently from non-blocking errors 46 ⟩
              ⟨ Build socket and address, then return 47 ⟩)))))))
```

This section exports `accept-socket`.

46. The `accept(2)` system call returns an error state even when the error is something we intended, such as the case with the EAGAIN and EWOULDBLOCK errors for non-blocking sockets. This doesn't make sense on a Scheme interface, so instead, we'll catch the situations where the errors are mundane and return these through the normal return channels, and only raise a real error for real errors.

⟨ Return intelligently from non-blocking errors 46 ⟩ ≡

```
(values
  #f
  (socket-raise/unless 'accept-socket 'accept err
                       $error-again $error-would-block))
```

This section captures `err`.

This code is used in section 45.

47. In the normal cases, we just need to translate the socket and extract out the address information.

⟨ Build socket and address, then return 47 ⟩ ≡

```
(values
  (make-socket ret
               (socket-domain sock)
               (socket-type sock)
               (socket-protocol sock))
  (let ([res (foreign->socket-address
              (socket-domain sock) addr addr-len)])
    (foreign-free addr)
    (foreign-free addr-len)
    res))
```

This section captures `sock, addr, addr-len, andret`.

This code is used in section 45.

48. Connecting to the world. We use `connect-socket` to connect a socket to an endpoint indicated by the given address.

```
(connect-socket socket address) => #t or condition
```

Normally a connection succeeds or fails. When it succeeds `connect-socket` returns true. When it fails, it will raise an error. However, if that error happens to be an in-progress message, then we don't raise an error and just return the condition to you. Since `connect-socket` is also a potentially blocking operation like `accpet-socket` we use the same technique to choose whether or not to call the special blocking optimized version of `connect(2)` which disables the foreign thread before calling `connect(2)`. This function corresponds to the `connect(2)` system call.

⟨ External procedure definitions 48 ⟩ +≡

```
(define (connect-socket sock addr)
  (let-values ([(fa fa-len) (socket-address->foreign addr)])
    (call-with-errno
      (lambda ()
        ((if (socket-nonblocking? sock)
             $connect
             $connect-blocking)
         (socket-fd sock) fa fa-len))
      (lambda (ret err)
        (foreign-free fa)
        (or (not (= ret $socket-error))
            (socket-raise/unless 'connect-socket
                                 'connect
                                 err
                                 $error-in-progress
                                 $error-would-block))))))
```

This section exports `connect-socket`.

49. Closing and shutting down sockets. "The normal `close(2)` system call works fine for closing down sockets, which are just file descriptors. We have a very light wrapping around this system call.

```
(close-socket socket)
```

`close-socket` does not return a value.

⟨ External procedure definitions 49 ⟩ +≡

```
(define (close-socket sock)
  (call-with-errno (lambda () ($close (socket-fd sock)))
    (lambda (ret err)
      (when (= ret $socket-error)
        (socket-error 'close-socket 'close err)))))
```

This section exports `close-socket`.

50. Sometimes you need more control for handling sockets, and the `shutdown-socket` procedure handles that. It allows you to specify how to shutdown a socket, so that you can selectively disable certain capabilities while leaving others around.

```
(shutdown-socket socket method)
```

The methods are constants defined below. Since they correspond closely to the existing `shutdown(2)` system call, the procedure isn't very complicated.

⟨ External procedure definitions 50 ⟩ +≡

```
(define (shutdown-socket sock how)
  (assert (shutdown-method? how))
  (call-with-errno
    (lambda ()
      ($shutdown (socket-fd sock) (socket-constant-value how)))
    (lambda (ret err)
      (when (= ret $socket-error)
        (socket-error 'shutdown-socket 'shutdown err)))))
```

This section exports `shutdown-socket`.

51. By default, we define the basic `shutdown-method` constant class, as well as shutdown methods for reading, writing, and reading/writing combined.

⟨ Datatype definitions 51 ⟩ +≡

```
(define-record-type shutdown-method (parent socket-constant))
```

This section exports `shutdown-method, make-shutdown-method, andshutdown-method?`.

52. ⟨ Socket constants 52 ⟩ +≡

```
(define shutdown-method/read
  (make-shutdown-method %shutdown/read))
(define shutdown-method/write
  (make-shutdown-method %shutdown/write))
(define shutdown-method/read&write
  (make-shutdown-method %shutdown/read&write))
```

This section exports `shutdown-method/read, shutdown-method/write, andshutdown-method/read&write`.

53. Sending data to sockets. While there are other, more convenient mechanisms for handling input and output, such as ports, you want to have a basic means of sending data natively to sockets.

```
(send-to-socket socket buffer address [flag ...]) => bytes sent
```

The buffer should be a bytevector containing the data that you want to send. The address and sockets are the standard data structures defined in this library. Any flags that you pass in should be `send-to-option` flags (they answer true to the predicate `send-to-option?`).

⟨ External procedure definitions 53 ⟩ +≡

```
(define (send-to-socket sock buf addr . flags)
  (assert (for-all send-to-option? flags))
  (let-values ([(fa fa-len) (socket-address->foreign addr)])
    (call-with-errno
      (lambda ()
        ⟨ Convert datatypes and jump to the right foreign function 54 ⟩)
      (lambda (res err)
        (foreign-free fa)
        (if (= res $socket-error)
            (socket-raise/unless 'send-to-socket 'sendto err
              $error-again $error-would-block)
            res)))))
```

This section exports `send-to-socket`.

54. Since `sendto(2)` could also block, we have to follow the same technique as with `accept-socket` to choose whether to go with a blocking optimized version or the straight `sendto(2)` call. Otherwise the basic mapping of the datatypes is fairly standard.

⟨ Convert datatypes and jump to the right foreign function 54 ⟩ ≡

```
((if (socket-nonblocking? sock) $sendto $sendto-blocking)
  (socket-fd sock) buf (bytevector-length buf)
  (fold-left (lambda (s v) (fxior s (socket-constant-value v)))
             0 flags)
  fa fa-len)
```

This section captures `sock, buf, flags, fa, andfa-len`.

This code is used in section 53.

55. There are a number of system depended flags that can be passed to `send-to-socket`, but only those reasonably portable ones are defined here.

⟨ Datatype definitions 55 ⟩ +≡

```
(define-record-type send-to-option (parent socket-constant))
```

This section exports `send-to-option, make-send-to-option, andsend-to-option?`.

56. ⟨ Socket constants 56 ⟩ +≡

```
(define send-to/dont-route
  (make-send-to-option %msg/dont-route))
(define send-to/out-of-band
  (make-send-to-option %msg/out-of-band))
```

This section exports `send-to/dont-routeandsend-to/out-of-band`.

57. Receiving over Sockets. Like sending, we want to be able to receive over sockets directly. We define `receive-from-socket` to correspond roughly to the `recvfrom(2)` system call.

