The TCP/IP UNAPI specification

Version 1.0. September 1, 2010
By Nestor Soriano, konamiman (at) konamiman (dot) com

1. Introduction

MSX-UNAPI is a standard procedure for defining, discovering and using new APIs (Application Program Interfaces) for MSX computers. The MSX-UNAPI specification is described in a separate document.

This document describes an UNAPI compliant API intended for software that implements a TCP/IP stack, that is, software that provides networking capabilities by using the IP family of protocols. The functionality provided by this API is focused mainly on communicating with other computers by using the TCP and UDP protocols, but there are also some additional routines that allow for example performing domain name resolution by querying DNS servers.

The intended client software applications for this API are networking related applications such as Telnet, FTP or e-mail clients. Any software willing to transmit or receive data by using the TCP or UDP protocols can make use of implementations of this specification.

This document is targeted at both developers of TCP/IP UNAPI implementations, and developers of client applications for these implementations.

1.1. Design goals

There were two main goals when designing this specification:

- *Simplicity*. This specification's intent is to provide the simplest API that will allow to develop useful networking applications for MSX computers.
- *Modularity*. Most of the capabilities provided by this API are optional, and there are means to get information about which capabilities are supported by a given implementation. This allows to create from minimal to complete implementations, as well as providing a clean way to develop an implementation in an incremental way.

1.2. Specification scope

In order to achieve the simplicity goal, this specification deals with the most basic capabilities required in order to develop client networking applications. These capabilities are:

- Communicating via TCP connections.
- Communicating via UDP datagrams.
- Converting domain names to IP addresses by querying DNS servers.
- Sending ICMP echo request messages (PINGs) and retrieving their answers.
- Communicating via raw IP datagrams.

Other capabilities that are usually part of TCP/IP stacks, but which are not necessary in order to develop most client applications, are not covered by this specification. In particular, this specification does NOT deal with:

- The link layer protocol / physical transport medium used (serial cable, modem, Ethernet network, wireless network, joystick cable, or whatever medium is used).
- The procedure for establishing and closing a network connection, if applicable.
- The configuration parameters of the implementation, other than basic parameters such as the IP addresses to use and wether to set these IP addresses manually or automatically (for example setting the user name and password for establishing a network connection is not covered).
- Sending and receiving ICMP messages, other than echo messages (PINGs).
- Configuring routing tables.
- Converting IP addresses into hardware addresses using ARP or an equivalent protocol, when applicable.

Note that this does not impose any restriction on implementations for actually providing these features. For example, an implementation may deal internally with ICMP messages; an Ethernet based implementation will most probably use ARP to convert IP addresses to hardware addreses; and it is expected that implementations will be delivered with the appropriate advanced configuration tools, when needed. The key concept is that these capabilities may exist but are not covered by this specification.

1.3. Modularity

In order to achieve the modularity goal, most of the capabilities defined in this specification are optional; implementations may choose to implement the full specification, or only a subset of it. Of course, the less capabilities are implemented by a given implementation, the greater are the chances that a particular client application will not work with it, especially the most basic capabilities. For example, an implementation not providing any support for TCP connections will not be very useful; on the other hand, an implementation that supports TCP and UDP but does not support raw IP connections will probably still work fine with most client applications.

The modularity feature is implemented in two ways:

1. There is a routine (TCPIP_GET_CAPAB, see Section 4.1.2) that returns a "capabilities vector". This vector holds one bit for each capability that is defined in this specification; when the bit is set it means that the capability is implemented.
2. All routines defined in this specification return an error code in register A. One of these codes is "Not implemented", and is returned whenever a routine related to an unimplemented capability is invoked (certain routines return this error or not depending on the input parameters).

For more details on which routines can be invoked (and how) depending on the supported capabilities, see the routines descriptions themselves.

2. API identifier and version

The API identifier for the specification described in this document is: "TCP/IP" (without the quotes). Remember that per the UNAPI specification, API identifiers are case-insensitive.

The TCP/IP API version described in this document is 1.0. This is the API specification version that the mandatory implementation information routine must return in DE (see

the UNAPI_GET_INFO routine in Section 4.1.1).

3. Error codes

All routines defined in this specification return an error code in register A. This section lists all the possible error codes; the numeric value, a mnemonic and a short description is provided for each one. Each routine description has an errors section which explains with detail which error codes can be returned for that routine, and for which reasons is each one returned.

Code	Mnemonic	Description
0	ERR_OK	Operation completed successfully
1	ERR_NOT_IMP	Capability not implemented
2	ERR_NO_NETWORK	No network connection available
3	ERR_NO_DATA	No incoming data available
4	ERR_INV_PARAM	Invalid input parameter
5	ERR_QUERY_EXISTS	Another query is already in progress
6	ERR_INV_IP	Invalid IP address
7	ERR_NO_DNS	No DNS servers are configured
8	ERR_DNS	Error returned by DNS server
9	ERR_NO_FREE_CONN	No free connections available
10	ERR_CONN_EXISTS	Connection already exists
11	ERR_NO_CONN	Connection does not exists
12	ERR_CONN_STATE	Invalid connection state
13	ERR_BUFFER	Insufficient output buffer space
14	ERR_LARGE_DGRAM	Datagram is too large
15	ERR_INV_OPER	Invalid operation

4. API routines

This version of the TCP/IP API consists of 30 mandatory routines, which are described below. API implementations may define their own additional implementation-specific routines, as described in the MSX-UNAPI specification.

Routines are grouped in subsections by related behavior. Useful information concerning all the routines on a given subsection is provided at the beginning of each subsection.

Some routines exchange data with the client application by using a memory buffer. Per the UNAPI specification, implementations may not allow the destination address to be a page 1 address (in the range 4000h-7FFFh). Client software should not use this range as destination address when invoking these routines, in order to correctly interoperate with such implementations.

4.1. Information gathering routines

These routines allow to obtain various information about the implementation capabilities, working parameters, and current state.

4.1.1. UNAPI_GET_INFO: Obtain the implementation name and version

```
Input:  A = 0
Output: A  = Error code
        HL = Address of the implementation name string
        DE = API specification version supported. D=primary, E=secondary.
        BC = API implementation version. B=primary, C=secondary.
```

This routine is mandatory for all implementations of all UNAPI compliant APIs. It returns basic information about the implementation itself: the implementation version, the supported API version, and a pointer to the implementation description string.

The implementation name string must be placed in the same slot or segment of the implementation code (or in page 3), must be zero terminated, must consist of printable characters, and must be at most 63 characters long (not including the terminating zero). Refer to the MSX-UNAPI specification for more details.

ERROR CODES

This routine never fails. ERR_OK is always returned.

4.1.2. TCPIP_GET_CAPAB: Get information about the TCP/IP capabilities and features

```
Input:  A = 1
        B = Index of information block to retrieve:
            1: Capabilities and features flags, link level protocol
            2: Connection pool size and status
            3: Maximum datagram size allowed
Output: A = Error code
        When information block 1 requested:
            HL = Capabilities flags
            DE = Features flags
            B  = Link level protocol used
        When information block 2 requested:
            B = Maximum simultaneous TCP connections supported
            C = Maximum simultaneous UDP connections supported
            D = Free TCP connections currently available
            E = Free UDP connections currently available
            H = Maximum simultaneous raw IP connections supported
            L = Free raw IP connections currently available
        When information block 3 requested:
            HL = Maximum incoming datagram size supported
            DE = Maximum outgoing datagram size supported
```

As explained in Section 1.3, the TCP/IP UNAPI specification is modular, meaning that implementators may choose to include only a certain funcionality subset in the developed implementations. This is the routine that gives information about the

capabilities actually available in the implementation in which it is invoked. It also provides information about other implementation working parameters that may be useful for client applications.

The **capabilities flags** is the most important piece of information, and should be retrieved by all client applications at startup time, before trying to actually perform any TCP/IP related operation. It consists of a bitfield in which each bit is associated to one of the capabilities provided by the routines described in this specification. When the bit is one, the capability is available, and client applications can safely invoke the routines that provide the capability. When the bit is zero, the capability is not implemented, and trying to invoke any of the associated routines will result in the routine returning a ERR_NOT_IMP error code. (Some routines depend on a given capability or not depending on the input parameters; more details are given on each routine description).

The capabilities flags are the following. Bit 0 is LSB of register L, bit 8 is LSB of register H.

