Network programming information

Introduction

The Master 10/100 ethernet module includes onboard software that runs a TCP/IP network stack. This is used by the supplied filing system, LANManFS, to share files with other computers on the network using the SMB protocol.

However, SMB is just one protocol of the many hundreds that can be sent over the network, and this document describes how access to the network can be included in other programs through use of a simple OSWord application programmer's interface (API).

The OSWord is closely modelled on the 'Berkeley Sockets API' which is the most commonly used means of writing network software, which should facilitate conversion of existing programs to work with the ethernet module. A detailed explanation of the operation of Berkeley Sockets is not provided here, it is assumed that the reader is familiar with their use.

Additional OSWord reason codes are provided to give access to the DNS portion of the network software to allow resolution of domain names to computer addresses, though in principle this could also be reimplemented directly using the API if desired.

Conventions used in this document

The following typographical conventions are used throughout this guide:

Hexadecimal numbers are prefixed with ampersand.

Decimal numbers have no prefix.

Binary numbers may be denoted with a leading percent and given in decending bit significant order (ie.for an eight bit number they will be written in the order %76543210).

Copyright

History

V0.10 First draft

V0.15 Finalised available calls and refreshed the BASIC function examples

V0.16 Added a section on the implemented MSG_ flags for Socket_Send and Socket_Recv

Calling the network API

OSWord

All of the functions available from the network interface are accessed by called OSWord 192 (&C0).

During processing interrupts will be enabled, therefore this reason code must not be called from an interrupt context and is not reentrant.

Use of the OSWord does not require the network filing system, LANManFS, to be the currently selected filing system at the time - the OSWord is independant of this as it is dealt with by the LANManager software.

Subreasons

To allow for many functions to be performed but without using up lots of OSWord numbers a one byte subreason code is included, allowing for up to 256 different pieces of functionality.

The subreason codes are further split into groups of 64 each, with detailed descriptions later in this document

&00-3F = Socket operations
&40-7F = Resolver operations
&80-BF = Reserved for future use
&C0-FF = Reserved for future use

unused or reserved subreason codes will return an error number at YX+3.

General format

On entry A = 192
X = low byte of address of a control block
Y = high byte of address of a control block
YX+0 = OSWord send block length (depends on subreason code)
YX+1 = OSWord return block length (depends on subreason code)
YX+2 = subreason code
YX+3 = must be zero
YX+4... = as required by the subreason code

On exit A = preserved
X = preserved
Y = preserved
YX+0 = preserved
YX+1 = preserved
YX+2 = set to zero
YX+3 = zero for OK, otherwise reports an error number
YX+4... = updated as determined by the subreason code

Note the first two values in the parameter block are required to instruct the Tube software how big the block is when the OSWord is issued from a coprocessor. Due to a limitation in the host side of the Tube software these values cannot exceed 128 bytes in either direction.

Presence

As LANManager zeros the reason code during the call it is possible to detect its presence by doing a Socket_Close operation with a socket number of -1, which will never be a valid socket number, and look for the YX+2 value changing.

Flags to Socket_Send and Socket_Recv

These two functions take optional flags which are a bit field of message options. The following three flags are the only flags supported by the TCP/IP network stack, and are a subset of those available in a full implementation of the Berkeley Sockets.

MSG_PEEK = 1 (flag bit 0)

The MSG_PEEK flag is used to receive data but not consume it from the network stack, this can be useful to inspect the first few bytes of a message before deciding how to process the remainder with a second call to the Socket_Recv function.

MSG_DONTWAIT = 8 (flag bit 3)

The MSG_DONTWAIT will use non blocking methods for Socket_Recv and return immediately, regardless of how the socket is configured.

MSG_MORE = 16 (flag bit 4)

The MSG_MORE flag hints to the Socket_Send function that this is not the last data to be sent. In practice this affects the state of the PSH flag in the TCP transaction, so that data can be more efficiently buffered.

Socket operations

Socket_Creat (&00)

Entry YX+4 = communications domain (2 for PF_INET)
YX+8 = socket type (1=stream; 2=datagram; 3=raw)
YX+12 = protocol, or zero for a default for the socket type chosen
Exit YX+4 = -1 if fails, otherwise the socket number created

C prototype `int socket(int domain, int type, int protocol);`
Description Create a new socket for subsequent use.

This function sets up a new socket and returns a handle for its future use. Only a limited number of sockets can be opened simultaneously.

Socket_Bind (&01)

Entry YX+4 = socket
YX+8 = pointer to socket address to bind to
sa+0 = size of socket address (usually 16)
sa+1 = address family (2 for AF_INET)
sa+2 = port number
sa+4 = IPv4 address
sa+8 = zero
sa+12 = zero
YX+12 = size of socket address (usually 16)
Exit YX+4 = -1 if fails

C prototype `int bind(int socket, struct sockaddr *addr, int size);`
Description Bind a socket to a specific local address.

Note that the size of the socket address structure is used twice, once in the structure itself, and also as the third parameter at YX+12.

