# Pawn

embedded scripting language

# Ethernet interface
**for the**
# H0420 MP3 controller

**Version 1.7**

**December 2008**

*ITB CompuPhase*

Eerste Industriestraat 19–21, 1401VL Bussum, The Netherlands (Pays Bas);
telephone: (+31)-(0)35 6939 261
e-mail: info@compuphase.com, WWW: http://www.compuphase.com



Requests for corrections and additions to the manual and the software can be directed to ITB CompuPhase at the above address.

Typeset with TEX in the "Computer Modern" and "Palatino" typefaces at a base size of 11 points.

# Contents

# Introduction

The H0420 Ethernet interface is a combination of an extension board (hardware) and extended firmware (software). The extension board plugs onto the H0420 MP3 controller and it provides a RJ-45 connector for a standard Ethernet network cable. The extended firmware contains a set of network functions that you can use from the script.

No network functionality is hard-coded in the Ethernet interface. All network functionality is under control of the script. In its current release, the network interface supports the TCP/IP protocol suite with the following functionality:

- ◇ TCP/IP core protocols (IP version 4), including the ARP, ICMP and UDP protocols.
- ◇ Support for dynamic configuration through DHCP, and AutoIP in absence of a DHCP server; lease times are handled.
- ◇ Support for multicast IP addresses and group memberships.
- ◇ For interoperability with Microsoft Windows hosts, NetBIOS Name Server requests are handled; DNS look-up is also present.
- ◇ PING transmit & response handling, for network diagnostics.
- ◇ SYSLOG client, for sending informational messages.
- ◇ Support for the SNTP (network time) protocol for synchronizing the internal clock (the firmware supports both a time client and a time server).
- ◇ Flexible and extensible SNMP agent support.
- ◇ TFTP client and server for simple file transport (as well as a simple form of "push" streaming).
- ◇ HTTP client, for downloading files; HTTP server (single session) for status or configuration.
- ◇ FTP server (single session) for file transfer.
- ◇ Shoutcast / Icecast client for streaming MP3 audio from the network ("pull" streaming).
- ◇ RTP protocol for "push" streaming of MP3 audio from the network.

## Usage

All scripts that use the network features must include the definition file (or "header file") for the network functionality. These scripts should have the following line near the top of the script:

Listing: **Initializing the network interface**

```
#include <tcpip>
```

Before using any of the network functions, the network interface must be initialized. This is done through the function `netsetup`. There are two ways to use `netsetup`: you can either give only a host name and have `netsetup` look up the network configuration from a DHCP server, or you can supply all the necessary information for a "fixed addressing" scheme. Examples are:

Listing: **Initializing the network interface**

```
// host name is MP3-Ctrl; IP address, gateway, DNS and netmask are
// looked up from DHCP
netsetup .hostname = !"MP3-Ctrl"

// host name is H0420, IP address = 192.168.0.123,
// gateway = 192.168.0.77, DNS = 192.168.0.99, netmask = 255.255.255.0
netsetup !"192.168.0.123", !"192.168.0.77", !"192.168.0.99",
         !"255.255.255.0", !"H0420"
```

If desired, the network can be cleaned up again with function `netshutdown`. However, this is rarely needed.

When initializing the network using DHCP, note that function `netsetup` returns *before* the DHCP handshaking is complete and the suitable addresses have been assigned. When the network status changes —such as DHCP completion, the script receives the event `@netstatus`. By implementing this function, the script can monitor network status, network errors and transfer progress. The function `netinfo` returns dynamic and static network information.

## Low-level interface

The network interface provides function for the low-level TCP/IP interface and for a selected set of the higher level protocols. The lower level interface allows to send and receive raw messages or data between the H0420 and external devices. Both the connection oriented TCP protocol and the datagram protocol UDP are supported. For opening a connection, use the function `netconnect` and for closing it use `netclose`. Only TCP connections need to be opened; UDP messages can be sent and received without opening a port. For sending a message, use `netsend`; and incoming data will be received by the event function `@netreceive`.

If you wish to act as a server, rather than a client, the script should call `netlisten` rather than `netconnect`. TCP connections that are "listened" to also need to be closed with `netclose`. For UDP servers, you must also call `netlisten` (unless you wish to listen to the default port 9930), but there is no need to close the connection.

Below is a skeleton of a script that implements a simple Telnet server. A Telnet server sets up a listening connection at port 23 and exchanges text messages with a Telnet client. The messages that a server receives are usually commands.

Listing: **Telnet server skeleton**

```
#include <tcpip>

@reset()
    {
    netsetup            /* configure the network using DHCP */
    }

@netstatus(NetStatus: code, status)
    {
    switch (code)
        {
        case NetAddrSet:
            {
            /* set up a listener on successful initialization */
            netlisten 23, TCP
            }
        }
    }

@netreceive(const buffer[], size, const source[])
    {
    if (size == 0)
        {
        /* special case, remote host just connected;
         * print a welcome message
         */
        netsend !"Welcome\r\n# ", _, source
        }
    else
        {
        /* normal case, data received */
        static line[100 char]
        strcat line, buffer
        if (strfind(line, "\r") >= 0 || strfind(line, "\n") >= 0)
            {
            /* we have received a full line, process it here */
             (... code omitted ...)
```

```
            line[0] = '\0'   /* prepare for next buffer */
            }
        }
    }
```

The script starts with setting up a network. Since the network is set up without any configuration options, the host must negotiate an IP address and other options via DHCP (if available) or AutoIP. When this negotiation ends, the script receives the `@netstatus` event with code `NetAddrSet` and the network configuration is complete. At this point, the script can set up a listener (function `netlisten`). As a side note: when using fixed addressing, network configuration is complete immediately after the call to `netsetup`.