> Bit 0: Send ICMP echo messages (PINGs) and retrieve the answers
> Bit 1: Resolve host names by querying a local hosts file or database
> Bit 2: Resolve host names by querying a DNS server
> Bit 3: Open TCP connections in active mode
> Bit 4: Open TCP connections in passive mode, with specified remote socket
> Bit 5: Open TCP connections in passive mode, with unsepecified remote socket
> Bit 6: Send and receive TCP urgent data
> Bit 7: Explicitly set the PUSH bit when sending TCP data
> Bit 8: Send data to a TCP connection before the ESTABLISHED state is reached
> Bit 9: Flush the output buffer of a TCP connection
> Bit 10: Open UDP connections
> Bit 11: Open raw IP connections
> Bit 12: Explicitly set the TTL and TOS for outgoing datagrams
> Bit 13: Explicitly set the automatic reply to PINGs on or off
> Bit 14: Automatically obtain the IP addresses, by using DHCP or an equivalent protocol
> Bit 15: Unused

The **features flags** provide additional information about the internal working parameters of the implementation. These parameters have no direct influence on the specification routines (no ERR_NOT_IMP error will ever be returned as a result of one of these features being missing), but client applications may indirectly make use of this information to decide how to behave.

The features flags are the following. Bit 0 is LSB of register E, bit 8 is LSB of register D.

> Bit 0: Physical link is point to point
> Bit 1: Physical link is wireless
> Bit 2: Connection pool is shared by TCP, UDP and raw IP (see explanation about maximum simultaneus connection support below)
> Bit 3: Checking network state requires sending a packet in looback mode, or other expensive (time consuming) procedure
> Bit 4: The TCP/IP handling code is assisted by external hardware
> Bit 5: The loopback address (127.x.x.x) is supported
> Bit 6: A host name resolution cache is implemented
> Bit 7: IP datagram fragmentation is supported
> Bit 8: User timeout suggested when opening a TCP connection is actually applied
> Bits 9-15: Unused

The **link level protocol used** byte may be one of the following. Future versions of this

specification may define additional codes.

> 0: Other/Unspecified
> 1: SLIP
> 2: PPP
> 3: Ethernet

The **connection pool size and status** block informs about how many TCP, UDP and raw IP connections can be handled simultaneously by the implementation, as well as how many free connections are currently available. If the TCP, UDP or raw IP protocols are not supported at all (as indicated in the capabilities flags), then the maximum and free connection count for that protocol should be returned as zero.

When the *"Connection pool is shared by TCP, UDP and raw IP"* feature bit is set, it means that there is one single group of free connections for both TCP, UDP and raw IP, rather than one separate group for each protocol. This implies that the "maximum simultaneuos connections" and "current free connections" values always will be the same for all three protocols, and opening a new connection for any of the protocols will cause all three free connection counters to decrease.

The **maximum datagram size allowed** block informs about the maximum size of IP datagrams (including IP and higher level protocol headers, but not including any link level header) the implementation can handle. The implementation may silently discard any incoming datagrams larger than the maximum incoming datagram size, but client applications may usually ignore this fact. More important is the maximum outgoing datagram size, since client applications need to take in account this value when performing an operation that involves directly sending datagrams (such as sending UDP data or raw IP datagrams).

All implementations are required to support a minimum datagram size of 576 bytes, as per the IP protocol specification.

ERROR CODES

- ERR_OK

The requested information block has been returned.

- ERR_INV_PAR

Invalid information block index specified.

4.1.3. TCPIP_GET_IPINFO: Get IP address

```
Input:  A = 2
        B = Index of address to obtain:
            1: Local IP address
            2: Peer IP address
            3: Subnet mask
            4: Default gateway
            5: Primary DNS server IP address
            6: Secondary DNS server IP address
Output: A = Error code
        L.H.E.D = Requested address
```

This routine returns one of the IP address parameters used by the implementation, as

currently configured. If an address is not configured, then it is returned as 0.0.0.0.

The addresses are returned in the format L.H.E.D. For example, 1.2.3.4 is returned as HL=0201h, DE=0403h. This makes easier to store and retrieve addresses in memory using simple `ld (IP),hl : ld(IP+2),de` or equivalent instructions.

ERROR CODES

- ERR_OK

The requested IP address has been returned.

- ERR_INV_PAR

An invalid value for B has been specified at input, or the specified address type does not make sense for the implementation (for example the subnet mask or the default gateway when the link layer protocol is PPP, or the peer address on an Ethernet network).

4.1.4. TCPIP_NET_STATE: Get network state

```
Input:  A = 3
Output: A = Error code
        B = Current network state:
            0: Closed
            1: Opening
            2: Open
            3: Closing
            255: Unknown
```

This ruotine returns the current state of the network availability. It is only possible to send and receive data when the network state is "Open"; it may or may not be possible as well when the returned state is "Unknown".

The "Closed" state may refer to a logical close (for example, no connection has been established in case of serial communications using the PPP protocol) or to a physical medium unavailability (for example, no cable is connected in case of an Ethernet network).

The "Opening" state means that the implementation is actively trying to reach the "Open" state, but the actions that are actually performed for this depend on each implementation. For example, it may mean "connecting to the ISP" in case of modem communications, or "obtaining addresses via DHCP" in case of an Ethernet network.

There are no restrictions on the possible transitions from any of these states to any other. For example, an implementation could pass from "Opening" to "Closed" again if an error is detected during the opening process. Or, it could toggle between "Open" and "Closed" directly if no special setup or shutdown process is required.

Implementations should return the *"Checking network state requires an expensive procedure"* feature flag set if calling this routine implies performing a time consuming procedure (such as sending a loopback packet), in order to inform client that it should not be called too often.

ERROR CODES

This routine never fails. ERR_OK is always returned.

4.2. ICMP echo request (PING) routines

These routines allow to send ICMP echo request messages (known as PINGs) and to retrieve the received responses.

4.2.1. TCPIP_SEND_ECHO: Send ICMP echo message (PING)

```
Input:  A  = 4
        HL = Address of echo parameters block
Output: A = Error code
```

This routine sends an ICMP echo request (a PING) with the specified parameters. A parameters block with the following format must be supplied:

> +0 (4): IP address of the destination machine
> +4 (1): TTL for the datagram
> +5 (2): ICMP identifier
> +7 (2): ICMP sequence number
> +9 (2): Data length, 0 to maximum datagram size - 28

It is possible to choose the size of the data area of the ICMP message, but not its contents; these must be always the byte sequence 0 1 2 ... 253 254 255 0 1 2... appropriately truncated to match the specified size.

The identifier and the data number can help on matching requests and replies (an echo reply will always have these values identical to the ones of its associated echo request), they can be any 16 bit number. The TTL value should normally be set to 255, in order to maximize the probability of the packet arriving to its destination.

Replies to ICMP echo messages (in the form of ICMP echo response messages) can be obtained by using the TCPIP_RCV_ECHO routine.

ERROR CODES

- ERR_OK

The ICMP echo message packet has been sent.

- ERR_NOT_IMP

The *"Send ICMP echo messages"* capability flag is not set. The implementation does not support sending ICMP echo messages.

- ERR_NO_NETWORK

There is no network connection available.

- ERR_LARGE_DGRAM

Invalid data length specified. The maximum data length allowed is equal to the maximum outgoing datagram size minus 28 (the size of the IP and ICMP headers

combined).

4.2.2. TCPIP_RCV_ECHO: Retrieve ICMP echo response message

```
Input:  A  = 5
        HL = Address for the echo parameters block
Output: A = Error code
```

This routine retrieves the parameters associated to the oldest received ICMP echo response message (usually received as a reply of an echo message sent via TCPIP_SEND_ECHO), and copies them to the address supplied in HL. It also removes the message information from the implementation internal buffers, so that the next call to this routine will either return information for the next message received, or a ERR_NO_DATA error if no more messages are available. Implementations are required to have internal buffer space to hold data for only one echo response message, but may choose to provide space for more than one; echo response messages received when the buffer space is full are silently discarded.

The parameters block has the same format as the one used by TCPIP_SEND_ECHO, only that now the parameters refer to the received echo response message, not to a message to be sent. It is not possible to retrieve the data part of the echo response messages received.

ERROR CODES

- ERR_OK

The ICMP echo response message parameters have been copied to the specified address.

- ERR_NOT_IMP

The *"Send ICMP echo messages"* capability flag is not set. The implementation does not support receiving ICMP echo response messages.

- ERR_NO_DATA

No new ICMP echo response messages have been received and the implementation's buffer for incoming messages is empty.

4.3. Host name resolution routines

These routines allow to resolve a host name, that is, to convert host names to the appropriate IP addresses. Depending on the supported capabilities, host names may be resolved by using a local hosts file or equivalent mechanism, by querying a DNS server, or both.