Socket_Listen (&02)

Entry YX+4 = socket
YX+8 = backlog of unaccepted connections to allow before rejecting
Exit YX+4 = -1 if fails

C prototype `int listen(int socket, int backlog);`
Description Switch a socket into listening for incoming connection attempts.

Only sockets opened and configured to tbe stream based sockets can be set to listen, datagram and raw sockets are connectionless and cannot be set to listen.

Socket_Accept (&03)

Entry YX+4 = socket
YX+8 = pointer to accepted address to fill
sa+0 = size of socket address (usually 16)
sa+1 = address family (2 for AF_INET)
sa+2 = port number
sa+4 = IPv4 address
sa+8 = zero
sa+12 = zero
YX+12 = pointer to an integer describing the size of socket address (usually 16)
Exit YX+4 = -1 if fails

C prototype `int accept(int socket, struct sockaddr *addr, int *size);`
Description Accept an incoming connection on an existing socket.
If there are no pending incoming connections this call will block until there is one. On accepting, the address details of the remote computer will be filled in at the block pointed to by YX+8.

Socket_Connect (&04)

Entry YX+4 = socket
YX+8 = pointer to socket address to connect to
sa+0 = size of socket address (usually 16)
sa+1 = address family (2 for AF_INET)
sa+2 = port number
sa+4 = IPv4 address
sa+8 = zero
sa+12 = zero
YX+12 = size of socket address (usually 16)
Exit YX+4 = -1 if fails

C prototype `int connect(int socket, struct sockaddr *addr, int size);`
Description Connect a socket to a specific remote address.
For raw and datagram style sockets this just notes the socket address for future use, for stream style sockets the remote computer is contacted to make the connection.

Socket_Recv (&05)

Entry YX+4 = socket
YX+8 = pointer to data buffer to receive into
YX+12 = buffer size
YX+16 = flags (usually 0)
Exit YX+4 = -1 if fails, otherwise number of bytes received

C prototype `int recv(int socket, char *msg, int len, int flags);`
Description Read data from the given socket.
This function attempts to read data or waits until some is ready. It is possible that zero bytes are returned, probably indicating that the remote computer has disconnected.

Socket_Recvfrom (&06)
Not currently supported, returns an error.

Socket_Recvmsg (&07)
Not supported, returns an error.

Socket_Send (&08)
Entry YX+4 = socket
YX+8 = pointer to data buffer to send
YX+12 = buffer size
YX+16 = flags (usually 0)
Exit YX+4 = -1 if fails, otherwise number of bytes sent

C prototype `int send(int socket, char *msg, int len, int flags);`
Description Send out data on the given socket.
For raw and datagram style sockets the message length must fit within one packet otherwise the request will be rejected, for stream style sockets as much as the message as possible will be queued and sent subject to available memory.

Socket_Sendto (&09)
Not currently supported, returns an error.

Socket_Sendmsg (&0A)
Not supported, returns an error.

Socket_Shutdown (&0B)
Entry YX+4 = socket
YX+8 = direction to shut (0=receive side; 1=transmit side; 2=both sides)
Exit YX+4 = -1 if fails

C prototype `int shutdown(int socket, int how);`
Description Shutdown part of a socket.
This allows a socket to be partially shut where the TCP/IP stack supports this. Caution should be taken as this does not actually close the socket, so does not free up any of the resources associated with the socket - see details of Close for how to do this.

Socket_Setsockopt (&0C)
Not currently supported, returns an error.

Socket_Getsockopt (&0D)
Not currently supported, returns an error.

Socket_Getpeername (&0E)
Not currently supported, returns an error.

Socket_Getsockname (&0F)
Not currently supported, returns an error.

Socket_Close (&10)

Entry YX+4 = socket
Exit YX+4 = -1 if fails

C prototype `int close(int socket);`
Description Close a socket.

As there are fixed number of sockets available it is important to remember to close sockets once any transactions are complete.

Socket_Select (&11)

Not supported, returns an error.

Socket_Ioctl (&12)

Not currently supported, returns an error.

Socket_Read (&13)

Not supported, returns an error.

Socket_Write (&14)

Not supported, returns an error.

Socket_Stat (&15)

Not supported, returns an error.

Socket_Readv (&16)

Not supported, returns an error.

Socket_Writev (&17)

Not supported, returns an error.

Resolver operations

Resolver_GetHostByName (&40)

Entry YX+8 = pointer to name to lookup

Exit YX+4 = pointer to name looked up
YX+8 = pointer to a null terminated list of pointers to aliases
YX+12 = IP address type returned (2 for AF_INET)
YX+16 = length of this address type (4 for IPv4)
YX+20 = pointer to a null terminated list of pointers to IP address(es)

C prototype `hostent *GetHostByName(char *name);`

Description Resolves a host name to a network address.

A control terminated name will be passed to the DNS resolver software built into the network module. This call will then wait for a result, and only return when a match is found or timeout occurs. The returned block at YX+8 onwards is a 'hostent' structure containing a list of IP addresses amongst other information, the lists will remain valid until the next resolver request.

