YASOL

Yet Another Socket Library for Forth

Rafael Gonzalez Fuentetaja

YASOL
User manual
Manual revision 1.00
12 February 2008

Table of Contents

1 Linux TCP and UDP Socket Bindings

Defines a socket wordset to be used in a variety of situations. The main I/O words have been defined with a similar stack effect as the *file* wordset. The *ior* codes are throwable.

Choosing names for this BDS style socket API has been problematic because it uses common words such as *bind* or *close* that could clash with other applications or wordlists. I have opted to append an `s` (for *socket*) to most of these words to keep them readable, concise and avoid name clashes. So *close* becomes *closes* and *bind* becomes *binds*.

Regarding portability, There has been no special effort to produce a portable library, only to make it work. It would probably be quite easy to port it to GForth. Some asumptions taken:

- 1 char = 1 byte = 8 bits.
- 1 cell = 4 bytes = 32 bits.
- There are words to read/write 16 bit values on unaligned addresses.
- IP addresses are IPv4 32 bit addresses, fitting into 1 cell.
- Structure defining words supplied by VFX Forth, whose `field` produces unaligned fields. Apparently, that's what was needed to clone some C wire structures.
- There is a facility to access external shared library. I have used the ones supplied bf VFX Forth.
- Used VFX extended locals notation in some cases to interface C system calls.
- There are available some convenient but non ANS standard words like `3drop` `4drop` `3dup` `4dup`.
- There is a word to print z-strings (actually, it is `.z$`).

1.1 Non-blocking sockets and multitasking

A non-blocking I/O wordset has been included to use together with cooperative multitaskers. Since *VFX Forth for Linux* includes a preemptive multitasker, blocking I/O can be used within each task and the non-blocking counterparts have, in principle, little interest. However, facilities like messages between tasks or task events are driven using the traditional `pause` word. If you plan to use such facilities, then you must use the non-blocking words to ensure that `pause` is called frequently enough.

1.2 Glossary

1.2.1 Handling Linux `errno` codes

Socket I/O uses a variety of system calls that may fail for a number of reasons. The following words help keep track of *errno* codes thrown as exceptions.

As far as I know, VFX do not have yet embedded all the *errno* codes as part of its error handling wordset, nor do not offer an easy way to automatically read the z-strings returned by *strerror()* and define them with `#AnonErr`. All of VFX's error defining words are parsing words.

Note: This section is likely to change as soon as possible.

```
-5000 Value errno-base
```

Base value for OS *errno* throw codes. Can be customized at runtime to coexist with other throw codes.

`: errno>excp \ errno -- n1`
Converts Linux *errno* values in throw codes.

`: excp>errno \ n1 -- errno`
Converts throw code back to genuine Linux *errno* values.

`AliasedExtern: errno int * __errno_location(void);`
`errno` is the well known *errno* C global variable used by libraries and system calls. Actually, it is a thread local (aka `USER`) variable. Can be read by `@` and written by `!`

`: throwable \ flag -- -ior`
Takes a boolean *flag* and makes it a throwable *ior* with the `False` makes a zero *ior*.

`Extern: char * strerror(int errnum); \ -- z-addr`
Get the human readable *errno* string. See *man strerror* for details.

`: .errno \ n1 --`
Given the Linux *n1 errno* number prints the associated z-string. A factor for the word below.

`: .errno-excp \ n1 --`
Prints the errno exception value thrown in *n1*. *n1* must be `< errno-base`.

1.2.2 Handling IP addresses

The following words help using and declaring IP addresses.

Reusable bits

`struct /hostent \ -- len`
Mimics the Linux *struct hostent*. See *man gethostbyname*.

`struct /sockaddr_in \ -- len`
Mimics the Linux *struct sockaddr_in*. See *man ip*.

`2 Constant PF_INET`
IP sockets family.

`0 Constant INADDR_ANY`
Used to listen to any IP address of a host.

`-1 Constant INADDR_BROADCAST`
The special broadcast address 255.255.255.255 encoded as binary.
The following constants are possible values left in *h_errno* C global variable. Comments are extracted form the `<netdb.h>` header file.

`-1 Constant NETDB_INTERNAL`
Internal error. See *errno*.

`0 Constant NETDB_SUCCESS`
No problem.

`1 Constant HOST_NOT_FOUND`
Authoritative Answer Host not found.

`2 Constant TRY_AGAIN`
Non-Authoritative: Host not found, or SERVERFAIL.

`3 Constant NO_RECOVERY`

Non recoverable errors, FORMERR, REFUSED, NOTIMP.

`4 Constant NO_DATA`

Valid name, no data record of requested type.

`4 Constant NO_ADDRESS`

No address, look for MX record. Currently, the same value as `NO_DATA`.

`AliasedExtern: h_errno int * __h_errno_location (void);`

`h_errno` is the *h_errno* C global variable. Actually, it is a thread local (aka `USER`) variable. Can be read by `@` and written by `!`

`AliasedExtern: ux-gethostbyname void *  gethostbyname(const char * name);`

See *man gethostbyname*.