4.3.1. TCPIP_DNS_Q: Start a host name resolution query

```
Input:  A  = 6
        HL = Address of the host name to be resolved, zero terminated
        B  = Flags, when set to 1 they instruct the resolver to:
            bit 0: Only abort the query currently in progress, if there
is any
```

```
                      (other flags and registers are then ignored)
              bit 1: Assume that the passed name is an IP address,
                     and return an error if this is not true
              bit 2: If there is a query in progress already,
                     do NOT abort it and return an error instead
Output:  A = Error code
         B = 0 if a query to a DNS server is in progress
             1 if the name represented an IP address
             2 if the name could be resolved locally
         L.H.E.D = Resolved IP address
                   (only if no error occurred and B=1 or 2 is returned)
```

This routine, together with its counterpart TCPIP_DNS_S explained later, allows to obtain the IP address associated to a given host name (for example "smtp.mailserver.com"), which may be up to 255 characters long. To achieve this, the DNS servers whose IP addresses are currently configured (see TCPIP_GET_IP routine) are queried. If the implementation supports a local hosts file or an equivalent mechanism (as indicated by the homonym capability flag), then it is queried first, and only when no match is found is the DNS server queried.

Strings that directly represent an IP address (for example "120.200.0.34") must also be accepted by this routine, making then easy to develop programs that accept both host names and IP addresses as a user supplied parameter. This functionality is mandatory even if the implementation does not support host name resolution by local hosts file nor by querying DNS servers (that is, even if the *"Resolve host names by querying a local hosts file"* and *"Resolve host names by querying a DNS server"* capability flags are both not set).

After the host name is examined, there are three possibilities regarding how the host name can be resolved:

1. *The name represents an IP address.* In that case, the result is directly returned in registers HL and DE, and B=1 is returned.
2. *The name can be resolved locally,* by using a local hosts file or the resolver cache. In that case, the result is directly returned in registers HL and DE, and B=2 is returned.
3. *A query to a DNS server is needed to resolve the name*. In that case, the query is started and B=0 is immediately returned; the query processing will continue in background. TCP_DNS_S must be invoked in order to know when the query has finished, and to get the result.

In the first two cases, the result must be cached so that a later call to TCPIP_DNS_S will return it as well. That way, client applications may ignore the fact that some names will be resolved locally (that is, they can ignore the returned values in B, HL and DE), and always invoke TCP_DNS_S to get the result of the name resolution process.

As stated above, the only mandatory capability of this routine is to parse strings representing IP addresses. Resolution of host names via local hosts file, and resolution via queries to DNS servers, are both optional capabilities. Depending of which of these capabilities are supported, implementations should behave as follows when a host name that does not represent an IP address is requested to be resolved:

- *None of the capabilities is supported:* An ERR_NOT_IMP error must be returned.
- *Only local hosts file is supported:* The host name must be searched in the local hosts file. If not found, a "Host name not found" error must be cached so a later call to TCPIP_DNS_S will return it. (See the description of TCPIP_DNS_S for details on this error)

- *Only queries to DNS servers are supported:* A query to the configured DNS server must be started.
- *Both capabilities are supported:* The host name must be searched in the local hosts file. If not found, a query to the configured DNS server must be started.

If there is already a query in progress when TCPIP_DNS_Q is executed, normally this query will be aborted in order to initiate the new one (only one query can be handled at a time). However, if B:2 is set at input and there is a query in progress, then this query will not be aborted and a ERR_QUERY_EXSITS will be returned instead.

How the implementation performs queries to DNS servers is outside this specification, however it is expected that all implementations will follow some basic procedures:

- The primary DNS server will be queried, and in case of failure, the secondary server will be queried.
- The "recursive query" flag in the packets send to the DNS servers will be set, but if the server does not support recursive queries and the address of another DNS server is obtained instead of the requested IP address, then this server will in turn be queried.
- A total timeout for the entire process will be implemented so that a client application invoking TCPIP_DNS_S in a loop will not hang forever waiting for a result.

However the implementation behaves, the process must be transparent for client applications, which will rely exclusively on the results returned by TCPIP_DNS_Q and TCPIP_DNS_S routines.

Note for implementors. The following algorithm should be followed when implementing this routine, to ensure that it behaves properly and returns the appropriate error codes depending on the input parameters and the supported capabilities.

```
Is B:0 set?
    Yes: Abort the current query, if any
         Return ERR_OK, B=0
Is there a query in progress?
    Yes: Is B:2 set?
         Yes: Return ERR_QUERY_EXISTS
         No: Abort the current query
Try to parse host name as an IP address
Success?
    Yes: Cache the result so that TCPIP_DNS_S will return it
         Return ERR_OK, B=1, and the address in HL,DE
B:1 was set?
    Yes: Return ERR_INV_IP
Are the local hosts file mechanism capability, and the querying DNS
servers capability, both NOT supported?
    Yes: Return ERR_NOT_IMP
Is the local hosts file mechanism supported?
    Yes: Is the host name in the local hosts file?
         Yes: Cache the result so that TCPIP_DNS_S will return it
              Return ERR_OK, B=2, and the address in HL,DE
Is DNS querying supported?
    No: Cache a "Host name not found error" so that TCPIP_DNS_S will
return it (see description of TCPIP_DNS_S)
         Return ERR_OK, B=0
Is DNS cache supported?
```

```
        Yes: Is the host name in the DNS cache?
            Yes: Cache the result so that TCPIP_DNS_S will return it
                Return ERR_OK, B=2, and the address in HL,DE
Is network available?
    No: Return ERR_NO_NETWORK
Is there any DNS server configured?
    No: return ERR_NO_DNS
Start querying the primary DNS server, or the secondary, if primary is not
configured
Return ERR_OK, B=0
```

Please note that the result is cached only if ERR_OK is returned. If any other error code is returned, then TCPIP_DNS_S must still return the same result as before TCPIP_DNS_Q was invoked.

Note for client application developers. Below is an example of a simple way of using the TCPIP_DNS_Q/S routines pair. For simplicity it is assumed that the implementation routines can be called directly, without need for slot or segment switching.

```
    ;* Start the query

    ld      hl,HOST_NAME
    xor     a
    call    TCPIP_DNS_Q
    or      a
    jp      nz,ERROR_Q     ;Jump to error management code

    ;* Wait until either a result or an error is returned

WAIT:
    xor     a
    call    TCPIP_DNS_S
    cp      1     ;Query is still in progress
    jr      z,WAIT
    cp      3     ;Query finished with error
    jp      z,ERROR_S  ;Jump to error management code
    ;From here, A can only be equal to 2
    ld      (IP_ADD),hl    ;Store the result
    ld      (IP_ADD+2),de
    ...

HOST_NAME: db     "name.host.com",0
IP_ADD:    ds     4
```

Note that in this example, we don't care about HOST_NAME representing an IP address or being resolved locally. We just invoke TCPIP_DNS_S to get the resolved IP address, without checking how it was obtained. This will be enough in most cases.

ERROR CODES

- ERR_OK

A DNS query is in progress, or the name resolution process has finished and the result has been cached for TCPIP_DNS_S.

- ERR_INV_PARAM

An unused flag was set in B.

- ERR_INV_IP

The host name does not represent an IP address and the routine was invoked with B:1 set.

- ERR_NOT_IMP

The host name does not represent an IP address, and neither the *"Resolve host names by querying a local hosts file"* nor the *"Resolve host names by querying a DNS server"* capabilities are supported.

- ERR_QUERY_EXISTS

The routine was invoked with B:0 set and there is another query in progress.

- ERR_NO_NETWORK

A query to a DNS server must start, but there is no network connectivity available.

- ERR_NO_DNS

A query to a DNS server must start, but there are no DNS server addresses configured.

4.3.2. TCPIP_DNS_S: Obtains the host name resolution process state and result

```
Input:  A = 7
        B = Flags, when set to 1 they instruct the resolver to:
          bit 0: Clear any existing result or error condition after the
execution
                (except if there is a query in progress)
Output: A = Error code
        B = DNS error code (when error is ERR_DNS)
        B = Current query status (when error is ERR_OK):
          0: There is no query in progress, nor any result nor error
code available
          1: There is a query in progress
          2: Query is complete
        C = Current query substatus (when error is ERR_OK and B=1):
          0: Unknown
          1: Querying the primary DNS server
          2: Querying the secondary DNS server
          3: Querying another DNS server
        C = Resolution proces type (when error is ERR_OK and B=2):
          0: The name was obtained by querying a DNS server
          1: The name was a direct representation of an IP address
          2: The name was resolved locally
        L.H.E.D = Resolved IP address (when error is ERR_OK and B=2)
```

This routine is the complementary of TCPIP_DNS_Q. It allow to obtain the status of the current host name resolution process (which was initiated via an invokation to TCPIP_DNS_Q); and if the query process has finished, it returns either the resulting IP addres or an appropriate error code.