Resolver_GetHost (&41)

Entry YX+4 = pointer to name to lookup

Exit YX+4 = pointer to name looked up
YX+8 = pointer to a null terminated list of pointers to aliases
YX+12 = IP address type returned (2 for AF_INET)
YX+16 = length of this address type (4 for IPv4)
YX+20 = pointer to a null terminated list of pointers to IP address(es)

C prototype `hostent *GetHost(char *name);`

Description Resolves a host name to a network address.

This is very similar to GetHostByName except that it returns immediately. If the name is already in the DNS cache the result will be filled in and YX+3 is zero, otherwise a request is issued and a 'resolver busy' error returned. Further calls to GetHost will update YX+3 until either a timeout occurs or the name is found.

This allows the request to be sent and the foreground program continue operating, compare this with GetHostByName which blocks until the name has been found.

Resolver_GetCache (&42)

Recognised, but does nothing, does not return an error.

Resolver_CacheControl (&43)

Recognised, but does nothing, does not return an error.

Example library functions

The following listing provides BASIC functions with the same names and parameters as the corresponding function in most libraries for the 'C' programming language as a convenience for use.

```
REM Network functions
REM (C)2010 SPROW
:
DEFFNgethost(name$)
wordblk?0=8:REM Parameters in
wordblk?1=24:REM Parameters out
wordblk?2=&41:REM Resolver_GetHost
wordblk?3=0:REM No error on entry
wordblk!4=nameblk
$nameblk=name$
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
IFwordblk?3<>0 THEN=0
=wordblk+4:REM Address not value
:
DEFFNcreat(pf%,type%,prot%)
wordblk?0=16:REM Parameters in
wordblk?1=8:REM Parameters out
wordblk?2=&00:REM Socket_Creat
wordblk?3=0:REM No error on entry
wordblk!4=pf%
wordblk!8=type%
wordblk!12=prot%
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
IFwordblk?3<>0 THEN=-1
=wordblk!4
:
DEFFNbind(handle%,addr%,addrlen%)
wordblk?0=16:REM Parameters in
wordblk?1=8:REM Parameters out
wordblk?2=&01:REM Socket_Bind
wordblk?3=0:REM No error on entry
wordblk!4=handle%
wordblk!8=addr%
wordblk!12=addrlen%
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
IFwordblk?3<>0 THEN=-1
=wordblk!4
:
DEFFNlisten(handle%,count%)
wordblk?0=12:REM Parameters in
wordblk?1=8:REM Parameters out
wordblk?2=&02:REM Socket_Listen
wordblk?3=0:REM No error on entry
wordblk!4=handle%
wordblk!8=count%
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
IFwordblk?3<>0 THEN=-1
=wordblk!4
:
DEFFNaccept(handle%,addr%,addrlenblk%)
```

```
wordblk?0=16:REM Parameters in
wordblk?1=8:REM Parameters out
wordblk?2=&03:REM Socket_Accept
wordblk?3=0:REM No error on entry
wordblk!4=handle%
wordblk!8=addr%
wordblk!12=addrlenblk%
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
IFwordblk?3<>0 THEN=-1
=wordblk!4
:
DEFFNconnect(handle%,addr%,addrlen%)
wordblk?0=16:REM Parameters in
wordblk?1=8:REM Parameters out
wordblk?2=&04:REM Socket_Connect
wordblk?3=0:REM No error on entry
wordblk!4=handle%
wordblk!8=addr%
wordblk!12=addrlen%
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
IFwordblk?3<>0 THEN=-1
=wordblk!4
:
DEFFNrecv(handle%,data%,len%,opts%)
wordblk?0=20:REM Parameters in
wordblk?1=8:REM Parameters out
wordblk?2=&05:REM Socket_Recv
wordblk?3=0:REM No error on entry
wordblk!4=handle%
wordblk!8=data%
wordblk!12=len%
wordblk!16=opts%
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
IFwordblk?3<>0 THEN=-1
=wordblk!4
:
DEFFNsend(handle%,data%,len%,opts%)
wordblk?0=20:REM Parameters in
wordblk?1=8:REM Parameters out
wordblk?2=&08:REM Socket_Send
wordblk?3=0:REM No error on entry
wordblk!4=handle%
wordblk!8=data%
wordblk!12=len%
wordblk!16=opts%
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
IFwordblk?3<>0 THEN=-1
=wordblk!4
:
DEFPROCshutdown(handle%,type%)
wordblk?0=12:REM Parameters in
wordblk?1=4:REM Parameters out
wordblk?2=&0B:REM Socket_Shutdown
wordblk?3=0:REM No error on entry
wordblk!4=handle%
```

```
wordblk!8=type%
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
ENDPROC
:
DEFPROCclose(handle%)
wordblk?0=8:REM Parameters in
wordblk?1=4:REM Parameters out
wordblk?2=&10:REM Socket_Close
wordblk?3=0:REM No error on entry
wordblk!4=handle%
A%=192:X%=wordblk:Y%=wordblk DIV256:CALL&FFF1
ENDPROC
```