`Extern: uint32  htonl(uint32 hostlong);`

See *man htonl*.

`Extern: uint16  htons(uint16 hosthort);`

See *man htons*.

`: c-string                          \ c-addr1 u -- c-addr2`

Converts a Forth string into an null terminated C string. The z-string *c-addr2*is stored in the `PAD`. Taken from GForth.

`: gethostbyname                                \ c-addr1 u -- addr2 ior`

Get a *struct hostent* from a symbolic name or IP dotted notation Returns the `/hostent` structure base address *addr2* and an *ior*. Negative *ior* values can be thrown as exceptions. Positive *ior* values denotes one of `HOST_NOT_FOUND`, `TRY_AGAIN`, `NO_RECOVERY`, `NO_DATA NO_ADDRESS` failed search results errors. *addr2* is undefined if *ior* `<>` 0.

`: host>addr                    \ c-addr1 u -- u1`

Converts an Internet host name into a binary *u1* IPv4 address. The resulting address is in network byte order. This word may throw *errno* exceptions if internal errors are detected. It may also call `abort"` with the texts shown in the constants glossary above if the search results are one of the following `HOST_NOT_FOUND`, `TRY_AGAIN`, `NO_RECOVERY`, `NO_DATA NO_ADDRESS` values.

`: port!                      \ port addr1 --`

Fills int the `/sockaddr_in` structure or `EndPoint:` given by *addr1* with the `PF_INET` protocol family and *port*.

`struct /endpoint                              \ -- addr`

Extends the `/sockaddr_in` structure with a `ep.rlen` field for convenience. This field is a convenient placeholder for storing returned length of a `/sockaddr_in` structure in various Linux system calls. Allocation of `/endpoint` structures can be made with `allocate` or `EndPoint:` (recommended).

Main words

`: EndPoint:                         \ "<name>" -- addr1`

Defining word to create an IP end point. The `/sockaddr_in` structure address *addr1* is returned just to be initialized by words `known-ip`, `unknown-ip any-ip` or `broadcast-ip`.

Examples:

```
s" localhost" 7624 EndPoint: remote known-ip \ for clients
7624 EndPoint: local any-ip                  \ for servers
EndPoint: receiver unknown-ip                \ to use in recvsfrom
```

`: known-ip                                  \ c-addr1 u1 port addr2 --`
An initializer to a `/sockaddr_in` given by *addr2* that declares a well known *port* plus *c-addr1 u1* IP address, typically used by clients to connect to remote peers or servers. Host string can be either a symbolic name or a IP dotted notation string. This word calls the resolver `host>addr` word, so it may throw *errno* exceptions.

`: any-ip                     \ port addr1 --`
An initializer to a `/sockaddr_in` given by *addr1* that declares an end point ready to listen to a given *port* but any IP address. Typically used by servers.

`: broadcast-ip                    \ port addr1 --`
An iniializer to a `/sockaddr_in` given by *addr1* that declares an end point ready to send broadcast messages to a given *port*. Can be used by clients or in UDP based application protocols to announce and discover.

`: unknown-ip                \ addr1 --`
An initializer to a `/sockaddr_in` given by *addr1* that declares an uninitialized end point to be filled by `binds`, `accepts` or `recvsfrom` words. Syntactic sugar.

1.2.3 TCP/IP Sockets

Reusable bits

Description of most words in this section is available by generating the additional detailed level of documentation.

`AliasedExtern: ux-socket int socket(int domain, int type, int protocol);`
See *man socket*.

`AliasedExtern: ux-bind int bind(int fd, void * my_addr, int addrlen);`
See *man bind*.

`AliasedExtern: ux-listen int listen(int fd, int backlog);`
See *man listen*.

`AliasedExtern: ux-connect int connect(int fd, const int * serv_addr, int addr_len);`
See *man connect*.

`AliasedExtern: ux-accept int accept(int fd, void * rem_addr, int * addr_len);`
See *man accept*.

`AliasedExtern: ux-close int close(int fd);`
See *man close*.

`AliasedExtern: ux-send int send(int s, const void * msg, int len, int flags);`
See *man send*.

`AliasedExtern: ux-recv int recv(int s, void * msg, int len, int flags);`
See *man recv*.

`Extern: int setsockopt(int fd, int level, int opt, void * val, int vallen);`
See *man setsockopt*.

`1 Constant SOL_SOCKET`
Select socket level in *get/setsockopt()*

`2 Constant SO_REUSEADDR`
Socket level option to reuse address. Typically used in server sockets.

`1 Constant SOCK_STREAM`
Stream type socket.

`2 Constant SOCK_DGRAM`
Datagram type socket.

`: reuse-IP \ fd -- ior`
Configure a socket *fd* to reuse its IP address through `setsockopt`. Usually done at servers. Returns a throwable *ior*.

Main words

`#6 Constant TCP`
Used to create a TCP socket with `socket`. Same value as Linux `#define IPPROTO_TCP`.

`#17 Constant UDP`
Used to create an UDP socket with `socket`. Same value as Linux `#define IPPROTO_UDP`.