As explained in the description of TCPIP_DNS_Q, in certain circumstances the host name can be resolved locally, without having to actually query any DNS server. In that case, the result is cached so that TCPIP_DNS_S returns it as well; that way, client applications do not need to worry about a DNS query being actually performed or not (see the sample code in the description of TCPIP_DNS_Q).

If B:0 is set at input, the current result is cleared after the routine returns, so that a later invocation of the routine will return B=0 (no query is in progress and no result is available). If B:0 is not set at input, successive invokations of the routine will always return the same result, until an invocation of TCPIP_DNS_Q that returns ERR_OK is performed. Note that B:0 is ignored if there is a query in progress.

If ERR_DNS is returned, then B returns an error code with more detailed error information. Below are the possible error codes that B can return; codes 1 to 15 are directly returned by a DNS server (as defined in RFC1035), other codes are generated by the implementation itself. Errors 1 and 4 should never be obtained if the implementation is working properly.

    0: Unknown error
    1: Invalid query packet format
    2: DNS server failure
    3: The specified host name does not exist
    4: The DNS server does not support this kind of query
    5: Query refused
    6-15: Undefined error codes
    16: One of the queried DNS servers did not reply
    17: Total process timeout expired
    18: Query cancelled by the user (TCPIP_DNS_Q was executed with B:0 set)
    19: Network connectivity was lost during the process
    20: The obtained reply did not contain REPLY nor AUTHORITATIVE
    21: The obtained reply is truncated

Note that this routine does never return ERR_NOT_IMP, since it is required that TCPIP_DNS_Q is implemented to at least parse strings representing IP addresses.

ERROR CODES

- ERR_OK

A DNS query is in progress, or a DNS query has finished without errors, or no query is in progress and results from a past query are not available.

- ERR_DNS

A DNS query has finished with errors, see register B for more details.

4.4. UDP protocol related routines

These routines allow to send and receive data by using the UDP protocol. Even if UDP is a datagram based, connectionless protocol, a connections-like mechanism is used; here an "UDP connection" is simply a pair formed by the local IP address + a local port number. When sending UDP data, the connection number and the remote IP address and port must be specified; when a UDP datagram is received on a given connection, the remote IP address and port are retrieved as well.

UDP connections are identified by connection numbers, which may be any number in the range 1 to 254. The numbers actually used depend on the implementation; client applications should handle these numbers as opaque values.

There is a "intended connection lifetime" parameter that is to be specified when opening the connection: *transient*, which means that the connection is intended to be closed when the client application finishes; and *resident*, which means that the connection is intended to be used by a *resident* program. The only behavioral difference between these two modes as far as this specification is concerned, is that all open transient connections are closed when TCPIP_UDP_CLOSE is invoked and zero is specified as the connection number, being the resident connections unaffected.

This specification does not define how implementations should handle incoming UDP datagrams whose destination port number is not associated with any open UDP connection. These datagrams can be silently discarded, or implementation specific routines can be defined to handle them.

4.4.1. TCPIP_UDP_OPEN: Open a UDP connection

```
Input:  A  = 8
        HL = Local port number (0FFFFh: random port)
        B  = Intended connection lifetime:
             0: Transient
             1: Resident
Output: A = Error code
        B = Connection number
```

This routine opens a new UDP connection. This process does not imply any type of negotiation or data exchange with any remote host; all that happens is that a connection number is associated with a local port number.

The port number should not be zero, as this means "unspecified port" in the UDP protocol, and this is not supported by this specification. Also, it should not be in the range 0xFFF0-0xFFFE, these port numbers are reserved for internal use of the implementations.

When 0FFFFh is specified as the port number, the implementation must use a randomly chosen port number. This port will be in the range 16384-32767, and will never be a port number used by another open UDP connection.

Once the connection is open, UDP datagrams data can be sent and received by using the TCPIP_UDP_SEND and TCPIP_UDP_RCV routines. The connection can be closed by using TCPIP_UDP_CLOSE. These routines expect the connection number returned by TCPIP_UDP_OPEN as an input parameter.

ERROR CODES

- ERR_OK

The connection has been created successfully. The returned connection number is valid.

- ERR_NOT_IMP

The *"Open UDP connections"* capability flag is not set. The implementation does not

support sending and receiving UDP datagrams.

- ERR_INV_PARAM

- The "intended connection lifetime" parameter has an invalid value.
- Zero has been specified as the local port.
- The local port is in the range 0xFFF0-0xFFFE.

- ERR_CONN_EXISTS

Other UDP connection exists which has the same local port assigned.

- ERR_NO_FREE_CONN

There are no free UDP connections available.

4.4.2. TCPIP_UDP_CLOSE: Close a UDP connection

```
Input:  A = 9
        B = Connection number
            0 to close all open transient UDP connections
Output: A = Error code
```

This routine closes the specified UDP connection. This process does not imply any type of negotiation or data exchange with any remote host; all that happens is that the connection number is freed and all the pending incoming datagrams for the connection are discarded.

All the existing connections that were open in "transient" lifetime mode are closed if zero is specified as the connection number.

ERROR CODES

- ERR_OK

The connection(s) has (have) been closed successfully.

- ERR_NOT_IMP

The *"Open UDP connections"* capability flag is not set. The implementation does not support sending and receiving UDP datagrams.

- ERR_NO_CONN

There is no connection open with the specified number.

4.4.3. TCPIP_UDP_STATE: Get the state of a UDP connection

```
Input:  A = 10
        B = Connection number
Output: A  = Error code
        HL = Local port number
        B  = Number of pending incoming datagrams
        DE = Size of oldest pending incoming datagram (data part only)
```

This routine retrieves information about the UDP connection with the specified number.

"Number of pending incoming datagrams" indicates how many UDP datagrams whose destination port matches the connection's associated port have been received and cached by the implementation; these datagrams can be retrieved by using the TCPIP_UDP_RCV routine. Implementations are required to have buffer space for only one incoming UDP datagram per connection, but may optionally have space for more. Datagrams received when this buffer is full are silently discarded.

"Number of pending incoming datagrams" may actually be smaller than the actual number of pending datagrams, if the implementation knows than there are received datagrams but has no means to know how many. The following rules apply: if no pending datagrams are available, then B=0; if at least one pending datagram is available, then B>0 but less than or equal to the number of pending datagrams. Applications willing to retrieve all the pending datagrams for a UDP connection should not rely on this number; instead, they should invoke TCPIP_UDP_RCV repeatedly until ERR_NO_DATA is returned.

"Size of oldest pending datagram" indicates the size of the data part of the datagram that will be retrieved by the next call to TCPIP_UDP_RCV. It is zero if the datagram contains no data apart from the UDP header, and contains no meaningful information if there are no pending datagrams (that is, if B=0 is returned).

ERROR CODES

- ERR_OK

The requested information has been retrieved.

- ERR_NOT_IMP

The *"Open UDP connections"* capability flag is not set. The implementation does not support sending and receiving UDP datagrams.

- ERR_NO_CONN

There is no connection open with the specified number.

4.4.4. TCPIP_UDP_SEND: Send an UDP datagram

```
Input:  A = 11
        B = Connection number
        HL = Address of datagram data
        DE = Address of parameters block
Output: A = Error code
```

This routine sends a block of data in the form od an UDP datagram. The local port used to compose the datagram is the one associated to the specified connection; the remote IP address and port are took from the parameters block. The format of the parameters block is as follows:

    +0 (4): Destination IP address
    +4 (2): Destination port
    +6 (2): Data length

The maximum allowed value for the datagram data length is equal to the maximum outgoing datagram size supported, minus 28 (the combined size of the IP and UDP headers).

Implementations are expected to send the datagram immediately. At most, the routine can block while waiting for another datagram transmission in progress to finish; or the datagram can be cached to be sent at a later time, provided that the time between routine invokation and actual datagram send is very small (unnoticeable for the user).

ERROR CODES

- ERR_OK

The datagram has been sent.

- ERR_NOT_IMP

The *"Open UDP connections"* capability flag is not set. The implementation does not support sending and receiving UDP datagrams.