```
(receive-from-socket socket count [flag ...]) => data address
```

`receive-from-socket` takes a socket, the number of bytes to receive, and a possibly zero set of flags. Each flag should be a `receive-from-option` flag. We define a minimal set of these flags below. The procedure returns a bytevector with the data and the address of the sender. Note that you cannot assume that the bytevector is the length of `count` because it is possible that fewer than `count` bytes may have been received.

If the socket passed to `receive-from-socket` is non-blocking, and there is no input to be read at the moment, then the data value returned will be false, and the `address` value will actually be a condition indicating the type of return error (EAGAIN or EWOULDBLOCK) that was returned from the `recvfrom(2)` system call. In the case of a real error, then the condition will not be returned in this manner, but raised and signalled.

Implementation Note: Right now the function defined below uses multiple bytevector copies; it would be much better to reduce the number of copies that are performed. I have done no formal performance testing of the effects of these copies, however.

⟨ External procedure definitions 57 ⟩ +≡

```
(define (receive-from-socket sock c . flags)
  (assert (for-all receive-from-option? flags))
  (let ([buf (make-bytevector c)]
        [addr-len (foreign-address-size (socket-domain sock))])
    (let ([addr (foreign-alloc addr-len)]
          [addr-len-buf (make-foreign-size-buffer addr-len)])
      (call-with-errno
        (lambda ()
          ⟨ Call $recvfrom or $recvfrom-blocking 58 ⟩)
        (lambda (n err)
          ⟨ Convert recvfrom returns to scheme versions 59 ⟩))))))
```

This section exports `receive-from-socket`.

58. The `receive-from-socket` procedure could potentially block, so we use the same technique as in `accept-socket`, and branch based on the blocking flag of the socket to either the straight `recvfrom(2)` system call or the specially wrapped blocking version.

⟨ Call $recvfrom or $recvfrom-blocking 58 ⟩ ≡

```
((if (socket-nonblocking? sock) $recvfrom $recvfrom-blocking)
  (socket-fd sock) buf c
  (fold-left (lambda (s v) (fxior s (socket-constant-value v)))
             0 flags)
  addr addr-len-buf)
```

This section captures `sock, buf, addr, addr-len-buf, flags, andc`.

This code is used in section 57.

59. To handle the scheme conversions, we need to make sure that we have the three different return behaviors. Firstly, we have the normal version when there is no error, where we need to create a bytevector of the right size and fill it in with just the correct amount of data. We also pass the address back with this. If on the other hand we have a blocking socket, we don't want to signal an error for cases when it is just a normal `EWOULDBLOCK` sort of return, so we handle that separately, but we do raise the error if it is a true error, and not something like `EAGAIN` or `EWOULDBLOCK`.

⟨ Convert recvfrom returns to scheme versions 59 ⟩ ≡

```
(if (= n $socket-error)
    (values
      #f
      (socket-raise/unless 'receive-from-socket 'recvfrom err
        $error-again $error-would-block))
    (values
      (if (< n c)
          (let ([res (make-bytevector n)])
            (bytevector-copy! buf 0 res 0 n)
            res)
          buf)
      (let ([res (foreign->socket-address
                   (socket-domain sock)
                   addr addr-len-buf)])
        (foreign-free addr)
        (foreign-free addr-len-buf)
        res)))
```

This section captures `n, err, c, buf, sock, addr, andaddr-len-buf`.

This code is used in section 57.

60. By default the following `receive-from-option` flags are defined.

⟨ Socket constants 60 ⟩ +≡

```
(define receive-from/out-of-band
  (make-receive-from-option %msg/out-of-band))
(define receive-from/peek
  (make-receive-from-option %msg/peek))
(define receive-from/wait-all
  (make-receive-from-option %msg/wait-all))
(define receive-from/dont-wait
  (make-receive-from-option %msg/dont-wait))
```

This section exports `receive-from/out-of-band, receive-from/peek, receive-from/wait-all, andreceive-from/dont-wait`.

61. ⟨ Datatype definitions 61 ⟩ +≡

```
(define-record-type receive-from-option (parent socket-constant))
```

This section exports `receive-from-option, make-receive-from-option, andreceive-from-option?`.

62. Maximum number of connects. There is a static header constant `SOMAXCONN` that indicates the maximum number of connections a socket can have going at a single moment. We do a simple wrapping around this and expose it to the user.

```
(socket-maximum-connections) => integer
```

The integer returned is the value of the `SOMAXCONN` constant.

⟨ External procedure definitions 62 ⟩ +≡

```
(define (socket-maximum-connections)
  %somaxconn)
```

This section exports `socket-maximum-connections`.

63. Handling socket options. Socket options are defined above. In this section I define the actual getter and setter that work on sockets. We also define some special functions to deal specifically with nonblocking and blocking sockets. The `get-socket-option` procedure takes in a socket and an option and returns the value of that option for that socket.

```
(get-socket-option socket socket-option) => value
```

The socket option should be a proper `socket-option` object. This function correlates to the `getsockopt(2)` system call.

⟨ Datatype definitions 63 ⟩ +≡

```
(define (get-socket-option sock opt)
  (let ([len (socket-option-foreign-size opt)])
    (let ([fbuf (foreign-alloc len)]
          [flen (make-foreign-size-buffer len)])
      (call-with-errno
        (lambda ()
          ($getsockopt (socket-fd sock)
            (socket-option-level opt)
            (socket-option-id opt)
            fbuf flen))
        (lambda (ret err)
          (if (= ret $socket-error)
              (begin (foreign-free fbuf) (foreign-free flen)
                (socket-error 'get-socket-option 'getsockopt err))
              (let ([res ((socket-option-foreign-converter opt)
                          fbuf
                          (foreign-size-buffer-value flen))])
                (foreign-free fbuf)
                (foreign-free flen)
                res)))))))
```

This section exports `get-socket-option`.

64. The `set-socket-option!` procedure takes a socket, socket option, and a value, and sets that socket option to that value for that socket. It has a slightly different interface from `setsockopt(2)` but it works the same way and uses it internally.