`: socket \ n1 -- fd ior`
Creates a TCP or UDP socket given the constant `TCP` or `UDP` passed in *n1*. Returns a descriptor *fd* and throwable *ior*. *fd* is undefined when the *ior* is not zero. Socket I/O on the returned *fd* is blocking by default. See *man socket*.

`: connects \ addr fd -- ior`
Client connects to a remote socket server. *addr* is a remote `/sockaddr_in` or `/endpoint` structure. Returns a throwable *ior*.

`: binds \ addr fd -- ior`
Binds the server socket to a port and address (ANY address usually). Reuse port+IP address if possible. *addr* can be either a `/sockaddr_in` or a `/endpoint` structure. Returns a throwable *ior*.

`#5 Value BackLog`
Backlog of pending connections to `listens`.

`: listens \ fd -- ior`
Server socket *fd* listens to incoming connections. Returns a throwable *ior*. See *man listen*.

`: accepts \ addr1 fd1 -- fd2 ior`
Accept an incoming socket connection. Returns a new socket descriptor *fd2* and a throwable *ior*. *addr1* **must** be a defined an `/endpoint` struct to be filled in with the incoming client IP address & port. *fd2* is undefined when *ior* is non-zero. Socket I/O on the returned *fd2* is blocking by default. See *man accept*.

`: closes \ fd -- ior`
Close the socket. Returns a throwable *ior*. See *man close*.

`$4000 Constant MSG_NOSIGNAL`
Transmission/reception flag to make `<recvs>` or `<sends>` not to throw a *SIGPIPE* signal when the other end closes a connection. Only applicable to connection oriented sockets. See *man send* or *man recv*.

`$02 Constant MSG_PEEK`
Reception flag to make `<recvs>` or `<recvsfrom>` peek at incoming data without dequeueing from system internal buffers. See *man recv*.

`$100 Constant MSG_WAITALL`
Reception flag to make `<recvs>` or `<recvsfrom>` block until all requested bytes have been read. See *man recv*.

`: <sends>  \ c-addr1 +n1 fd n2 -- +n3 ior`
Send *n1* bytes starting from *c-addr1* through socket given by *fd*. Transmission flags must be or-ed in a single *n2*. The actual amount sent is returned in *n3*. In certain conditions, *n3* can be < *n1*. Returns a throwable *ior*. *n3* is undefined if ior is non-zero. See *man send*.

`: sends  \ c-addr1 +n1 fd -- +n2 ior`
The common use case for transmission. `SIGPIPE` is disabled. If the remote end closes connection, *ior* will contain a `EPIPE` throwable errno code. See *man send*. Defined as :

```
MSG_NOSIGNAL <sends> ;
```

`: <recvs>  \ c-addr1 +n1 fd n2 -- +n3 ior`
Receive through socket given by *fd* an amount of *n1* bytes and copy them into *c-addr1*. Reception flags must be or-ed in a single *n2*. Returns the actual amount received in *n3* and a throwable *ior*. If there is no data, the socket will block until something is available. If the socket is non-blocking and there is no data, *ior* will contain a `EAGAIN` throwable exception. See *man recv*.

`#32 Constant EPIPE`
errno code returned when the other end closes a connection.

`: epipe?  \ #read ior1 -- #read ior2`
Generate an artificial, throwable `EPIPE` errno code when the remote end closes the connection and signals are disabled. This happens when `<recvs>` returns `0 0`.

`: recvs  \ c-addr1 +n1 fd -- +n2 ior`
The common use case for reception. If the remote side closes the connection, `recvs` generates an artificial errno code through `epipe?`. Defined as :

```
MSG_NOSIGNAL <recvs> epipe? ;
```

`: recvs-all  \ c-addr1 +n1 fd -- +n2 ior`
Blocks until **all** *n1* bytes are received. If the remote side closes connection, \fo{recvs-all} generates an artificial errno code through `epipe?`. Defined as :

```
[ MSG_WAITALL MSG_NOSIGNAL or ] literal <recvs> epipe? ;
```

1.2.4 UDP Sockets

This section defines additional words to handle UDP sockets.

Reusable bits

`AliasedExtern: ux-sendto int sendto(int, const void *, int, int, const void *, int);`
See *man sendto*.

`AliasedExtern: ux-recvfrom int recvfrom(int, void *, int, int, void *, int *);`
See *man recvfrom*.

Main words

`: sendsto  \ c-addr1 +n1 addr2 fd -- +n2 ior`
Send *n1* bytes starting from *c-addr1* through socket given by *fd* to an `/endpoint` or `/sockaddr_in` given by *addr2*. *n2* is the actual amount of data sent. Returns a throwable *ior*. *n2* is undefined if *ior* is non-zero. See *man sendto*.

`: <recvsfrom> { c-addr1 n1 addr2 fd n2 -- +n3 ior }`
Receive data from a remote `/endpoint` *addr2*. *n1* is the amount to read through socket *fd* and *c-addr1* is the receiving buffer. Recepcion flags in *n2* must be or-ed together. Returns the actual amount received in *n3* and a throwable *ior*. See *man recvfrom*.