Function `@netreceive` gets an event if data is received. The data may arrive character by character, or it may arrive is blocks or text lines (this is how the Telnet protocol works). The `@netreceive` function must collect the blocks of data and process any full line that is received. Any response from the script can be sent via `netsend`.

Immediately after a remote Telnet client connects, `@netreceive` also receives an event, but without any data. It is up to the script to decide how to respond. For a Telnet server, it is common to print a welcome message and a prompt.

Not shown in the skeleton is the way to close the connection. If the remote Telnet client closes the connection, there is nothing for the script to do: the listening socket will be notified about the closed connection. If the script must take the initiative to closing the connection, however, it must call `netclose` on the socket that was returned by the earlier call to `netlisten`. If you wish to accept a subsequent (new) incoming connection after having closed the active connection, the script should call `netlisten` again after the call to `netclose`.

## High-level interface

The firmware has built-in protocol handlers for the following services:

- ◇ TFTP client — `netdownload` or `netupload`
- ◇ TFTP server — `@nettransfer`
- ◇ HTTP client — `netdownload`
- ◇ HTTP server — `@nettransfer`

| | |
|---|---|
| ◇ FTP server | `@nettransfer` |
| ◇ Shoutcast / Icecast client | `netstream` or `play` |
| ◇ RTP client | `netstream` or `play` |
| ◇ Syslog client | `netsyslog` |
| ◇ SNTP client | `netsynctime` |
| ◇ SNTP server | *automatic* |
| ◇ ICMP client (ping only) | `netping` |
| ◇ ICMP server (ping only) | *automatic* |
| ◇ SNMP agent | `@netsnmp` |
| ◇ SNMP traps | `netsnmptrap` |

To enable a file server, the script must implement the function `@nettransfer`. The SNTP and ICMP servers are always enabled, and they allow a host on the network to query the time of the H0420 device and to "ping" the H0420. Function `netdownload` allows to download from both HTTP and TFTP servers. The function gets the protocol to use from the URL.

When you call the functions `netsynctime` or `netping`, the reply of the remote host is received as an event, through `@netstatus`. The functions `netsynctime` and `netping` are asynchronous: they return immediately (before a reply from the remote host is received).

## MP3 audio streams

The H0420 E-series can play MP3 audio that is streamed to the device. There are four protocols for streaming: direct streaming via TFTP, direct streaming via RTP, and buffered streaming with a progressive HTTP protocol (e.g. Shoutcast), and buffered streaming via standard HTTP. TFTP streaming and HTTP streaming (progressive or standard) can be used concurrently.

### • Progressive HTTP versus standard HTTP

Progressive and standard HTTP streaming have are similar in that the script uses functions `play` or `netstream` in both cases and that a stream queue must be prepared in both cases.

There are also important differences. To begin with, the server set-up is different: you need a HTTP server for standard HTTP streaming and a Shoutcast/Icecast server for progressive HTTP. Standard HTTP streams

play MP3 *files* over the network, from start to finish —you do not have the option start at an arbitrary position in the file. The "standard" HTTP streaming is therefore not suitable for live streaming.

The main advantages of standard HTTP streaming are that HTTP servers are more readily available (e.g. in "shared hosting" accounts) than streaming audio servers, and that standard HTTP streaming allows the client (i.e. the "web radio") to choose the tracks to play; a progressive HTTP stream plays back what the server pushes into the channel.

To use either kind of HTTP streaming, first the CompactFlash card must be prepared. The card must contain a file called "stream.swp" of an appropriate size and this file *must be unfragmented*. The H0415E/2 product comes with a utility, `PrepareStream`, that creates a stream file of an appropriate size that complies with the requirements for HTTP streaming. The utility can be found on the CD-ROM that comes with the product.

When using progressive HTTP, a relatively small stream queue of 512 kiB is sufficient. For standard HTTP streaming, a larger queue has the advantage that the complete track is downloaded in "burst mode" when it fits in the stream queue —the advantage is that quick downloads are less prone to dropped or stalling connections. In general, if you can spare the space on the CompactFlash card, a big stream queue is best.

- **Streaming with progressive HTTP**

The most common streaming method is a variation on the protocol used by all web browsers (Mozilla Firefox, Internet Explorer, Opera, etc.): the HTTP protocol. For MP3 streaming, ubiquitous stream servers are Shoutcast and Icecast, both of which use the progressive HTTP protocol.

Progressive HTTP is more suitable for streaming over a WAN or the Internet because it is buffered. This, in turn, requires that a suitable queue is prepared on the CompactFlash card —see the preceding section. For progressive HTTP, a stream queue size of 512 kiB works well in most cases, but larger stream queues never hurt. You can optionally also monitor the queue status to decide when to start playing the stream.

Like standard HTTP, progressive HTTP is a "pull" protocol: the H0420 initiates the connection to a stream server.

You connect to a stream with the function `netstream` or function `play`. Both functions start filling the stream queue and both start playing audio

from the stream queue when it reaches a certain level. Function `netstream` allows you to specify how many kilobytes must be in the stream queue before starting to play the stream (function `play` fixes this at 128 kiB). In addition, `netstream` can buffer (or re-buffer) a stream while audio is still playing — `play` will stop audio output before starting up the stream.

With `netstream`, you can select at which queue level you wish to start playing the stream. When you wait until the stream queue is 256 kiB full, you are relatively insensitive to network stalls (due to congestion or bad reception), but there is a high "latency" between the connection to the stream and the audio actually coming out of the speakers. This latency is because the queue needs to be filled first. You can choose to reduce the latency by starting to play the stream at a queue level of 32 kiB, at the risk that a network stall causes a gap in the audio or a disconnection from the stream.

The number of seconds in the stream queue depends on the amount of data in the queue and the bit rate. At the common MP3 bit rate 128 kb/s, the player processes 16,000 bytes per second.