- ERR_NO_CONN

There is no connection open with the specified number.

- ERR_NO_NETWORK

No network connectivity is currently available.

- ERR_LARGE_DGRAM

The data length specified in the parameters block is larger than the maximum outgoing datagram size supported minus 28.

4.4.5. TCPIP_UDP_RCV: Retrieve an incoming UDP datagram

```
Input:  A = 12
        B = Connection number
        HL = Address for datagram data
        DE = Maximum data size to retrieve
Output: A = Error code
        L.H.E.D = Source IP address
        IX = Source port
        BC = Actual received data size
```

This routine retrieves the data part and information parameters of the oldest received UDP datagram for the given connection, and removes the datagram from the implementation's internal buffer. Implementations are required to have buffer space for only one incoming UDP datagram per connection, but may optionally have space for more. Datagrams received when this buffer is full are silently discarded.

The data part of the datagram, without the IP and UDP headers, will be copied to the address pointed by HL. The entire datagram data part will be copied if its length is smaller or equal than the "maximum data size" parameter passed in DE; otherwise, only the first DE bytes will be copied and the rest of the datagram data will be discarded (DE=0FFFFh can be specified with the meaning of "get the whole datagram"). No data

will be copied at all if DE=0. Remember that it is possible to check the datagram data size in advance by using TCPIP_UDP_STATE.

Information about the originator of the datagram will be returned in registers HL, DE, and IX. Register BC will contain the datagram data size as it was received, which may be larger than the number of bytes actually retrieved.

ERROR CODES

- ERR_OK

The datagram data and information have been retrieved.

- ERR_NOT_IMP

The *"Open UDP connections"* capability flag is not set. The implementation does not support sending and receiving UDP datagrams.

- ERR_NO_CONN

There is no connection open with the specified number.

- ERR_NO_DATA

There are no pending incoming datagrams for the specified connection.

4.5. TCP protocol related routines

These routines allow to send and receive data by using the TCP protocol. TCP is a connection based protocol, and thus this specification defines TCP connections as the way to manage TCP communications. A connection is defined by two pairs of IP address + port, one for the local computer and another one for the remote computer.

TCP connections are identified by connection numbers, which may be any number in the range 1 to 254. The numbers actually used depend on the implementation; client applications should handle these numbers as opaque values.

There is a "intended connection lifetime" parameter that is to be specified when opening the connection: *transient*, which means that the connection is intended to be closed when the client application finishes; and *resident*, which means that the connection is intended to be used by a *resident* program. The only behavioral difference between these two modes as far as this specification is concerned, is that all open transient connections are closed when TCPIP_TCP_CLOSE is invoked and zero is specified as the connection number, being the resident connections unaffected. The same applies to TCPIP_TCP_ABORT.

4.5.1. TCPIP_TCP_OPEN: Open a TCP connection

```
Input:  A  = 13
        HL = Address of parameters block
Output: A = Error code
        B = Connection number
```

This routine opens a new TCP connection using the information specified in the

parameters block. The format of this block is:

> +0 (4): Remote IP address (0.0.0.0 for unspecified remote socket)
> +4 (2): Remote port (ignored if unspecified remote socket)
> +6 (2): Local port, 0FFFFh for a random value
> +8 (2): Suggestion for user timeout value
> +10 (1): Flags:
>> bit 0: Set for passive connection
>> bit 1: Set for resident connection

If the connection is open in active mode, a remote IP address and a remote port must be specified. The implementation must start trying to actively establish a connection (that is, a SYN segment must be sent and connection must enter the SYN-SENT state), and must continue processing the connection establishment in background.

If the connection is open in passive mode, there is the option to leave the remote IP address unspecified (by specifying it as 0.0.0.0), meaning that a connection from any host will be accepted; the remote port is ignored in this case. The implementation must simply enter the LISTEN state when a connection is open in passive mode.

Only one connection can exist with the same combination of local port, remote IP address and remote port. Passive connections with unspecified remote socket are an exception: more than one connection can exist with the same local port and 0.0.0.0 as the remote IP address. In this case, when a connection request is received for that local port, the implementation must choose one of these connections, and there is no way for client applications to decide which one is chosen.

When 0FFFFh is specified as the local port, the implementation must use a randomly chosen port number. This port will be in the range 16384-32767, and will never be a port number used by another open TCP connection.

Although the TCP standard allows to transform a passive connection into an active one by opening it again, this is not allowed in this specification. On the other hand, sending data to an active connection before it reaches the ESTABLISHED state may or may not be supported by the implementation, depending on the value of the *Send data to a TCP connection before the ESTABLISHED state is reached"* capability flag. When this is supported, data is cached internally by the implementation, and then sent when the ESTABLISHED state is reached.

The "user timeout" is a suggested value for a timer that is activated when new data is sent to the connection, and is stopped when the acknowledgment (ACK) arrives for that data. If this timer expires, the host is considered to be unreachable and the connection is aborted. This timer is also applied to the connection initiation (that is, it is also the maximum time that can elapse until the acknowledgment for a sent SYN segment arrives). The specified value is interpreted as follows:

- 1 to 1080: Value for the timer in seconds (1 second to 18 minutes).
- 0: Use the default value for the timer, as chosen by the implementation.
- 0FFFFh: Infinite, data is retransmitted indefinitely until either the acknowledgment arrives, the connection is aborted or an RST segment arrives.

Note that this value is just a suggestion: implementations may ignore it and use their own value, or they may not implement this timeout at all.

Once the connection is open, TCP data can be sent and received by using the TCPIP_TCP_SEND and TCPIP_TCP_RCV routines. The connection can be closed by using TCPIP_UDP_CLOSE, or aborted by using TCPIP_TCP_ABORT. These routines expect the connection number returned by TCPIP_TCP_OPEN as an input parameter.

- ERR_OK

The connection has been open with the specified parameters. The returned connection number is valid.

- ERR_NOT_IMP

- The connection is required to be open in active mode, but the *"Open TCP connections in active mode"* capability flag is not set.
- The connection is required to be open in passive mode with specified remote socker, but the *"Open TCP connections in passive mode, with specified remote socket"* capability flag is not set.
- The connection is required to be open in passive mode with unspecified remote socker, but the *"Open TCP connections in passive mode, with unspecified remote socket"* capability flag is not set.

- ERR_INV_PARAM

- An invalid value for the user timeout was specified.
- 0.0.0.0 was specified as the remote IP address when trying to open a connection in active mode.
- An unused flag was set.

- ERR_NO_NETWORK

No network connectivity is currently available.

- ERR_CONN_EXISTS

There is a TCP connection already open with the same combination of local port, remote IP address and remote port.

- ERR_NO_FREE_CONN

There are no free TCP connections available.

4.5.2. TCPIP_TCP_CLOSE: Close a TCP connection

```
Input:  A  = 14
        B = Connection number
            0 to close all open transient TCP connections
Output: A = Error code
```

This routine initiates the closing procedure for a TCP connection. As per the TCP protocol, closing a connection implies that it is not possible to send new data to it, but the other side can still sending new data (and we can still receiving it); the connection will not be freed until both sides have exchanged a FIN segment. To immediately destroy the connection, TCPIP_TCP_ABORT should be used instead.

All the existing connections that were open in "transient" lifetime mode are closed if zero is specified as the connection number.

ERROR CODES

- ERR_OK

The connection has been closed.

- ERR_NOT_IMP

The implementation does not support TCP connections at all (none of the *"Open TCP connections in active mode"*, *"Open TCP connections in passive mode, with specified remote socket"* or *"Open TCP connections in passive mode, with unspecified remote socket"* capability flags is set).

- ERR_NO_CONN

There is no connection open with the specified number.

4.5.3. TCPIP_TCP_ABORT: Abort a TCP connection

```
Input:  A  = 15
        B = Connection number
            0 to abort all open transient TCP connections
Output: A = Error code
```

This routine aborts the TCP connection with the specified number. An RST segment is sent if appropriate (according to the TCP protocol specification), and then the connection is freed immediately.

All the existing connections that were open in "transient" lifetime mode are aborted if zero is specified as the connection number.

ERROR CODES

- ERR_OK

The connection has been aborted.

- ERR_NOT_IMP

The implementation does not support TCP connections at all (none of the *"Open TCP connections in active mode"*, *"Open TCP connections in passive mode, with specified remote socket"* or *"Open TCP connections in passive mode, with unspecified remote socket"* capability flags is set).

- ERR_NO_CONN

There is no connection open with the specified number.