`: recvsfrom \ c-addr1 +n1 addr2 fd -- +n2 ior`
The common use case for reception. Equivalent to `<recvsfrom>` when *n2* = 0. See `<recvsfrom>`.

`: recvsfrom-all \ c-addr1 +n1 addr2 fd -- +n2 ior`
Blocks until all *n1* bytes are received. Equivalent to `<recvsfrom>` when *n2* = `MSG_WAITALL` flag. See `<recvsfrom>`.

1.2.5 Non-blocking connection-oriented socket I/O

The following words are intended for use with a cooperative multitasker using `pause` and non-blocking sockets, although they also work with blocking sockets without multitasker. All `xxx-mt` words are the multitasked counterparts of `xxx` words.

NOTE 03Jan07: xxx-mt words have been refactored to avoid throwing exceptions from the inside words. The internal ior codes have been proagated to the outermost word instead. This has caused a lot of headaches and have added significant complexity to xxx-mt words.

Reusable bits

`Extern: int fcntl(int fd, int cmd, LONG arg);`
See *man fcntl*.

`3 Constant F_GETFL`
File control 'get flags' command. See *man fcntl*.

`4 Constant F_SETFL`
File control 'set flags' command. See *man fcntl*.

`$800 Constant O_NONBLOCK`
Mark a file descriptor as non-blocking. To perform a multitasked I/O, instead of the usual multiplexed I/O using `select()`. See *man open*.

`: fcntl@ \ fd -- n1 ior`
Get the socket file control flags (only handles O_APPEND, O_ASYNC, O_NONBLOCK, O_DIRECT). Returns a throwable *ior*. *n1* is undefined if *ior* <> 0. See *man fcntl*.

`: fcntl! \ n1 fd -- ior`
Set the socket file control flags (only handles O_APPEND, O_ASYNC, O_NONBLOCK, O_DIRECT). *n1* is the new flag value. Returns a throwable *ior*. See *man fcntl*.

`11 Constant EAGAIN \ Try Again`
errno code returned for non-blocking I/O in `accepts sends recvs sendsto recvsfrom`.

`114 Constant EALREADY`
errno code returned for non-blocking socket when performing `connects` when former `connects` request is still in progress.

`115 Constant EINPROGRESS \ Operation now in progress`
errno code returned for non-blocking socket when performing `connects` request that would need to wait for completion.

`: again? \ ior1 -- ior2 flag`

Process *ior1* to filter the `EAGAIN` value. In this case, *flag* is true and *ior2* is zero. Any other *ior1* will produce the same value in *ior2* and a false *flag*.

```
: in-progress?                      \ ior1 -- ior2 flag
```

Process *ior1* to filter the `EINPROGRESS` or `EALREADY` values when using `connects`. In this case, *flag* is true and *ior2* is zero. Any other *ior1* will produce the same value in *ior2* and a false *flag*. Please note that `connects` can return `EAGAIN` but this denotes a different error situation (see *man connect*.)

Main words

```
: -blocking                 \ fd -- ior
```

Configure the socket as non blocking. Returns a throwable *ior*.

```
: +blocking                 \ fd -- ior
```

Configure the socket as blocking. Returns a throwable *ior*.

```
: peeks?                   \ fd -- flag ior
```

Peek into the socket system buffer to check for data available. Returns a true *flag* if one or more bytes are waiting to be read and a throwable *ior*.

```
1 Value #ms-delay
```

poll period in non-blocking `xxx-mt` I/O words below setting this value to 0 increases the CPU load. The default value is enough to produce a negligible load but with a reduced performace.

```
: delayed           \ --
```

By using this word, the caller is delayed `#ms-delay ms`. and allows task switching if multitasking is enabled.

```
: connects-mt                      \ addr1 fd1 -- ior
```

Non-blocking I/O counterpart of `connects`. This word loops over `connects` and uses internally `delayed`.

```
: accepts-mt                      \ addr1 fd1 -- fd2 ior
```

Non-blocking I/O counterpart of `accepts`. This word loops over `accepts` and uses internally `delayed`.

```
: (sends-mt)                \ c-addr1 +n1 fd ior1 -- c-addr2 n2 fd ior2
```

This word is an internal factor of `sends-mt` for non-blocking I/O and should not be used. Tries to send *n1* bytes to socket given by *fd* and loops over until the socket says it would not block. Returns the remaining buffer *c-addr2 n2* to be sent.

```
: (recvs-mt)                \ c-addr1 n1 fd ior1 -- c-addr2 +n2 fd ior2
```

This word is an internal factor of `recvs-mt` for non-blocking I/O and should not be used. Tries to receive *n1* bytes to socket given by *fd* and loops over until the socket says it would not block. Returns the remaining buffer *c-addr2 n2* to be received.

```
: #bytes-io                              \ c-addr1 n1 x1 ior n2 -- n3 ior
```

This word is a factor for some loop epilogs below and should not be used. Returns the number of bytes processed in I/O in *n3* from the total requested *n2* and the remaining amount *n1*. Reorders the result to place *ior* on TOS. *x1* is a don't care cell left in the loop exist that gets dropped in the way.

```
: sends-mt                               \ c-addr1 n1 fd -- n2 ior
```