A Shoutcast server will typically enter "burst mode" immediately after establishing a connection. In burst mode, the server sends up to 256 kiB as quickly as possible, and then switches to stream mode where the transfer speed is equivalent to the audio bit rate. Although newer Icecast servers also use burst mode, an older Icecast server streams at the speed of the audio bit rate from the very beginning. If you know that you are connecting to an old Icecast server, you may wish to fill the queue to 256 kiB before starting to play the stream. Similarly, for a Shoutcast server, you may start to play at a queue fill level of 64 kiB, because the queue will grow quickly in burst mode. If you do not know what server the device connects to, waiting until a fill level of 128 kiB is a fair trade-off: it is a safe margin for an Icecast server, and not cause a great delay for a Shoutcast server —it fills the queue to this level quickly anyway, because of burst mode.

With function `play`, all that is required is that you pass in an URL to the stream. The URL prefixes "`http://`" and "`icy://`" are equivalent, except that the default port number for "`http://`" is 80 and that for "`icy://`" is 8000.

Listing: **Streaming with HTTP**

```
play !"icy://224.82.71.81:8080/"
```

The H0420 supports meta-data in the stream. This meta-data is textual data, usually containing the title of the song and the name of the artist or the band, that the streaming server inserts into the audio stream at regular intervals. When a stream is playing, a script can retrieve that data from the function `taginfo`. Although meta-data is not technically a "tag", the two concepts share the same purpose and most streaming servers retrieve the meta-data from the tags of the tracks.

### • Restarting a HTTP stream

The `netstream` function is more specialized than function `play` for streaming: it has a parameter for the amount of data (in kiB) in the stream queue before playing starts and it can start buffering a stream while audio is still playing. The previous section already discussed the relation between the queue fill level and audio latency. This section focusses on the second feature —which is particularly useful for reliable streaming from progressive HTTP servers (Shoutcast/Icecast servers).

HTTP is a simple protocol on top of TCP. There are no particular reasons why a TCP connection may not be kept open indefinitely, but the protocol was not designed for continuous never-ending transfers. In practice, TCP connections get dropped on occasion. This may happen, among other reasons, because of server load or time-outs in NAT routers, a gateway in the middle (a "hop" ) that goes off-line, or a host switching to a different network (this happens with mobile devices that are "on the road").

When the H0420 is playing a stream and the connection for the stream gets disrupted, the H0420 will continue to play the remainder of the audio in the stream queue. No new data will arrive into the queue, however. The only way to "fix" a broken connection is to set up a new connection and restart the stream. The advantage that `netstream` has to `play` in this situation is that `netstream` can continue to play the remainder of the stream while the stream is restarted. In other words, `netstream` avoids (or at least minimizes) a silent gap during the re-opening of the stream.

The following code snippet illustrates a the concept:

Listing: **Monitoring and restarting a HTTP stream**

```
const StreamUrl[]       = !"icy://192.168.1.22"
const StreamBufferLimit = 128
```

```
@main()
    {
    netsetup
    settimer 1000
    }

@timer()
    {
    static StartDelay = 0
    const LowBufferLimit    = StreamBufferLimit / 4

    if (netinfo(LinkStatus) != 0 && netinfo(GatewayIP) != 0)
        {
        if (StartDelay == 0 && netinfo(StreamQueue) < LowBufferLimit)
            {
            StartDelay = 10
            netstream StreamUrl, StreamBufferLimit
            }
        }

    if (StartDelay > 0)
        StartDelay--
    }
```

The script initializes a timer. The event function `@timer` checks whether network is ready. The script assumes that a DHCP server is present, so that it will have a gateway address once the DHCP handshake completes.* The first time that it drops through the first "`if`" that checks the `LinkStatus` and the `GatewayIP`, the fill level of the stream queue is zero bytes. It will therefore drop through the second "`if`" as well and start the stream. It also sets a local variable, `StartDelay`, because on the next timer event —one second later, the stream has just started and the stream queue may not have received the first 32 kiB of the stream data yet.† We should give the stream a chance to fill the queue. Hence, the script makes sure that it does not restart a stream within 10 seconds since the last start.

When the stream is playing, the queue fill level will normally stay relatively stable, and that level will be either close to the queue limit set in function `netstream`, or it may be higher if the streaming server uses a burst mode to a higher fill level. If the stream queue fill level drops below 25% of the

* an alternative would be to implement the `@netstatus` function and wait for the `NetAddrSet` event, see page 3.

† Since `StreamBufferLimit` is defined at 128 kiB, `StreamBufferLimit` divided by 4 is 32 kiB.

level set in `netstream`, the connection probably has a problem. The script detects this situation and restarts the stream.

If a reconnection succeeds, the H0420 picks up the stream from the server again. If the reconnection was quick enough to avoid the stream queue to empty completely, there will be no gap in the audio (i.e. no silent period). However, due to the buffering scheme of progressive HTTP streaming, the position in the track where the stream is picked up will not match precisely the position where the connection was broken. As a result, there will be a glitch in the audio shortly after the successful reconnection.

Restarting a stream is only useful when the server uses burst mode. If the server *does not* use burst mode, the stream queue receives new data at the bit rate of the audio, which means that the stream queue cannot grow and play at the same time. Restarting a stream is also only useful for *progressive* HTTP streaming: when restarting a standard HTTP stream, the stream restarts from the beginning of the track, which is not what you want.

- **Tips for progressive HTTP streaming**

◇ To keep playing a local track while the stream queue fills up, use `netstream` instead of function `play`.

◇ To detect a disconnection from the stream, implement the event function `@audiostatus` and watch for the "`Stopped`" signal. If this signal arrives and you were streaming, the stream was disconnected.

◇ While playing a stream, you can also monitor the fill level of the stream queue with function `netinfo` and call `netstream` on the same stream again when it drops below a certain level. Doing this *refreshes* the stream.