```
(set-socket-option socket socket-option value)
```

The `set-socket-option!` procedure does not return a value.

⟨ Datatype definitions 64 ⟩ +≡

```
(define (set-socket-option! sock opt val)
  (let-values ([(buf buf-len) ((socket-option-foreign-maker opt) val)])
    (call-with-errno
      (lambda ()
        ($setsockopt
          (socket-fd sock)
          (socket-option-level opt)
          (socket-option-id opt)
          buf buf-len))
      (lambda (ret err)
        (foreign-free buf)
        (when (= ret $socket-error)
          (socket-error 'set-socket-option! 'setsockopt err))))))
```

This section exports `set-socket-option!`.

65. Blocking Sockets. The following options allow you to determine if a socket is blocking or nonblocking and set the nonblocking state of the socket. Use of these options is not recommended or encouraged. They exist here for implementing some low-level behavior for higher level abstractions.

⟨ Datatype definitions 65 ⟩ +≡

```
(define (set-socket-nonblocking! sock val)
  (call-with-errno
    (lambda ()
      (%set-blocking (socket-fd sock) (not val)))
    (lambda (ret err)
      (when (= ret $socket-error)
        (socket-error 'set-socket-nonblocking! 'fcntl err))
      (socket-nonblocking?-set! sock val))))
```

This section exports `set-socket-nonblocking!`.

66. Handling Socket Errors. Most of the underlying system calls in this library signal an error with some value and then set `errno`. Most of these errors should be raised, but on system calls that could block, but don't because a socket has been set to non-blocking, we should not do so. Generally, I leave how to raise or return the error up to the individual functions. However, all of them need a nice way to obtain a given Scheme condition rather than some obscure error number. I've normalized all of the error reporting from this library into a single condition type. It's meant to enable you to see who signalled the condition, what system call they were trying to work with, the type of error, and any system messages that could be derived.

⟨ Datatype definitions 66 ⟩ +≡

```
(define-condition-type &socket &condition make-socket-condition socket-condition?
  (who socket-condition-who)
  (syscall socket-condition-syscall)
  (type socket-condition-type)
  (msg socket-condition-message))
```

This section exports `&socket`, `make-socket-condition`, `socket-condition?`, `socket-condition-who`, `socket-condition-syscall`, `socket-condition-type`, and`socket-condition-message`.

67. For functions that want to raise an error regardless of the return type, they can use `socket-error`.

⟨ Internal procedure definitions 67 ⟩ +≡

```
(define (socket-error who call errval)
  (raise (make-socket-condition who call errval (errno-message errval))))
```

This section exports `socket-error`.

68. Some functions, however, may want to return the conditions instead of raising the error. The following provides this functionality. It allows you to selectively return some errors as conditions, and others as raised errors. This is useful for the non-blocking socket interface, where `EAGAIN` and `EWOULDBLOCK` errors should return without raising an error condition.

⟨ Internal procedure definitions 68 ⟩ +≡

```
(define (socket-raise/unless who call errval . vals)
  (let ([cnd (make-socket-condition who call errval (errno-message errval))])
    (if (memv (socket-condition-type cnd) vals) cnd (raise cnd))))
```

This section exports `socket-raise/unless`.

69. Low-level Interactions. Our discussion of sockets thus far has precluded the details of low-level access to the system libraries. Now we need to consider these system functions. The first and most obvious reason for accessing the system libraries is the system constants that are stored there. We base a number of elements on constants, and we need these constants to do our work. We use a form called `define-foreign-values` to help extract these elements from the system. Additionally, there are a few procedures that are necessary at runtime, which help us to support blocking I/O calls on Threaded systems. We separate out the foreign values that we can determine at compile time from those that we need at runtime into two libraries. The compile time values are stored in a file called "socket-ffi-values" and the runtime "stub" is stored in a file called "sockets-stub". The source for these files is generated as a side effect of tangling this document. However, the names of these files will be differrent depending on what platform compiled them. Moreover, while we have to ensure that we load the compile time files at compile time, we have to ensure that we do not need to do this at runtime. We need to compile things based on the machine type we receive, so it is helpful to have a nice way of asking about that. We also want an easy way to stay in definition mode.

⟨ Foreign code utilities 69 ⟩ +≡
```
(define-syntax (on-machine x)
  (syntax-case x (else)
    [(_) #'(begin)]
    [(_ (else exp)) #'(begin (define-syntax (t x) exp) (t))]
    [(_ ((type ...) exp) rest ...)
     (if (memq (machine-type) (syntax->datum #'(type ...)))
         #'(begin (define-syntax (t x) exp) (t))
         #'(on-machine rest ...))]))
(define-syntax fake-define
  (syntax-rules ()
    [(_ exp ...) (define dummy (begin exp ... (void)))]))
```
This section exports .

70. Firstly, we need to first load the normal standard library, which is slightly different on different platforms.

⟨ Foreign code utilities 70 ⟩ +≡
```
(on-machine
  [(i3nt ti3nt a6nt ta6nt) #'(fake-define (load-shared-object "crtdll.dll"))]
  [(i3le ti3le a6le ta6le) #'(fake-define (load-shared-object "libc.so.6"))]
  [(i3osx ti3osx a6osx ta6osx)
   #'(fake-define (load-shared-object "libc.dylib"))]
  [else #'(fake-define (load-shared-object "libc.so"))])
```
This section exports .

71. Next, we need to load the FFI values that are used at compile time. We have do use a different extension based on the machine type that we have.

⟨ Foreign code utilities 71 ⟩ +≡
```
(on-machine
  [(i3nt ti3nt a6nt ta6nt)
   (begin (load-shared-object "socket-ffi-values.dll") #'(fake-define))]
  [(i3osx ti3osx ta6osx a6osx)
   (begin (load-shared-object "socket-ffi-values.dylib") #'(fake-define))]
  [else (begin (load-shared-object "socket-ffi-values.so") #'(fake-define))])
```
This section exports .

72. On the threaded versions, we must load a stub file that allows us to deal with the blocking FFI calls appropriately.

⟨ Foreign code utilities 72 ⟩ +≡

```
(on-machine
  [(ti3nt ta6nt) #'(fake-define (load-shared-object "sockets-stub.dll"))]
  [(ti3osx ta6osx) #'(fake-define (load-shared-object "sockets-stub.dylib"))]
  [else
    (if (threaded?)
        #'(fake-define (load-shared-object "sockets-stub.so"))
        #'(fake-define))])
```

This section exports .

73. Binding Foreign Values. We define a single syntax `define-foreign-values` that binds identifiers to values computed by a foreign function. The basic idea is to provide a foreign shared object, the name of a function, and its return type, and then to use that function to get various values of that return type bound to the identifiers that we provide. This could be used for getting things like the sizes or values of various structures, for example. Our syntax follows the following:

```
(define-foreign-values <shared-object> <proc-name> <return-type>
                       <binding> ...)
```

Here `<shared-object>` is a string that will be used in a call to `load-shared-object`. The `<proc-name>` takes two forms, one with a convention, and the other without.