Non-blocking I/O counterpart of `sends`. This word loops over `sends` and uses internally `delayed`.

```
: recvs-mt                               \ c-addr1 +n1 fd -- +n2 ior
```

Non-blocking I/O counterpart of `recvs`. This word loops over `recvs` and uses internally `delayed`.

```
: halt-loop?   \ n2 fd ior -- n2 ior fd flag
```

This word is an internal factor of `recvs-all-mt`. Do not use. This word factors the loop end detection while preserving stack ordering. Returns true if *ior* < 0 or *n2* = 0, false otherwise.

```
: recvs-all-mt                          \ c-addr1 +n1 fd -- +n2 ior
```

Non-blocking I/O counterpart of `recvs-all`. This word loops over `recvs` and uses internally `delayed`.

1.2.6 Non-blocking connectionless socket I/O

The following words are intended for use with a cooperative multitasker using `pause` and non-blocking sockets, although they also work with blocking sockets without multitasker. All `xxx-mt` words are the multitasked counterparts of `xxx` words.

Reusable bits

```
: lover                                 \ x1 x2 x3 -- x1 x2 x3 x1
```

"long" `over`, although the name actually suggest something different :-)

```
: halt2-loop?                  \ n2 addr2 fd ior -- n2 addr2 ior fd flag
```

This word is an internal factor of `recvs-all-mt` and `recvsfrom-all-mt`. Do not use. This word factors the loop end detection while preserving stack ordering. Returns true if *ior* < 0 or *n2* = 0, false otherwise.

```
: (sendsto-mt)  \ c-addr1 +n1 addr2 fd ior -- c-addr3 +n2 addr2 fd ior
```

This word is an internal factor of `sendsto-mt` for non-blocking I/O and should not be used. Tries to send *n1* bytes to socket *fd* and loops over and over again until socket says it would not block. Returns the remaining buffer *c-addr3 n2* to be sent. *addr2* denotes the destination IP `/endpoint`. *ior* is propagated from `sendsto`.

```
: (recvsfrom-mt)   \ c-addr1 +n1 addr2 fd ior -- c-addr1 +n2 addr2 fd ior
```

This word is an internal factor of `recvsfrom-mt` for non-blocking I/O and should not be used. Tries to receive +*n1* bytes into buffer *c-addr1* from socket *fd* and loops over and over again until socket says it would not block. Returns the remaining buffer *c-addr3 n2* to be received. *addr2* denotes the destination IP `/endpoint`. *ior* is propagated from `recvsfrom`.

Main words

```
: sendsto-mt                            \ c-addr1 +n1 addr2 fd -- +n2 ior
```

Non-blocking I/O counterpart of `sendsto`. This word loops over `sendsto` and uses internally `delayed`.

```
: recvsfrom-mt                          \ c-addr1 +n1 addr2 fd -- +n2 ior
```

Non-blocking I/O counterpart of `recvsfrom`. This word loops over `recvsfrom` and uses internally `delayed`.

```
: recvsfrom-all-mt                      \ c-addr1 +n1 addr2 fd -- +n2 ior
```

Non-blocking I/O counterpart of `recvsfrom-all`. This word loops over `recvsfrom-all` and uses internally `delayed`.

Warning: Unpredictable results are obtained if several simultaneous clients send their data to the same connectionless socket in the receiving side. Data will be mixed in the same buffer and a wrong byte count will be reported.

1.2.7 Multiplexed I/O

An alternative to multitasked I/O when blocking sockets are used. The *poll()* system call is easier to implement than *select()*.

Reusable bits

Internal documentation.

`AliasedExtern: ux-poll int poll(void * ufds, unsigned int nfds, int timeout);`
See *man poll*.

Main words

`$0001 Constant POLLIN`
On `pollfd.events`: notify when data ready. On `pollfd.revents`: data ready to read.

`$0004 Constant POLLOUT`
On `pollfd.events`: notify when write will not block. On `pollfd.revents`: write will not block.

`$0008 Constant POLLERR`
On `pollfd.revents`: Error condition.

`$0010 Constant POLLHUP`
On `pollfd.revents`: Hangup. Remote socket closed.

`$0020 Constant POLLNVAL`
On `pollfd.revents`: Invalid request. File descriptor is not open.

```
struct /pollfd                          \ -- len
```
Mimics the Linux *struct pollfd*. See *man poll*.
```
   1 cells field pollfd.fd              \ file descriptor
   2 chars field pollfd.events          \ requested events
   2 chars field pollfd.revents         \ returned events
end-struct


: polls                                   \ addr1 n1 n2 -- n3 ior
```
Poll the system for I/O events. An array of `/pollfd` structures is passed starting at *addr1*. *n1* is the array length in structure units (not bytes). *n2* is a timeout in milliseconds. *n2* `<` 0 indicates an infinite timeout. *n3* is the number of structures having `pollfd.revents <>` 0 If *n3* = 0, a timeout has occured. If *n3* `<` 0, an internal error has occured and *ior* contains a throwable *errno* code. See *man poll*.