◇ To signal a failed connection to a stream:

a) check the return value of `netstream`; it returns `false` if it cannot connect to the server;

b) `@netstatus` gets the event `NetStreamBuffer` with status 0 (stream queue 0% full), which means that the remote stream server replied with an error or reset the connection.

◇ To monitor the level to which the stream queue is full, call `netinfo` with code `StreamQueue`.

◇ To abort a stream, call `netstream("")`. This stops the stream. The audio will continue playing for a few seconds, because there is likely still data in the stream queue. You can wait until it runs out, or call the function `stop`.

**• Streaming via TFTP**

The simplest way to "stream" an MP3 file to the H0420 is to transfer it to the H0420 with the TFTP protocol and the destination filename `"stream:"` Note that the trailing colon in the name is required. For streaming over TFTP, all that is required is a TFTP client that allows you to set a specific target filename.

TFTP streaming is practical to send announcements by the network to players that normally play tracks from a CompactFlash card or use progressive HTTP streaming. Since the TFTP streaming method is non-buffered, it is not suitable for networks with a high latency, such as a WAN or the Internet.

TFTP streaming, as implemented in the H0420, is a "push" protocol, which means that the remote host initiates the streaming connection. In other words, you tell the H0420 what file to stream in (by starting a TFTP session); the H0420 does not ask for the stream.

In contrast to progressive HTTP streaming, audio data that is streamed over TFTP does not pass through the CompactFlash card. The TFTP server implemented in the H0420 uses the lock-step mechanism in the TFTP protocol to accept the audio data in when needed.

**• Streaming with RTP**

The "Real-time Transport Protocol" (RTP) is designed for quick transfer of multimedia data, where transfer speed is more important than integrity of the data. Occasionally, a packet with audio information may get lost with RTP. On the other hand, latency is much lower than in *reliable* transport protocols such as HTTP and the protocol overhead is lower too —which also reflects in lower bandwidth requirements. RTP is furthermore a suitable protocol for multicasting, which may significantly reduce bandwidth requirements.

There are various devices that can stream audio data onto the network using RTP. A PC application (on Microsoft Windows) that creates an RTP stream from MP3 tracks is "LiveCaster".

RTP is a non-buffered "push" protocol. No "stream queue" needs to be prepared on the CompactFlash card, and no stream needs to be initialized. To play an RTP stream, the script only needs to call the standard function `play` with an RTP URL instead of a filename. For example, the following snippet starts playing the stream from the server at "224.82.71.81" on port 56952:

Listing: **Streaming with RTP**

```
play !"rtp://224.82.71.81:56952/"
```

No standard port is defined for the RTP protocol, which is why you usually have to give an explicit port number. If you omit the port, the H0420 MP3 controller uses port 5004 for RTP packets.

The controller automatically detects multicast addresses and sends out a multicast group announcement for the service if needed. If the remote address is an unicast address, no group announcement is sent.

The H0420 MP3 controller is compatible with the Barix extension of the RTP protocol, where the host has to request the stream from the server first. The Barix RTP variant is often better able to get audio data through a NAT router than the standard RTP protocol, but it may be limited to unicast applications. To use the Barix RTP variant, specify the protocol prefix "`brtp://`" in the `play` command (instead of "`rtp://`").

## Transferring files

The script supports the HTTP protocol for downloading files from a web server and the TFTP protocol for downloading and uploading files from and to a TFTP server. Authenticated file transfers are currently not supported.

To initiate the file transfers, the script uses the functions `netdownload` and `netupload`. These functions are *asynchronous*, meaning that the function returns *before* the file transfer is complete. Once the transfer completes, the script receives an event through the `@netstatus` function —the respective event codes are `NetHttpDone` and `NetTftpDone`.

These functions initiate the file transfer and thereby act as a "client". The script can also wait for an incoming request (from a remote host) to transfer a file, by setting up a server. See the section "HTTP, FTP and TFTP servers" on page 16 for this functionality.

## Monitoring and configuration with SNMP

SNMP stands for "Simple Network Management Protocol". This protocol allows remote monitoring and configuration of network devices. For this to work, the network device must be equipped with an SNMP agent. To implement an SNMP (version 1) agent in the H0420, you need a script that contains the `@netsnmp` function and a MIB file.

With SNMP, a *monitor* sends out queries at regular intervals to request the status of one or more parameters of one or more devices. The A query may also contain a new value for a parameter. Each device contains an SNMP agent that receives the queries and responds to it. This is the task of the `@netsnmp` function: return and optionally change values of requested parameters.

SNMP works with "communities", where the name of a community functions as a password. The SNMP browser sets the community name and the SNMP agent decides whether that community name is given read or write access —or neither. See function `@netsnmpcfg` to set community strings for the SNMP agent in the H0420.

For reasons of efficiency, SNMP exchanges all device parameters as numbers. So 1 may stand for "device status" and 12 for "current volume setting". An SNMP browser or SNMP monitor that you use on your workstation to control the device shows the same parameters with their names. To map "magic" numbers to human-readable names (and vice versa), the SNMP browser/monitor needs a MIB file.

The MIB ("Management Information Base") file is a plain text file that contains the definitions of the settings that the H0420 MP3 controller can return. These settings depend on the script. You can build a script that allows a user to select tracks, set volume and balance and other audio parameters, or build a script that allows a user to query information such as *up-time*, network traffic and recent status changes. The script, and in particular the event function `@netsnmp`, determine how the H0420 MP3 controller responds to queries and which requests it supports.

Obviously, the definitions in the MIB file must be in conformance with the implementation of the `@netsnmp` function in the script. Part of the MIB file needed for the H0420 is fixed, but another part is flexible because the scripting capabilities of the H0420 are flexible too.

- **The MIB file**

The template MIB file, onto which you will base your specific MIB files is below. You will find this template MIB file on the CD-ROM that comes with the product (in the "examples" subdirectory).