4.5.4. TCPIP_TCP_STATE: Get the state of a TCP connection

```
Input:  A = 16
        B = Connection number
        HL = Pointer in TPA for connection information block
             (0 if not needed)
Output: A  = Error code
        B  = Connection state
```

```
C  = Close reason (only if ERR_NO_CONN is returned)
HL = Number of total available incoming bytes
DE = Number of urgent available incoming bytes
IX = Available free space in the output buffer
     (0FFFFh = infinite)
```

This routine returns information about the current state of an open TCP connection. It is useful mainly to know if data can be sent to the connection, and if so how many output buffer space is available; as well as to know if incoming data is available.

The connection information block is 8 bytes long. If a non-zero pointer is passed in HL, and if no error is returned, it will be filled with the following information about the connection:

> +0 (4): Remote IP address
> +4 (2): Remote port
> +6 (2): Local port

The **connection state** value encodes the current connection state as defined in the TCP specification. It will be one of the following values:

> 0: Unknown
> 1: LISTEN
> 2: SYN-SENT
> 3: SYN-RECEIVED
> 4: ESTABLISHED
> 5: FIN-WAIT-1
> 6: FIN-WAIT-2
> 7: CLOSE-WAIT
> 8: CLOSING
> 9: LAST-ACK
> 10: TIME-WAIT

The **close reason** value is meaningful only when ERR_NO_CONN is returned in A, and contains an optional indication of why the connection is closed. It may be one of the following values ("Unknown" should be returned if the implementation does not support reporting close reasons):

> 0: Unknown
> 1: This connection has never been used since the implementation was initialized.
> 2: The TCPIP_TCP_CLOSE method was called.
> 3: The TCPIP_TCP_ABORT method was called.
> 4: A RST segment was received (the connection was refused or aborted by the remote host).
> 5: The user timeout expired.
> 6: The connection establishment timeout expired.
> 7: Network connection was lost while the TCP connection was open.
> 8: ICMP "Destination unreachable" message received.

The **Number of total available incoming bytes** value indicates how many bytes have been received by this connection and can be retrieved by using the TCPIP_TCP_RCV routine. The **Number of urgent available incoming bytes** value indicates how many of these bytes are urgent data (this value will always be zero if the implementation does not support TCP urgent data, as per the *"Send and receive TCP urgent data"* capability flag).

The **Available free space in the output buffer** value indicates how many data can be queued by calling the TCPIP_TCP_SEND routine. This free space decreases when a call to

that routine is made, and increases when the buffered data is sent an acknowledged. A value of 0FFFFh means that there is no practical buffer space limit and a call to TCPIP_TCP_SEND will never return an ERR_BUFFER error.

ERROR CODES

- ERR_OK

The connection has been aborted.

- ERR_NOT_IMP

The implementation does not support TCP connections at all (none of the *"Open TCP connections in active mode"*, *"Open TCP connections in passive mode, with specified remote socket"* or *"Open TCP connections in passive mode, with unespecified remote socket"* capability flags is set).

- ERR_NO_CONN

There is no connection open with the specified number.

4.5.5. TCPIP_TCP_SEND: Send data a TCP connection

```
Input:  A  = 17
        B  = Connection number
        DE = Address of the data to be sent
        HL = Length of the data to be sent
        C  = Flags:
             bit 0: Send the data PUSHed
             bit 1: The data is urgent
Output: A = Error code
```

This routine enqueues data to be sent by a TCP connection. The data is sent in the form of TCP segments at the appropriate times, depending on the sending window available, the data acknoledgements received from the remote host, and the possible data sending optimization algorithms used by the implementation (see the TCP protocol specification for more details).

All implementations must accept data to be enqueued when the connection is in the ESTABLISHED or CLOSE-WAIT state. Optionally, implementations may accept also data when the connection is in the SYN-SENT or SYN-RECEIVED state (the *"Send data to a TCP connection before the ESTABLISHED state is reached"* capability flag must be set); in this case, enqueued data is sent as soon as the ESTABLISHED state is set. In all other connection states, this routine must return a ERR_CONN_STATE error.

Implementations will tipically have a limited buffer space to enqueue TCP outgoing data. The free space on this buffer can be checked by invoking the TCPIP_TCP_STATE routine. Trying to enqueue more data than buffer space available will result on a ERR_BUFFER error being returned. All implementations are required to have at least 512 bytes of TCP output buffer space.

If the *"Explicitly set the PUSH bit when sending TCP data"* capability flag is set, it is possible to indicate that the data is intended to be PUSHed; in this case, the data enqueued by the involved call and any other data previously enqueued is sent immediately (bypassing any possible data grouping algorithm being used), and the PUSH bit is set on the outgoing datagrams for this data. If the capability is not supported, the

PUSH flag is ignored.

If the *"Send and receive TCP urgent data"* capability flag is set, then all the data enqueued by the involved call will be set in segments with the URG bit set. If the capability is not supported, the "urgent data" flag is ignored.

Note for client application developers. The following pseudocode shows the recommended approach for sending data to a TCP connection. The idea is to feed the connection with data until the output buffer space is full (which is signaled by ERR_BUFFER), and then to give the implementation an opportunity to send the data and free the buffer (which is achieved by calling TCPIP_WAIT_INT). BlockSize can be any value between 1 and 512. A, B, DE and HL refer to the Z80 registers.

```
BlockSize = 128
RemainigSize = (amount of data to send)
DataAddress = (address of data to send)

do
    do
        B = ConnectionNumber
        DE = DataAddress
        HL = min(RemainingSize, BlockSize)
        call TCPIP_TCP_SEND
        error = A
        if(error = ERR_BUFFER)
            call TCPIP_TCP_WAIT
    while(error = ERR_BUFFER)

    RemainigSize = RemainingSize - HL
    DataAddress = DataAddress + HL
while(RemainingSize > 0 and error != 0)

if(error != 0)
    ProcessError(error)
```

ERROR CODES

- ERR_OK

The data has been successfully enqueued for sending.

- ERR_NOT_IMP

The implementation does not support TCP connections at all (none of the *"Open TCP connections in active mode"*, *"Open TCP connections in passive mode, with specified remote socket"* or *"Open TCP connections in passive mode, with unespecified remote socket"* capability flags is set).

- ERR_NO_CONN

There is no connection open with the specified number.

- ERR_INV_PARAM

An unused bit was set in the flags byte.

- ERR_CONN_STATE

- The connection is in the SYN-SENT or SYN-RECEIVED state, and the *"Send data to a TCP connection before the ESTABLISHED state is reached"* capability flag is not set.
- The connection is in the LISTEN, FIN-WAIT-1, FIN-WAIT-2, CLOSING, LAST-ACK or TIME-WAIT state.

- ERR_BUFFER

There is not enough free output buffer to enqueue all the specified data.

4.5.6. TCPIP_TCP_RCV: Receive data from a TCP connection

```
Input:   A  = 18
         B  = Connection number
         DE = Address for the retrieved data
         HL = Length of the data to be obtained
Output:  A  = Error code
         BC = Total number of bytes that have been actually retrieved
         HL = Number of urgent data bytes that have been retrieved
              (placed at the beginning of the received data block)
```

This routine copies to the specified address as much enqueued incoming data bytes as specified in the HL register. If there is less data available, then all the available data is retrieved, and if no data is available at all, then no data is retrieved; in any case, the number of bytes actually retrieved is returned in BC. Having retrieved less data than requested or having retrieved no data at all is not considered an error.

It is possible to check how many incoming data bytes are available by invoking the TCPIP_TCP_STATE routine, but this is not mandatory since as mentioned above, trying to get more data than available is allowed.

If the *"Send and receive TCP urgent data"* capability flag is set, then it is possible than part of the received data is urgent. In that case, the urgent data is placed at the beginning of the retrieved data block, and a value will be returned in HL telling how many of the retrieved bytes are urgent. HL will be zero if no urgent data has been received, or if receiving urgent data is not supported by the implementation.

Note that unlike when sending TCP data, it is not necessray for client applications to worry about the implementation's buffer space for incoming data to be exhausted, since this is handled by the TCP protocol itself (if the buffer is full, the implementation will simply close the receiving window and the remote host will not send more data). Also, this routine can be safely called regardless of the state the TCP connection is in; if the current state does not allow receiving data, then simply no data will be retrieved at all and BC will be returned as zero.

ERROR CODES

- ERR_OK

The data has been successfully retrieved.

- ERR_NOT_IMP

The implementation does not support TCP connections at all (none of the *"Open TCP connections in active mode"*, *"Open TCP connections in passive mode, with specified remote socket"* or *"Open TCP connections in passive mode, with unspecified remote socket"* capability flags is set).