NOTE: Tests made shows that when a remote side closes the socket poll returns with an `POLLIN` event instead of `POLLHUP`. Performing a `recvs` on that socket returns a zero length.

2 GENIO Socket Device

This Generic IO Device operates on a TCP socket for input and output. Flags specify whether to create a client or server socket and also the blocking/non-blocking mode.

2.1 Socket Device Creation

In order to create a named socket device on the dictionary, use the `SOCKDEV:` definition given later. These devices can be created in the heap using `allocate`, and must be initialized using `initSockDev`.

Under GENIO, sockets are opened by using either `open-gen` or `open-gio`. The following flags control options at open time:

- `SOCKDEV_NONBLOCK`, to open the socket in non-blocking mode.
- `SOCKDEV_SVR`, to open the socket in server mode using `accepts`. Needs to pass the parent socket *fd*.
- `SOCKDEV_ECHO` to echo characters to remote peer socket when using ANS `ACCEPT` word on this socket.

When using this driver to program server sockets, it is up to the developer to set up and configure the parent socket with `socket`, `binds` and `listens` or `-blocking`. An example:

```
  -1 Value master
  7624 EndPoint: local any-ip

: new-master   ( -- )
   TCP socket throw to master
   local master binds throw
   master listens throw ;
```

2.2 Exceptions

When examining the stack signature of GENIO drivers, there were no place to report I/O errors. Two options were available: either ignore them or to throw exceptions. I have chosen the latter strategy as a way to detect bugs and unexpected situations.

The following **xxx-gio** words may throw exceptions when using the underlying `sockets` vocabulary:

`open-gio`, `close-gio`, `read-gio`, `readex-gio`, `write-gio`, `key?-gio`, `key-gio`, `accept-gio`, `emit-gio`, `type-gio`, `cr-gio`, `lf-gio`, `ff-gio` and `bs-gio`.

The following **xxx-gio** words throw `abort" operation not supported"` exceptions:

`ekey-gio`, `ekey?-gio`, `bell-gio`, `setpos-gio`, `getpos-gio` and `ioctl-gio`,

The following **xxx-gio** words do nothing:

`flushOP-gio`, `init-gio`, `term-gio` and `config-gio`.

2.3 Glossary

`MODULE GENIO-SOCK`
GENIO Module name for socket I/O.

`struct /SockDev \ -- len ;`
The socket SID structure, passed around as a handle by all **xxx-gio** words.

```
  gen-sid +                             \ reuse field names of GEN-SID
   1 cells field sd.flags               \ Mode flags (see below).
end-struct
```

`1 Constant SOCKDEV_NONBLOCK`
Non-Blocking bit for *fam* flag in `open-gio`. Also stored in `sd.flags`.

`2 Constant SOCKDEV_SVR`
Server bit for *fam* flag in `open-gio`. Also stored in `sd.flags`.

`4 Constant SOCKDEV_ECHO`
Echo bit in *fam* flag in `open-gio`. Makes `ACCEPT` echo character back to the sender socket. Option available for server sockets only. Also stored in `sd.flags`.

2.3.1 Blocking/Non-Blocking socket API vectors

Handling blocking or non-blocking socket connections is deferred until open time. It makes much easier life to code several entry points with a slight overhead.

`Defer socket-connect                   \ addr fd -- ior`
Vector for `connects` or `connects-mt`.

`Defer socket-accept                    \ addr1 fd1 -- fd2 ior`
Vector for `accepts` or `acceptss-mt`.

`Defer socket-send                      \ c-addr1 +n1 fd -- +n2 ior`
Vector for `sends` or `sends-mt`.

`Defer socket-recv                      \ c-addr1 +n1 fd -- +n2 ior`
Vector for `recvs` or `recvs-mt`.

`Defer socket-recv-all                  \ c-addr1 +n1 fd -- +n2 ior`
Vector for `recvs-all` or `recvs-all-mt`.

`Defer sd-key?                          \ sid -- flag`
Vector for `(sd-key?)` or `(sd-key?-mt)`.

2.3.2 Common, low level factors

`: (sd-key?)          \ sid -- flag`
Test for any character received by configuirng the socket as non-blocking mode, do a peek and check for EAGAIN errno. Then, the socket is configured as blocking again.

`: (sd-key?-mt)         \ sid -- flag`
More efficient operation when socket is already open in non-blocking mode.

`: NotSockDev \ --`
Issue a SockDev error message.

`: (sd-create) \ addr len sid -- addr fd sid 0 | sid ior`
Creates the socket. *len* is unusued but *addr* will be used later,

`: (sd-connect) \ addr fd sid -- fd sid 0 | sid ior`
Connects or closes the socket if not succesfull. *addr* is a `/sockaddr_in`, `/enpoint` structure or `NULL`.

`: (sd-accepts) \ addr len sid -- fd sid 0 | sid ior`
Creates a new child socket from parent socket. *addr* is an `/enpoint` structure or `NULL`. *len* is the parent spcket descriptor

`: (sd-gen!) \ fd sid -- sid 0`
Sets the internal handle to be the file descriptor returned by socket creation words.