```
(<conv> <name-string>)
<name-string>
```

The `<name-string>` should be a string as passed to the function name of `foreign-procedure`. The `<conv>` is a convention as listed by the Chez Scheme User's Guide in `foreign-procedure`. This is used mostly on Windows where there are various calling conventions.

The `<return-type>` is just a valid `foreign-procedure` return type specifier.

Each `<binding>` should be an identifier. Each `<binding>` will be bound to the value returned by calling the foreign procedure on the string representation of the `<binding>`.

74. Let's first get the verifier for the syntax forms out of the way. We use this to verify our syntax later.

⟨ Verify DFV syntax 74 ⟩ ≡

```
(and (identifier? #'conv)
     (memq (syntax->datum #'conv) '(__cdecl __stdcall __com))
     (string? (syntax->datum #'shared-object))
     (string? (syntax->datum #'proc-name))
     (identifier? #'type)
     (for-all identifier? #'(binding ...)))
```

This section captures `conv, shared-object, proc-name, type, andbinding`.

This code is used in section 76.

75. Since these shared objects are not going to be distributed with the rest of the code, it's fair to assume that they won't be in the normal shared library paths. To make it possible to load these correctly then, I want to have a resolving procedure that can be used to find things in the current `source-directories`. This parameter is modified when the compiler is loading in libraries to make sure that I can load things in relative to the library's location. We need this at the meta level since that is where we are going to be doing the loading and importing.

⟨ Foreign code utilities 75 ⟩ +≡

```
(meta define (resolve name)
  (let loop ([dirs (source-directories)])
    (cond
      [(not (pair? dirs)) name]
      [(let ([path (format "~a~a~a" (car dirs) (directory-separator) name)])
         (and (file-exists? path) path))]
      [else (loop (cdr dirs))])))
```

This section exports `resolve`.

76. The actual `define-foreign-values` macro is fairly simple. We need to have access to the resolver, but in general, we want to create a special `get-ffi-value` function that, when called with an identifier, will return back the result of calling the foreign code. Once that's defined, we simply call it on each of the bindings. Since the `foreign-procedure` code is different depending on whether we have a convention or not, we'll define that helper here.

⟨ Foreign code utilities 76 ⟩ +≡

```
(define-syntax define-foreign-values
  (syntax-rules ()
    [(_ shared-object (conv proc-name) type binding ...)
     ⟨ Verify DFV syntax 74 ⟩
     (begin
       (meta define %get-ffi-value
         (begin
           (load-shared-object (resolve shared-object))
           (foreign-procedure conv proc-name (string) type)))
       ⟨ Define get-ffi-value 77 ⟩
       (define-bindings get-ffi-value binding ...))]
    [(_ shared-object proc-name type binding ...)
     (with-syntax ([conv #'__cdecl])
       ⟨ Verify DFV syntax 74 ⟩)
     (begin
       (meta define %get-ffi-value
         (begin
           (load-shared-object (resolve shared-object))
           (foreign-procedure proc-name (string) type)))
       ⟨ Define get-ffi-value 77 ⟩
       (define-bindings get-ffi-value binding ...))]))
```

This section exports `define-foreign-values`.

77. What about `get-ffi-value`? This syntactic function needs to convert the incoming binding into a string and pass it through to the getter. It then needs to bind the resulting value to the original name provided.

⟨ Define `get-ffi-value` 77 ⟩ ≡

```
(define-syntax (get-ffi-value x)
  (syntax-case x ()
    [(k name) (identifier? #'name)
     #`'#,(datum->syntax #'k
            (%get-ffi-value
             (symbol->string (syntax->datum #'name))))]))
```

This section captures `%get-ffi-value`.

This section exports `get-ffi-value`.

This code is used in section 76.

78. Finally, we need a definition for `define-bindings`.

⟨ Foreign code utilities 78 ⟩ +≡

```
(define-syntax define-bindings
  (syntax-rules ()
    [(_ get) (begin)]
    [(_ get binding) (define binding (get binding))]
    [(_ get binding rest ...)
     (begin (define binding (get binding))
       (define-bindings get rest ...))]))
```

This section exports `define-bindings`.

79. Foreign Socket Constants. We have the following constants that must be defined for our library.

⟨ Foreign constants 79 ⟩ ≡

```
(define-syntax (define-constants x)
  (syntax-case x ()
    [(k head ...)
     (datum->syntax #'k
       `(define-foreign-values ,@(syntax->datum #'(head ...)) int
          $error-again $error-in-progress $error-would-block #;$file-get-flag
          $file-set-flag $format-message-allocate-buffer
          $format-message-from-system $ipproto-ip $ipproto-raw $ipproto-tcp
          $ipproto-udp $option-non-blocking $socket-error $sol-socket
          %ai/canonname %ai/numerichost %ai/passive %msg/dont-route %msg/dont-wait
          %msg/out-of-band %msg/peek %msg/wait-all %shutdown/read
          %shutdown/read&write %shutdown/write %socket-domain/internet
          %socket-domain/internet-v6 %socket-domain/local %socket-domain/unix
          %socket-type/datagram %socket-type/random %socket-type/raw
          %socket-type/sequence-packet %socket-type/stream %somaxconn af-inet
          af-unix invalid-socket size-of/addr-in size-of/addr-un size-of/addrinfo
          size-of/integer size-of/ip size-of/pointer size-of/port size-of/protoent
          size-of/sa-family size-of/size-t size-of/sockaddr-in size-of/sockaddr-un
          size-of/socklen-t size-of/wsa-data unix-max-path))]))
(meta-cond
  [(windows?)
   (define-constants "socket-ffi-values.dll" (__cdecl "_get_ffi_value"))]
  [else
   (define-constants "socket-ffi-values.so" "get_ffi_value")])
```

This section exports $error-again, $error-in-progress, $error-would-block, $file-set-flag, $format-message-allocate-buffer, $format-message-from-system, $ipproto-ip, $ipproto-raw, $ipproto-tcp, $ipproto-udp, $option-non-blocking, $socket-error, $sol-socket, %ai/canonname, %ai/numerichost, %ai/passive, %msg/dont-route, %msg/dont-wait, %msg/out-of-band, %msg/peek, %msg/wait-all, %shutdown/read, %shutdown/read&write, %shutdown/write, %socket-domain/internet, %socket-domain/internet-v6, %socket-domain/local, %socket-domain/unix, %socket-type/datagram, %socket-type/random, %socket-type/raw, %socket-type/sequence-packet, %socket-type/stream, %somaxconn, af-inet, af-unix, invalid-socket, size-of/addr-in, size-of/addr-un, size-of/addrinfo, size-of/integer, size-of/ip, size-of/pointer, size-of/port, size-of/protoent, size-of/sa-family, size-of/size-t, size-of/sockaddr-in, size-of/sockaddr-un, size-of/socklen-t, size-of/wsa-data, andunix-max-path.

This code is used in section 1.

80. We use the following C code for the foreign values shared object.

⟨ *socket-ffi-values.c* 80 ⟩ ≡