Listing: **Template MIB file**

```
--
-- A template SNMP MIB file for use with the H0420
--
-- Copyright (c) 2007-2008 ITB CompuPhase
--

-- ================================================================
-- This part should remain unchanged
-- ================================================================
COMPUPHASE-MIB DEFINITIONS ::= BEGIN

IMPORTS
        enterprises, IpAddress, Counter, TimeTicks
                FROM RFC1155-SMI
        OBJECT-TYPE
                FROM RFC-1212
        DisplayString
                FROM RFC-1213;

compuphase    OBJECT IDENTIFIER ::= { enterprises 28388 }
products      OBJECT IDENTIFIER ::= { compuphase 1 }
h0420         OBJECT IDENTIFIER ::= { products 20 }

-- ================================================================
-- The part below is specific to the application, and it must be
-- in conformance with the script
-- ================================================================

-- Add your definitions here...

-- ================================================================
-- End of the application-specific definitions
-- ================================================================

END
```

The definitions in the MIB file are written in "Abstract Syntax Notation One", or ASN.1. Information on the ASN.1 syntax can be found in various books and on the Internet, including tutorials and the original definitions in RFCs.

When writing the MIB file, please note that the H0420 implementation of the SNMP agent only supports whole numbers and (octet/character) strings. The H0420 does not support "sequence" types for user data. In the MIB file,

you may also use derived types as `Counter`, `Gauge`, `TimeTick` and `IpAddress`, which are basically different representations of integer values.

Below is a very brief implementation of the `@netsnmp` function. It handles only two fields: the title of the track currently playing (this is a read-only) property and the volume level —a read-write property.

Listing: **Minimal SNMP agent**

```
@netsnmp(item, data[], size)
    {
    switch (item)
        {
        case 1: // title, read-only
            taginfo ID3_Title, data, size

        case 3:
            if (size == 0)
                setvolume strval(data)
            else
                {
                new value
                getvolume value
                strformat data, size, true, !"%d", value
                }

        default:
            return false
        }
    return true
    }
```

The definitions to put in the MIB file are below (these definitions must be merged in the template MIB file, see page 14):

Listing: **MIB file extract, matching the above minimal SNMP agent**

```
Title OBJECT-TYPE
SYNTAX   OCTET STRING
ACCESS read-only
STATUS mandatory
DESCRIPTION "Track title"
::= { h0420 1 }

Volume OBJECT-TYPE
SYNTAX   INTEGER(0..100)
ACCESS read-write
STATUS mandatory
DESCRIPTION "Audio volume (0..100)"
::= { h0420 3 }
```

## HTTP, FTP and TFTP servers

To enable the built-in HTTP, FTP and/or TFTP servers, the script must implement the `@nettransfer` function. The HTTP, FTP and TFTP protocols are file transfer protocols. The FTP and TFTP servers allow read and write requests, while the HTTP server only supports read requests (i.e. "downloads" or page views). Only the FTP server requires a log-in before allowing file transfers. The script may optionally also implement the `@netstatus` function to receive an event on the completion of the transfer.

To have the script initiate the file transfer itself, rather than wait for an incoming request, see section "Transferring files" on page 12.

The purpose of the `@nettransfer` function is to let the script either allow or refuse the request. In the case of a HTTP server, the script may also process any parameters on the URL (before acknowledging or cancelling the transfer).

### • TFTP server

The following implementation of `@nettransfer` enables the TFTP server, but cancels any HTTP requests that it receives. Read and write requests are accepted in the "user" subdirectory, and cancelled for other areas on the memory card of the H0420.

Listing: **Handling TFTP requests**

```
bool: @nettransfer(path[], NetRequest:code, socket)
    {
    /* HTTP requests are denied (only accept HTTP requests) */
    if (code != NetTftpGet && code != NetTftpPut)
        return false

    /* only up/downloading to/from "user" is allowed */
    if (strcmp(path, !"user/", true, 5) != 0)
        return false

    return true         /* allow this transfer */
    }
```

TFTP has no concept of a "current directory". Instead, the full path of the filename to "put" or to "get" must be specified. Some TFTP clients allow you to type in only a single name, for both the source and the destination. This is inconvenient if you wish to transfer a file to or from a different directory on the PC than on the memory card of the H0420. A free TFTP client that allows separate paths and names for the local and remote hosts is TFTPD32 by Philippe Jounin.

- **HTTP server**

From the viewpoint of the PAWN script is a web server very similar to a TFTP server. For a HTTP server, you also need to implement the `@nettransfer` function, but now enabling the HTTP requests instead of (or in addition to) the TFTP requests.

HTTP clients, such as a browser like Mozilla Firefox or Microsoft Internet Explorer, may pass parameters to a server accompanying the request. The H0420 supports URL parameters on "GET" requests and passes the full URL to the `@nettransfer` function. In `@nettransfer`, you can process and save these parameters. The browser may then obtain the processed results with a subsequent file transfer or through an embedded request using the XMLHttpRequest method supported by most browsers.

Listing: **Handling HTTP requests**

```
bool: @nettransfer(path[], NetRequest:code, socket)
    {
    if (code != NetHttpGet)
        return false    /* deny non-HTTP transfers */

    /* get and save any parameters */
    new idx = strfind(path, !"?");
    if (idx >= 0)
        {
        new params[100 char]
        strmid params, path, idx + 1
        /* write the parameter in a file (without further processing) */
        new File: handle = fopen(!"params.txt", io_write)
        if (handle)
            {
            fwrite handle, params
            fclose handle
            }
        }

    return true         /* allow this transfer */
    }
```

The script presented above saves any parameters into a text file, without processing the parameters in any way. If you do not plan to handle URL parameters, you can remove the entire section —making the `@nettransfer` as simple as:

Listing: **Handling HTTP requests ignoring any URL parameters**

```
bool: @nettransfer(path[], NetRequest:code, socket)
    return (code == NetHttpGet)         /* allow HTTP, deny TFTP */
```

- **FTP server**

Like the HTTP and TFTP servers, the FTP server passes through the `@nettransfer` function. In the implementation of this function in the PAWN script, it must respond to several FTP requests. The FTP protocol has a login handshake, and it allows you to set one or more usernames and passwords for all users that you wish to grant access. Only one user can be connected to the FTP server at a time.