- ERR_NO_CONN

There is no connection open with the specified number.

4.5.7. TCPIP_TCP_FLUSH: Flush the output buffer of a TCP connection

```
Input:   A = 19
         B = Connection number
Output:  A = Error code
```

This routine flushes the output buffer of a TCP connection, that is, removes from the appropriate output buffer the data that has been enqueued to be sent by TCPIP_TCP_SEND but has not been sent yet.

ERROR CODES

- ERR_OK

The data has been successfully retrieved.

- ERR_NOT_IMP

- The implementation does not support TCP connections at all (none of the *"Open TCP connections in active mode"*, *"Open TCP connections in passive mode, with specified remote socket"* or *"Open TCP connections in passive mode, with unspecified remote socket"* capability flags is set).
- The *"Flush the output buffer of a TCP connection"* capability flag is not set.

- ERR_NO_CONN

There is no connection open with the specified number.

4.6. Raw IP connections related routines

These routines allow to send and receive data by using a transport protocol which is not TCP nor UDP. This is achieved by opening a raw IP connection for a given protocol code. When sending data, the implementation will add a IP header with the given protocol code and the specified destination IP address to the supplied block of data, and the result will be sent as a IP datagram; receiving data works by the implementation capturing and enqueueing all incoming datagrams having the given protocol specified in the IP header, and returning them when the client application requests them.

Raw IP connections are identified by connection numbers, which may be any number in the range 1 to 254. The numbers actually used depend on the implementation; client applications should handle these numbers as opaque values.

There is a "intended connection lifetime" parameter that is to be specified when opening

the connection: *transient*, which means that the connection is intended to be closed when the client application finishes; and *resident*, which means that the connection is intended to be used by a *resident* program. The only behavioral difference between these two modes as far as this specification is concerned, is that all open transient connections are closed when TCPIP_RAW_CLOSE is invoked and zero is specified as the connection number, being the resident connections unaffected.

4.6.1. TCPIP_RAW_OPEN: Open a raw IP connection

```
Input:   A = 20
         B = Transport protocol code
         C = Intended connection lifetime:
             0: Transient
             1: Resident
Output:  A = Error code
         B = Connection number
```

This routine opens a raw IP connection for the specified transport protocol code. This process does not imply any type of negotiation or data exchange with any remote host; all that happens is that a connection number is associated with the protocol code.

While the connection is open, the implementation will capture and enqueue all the incoming datagrams that have the specified value in the protocol field; the client application can retrieve them by using the TCPIP_RAW_RCV routine. Also, it is possible to send datagrams which will have the specified value in the protocol field of the IP header by using the TCPIP_RAW_SEND routine, and to close the connection by using TCPIP_RAW_CLOSE. These routines expect the connection number returned by TCPIP_RAW_OPEN as an input parameter.

If the transport protocol code for TCP, UDP or ICMP is specified, the implementation behavior is undefined. It could capture only the packets that are not associated to any open TCP or UDP connection, or it could effetively capture all the incoming datagrams for the specified protocol and render all open connections for that protocol useless.

It is not possible to open more than one connection for the same protocol code.

ERROR CODES

- ERR_OK

The connection has been successfully open. The returned connection number is valid.

- ERR_NOT_IMP

The *"Open raw IP connections"* capability flags is not set. The implementation does not support raw IP connections.

- ERR_CONN_EXISTS

A raw IP connection for the specified protocol already exists.

- ERR_NO_FREE_CONN

There are no free raw IP connections available.

4.6.2. TCPIP_RAW_CLOSE: Close a raw IP connection

```
Input:  A = 21
        B = Connection number
            0 to close all open transient UDP connections
Output: A = Error code
```

This routine closes the specified raw IP connection. This process does not imply any type of negotiation or data exchange with any remote host; all that happens is that the connection number is freed and all the pending incoming datagrams for the connection are discarded. Also, the implementation will stop capturing datagrams with the protocol number associated to the connection in the IP header.

All the existing connections that were open in "transient" lifetime mode are closed if zero is specified as the connection number.

ERROR CODES

- ERR_OK

The connection has been closed successfully.

- ERR_NOT_IMP

The *"Open raw IP connections"* capability flags is not set. The implementation does not support raw IP connections.

- ERR_NO_CONN

There is no connection open with the specified number.

4.6.3. TCPIP_RAW_STATE: Get the state of a raw IP connection

```
Input:  A = 22
        B = Connection number
Output: A  = Error code
        B  = Associated protocol code
        HL = Number of pending incoming datagrams
        DE = Size of the oldest pending incoming datagram
```

This routine retrieves information about the raw connection with the specified number.

"Number of pending incoming datagrams" indicates how many datagrams whose protocol code in the IP header matches the connection's associated protocol code have been received and cached by the implementation; these datagrams can be retrieved by using the TCPIP_RAW_RCV routine. Implementations are required to have buffer space for only one incoming IP datagram per connection, but may optionally have space for more. Datagrams received when this buffer is full are silently discarded.

"Size of oldest pending datagram" indicates the size of the data part of the datagram that will be retrieved by the next call to TCPIP_RAW_RCV. It is zero if the datagram contains no data apart from the IP header itself, and contains no meaningful information if there are no pending datagrams (that is, if B=0 is returned).

ERROR CODES

- ERR_OK

The requested information has been retrieved.

- ERR_NOT_IMP

The *"Open raw IP connections"* capability flags is not set. The implementation does not support raw IP connections.

- ERR_NO_CONN

There is no connection open with the specified number.

4.6.4. TCPIP_RAW_SEND: Send a raw IP datagram

```
Input:  A  = 23
        B  = Connection number
        HL = Address of datagram data
        DE = Address of parameters block
Output: A = Error code
```

This routine sends a block of data in the form of an IP datagram. The implementation will prepend the passed data block with a IP header composed from the protocol code associated to the connection and the data in the parameters block. The format of the parameters block is as follows:

+0 (4): Destination IP address
+4 (2): Data length

The maximum allowed value for the datagram data length is equal to the maximum outgoing datagram size supported, minus 20 (the size of the IP header).

Implementations are expected to send the datagram immediately. At most, the routine can block while waiting for another datagram transmission in progress to finish; or the datagram can be cached to be sent at a later time, provided that the time between routine invokation and actual datagram send is very small (unnoticeable for the user).

ERROR CODES

- ERR_OK

The datagram has been sent.

- ERR_NOT_IMP

The *"Open raw IP connections"* capability flags is not set. The implementation does not support raw IP connections.

- ERR_NO_CONN

There is no connection open with the specified number.

- ERR_NO_NETWORK

No network connectivity is currently available.

- ERR_LARGE_DGRAM

The data length specified in the parameters block is larger than the maximum outgoing datagram size supported minus 20.

4.6.5. TCPIP_RAW_RCV: Retrieve an incoming raw IP datagram

```
Input:  A  = 24
        B  = Connection number
        HL = Address for datagram data
        DE = Maximum data size to retrieve
Output: A = Error code
        L.H.E.D = Source IP address
        BC = Actual received data size
```

This routine retrieves the data part and information parameters of the oldest received IP datagram for the given connection, and removes the datagram from the implementation's internal buffer. Implementations are required to have buffer space for only one incoming IP datagram per connection, but may optionally have space for more. Datagrams received when this buffer is full are silently discarded.

The data part of the datagram, without the IP header, will be copied to the address pointed by HL. The entire datagram data part will be copied if its length is smaller or equal than the "maximum data size" parameter passed in DE; otherwise, only the first DE bytes will be copied and the rest of the datagram data will be discarded (DE=0FFFFh can be specified with the meaning of "get the whole datagram"). No data will be copied at all if DE=0. Remember that it is possible to check the datagram data size in advance by using TCPIP_RAW_STATE.

Information about the originator of the datagram will be returned in registers HL and DE. Register BC will contain the datagram data size as it was received, which may be larger than the number of bytes actually retrieved.

ERROR CODES

- ERR_OK

The datagram data and information has been retrieved.

- ERR_NOT_IMP

The *"Open raw IP connections"* capability flags is not set. The implementation does not support raw IP connections.

- ERR_NO_CONN

There is no connection open with the specified number.

- ERR_NO_DATA

There are no pending incoming datagrams for the specified connection.

4.7. Configuration related routines

These routines allow to configure various working parameters of the implementation.