```
#ifdef __NT__
#define WIN32
#define EXPORTED __declspec ( dllexport ) int cdecl
#endif
#ifdef __WINDOWS__
#define WIN32
#define EXPORTED __declspec ( dllexport ) int cdecl
#endif

#ifndef WIN32
#define EXPORTED int
#endif

#ifdef WIN32
#include <stddef.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#else
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <sys/unistd.h>
#include <sys/fcntl.h>
#endif

EXPORTED get_ffi_value(const char *val) {
  struct sockaddr_in sai;
#ifndef WIN32
  struct sockaddr_un sau;
#endif

  if (!strcmp(val, "$ipproto-ip")) return IPPROTO_IP;
  if (!strcmp(val, "$ipproto-raw")) return IPPROTO_RAW;
  if (!strcmp(val, "$ipproto-tcp")) return IPPROTO_TCP;
  if (!strcmp(val, "$ipproto-udp")) return IPPROTO_UDP;
  if (!strcmp(val, "$sol-socket")) return SOL_SOCKET;
  if (!strcmp(val, "%ai/canonname")) return AI_CANONNAME;
  if (!strcmp(val, "%ai/numerichost")) return AI_NUMERICHOST;
  if (!strcmp(val, "%ai/passive")) return AI_PASSIVE;
  if (!strcmp(val, "%msg/dont-route")) return MSG_DONTROUTE;
  /* if (!strcmp(val, "%msg/dont-wait")) return MSG_DONTWAIT;
  if (!strcmp(val, "%msg/wait-all")) return MSG_WAITALL; */
  if (!strcmp(val, "%msg/out-of-band")) return MSG_OOB;
  if (!strcmp(val, "%msg/peek")) return MSG_PEEK;
  if (!strcmp(val, "%socket-domain/internet")) return AF_INET;
  if (!strcmp(val, "%socket-domain/internet-v6")) return AF_INET6;
  if (!strcmp(val, "%socket-type/datagram")) return SOCK_DGRAM;
```

```
  if (!strcmp(val, "%socket-type/random")) return SOCK_RDM;
  if (!strcmp(val, "%socket-type/raw")) return SOCK_RAW;
  if (!strcmp(val, "%socket-type/sequence-packet")) return SOCK_SEQPACKET;
  if (!strcmp(val, "%socket-type/stream")) return SOCK_STREAM;
  if (!strcmp(val, "%somaxconn")) return SOMAXCONN;
  if (!strcmp(val, "af-inet")) return AF_INET;
  if (!strcmp(val, "size-of/addr-in")) return sizeof(struct sockaddr_in);
  if (!strcmp(val, "size-of/addrinfo")) return sizeof(struct addrinfo);
  if (!strcmp(val, "size-of/integer")) return sizeof(int);
  if (!strcmp(val, "size-of/ip")) return sizeof(struct in_addr);
  if (!strcmp(val, "size-of/pointer")) return sizeof(void *);
  if (!strcmp(val, "size-of/port")) return sizeof(sai.sin_port);
  if (!strcmp(val, "size-of/protoent")) return sizeof(struct protoent);
  if (!strcmp(val, "size-of/size-t")) return sizeof(size_t);
  if (!strcmp(val, "size-of/sockaddr-in")) return sizeof(struct sockaddr_in);
  if (!strcmp(val, "size-of/socklen-t")) return sizeof(socklen_t);
  /* if (!strcmp(val, "$file-get-flag")) return F_GETFL; */
#ifdef WIN32
  if (!strcmp(val, "size-of/sa-family")) return sizeof(unsigned short);
  if (!strcmp(val, "%shutdown/read")) return SD_RECEIVE;
  if (!strcmp(val, "%shutdown/read&write")) return SD_BOTH;
  if (!strcmp(val, "%shutdown/write")) return SD_SEND;
  if (!strcmp(val, "invalid-socket")) return INVALID_SOCKET;
  if (!strcmp(val, "$file-set-flag")) return FIONBIO;
  if (!strcmp(val, "$option-non-blocking")) return 1;
  if (!strcmp(val, "$error-again")) return WSAEWOULDBLOCK;
  if (!strcmp(val, "$error-in-progress")) return WSAEINPROGRESS;
  if (!strcmp(val, "$error-would-block")) return WSAEWOULDBLOCK;
  if (!strcmp(val, "$socket-error")) return SOCKET_ERROR;
  if (!strcmp(val, "$format-message-allocate-buffer"))
    return FORMAT_MESSAGE_ALLOCATE_BUFFER;
  if (!strcmp(val, "$format-message-from-system"))
    return FORMAT_MESSAGE_FROM_SYSTEM;
  if (!strcmp(val, "size-of/wsa-data")) return sizeof(WSADATA);
#else
  if (!strcmp(val, "size-of/sa-family")) return sizeof(sa_family_t);
  if (!strcmp(val, "%shutdown/read")) return SHUT_RD;
  if (!strcmp(val, "%shutdown/read&write")) return SHUT_RDWR;
  if (!strcmp(val, "%shutdown/write")) return SHUT_WR;
  if (!strcmp(val, "invalid-socket")) return -1;
  if (!strcmp(val, "$file-set-flag")) return F_SETFL;
  if (!strcmp(val, "$option-non-blocking")) return O_NONBLOCK;
  if (!strcmp(val, "$error-again")) return EAGAIN;
  if (!strcmp(val, "$error-in-progress")) return EINPROGRESS;
  if (!strcmp(val, "$error-would-block")) return EWOULDBLOCK;
  if (!strcmp(val, "$socket-error")) return -1;
  if (!strcmp(val, "%socket-domain/local")) return AF_LOCAL;
  if (!strcmp(val, "%socket-domain/unix")) return AF_UNIX;
  if (!strcmp(val, "af-unix")) return AF_UNIX;
  if (!strcmp(val, "size-of/sockaddr-un")) return sizeof(struct sockaddr_un);
  if (!strcmp(val, "unix-max-path")) return sizeof(sau.sun_path);
  if (!strcmp(val, "size-of/addr-un")) return sizeof(struct sockaddr_un);
```