`: open-client \ addr len sid -- sid ior`
Open socket device and connect to remote `/sockaddr_in` or `/endpoint` *addr* for client sockets. *len* is unusued. Leave it to -1.

`: open-server \ addr len sid -- sid ior`
Open socket device for children server sockets. *addr* is an `/enpoint` structure or `NULL`. *len* is the parent socket descriptor returned beforehand by `socket`.

2.3.3 Socket driver Entry points

`: sd-close \ sid -- ior`
Closes the socket and initializes the gen-handle back to invalid state.

`: sd-read \ addr len sid -- ior`
Read **all** the bytes up to *len* from the socket device.

`: sd-readex \ addr len sid -- #read ior`
Read **all** the bytes up to *len* from the socket device. May return less bytes than requested.

`: sd-write \ addr len sid -- ior`

`: sd-key \ sid -- char`

`: sd-ekey \ sid -- echar`
Not supported.

`: sd-ekey? \ sid -- flag`
Not supported.

`: sd-emit \ char sid --`

`: sd-emit? \ sid -- flag`
ALways true. does not check for full Tx buffer.

`: sd-type \ addr len sid --`

`: sd-cr \ sid --`
Send CR+LF characters

`: sd-lf \ sid --`
Linefeed.

`: sd-ff \ sid -- ; page/cls on display devices`

`: sd-bs \ sid -- ; destructive on display devices`

`: sd-bell \ sid -- ; audible beeper`

Not supported.

```
: sd-setpos        \ x y mode sid -- ior
```
Not supported.

```
: sd-getpos        \ mode sid -- x y ior
```
Not supported.

```
: sd-ioctl         \ addr len fn sid -- ior
```
Not supported. All is done at open time.

```
: sd-flushOP       \ sid -- ior
```
Not supported.

```
: sd-init          \ addr len sid -- ior
```
Not supported.

```
: sd-term          \ sid -- ior
```
Not supported.

```
: sd-config        \ sid -- ior ; produces a dialog
```
Not supported.

Support for ACCEPT

For the sd-accept entry points, we have deconstructed its loop into pieces.

```
: wait-key                    \ sid -- sid
```
Wait until next character arrives.

```
: /loop                            \ addr1 len sid - cnt sid addr2 addr3
```
Initializes accept loop, converting (start, count) pair into (starting , ending) addresses *addr2,addr3*.

```
: cr?                                          \ c -- c flag
```
Test for carriage return. Returns true id `CR` is detected.

```
: ?echo  \ cnt sid c -- cnt sid c
```
Conditionally echoes back the character, depending on the `SOCKDEV_ECHO` flag.

```
: ?backspace   \ cnt1 sid c -- cnt2 sid index
```
Handle possible backspace character *c* by going back into the array and decreasing by 1 the count and loop index. Normal operation increases by 1 the count and loop index.

```
: discard-lf                         \ cnt sid c -- cnt sid
```
c is a *CR*, therefore, read *LF* and discard it.

```
: sd-accept                          \ addr len sid -- #read
```
Accept chars from socket until *cr* is detected. Returns number of characters read.

2.3.4 Open and vector table

```
: +block-all
```
Set vectors for blocking mode.

```
: -block-all
```
Set vectors for blocking mode.

```
: sd-open       \ addr len fam sid -- sid ior
```

addr is an `/endpoint` structure. *fam* containst the two mode bits defined above. *len* is -1 for client socke or else the parent *fd*. when opening a child server socket. *sid* is the */SockDev* structure.

```
create sd-vectors                          \ -- ; Exec: -- addr
```

Table of GENIO execution tokens.

2.3.5 Device Creation

```
: initSockDev   \ sid --
```

Initialise the sid for a socket device. Use it to initialize the structure has been allocated from the heap.

```
: SockDev:      \ "name" -- ; Exec: -- sid
```

Create a Socket based Generic IO device in the dictionary.

3 Examples

Some straightforward examples.

3.1 Very simple TCP client example.

Shows usage of basic words.

```
\ ================================================================================

include ../src/sockets.fs

-1 Value mysock

s" localhost" 7624 EndPoint: remote known-ip

: (client)
   TCP socket throw to mysock
   remote mysock connects throw
   cr ." Connected to server" cr
   s" Hello TCP World!" mysock sends throw .
   mysock closes throw
;

: client
   ['] (client) catch
   dup errno-base < if .errno-excp else throw then ;

client
```

3.2 Very simple TCP server example.

Shows non-blocking use of sockets. Child sockets are blocking even if the master is not. `recv-mt` returns when something is available, even if we asked to read `BufSize` bytes.

```
\ ================================================================================

include ../src/sockets.fs

-1 Value master
-1 Value slave

7624 EndPoint: local any-ip
EndPoint: remote unknown-ip

256 Constant BufSize

BufSize Buffer: RxBuffer
```

```
: (server)
   TCP socket throw to master
   local master binds throw
   master listens throw

   remote master accepts-mt throw to slave
   slave -blocking throw
   cr ." A client has connected" cr
   RxBuffer BufSize slave recvs-mt throw
   RxBuffer swap type cr
;

: server
   ['] (server) catch
   dup errno-base < if .errno-excp else throw then ;

server
```

3.3 Multitasked TCP client example.

This example shows a multitasked client using the VFX preemptive multitasker. Upon starting the client, the Forth console is still available and active tasks are shown using multitasker's `.tasks` word.