4.7.1. TCPIP_CONFIG_AUTOIP: Enable or disable the automatic IP addresses retrieval

```
Input:  A = 25
        B = 0: Get current configuration
            1: Set configuration
        C = Configuration to set (only if B=1):
            bit 0: Set to automatically retrieve
                   local IP address, subnet mask and default gateway
            bit 1: Set to automatically retrieve DNS servers addresses
            bits 2-7: Unused, must be zero
Output: A = Error code
        C = Configuration after the routine execution
            (same format as C at input)
```

This routine configures how the various IP addresses used by the system should be obtained. Two groups of addresses can be configured sepparately: the first group is composed of the local IP address, the subnet mask and the default gateway address (the latter two are ignored when they make no sense for the implementation, for example in case of a PPP link); the second group is composed of the primary and secondary DNS server addresses.

This routine is intended to be called in two scenarios:

- To set "automatic" when the current setting is "manual" and all the involved addresses are 0.0.0.0. In this case, the implementation must try to automatically obtain the addresses from a remote source (tipically a DHCP server); in case of failure in this procedure, the involved addresses must remain to 0.0.0.0. The implementation should retry the address retrieval procedure a reasonable number of times because giving out in case of failure.

- To set "manual" when the current setting is "automatic" and no IP addresses have been obatined yet (that is, the involved addresses are still 0.0.0.0).

Behavior in other cases in undefined. In particular:

- When changing from "manual" to "automatic" and the involved IP addresses already have a value, the implementation may either discard the addresses values and start the address retrieval procedure, or use the already configured addresses as if they were obtained automatically.

- When changing from "automatic" to "manual" and the involved IP addresses already have a value, the implementation may either maintain the obtained addresses or reset them to 0.0.0.0.

Addresses on a group not set to automatic retrieving are expected to be manually set by the user, by using the TCPIP_CONFIG_IPS routine, after the implementation is installed/initialized; otherwise they will remain with the value 0.0.0.0 indefinitely.

Implementations supporting automatic address retrieval should default to automatically retrieve all addresses when installed/initialized.

ERROR CODES

- ERR_OK

The setting has been successfully changed or the current setting has been returned.

- ERR_NOT_IMP

The *"Automatically obtain the IP addresses"* capability flags is not set. The implementation does not support automatical retrieval of IP addresses.

- ERR_INV_PAR

An invalid value for B or C has been specified at input.

4.7.2. TCPIP_CONFIG_IP: Manually configure an IP address

```
Input:  A = 26
        B = Index of address to set:
            1: Local IP address
            2: Peer IP address
            3: Subnet mask
            4: Default gateway
            5: Primary DNS server IP address
            6: Secondary DNS server IP address
        L.H.E.D = Address value
Output: A = Error code
```

This routine allows to manually set the value of one of the IP addresses used by the system. The addresses group of the involved address should have been previously configured as "manual setting" (see TCPIP_CONFIG_AUTOIP routine), otherwise the behavior is undefined.

The addresses are supplied in the format L.H.E.D. For example, 1.2.3.4 would be specified as HL=0201h, DE=0403h.

All addresses should default to 0.0.0.0 when the implementation is installed/initialized, unless the implementation installer provides a mechanism to explicitly set the IP addresses at install time.

This routine is intended to be invoked when the address is configured for manual setting (see the TCPIP_CONFIG_AUTOIP rotuine). If invoked then the address is configured for automatic retrieval, the behavior is undefined: the implementation may either replace the obtained address with the one supplied, or may simply ignore the routine call and preserve the previous address.

ERROR CODES

- ERR_OK

The address has been successfully set.

- ERR_INV_PAR

An invalid value for B has been specified at input, or the specified address type does not

make sense for the implementation (for example the subnet mask or the default gateway when the link layer protocol is PPP, or the peer address on an Ethernet network).

4.7.3. TCPIP_CONFIG_TTL: Get/set the value of TTL and TOS for outgoing datagrams

```
Input:  A = 27
        B = 0: Get current values
            1: Set values
        D = New value for TTL (only if B=1)
        E = New value for ToS (only if B=1)
Output: A = Error code
        D = Value of TTL after the routine execution
        E = Value of ToS after the routine execution
```

This routine gets or sets the value of the TTL (Time to live) and ToS (Type of service) that are used by the implementation for all the outgoing datagrams. These values apply to all datagrams, it is not possible to set the value per datagram, per connection or per protocol.

It is not possible to change only the TTL or only the ToS. If only one of the values is to be changed, then the routine must be called with B=0, the desired value must be changed (D or E), and then the routine must be called with B=1.

Implementations should default to TTL=64 and ToS=0 when installed/initialized.

ERROR CODES

- ERR_OK

The values have been successfully set or obtained.

- ERR_INV_PAR

An invalid value for B has been specified at input.

- ERR_NOT_IMP

B=1 at input but the *"Explicitly set the TTL and TOS for outgoing datagrams"* capability flags is not set. The implementation does not support explicitly setting these values.

4.7.4. TCPIP_CONFIG_PING: Get/set the automatic PING reply flag

```
Input:  A = 28
        B = 0: Get current flag value
            1: Set flag value
        C = New flag value (only if B=1):
            0: Off
            1: On
Output: A = Error code
        C = Flag value after the routine execution
```

This routine gets or sets the value of the automatic PING reply flag. When it is ON, ICMP echo request messages (PING requests) received by the implementation are automatically replied (an ICMP echo reply message is sent in response, immediately or as soon as possible). When it is OFF, ICMP echo request messages received are ignored.

If this flag cannot be changed by the user (because PING requests are always replied or always ignored), this routine must return ERR_NOT_IMP when invoked with B=1, but must still return the appropriate flag value when invoked with B=0.

The value of this flag should default to ON (if supported) when the implementation is installed/initialized.

ERROR CODES

- ERR_OK

The flag value has been successfully set or obtained.

- ERR_NOT_IMP

B=1 at input but the *"Explicitly set the automatic reply to PINGs on or off"* capability flags is not set. The implementation does not support explicitly changing the flag value.

- ERR_INV_PAR

An invalid value for B or C has been specified at input.

4.8. Miscellaneous routines

These routines perform various tasks on the implementation.

4.8.1. TCPIP_WAIT: Wait for a processing step to run

```
Input:  A = 29
Output: A = Error code
```

Implementations of the TCP/IP UNAPI specification are expected to be resident programs, that is, software that runs in background, aside from the user's code running in foreground.

Usually, and especially for implementations which are not assisted by external hardware, the "run in background" behavior is achieved by hooking to the standard 50 or 60Hz timer interrup available on the MSX system. The TCP/IP processing (retrieving datagrams, processing them, sending new datagrams, updating timers, and so on) is then done in "processing steps", one per timer interrupt, in which a series of operations are performed depending on the available incoming datagrams and the state of timers and other internal variables.

Since the timer interrupts may happen at any moment, implementations need to take in account the case in which the interrupt happens in the middle of the execution of one of the exposed routines, so that the processing done during the interrupt does not conflict with the (possibly half-done) processing performed by the routine, especially when the internal state of the implementation is modified by the routine.

The easiest way to avoid such conflicts is to simply set a flag when one of the exposed routines starts its execution, and to reset it when the routine execution finishes. Then, when the implementation's timer interrupt service routine starts, it checks the flag, and when it is set, the processing step is delayed to the next timer interrupt.

This poses a problem: if the client software executes implementation routines in quick succession (for example, inside a loop that waits for incoming data from a connection), then no processing steps may be done in a long time, which can result in data loss or other misbehavior of the implementation.

The TCPIP_WAIT routine helps to solve this problem. On implementations that perform its internal processing in processing steps, this routine should block until the next processing step is finished. Client applications that repeatedly execute implementation routines in a loop should call this routine once per loop iteration, to ensure that the implementation's processing steps are performed.

On implementations that do not perform work based on processing steps (for example implementations whose processing is done entirely by external hardware and are therefore not tied to the timer interrupt), this routine can simply do nothing.

This routine should return with interrupts enabled.

Note for implementors. The easiest way to implement this routine is to look at the two byte system variable stored at address 0xFC9E. The value of this variable changes whenever a timer interrupt occurs. Thus, it is enough to continuously look at the variable value until it changes. Here is a sample implementation of this concept:

```
TIMER:          equ             #FC9E

WAIT:
        ei
        ld          de,(PREV_TIMER)
WAIT2:
        ld          hl,(TIMER)
        ld          (PREV_TIMER),hl
        ld          a,h
        cp          d
        ret          nz
        ld          a,l
        cp          e
        ret          nz
        jr          WAIT2


PREV_TIMER:
        dw          0
```

ERROR CODES

This routine never fails. ERR_OK is always returned.