```
#endif

  return 0;
}
```

81. Foreign Stub File. We have a C file separate from the foreign values library that is expected to be available at runtime. It's not necessary to have the foreign values shared object available at runtime, since all of that operates at compile time, but it is necessary to have the stub file around for persistent code. We set up the headers for that file here.

⟨ *sockets-stub.c* 81 ⟩ ≡

```
#ifdef __NT__
#define WIN32
#elif defined __WINDOWS__
#define WIN32
#endif

#ifdef WIN32

#define EXPORTED(type) __declspec ( dllexport ) type cdecl

#include <stddef.h>
#include <winsock2.h>
#include <ws2tcpip.h>

typedef int ssize_t;

#define SCHEME_STATIC

#else

#define EXPORTED(type) type

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <netinet/in.h>
#include <netdb.h>
#include <errno.h>
#include <sys/unistd.h>
#include <sys/fcntl.h>

#endif

#ifdef WIN32
extern __declspec(dllimport) int cdecl Sactivate_thread(void);
extern __declspec(dllimport) void cdecl Sdeactivate_thread(void);
#else
#include "scheme.h"
#endif
```

See also sections 83and86.

82. Dealing with errors. We need to provide some mechanism for getting the error report for foreign code. This is different on Windows than on UNIX systems.

⟨ Foreign functions 82 ⟩ ≡
```
(meta-cond
  [(windows?) ⟨ Define Windows foreign error handler 85 ⟩]
  [else ⟨ Define POSIX foreign error handler 84 ⟩])
(define (call-with-errno thunk receiver)
  (call-with-values
   (lambda () (critical-section (let ([v (thunk)]) (values v (errno)))))
   receiver))
```
This section exports `call-with-errnoanderrno-message`.

See also sections 88, 89, 90, 92, 93, 94, 95, 96, 97, 98, 99, 100, and101.

This code is used in section 1.

83. On the POSIX systems we are allowed to just use the value of `errno` so we do this through the stub file.

⟨ *sockets-stub.c* 83 ⟩ +≡
```
EXPORTED(int)
get_errno()
{
        return errno;
}
```

84. ⟨ Define POSIX foreign error handler 84 ⟩ ≡
```
(define errno (foreign-procedure "get_errno" () int))
(define errno-message (foreign-procedure "strerror" (int) string))
```
This section exports `errnoanderrno-message`.

This code is used in section 82.

85. In the case of Windows systems, you are not supposed to use the `errno` value to get the value. instead, you are supposed to use `WSAGetLastError`.

⟨ Define Windows foreign error handler 85 ⟩ ≡

```
(define errno (foreign-procedure "WSAGetLastError" () int))
(define errno-message
  (let ([$format-message
         (foreign-procedure __stdcall "FormatMessageA"
                            (unsigned-32 uptr unsigned-32
                             unsigned-32 uptr unsigned-32 uptr)
                            unsigned-32)]
        [$local-free (foreign-procedure __stdcall "LocalFree" (uptr) void)])
    (lambda (num)
      (let* ([ptr (make-foreign-pointer)]
             [ret-res ($format-message
                       (bitwise-xor $format-message-allocate-buffer
                                    $format-message-from-system)
                       0 num 0 ptr 0 0)]
             [res (and (not (zero? ret-res))
                       (get-foreign-string (foreign-pointer-value ptr)))])
        ($local-free (foreign-pointer-value ptr))
        (foreign-free ptr)
        res))))
```

This section exports `errnoanderrno-message`.

This code is used in section 82.

86. Supporting blocking operations. The following C functions are necessary to support potentially blocking socket operations in the threaded versions of Chez Scheme. Each one uses the following macro to guard the main expression that could block.

⟨ *sockets-stub.c* 86 ⟩ +≡

```
#define GUARD(exp) do { \
    Sdeactivate_thread(); \
    exp; \
    Sactivate_thread(); \
    } while (0)

/* Blocking Accept */
EXPORTED(int)
accept_block(int fd, struct sockaddr *addr, socklen_t *addrlen) {
        int ret;
        GUARD(ret = accept(fd, addr, addrlen));
        return ret;
}

/* Blocking Connect */
EXPORTED(int)
connect_block(int fd, const struct sockaddr *addr, socklen_t addrlen) {
        int ret;
        GUARD(ret = connect(fd, addr, addrlen));
        return ret;
}

/* Blocking Receive */
EXPORTED(ssize_t)
recvfrom_block(int fd, void *buf, size_t len, int flags,
    struct sockaddr *src_addr, socklen_t *addrlen) {
        ssize_t ret;
        GUARD(ret = recvfrom(fd, buf, len, flags, src_addr, addrlen));
        return ret;
}