After login, the `@nettransfer` function may also allow or block any file transfer command (upload or download) as well as file deletion. In addition, the FTP server supports the SITE command, which you can use to send arbitrary commands to the script from within an FTP client (not all FTP clients support the SITE command).

Listing: **Handling FTP requests**

```
bool: @nettransfer(path[], NetRequest:code, socket)
    {
    switch (code)
        {
        case NetFtpLogin:
            {
            /* read the username:password string from an INI file */
            new ftplogin[30 char]
            readcfg .key=!"ftplogin", .value=ftplogin, .pack=true

            /* accept a matching login, or accept all login's if
             * no username:password was set in the INI file
             */
            return strlen(ftplogin) == 0 || strcmp(path,ftplogin) == 0
            }

        case NetFtpGet, NetFtpDelete, NetFtpPut:
            return true /* accept all file commands */

        case NetFtpCmd:
            if (strcmp(path, !"RESET") == 0)
                {
                reset   /* host command = "SITE RESET" */
                return true
                }
        }

    return false        /* deny all non-FTP transfers */
    }
```

# Function reference

## Public functions

**@netreceive** A data packet is received

Syntax: `@netreceive(const buffer[], size, const source[])`

| | |
|---|---|
| `buffer` | The data received. Depending on the protocol, this may be text or numeric data. See the notes, below, for details. |
| `size` | The size of the data in `buffer`, in cells. Each cell holds four bytes or four characters. This parameter may be zero on a "passive connect", see the notes, below. |
| `source` | For UDP connections, this field is the IP address and the port number of the sender, where the IP address and the port are separated by a colon (for example: "`192.168.10.29:9930`"). For TCP connections, this field is a "`#`" followed by the socket number returned by `netlisten`. |

Returns: The return value of this function is currently ignored.

Notes: If the received data is ASCII text, parameter `buffer` holds a packed string that may not be zero-terminated. Use the `size` parameter to determine the number of cells of data in the buffer. If the received data is not text, it is assumed to consist of 32-bit values that are send in "network byte order" (Big Endian).

Before being able to receive packets, the script should call `netconnect` to open a connection, or call `netlisten` to allow a remote host to connect.

When the script is listening on a TCP socket and a remote device connects to this socket (i.e., a passive connect), the `@netreceive` function is called with the `size` parameter set

to zero. A script can use this special case to send a greeting message to the remote host on connect.

Example: See the Telnet server (skeleton) on page 3.

See also: `netlisten`

---

**@netsnmp** An SNMP request is received

Syntax: `bool: @netsnmp(item, data[], size)`

| | |
|---|---|
| `item` | The numeric identifier of the item. |
| `data` | Either the new data to write to the item (SET request), or the buffer to read the current value of the item into (GET request). |
| `size` | If zero, this is a SET request and `data` is a zero-terminated string that holds the new data for the item. If non-zero, this is a GET request and this parameter holds the size of the `data` array in cells. |

Returns: The function should return `true` if it can fulfil the request and `false` on failure. In particular, if `item` has an unknown or unsupported value, this function should return `false`.

Notes: The same function is used for querying parameters and for setting them. The distinction between the two operations is in the `size` parameter. If it is zero, the request is a SET operation; otherwise it is a GET operation.

The contents of parameter `data` may be a text string, a number or an IP address, depending on the definition of the item. For SET requests, numbers and IP addresses are encoded as text strings. For GET requests, the script should store the requested information in parameter `data` as a text string.

The definition of the type of each item is in the MIB file. It is the responsibility of the programmer to have a matching MIB file to the implementation of this `@netsnmp` function.

Example: See the code and associated MIB snippets on page 15.

See also: `netsnmptrap`

**@netstatus** Network status changed/event occurred

Syntax: `@netstatus(NetStatus: code, status)`

`code` The code of the event or status change; it is one of the following:

`NetLink` (0)
: Physical link status; parameter `status` = 0 (disconnected) or 1 (connected).

`NetPing` (1)
: Ping reply (see `netping`); parameter `status` = ping sequence number.

`NetAddrSet` (2)
: The IP address is set; this code is useful for DHCP configuration because is signals that the network is ready for sending and receiving packets; parameter `status` holds the IP address as a 32-bit integer value.

`NetTimeSync` (3)
: H0420 clock synchronized with remote host (this time is in UTC, you may need to adjust the clock for the time zone or daylight saving time); parameter `status` = 0.

`NetLeaseExp` (4)
: The DHCP lease is expired or the link-local lease is expired; parameter `status` = 0.

`NetTftpDone` (5)
: TFTP transfer has finished; parameter `status` = 0.

`NetHttpDone` (6)
: HTTP transfer has finished; parameter `status` = 0.

`NetStreamQueue` (7)
: Stream queue mark reached; parameter `status` = level (in kilobytes), it is zero

if the remote server rejected the stream request.

`NetFtpDone` (8)
: FTP transfer has finished; parameter `status` = 0.

`status`
: The value associated with the status, its meaning depends on the event code.

Returns: The return value of this function is currently ignored.

Notes: Link-local addresses have a fixed lease of 10 minutes. DHCP leases depend on the configuration of the DHCP server.

Example: See the Telnet server (skeleton) on page 3.

See also: `netclose`, `netinfo`, `netping`, `netsetup`, `netstream`, `netsynctime`

---

**@nettransfer** A file transfer request was received

Syntax: `bool: @nettransfer(path[], code)`

`path`
: The full path to the requested file, for HTTP this may include any parameters. The script may modify this parameter, which is useful for redirecting a file, for example.