I/O through sockets is done using a GENIO `/SockDev` driver, so words like `cr`, `type`, or `accept` are available. Note that this example does not use event passing nor messages, so there is no need to call `pause` and blocking sockets are used. Also note that wihin the task, text can be printed in the console surronded by a `[io, io]` context.

```
\ ========================================================================

include /usr/share/doc/VfxForth/Lib/Lin32/MultiLin32.fth
include ../src/sockets.fs
include ../src/Genio/sockets.fs

0 Constant myflags

s" localhost" 7624 EndPoint: remote known-ip \ the "remote" end point

SockDev: mysock

\ ---------------------------------------


: default-io    \ -- ; set up default I/O
   xconsole setIO ;

: suicide
   pause termThread 0 pthread_exit ;
```

```
: run-task \ xt --
   \ wrapper around real task code given by xt
   catch default-io dup errno-base <= if dup .errno-excp then suicide ;

\ -------------------------------------------------------

: .connect
   cr ."  Connected to server " cr ;

: .selfmsg             \ --
   ." task " self ." sent a message" cr ;

: task1-init
   mysock SetIO
   remote -1 myflags open-gen throw drop
   [io default-io .connect io]
;

: (task1-action)
   task1-init
   begin
      s" Hello TCP World!" type cr      \ send to remote
      [io default-io .selfmsg io]
      3000 ms
   again
;

: task1-action
   ['] (task1-action) run-task ;

\ -------------------------------------------------------


task task1

' task1-action task1 initiate
```

3.4 Simple TCP server example.

This example shows a multitasked server using the VFX preemptive multitasker. A master server thread is created to listen to incoming connections and spawns dynamically created slave threads. Slave threads get the text from a client and print it in the Forth console.

I/O through sockets is done using a GENIO `/SockDev` driver, so words like `cr`, `type`, or `accept` are available. Note that this example does not use event passing nor messages, so there is no need to call `pause` and blocking sockets are used. When remote clients close connections, slave tasks print exceptions and die.

Upon starting the server, the Forth console is still available and active tasks are shown using multitasker's `.tasks` word.

```
\ ========================================================================

include /usr/share/doc/VfxForth/Lib/Lin32/MultiLin32.fth
include ../src/sockets.fs
include ../src/Genio/sockets.fs


SOCKDEV_SVR Constant myflags

-1 Value master                         \ master listening socket fd

7624 EndPoint: local any-ip
EndPoint: remote unknown-ip

256 Constant BufSize
BufSize +User RxBuffer                  \ slave tasks reception buffers

\ ------------------------------------------------------------

variable  sock-sid                      \ shared between tasks
semaphore sock-sem                      \ to syncronize access

: sock-sid@          \ -- sid
   sock-sem request sock-sid @ sock-sem signal ;

: sock-sid!          \ sid --
   sock-sem request sock-sid ! sock-sem signal ;

\ ------------------------------------------------------------
\ COMMON WORDS
\ ------------------------------------------------------------

: default-io          \ The default Forth console I/O
   xconsole SetIO ;

: suicide             \ The way a task terminates itself cleanly
   pause termThread 0 pthread_exit ;

: is-action \ xt -- ; wrapper around real action code given by xt
   catch
   default-io dup errno-base <= if .errno-excp else drop then suicide ;

\ ------------------------------------------------------------
\ SLAVE TASKS
\ ------------------------------------------------------------

: new-task  \  -- tcb ; creates an unnamed, dynamically allocated task
   /TCB protAlloc dup initTCB ;


: .selfmsg             \ #nread --
   dup if ." task " self . RxBuffer swap type cr else ." Read 0!" cr then ;
```

```
: (slave-task)         \ slave task action
   sock-sid@ setIO
   begin
      RxBuffer BufSize accept
      [io  default-io .selfmsg  io]
   again
;

: slave-task          \ slave task wrapper
   ['] (slave-task) is-action ;

\ -------------------------------------------------------------
\ MASTER TASK
\ -------------------------------------------------------------

: new-master            \ creates the master socket, in blocking mode
   TCP socket throw to master
   local master binds throw
   master listens throw
;

: new-sockdev          \ -- sid ; dinamically allocated /sockdev
   /SockDev protAlloc dup initSockDev ;

: .connect
   cr ."  A client has connected " cr ;

: (server)      \ Main server action
   new-master
   begin
      new-sockdev dup >r sock-sid!
      remote master myflags r> open-gio throw
      .connect
      ['] slave-task new-task initiate
   again
;

: server
   ['] (server) is-action ;

: myshutdown
   sock-sem ShutSem ;

\ Main task creation and startup. Leaves a fully operating Forth console

sock-sem InitSem
task server-task
' server server-task initiate
' myshutdown AtExit
```