/* Blocking Send To */
EXPORTED(int)
sendto_block(int fd, const void *buf, size_t len, int flags,
    const struct sockaddr *dest_addr, socklen_t addrlen) {
        ssize_t ret;
        GUARD(ret = sendto(fd, buf, len, flags, dest_addr, addrlen));
        return ret;
}
```

87. Foreign procedures. We rely on quite a few foreign procedures for our ditry work. Unfortunately, supporting Windows makes our life miserable, yet again. To aleviate some of this misery, the following macros allow me to hide away some of the details of supporting both platforms.

⟨ Foreign code utilities 87 ⟩ +≡
```
(define-syntax define-ffi
  (syntax-rules ()
    [(_ name ffiname in out)
     (define name
       (meta-cond
        [(windows?) (foreign-procedure __stdcall ffiname in out)]
        [else (foreign-procedure ffiname in out)]))]))
```

This section exports `define-ffi`.

88. Here is our set of foreign procedures that we rely on.

⟨ Foreign functions 88 ⟩ +≡
```
(define-ffi $socket "socket" (fixnum fixnum fixnum) fixnum)
(define-ffi $getaddrinfo "getaddrinfo" (string string uptr uptr) int)
(define-ffi $getprotobyname "getprotobyname" (string) uptr)
(define-ffi $getprotobynumber "getprotobynumber" (fixnum) uptr)
(define-ffi $bind "bind" (fixnum uptr fixnum) fixnum)
(define-ffi $listen "listen" (fixnum fixnum) fixnum)
(define-ffi $accept "accept" (fixnum uptr uptr) fixnum)
(define-ffi $connect "connect" (fixnum uptr fixnum) fixnum)
(define-ffi $shutdown "shutdown" (fixnum fixnum) fixnum)
(define-ffi $sendto "sendto" (fixnum u8* fixnum fixnum uptr fixnum) fixnum)
(define-ffi $recvfrom "recvfrom" (fixnum u8* fixnum fixnum uptr uptr) fixnum)
(define-ffi $getsockopt "getsockopt" (int int int uptr uptr) int)
(define-ffi $setsockopt "setsockopt" (int int int uptr int) int)
(meta-cond
  [(windows?)
   (define $gai_strerror errno-message)
   (define-ffi $close "closesocket" (unsigned) int)
   (define-ffi $fcntl "ioctlsocket" (unsigned unsigned unsigned) int)]
  [else
   (define $gai_strerror (foreign-procedure "gai_strerror" (int) string))
   (define-ffi $close "close" (int) int)
   (define-ffi $fcntl "fcntl" (int int long) int)])
```

This section exports `$getaddrinfo`, `$gai_strerror`, `$socket`, `$getprotobyname`, `$getprotobynumber`, `$bind`, `$listen`, `$accept`, `$connect`, `$close`, `$shutdown`, `$sendto`, `$recvfrom`, `$getsockopt`, `$setsockopt`, and`$fcntl`.

89. Unfortunately, there are some things that Windows just doesn't support at all.

⟨ Foreign functions 89 ⟩ +≡
```
(meta-cond
  [(windows?)
   (define ($getprotoent) (unsupported-feature '$getprotoent))
   (define ($setprotoent) (unsupported-feature '$setprotoent))
   (define ($endprotoent) (unsupported-feature '$endprotoent))]
  [else
   (define $getprotoent (foreign-procedure "getprotoent" () uptr))
   (define $setprotoent (foreign-procedure "setprotoent" (boolean) void))
   (define $endprotoent (foreign-procedure "endprotoent" () void))])
```

This section exports `$getprotoent`, `$setprotoent`, and`$endprotoent`.

90. There are four procedures that could potentially block for long times when we use them, and these require that we deactivate the blocked foreign thread before we block, so that the rest of Chez Scheme can work. This is only necessary if we are on a threaded version of Chez Scheme,

⟨ Foreign functions 90 ⟩ +≡
```
(meta-cond
  [(threaded?)
   (define $accept-blocking
     (foreign-procedure "accept_block" (fixnum uptr uptr) int))
   (define $connect-blocking
     (foreign-procedure "connect_block" (fixnum uptr fixnum) int))
   (define $sendto-blocking
     (foreign-procedure "sendto_block"
                        (fixnum u8* fixnum fixnum uptr fixnum)
                        int))
   (define $recvfrom-blocking
     (foreign-procedure "recvfrom_block"
                        (fixnum u8* fixnum fixnum uptr uptr)
                        int))]
  [else
   (define $accept-blocking
     (foreign-procedure "accept" (fixnum uptr uptr) int))
   (define $connect-blocking
     (foreign-procedure "connect" (fixnum uptr fixnum) int))
   (define $sendto-blocking
     (foreign-procedure "sendto" (fixnum u8* fixnum fixnum uptr fixnum) int))
   (define $recvfrom-blocking
     (foreign-procedure "recvfrom" (fixnum u8* fixnum fixnum uptr uptr) int))])
```

This section exports `$accept-blocking`, `$connect-blocking`, `$sendto-blocking`, and`$recvfrom-blocking`.

91. Foreign Data Utilities. The following functions help in manage the foreign data structures that map to our normal data definitions. This includes constructing, accessing, and deallocating them. Most foreign data structures are allocated on the heap and can thus be removed by using the `foreign-free` function.

92. Foreign UNIX Addresses. To start, let's define a constructor for the foreign UNIX address structures. We are copying the following C structure:

```
#define UNIX_PATH_MAX 108
struct sockaddr_un {
sa_family_t sun_family;               /* AF_UNIX */
char        sun_path[UNIX_PATH_MAX];  /* pathname */
};
```

The `sun_family` always maps to the `af-unix` constant and the size of the path is restricted to `unix-max-path`.

⟨ Foreign functions 92 ⟩ +≡

```
(define make-foreign-unix-address
  (let ([$strcpy (foreign-procedure "strcpy" (uptr string) void)])
    (lambda (path)
      (let ([res (foreign-alloc size-of/sockaddr-un)]
            [path-len (string-length path)])
        (assert (< path-len unix-max-path))
        (foreign-set! 'unsigned-short res 0 af-unix)
        ($strcpy (+ res size-of/sa-family) path)
        res))))
```

This section exports `make-foreign-unix-address`.

93. We have only one accessor for the structure, and we comfortably deviate from the C name here for the sake of normal Scheme naming patterns.

⟨ Foreign functions 93 ⟩ +≡

```
(define foreign-unix-address-path
  (let ([$strncpy (foreign-procedure "strncpy" (u8* uptr fixnum) string)])
    (lambda (addr)
      ($strncpy (make-bytevector unix-max-path 0)
                (+ addr size-of/sa-family)
                unix-max-path))))
```

This section exports `foreign-unix-address-path`.

94. Foreign INET4 Addresses. Next, we define the constructor for INET4 foreign addresses. We are following this pattern of the INET4 internet address C structure.

```
struct sockaddr_in {
sa_family_t     sin_family; /* address family: AF_INET */
in_port_t       sin_port;   /* port in network byte order */
struct in_addr  sin_addr;   /* internet address */
};

struct in_addr {
uint32_t        s_addr;     /* address in network byte order */
};
```

⟨ Foreign functions 94 ⟩ +≡

```
(define (make-foreign-ipv4-address port ip)
  (let ([res (foreign-alloc size-of/sockaddr-in)])
    (foreign-set! 'unsigned-short res 0 af-inet)
    (foreign-set! 'unsigned-16
                  res
                  (foreign-sizeof 'unsigned-short)
                  (host->network/u16 port))
    (foreign-set! 'unsigned-32
                  res
                  (+ (foreign-sizeof 'unsigned-short)
                     (foreign-sizeof 'unsigned-16))
                  (host->network/u32 ip))
    res))
```

This section exports `make-foreign-ipv4-address`.

95. There are two accessors for the port and the internet address.

⟨ Foreign functions 95 ⟩ +≡

```
(define (foreign-ipv4-address-ip addr)
  (network->host/u32
    (foreign-ref 'unsigned-32
                 addr
                 (+ (foreign-sizeof 'unsigned-short)
                    (foreign-sizeof 'unsigned-16)))))

(define (foreign-ipv4-address-port addr)
  (network->host/u16
    (foreign-ref 'unsigned-16
                 addr
                 (foreign-sizeof 'unsigned-short))))
```

This section exports `foreign-ipv4-address-ipandforeign-ipv4-address-port`.

96. The above utilities require tools for converting too and from the network byte orders. We do this using the following helpers.

⟨ Foreign functions 96 ⟩ +≡