`code`
: The code of the event or status change; it is one of the following:

`NetTftpGet` (1)
: The remote host requests to receive this file from the H0420, using the TFTP protocol.

`NetTftpPut` (2)
: The remote host requests to transmit this file to the H0420, using the TFTP protocol.

`NetHttpGet` (3)
: The remote host requests to receive this file from the H0420, using the HTTP protocol.

`NetFtpLogin` (5)
: The remote host requests to log in as an FTP user. The `path` parameter contains the username and password, separated with a colon ("*user:password*").

`NetFtpGet` (6)
: The remote host requests to receive this file from the H0420, using the FTP protocol.

`NetFtpPut` (7)
: The remote host requests to transmit this file to the H0420, using the FTP protocol.

`NetFtpDel` (8)
: The remote host requests to delete this file from the H0420 server (using the FTP protocol).

`NetFtpCmd` (9)
: The remote host has sent a SITE command. The `path` parameter contains the text of the SITE command, excluding the keyword SITE.

Returns: The function should return `true` if it can fulfil the request and `false` on failure.

Notes: On a GET request, if the file cannot be found, the TFTP, HTTP or FTP server in the H0420 will always return an appropriate error code. It is not necessary to verify the presence of the files.

Any parameters on the URL, for a HTTP request, may be used by the script to adjust settings. Web forms often use parameters on the URL to pass data from the client to the server.

If you do not implement this function, all TFTP, HTTP and FTP server requests are denied. The FTP server can only handle one user at a time. A login request while there is already a connection open is denied. Some modern FTP clients issue

a second (or third...) login for every file transfer; this option must be disabled for the FTP server in the H0420.

Example: See the code snippets on page 16 and page 17.

See also: `netdownload`, `netupload`

## Native functions

**netclose** — Close a socket

Syntax: `bool: netclose(socket)`

`socket` The socket number to close. This value must have been returned by an earlier call to a function that opens a socket (see `netconnect` and `netlisten`).

Returns: `true` on success and `false` on failure.

Notes: When closing a "listening" connection, the ability for remote hosts to connect is disabled. To close the active connection with a remote host, but remain available to new connections, call `netlisten` after the call to `netclose`.

See also: `netconnect`, `netlisten`

**netconnect** — Open a connection / socket

Syntax: `netconnect(const remote_addr[])`

`remote_addr` The IP address and (optionally) the port number to connect to. An example of a full address is "193.54.119.12:23", where the host is at IP address 193.54.119.12 and the service is at port number 23. If the port number is absent, the function connects to the default port 9930. Instead of an IP address, you may also give a domain name, as in "server.mydomain.com:2".

Returns: The function returns a socket number of the open is successful, or zero on failure.

Notes: This function opens a socket and sets up a transfer to a remote host. That is, it sets up an *outgoing* connection. See `netlisten` to handle *incoming* connections.

See also: `netclose`, `netsend`

---

**netdownload** Download a file

Syntax: `netdownload(const url[], const filename[]=!"")`

`url` The full network path of the file to download, preferably including the protocol prefix. For example, to download the file "loops.mp3" from www.soundclips.com, the URL would be: "http://www.soundclips.com/loops.mp3".

`filename` The local filename to store the downloaded file in. This name may optionally include a directory.

Returns: The function returns 0 on error (unable to connect to the host, or file not found) and a socket number on success.

Notes: To download from a HTTP server, use the protocol designator "http://". To download a file from a TFTP server, the protocol designator is "tftp://". TFTP transfers are usually faster than HTTP transfers, especially on local networks (LAN).

The function returns *before* the download is complete. When the download completes, you will receive the event `@netstatus` with code `NetHttpDone` or `NetTftpDone`. You can abort a transfer by calling `netclose` on the returned socket number.

See also: `@netstatus`, `netclose`, `netupload`

**netinfo** Get network status information

Syntax:
```
netinfo(NetInfo: code,
        string[]=!"", size=sizeof string)
```

`code` The kind of data to return, it must be one of the following:

`LinkStatus` (0)
: The status of the physical link: 0 if the device has no good (physical) connection to a network (LAN or WAN), and 1 if the physical link is present. A bad physical link usually indicates that the device is disconnected or that the cable is defective.

`IPaddress` (1)
: The IP address of this host.

`SubnetMask` (2)
: The subnet mask matching the IP address.

`GatewayIP` (3)
: The address of the gateway.

`DNS_IP` (4)
: The address of the domain name server.

`MACaddr` (5)
: The hardware (MAC) address; this information is only returned as a string.

`HostName` (6)
: The name of the H0420 device as known on the network; this item is only returned as a string.

`StreamQueue` (7)
: The level to which the stream queue is filled, in the context of progressive HTTP streaming. This value is in kilobytes, so when the return value is 98, there is 98 kiB left in the queue at the time of the call.

In case that you need to know the maximum size of the stream queue, use `fstat` on the file "`stream.swp`".

`PacketLoss` (8)
: The number of RTP packets lost since the last request; in the context of RTP streaming. This "lost packets" count is reset to zero after this call.

`LeaseTime` (9)
: The time that is left before the lease expires (in seconds).

`string` If provided (and of suitable length), the item is stored in a formatted way in this string.

`size` The size of the `string` parameter, in cells. Since the function will store the data in parameter `string` as a packed string, four characters fit into a single cell.

Returns: The requested value, or zero on error.

Notes: The function returns the data as a number (except for the codes `MACaddr` and `HostName`). If a string of suitable length is provided, the function also stores the value as a formatted number. IP addresses are stored in the `string` parameter as dotted numbers (for example: "192.168.1.16").

See also: `@netstatus`, `netsetup`, `netstream`

---

**netlisten** Open a "listening" connection

Syntax: `netlisten(port=9930, NetProtocol: protocol=UDP)`

`port` The number of the port to listen to. The default port is 9930.