```
(define host->network/u16
  (if (eq? (native-endianness) (endianness big))
      (lambda (x) x)
      (lambda (x)
        (let ([buf (make-bytevector 2)])
          (bytevector-u16-set! buf 0 x (endianness big))
          (bytevector-u16-ref buf 0 (native-endianness))))))

(define host->network/u32
  (if (eq? (native-endianness) (endianness big))
      (lambda (x) x)
      (lambda (x)
        (let ([buf (make-bytevector 4)])
          (bytevector-u32-set! buf 0 x (endianness big))
          (bytevector-u32-ref buf 0 (native-endianness))))))

(define network->host/u32 host->network/u32)
(define network->host/u16 host->network/u16)
```

This section exports `host->network/u16`, `host->network/u32`, `network->host/u16`, and`network->host/u32`.

97. Foreign address information structures. These procedures are used to manage the foreign address information structures.

⟨ Foreign functions 97 ⟩ +≡

```
(define (make-foreign-pointer)
  (foreign-alloc (foreign-sizeof 'void*)))

(define (foreign-pointer-value x)
  (foreign-ref 'void* x 0))

(define family-offset (foreign-sizeof 'int))
(define type-offset (+ family-offset (foreign-sizeof 'int)))
(define proto-offset (+ type-offset (foreign-sizeof 'int)))
(define addrlen-offset (+ proto-offset (foreign-sizeof 'int)))
(define addr-offset (+ addrlen-offset (foreign-sizeof 'unsigned-long)))
(define name-offset (+ addr-offset (foreign-sizeof 'void*)))
(define next-offset (+ name-offset (foreign-sizeof 'void*)))

(define (make-foreign-address-info
          flags family type proto addrlen addr name next)
  (let ([res (foreign-alloc size-of/addrinfo)])
    (foreign-set! 'int res 0 flags)
    (foreign-set! 'int res family-offset family)
    (foreign-set! 'int res type-offset type)
    (foreign-set! 'int res proto-offset proto)
    (foreign-set! 'unsigned-long res addrlen-offset addrlen)
    (foreign-set! 'void* res addr-offset addr)
    (foreign-set! 'void* res name-offset name)
    (foreign-set! 'void* res next-offset next)
    res))

(define (foreign-address-info-canonical-name addrinfo)
  (let ([ptr (foreign-ref 'void* addrinfo name-offset)])
    (if (zero? ptr) #f (get-foreign-string ptr))))

(define (foreign-address-info-domain addrinfo)
  (foreign-ref 'int addrinfo family-offset))
(define (foreign-address-info-type addrinfo)
  (foreign-ref 'int addrinfo type-offset))
(define (foreign-address-info-protocol addrinfo)
  (foreign-ref 'int addrinfo proto-offset))
(define (foreign-address-info-address addrinfo)
  (foreign-ref 'void* addrinfo addr-offset))
(define (foreign-address-info-address-length addrinfo)
  (foreign-ref 'unsigned-long addrinfo addrlen-offset))
(define (foreign-address-info-next addrinfo)
  (foreign-ref 'void* addrinfo next-offset))
```

This section exports `make-foreign-pointer`, `foreign-pointer-value`, `make-foreign-address-info`, `foreign-address-info-canonical-name`, `foreign-address-info-next`, `foreign-address-info-domain`, `foreign-address-info-type`, `foreign-address-info-protocol`, `foreign-address-info-address`, `andforeign-address-info-address-length`.

98. Foreign protocol database structures. The following functions are needed to manage foreign protocol databases.

⟨ Foreign functions 98 ⟩ +≡
```
(define (foreign-protocol-entry-name x)
  (get-foreign-string (foreign-pointer-value x)))

(define (foreign-protocol-entry-aliases x)
  (do ([ptr (foreign-ref 'void* x (foreign-sizeof 'void*))
            (+ ptr (foreign-sizeof 'void*))]
       [res '()  (cons (get-foreign-string (foreign-pointer-value ptr))
                       res)])
      [(zero? (foreign-pointer-value ptr)) (reverse res)]))

(define (foreign-protocol-entry-protocol x)
  (foreign-ref 'int x (* 2 (foreign-sizeof 'void*))))
```
This section exports `foreign-protocol-entry-name`, `foreign-protocol-entry-aliases`, `andforeign-protocol-entry-protocol`.

99. Managing foreign strings and bufers. It's somewhat awkward to get a C string, so we use the following function to abstract this functionality.

⟨ Foreign functions 99 ⟩ +≡
```
(define get-foreign-string
  (let ([$strlen (foreign-procedure "strlen" (uptr) fixnum)]
        [$strcpy (foreign-procedure "strcpy" (u8* uptr) string)])
    (lambda (x)
      (let* ([len ($strlen x)]
             [buf (make-bytevector (1+ len))])
        ($strcpy buf x)))))
```
This section exports `get-foreign-string`.

100. We sometimes need to allocate buffers of appropriate size and type. Such as those used for value-result parameters in foreign C functions. We define some conveniences here for doing this.

⟨ Foreign functions 100 ⟩ +≡
```
(define (make-foreign-size-buffer size)
  (let ([res (foreign-alloc (foreign-sizeof 'unsigned-long))])
    (foreign-set! 'unsigned-long res 0 size)
    res))

(define (foreign-size-buffer-value buf)
  (foreign-ref 'unsigned-long buf 0))
```
This section exports `make-foreign-size-bufferandforeign-size-buffer-value`.

101. Blocking sockets. The following helper is used to manage blocking sockets.

⟨ Foreign functions 101 ⟩ +≡

```
(meta-cond
  [(windows?)
   (define (%set-blocking fd yes?)
     (let ([buf (make-foreign-size-buffer (if yes? 0 1))])
       ($fcntl fd $file-set-flag buf)))]
  [else
    (define (%set-blocking fd yes?)
      ($fcntl fd $file-set-flag (if yes? 0 $option-non-blocking)))])
```

This section exports `%set-blocking`.

102. Windows Initialization. When working on Windows, there is an initialization sequence that we have to do in order to use any of the sockets functionality. It looks like this:

⟨ Foreign code initialization 102 ⟩ ≡

```
(meta-cond
  [(windows?)
   (define $wsa-startup
     (foreign-procedure __stdcall "WSAStartup"
       (unsigned-16 uptr)
       int))
   (define winsock-version (+ (bitwise-arithmetic-shift 2 8) 2))
   (let ([buf (foreign-alloc size-of/wsa-data)])
     ($wsa-startup winsock-version buf)
     (foreign-free buf))]
  [else (void)])
```

This section exports .

This code is used in section 1.

103. Index.

Native Scheme Sockets for Chez Scheme

(Version 2.0)