`protocol` Must be either `TCP` or `UDP`.

Returns: The socket number, or zero on error.

Notes: A "listening connection" is needed to accept *incoming* connections. For *outgoing* connections, see `netconnect`. Both incoming and outgoing connections need the `@netreceive` function to handle received data. When a remote host connects to a listening socket, this is also called a "passive connect".

By default, a listening connection is already set up on the UDP port 9930. In order to listen to a different port, or to listen on a TCP connection, you need to call `netlisten` explicitly.

The function returns a socket number that was opened for the listener. To stop listening on the port, close this socket number with `netclose`. After closing a listening socket, an external host can no longer connect to the MP3 controller (and send it data). In order to close a connection and return to a listening state, first call `netclose` and then call `netlisten` again to set up a new listener.

Example: See the Telnet server (skeleton) on page 3.

See also: `@netreceive`, `netclose`, `netconnect`

---

**netping** "Ping" remote host

Syntax: `bool: netping(const remote_addr[], sequence=0)`

`remote_addr` The IP address or the domain name of the remote host to send a ping request to. No port number may be attached to the IP address or domain name.

`sequence` An arbitrary number that allows you to match the ping response to a request, in case you send multiple "pings".

Returns: `true` if the "ping" message could be sent, `false` if sending the message failed.

Notes: The first step in diagnosing a network problem often is to send a "ping" message. If the message can be sent and a reply is received within (at most) a few seconds, the core protocols of the TCP/IP protocol suite are working and the remote host is up.

If a call to `netping` fails, this indicates one of the following:

- ◇ **Physical connection down**: no cable is connected to the device, the cable is damaged, the network switch or hub is down, ...
- ◇ **No gateway**: the IP address in `remote_addr` lies in a different network than this host and the gateway is misconfigured or non-responding. This situation may also occur when this host has obtained a link-local address and it is trying to reach computers outside the link-local address range.
- ◇ **ARP failure**: the IP address in `remote_addr` is in the same network as this host, but the remote host does not respond to address look-up queries (ARP). This usually means that the remote host is down.
- ◇ **DNS/NetBIOS failure**: if you passed in a domain name in parameter `remote_addr` (instead of an IP address), this name could not be resolved to an IP address using DNS and/or NetBIOS queries.

Even if the "ping" message was transmitted successfully, function `netping` returns immediately after sending the ping request; it does not wait for a reply. If the remote host responds to the ping request, the returned answer will fire the event `@netstatus` with code `NetPing` and the `status` parameter set to the sequence number of the corresponding call to `netping`.

See also: `@netstatus`, `netinfo`

**netsend** Send a packet

Syntax:
```
bool: netsend(const buffer[], size=sizeof buffer,
              const remote_addr[])
```

`buffer` The data to send to a remote host.

`size` The size of the buffer in cells.

`remote_addr` Either an IP address and a port, for sending an UDP datagram, or a socket number for sending a TCP message —see the notes for details.

Returns: `true` on success and `false` on failure.

Notes: When sending an UDP message, the remote address should have the form like "193.54.119.12:23", where the host is at IP address "193.54.119.12" and the service is at port number 23. You may give a domain name, like "server.mydomain.com:23", instead of an IP address. If the port number is absent, the function connects to the default port 9930.

For sending a TCP message, the `remote_addr` parameter must contain only a socket number, optionally prefixed with a "`#`". For example, when sending on socket 3, `remote_addr` could have the value "`#3`". See `netsocket` to convert socket numbers to a string with a "`#`" prefix.

TCP connections must be set up before any data can be sent, see function `netconnect`.

The `netsend` function sends numeric data in parameter `buffer` as 32-bit values in "network byte order" (Big Endian). When sending text data, the text is padded to a multiple of four bytes (the size of a PAWN cell).

Example: See the Telnet server (skeleton) on page 3.

See also: `@netreceive`, `netconnect`, `netsocket`

**netsetup** Initialize the network

Syntax:
```
bool: netsetup(const ip_address[]=!"",
               const gateway_address[]=!"",
               const dns_address[]=!"",
               const subnet_mask[]=!"",
               const hostname[]=!"")
```

`ip_address` The IP address of this host (the MP3 controller), or empty to have it looked up from a DHCP server.

`gateway_address` The IP address of the gateway, or empty to have it looked up from a DHCP server.

`dns_address` The IP address of the DNS server, or empty to have it looked up from a DHCP server.

`subnet_mask` The network mask in "dotted format" (see below), or empty to have it looked up from a DHCP server.

`hostname` The name of this host. This name is used for the DHCP request and for the DNS and NetBIOS look-ups. If left empty, the standard name is "H0420".

Returns: `true` on success and `false` on failure.

Notes: All IP addresses should be in "dotted format", meaning four decimal numbers in the range of 0 to 255 separated by periods. An example is `192.168.10.29`.

You should avoid doing partial DHCP look-ups —either leave the first three parameters of this function empty, in order to have them provided by a DHCP server, or specify all three: the host IP address, the gateway address and the DNS server address. For common networks, the function can establish the network mask automatically, but if known, it is best to specify it as well.

If no IP addresses are given, and DHCP fails too, the H0420 assigns a "link-local" address to itself. Link-local addresses

are only valid inside a LAN (the link-local address range is non-routable). The H0420 will not have access to the Internet when it has a link-local address. The link-local address scheme is also known as "AutoIP" and "APIPA".

Example: See the code snippets on page 2 and page 8.

See also: `netshutdown`

---

**netshutdown** Close the network interface

Syntax: `netshutdown()`

Returns: This function currently always returns 0.

Notes: This function closes down the network support and frees all resources.

See also: `netsetup`

---

**netsnmpcfg** Set the communities (passwords) for SNMP

Syntax:
```
netsnmpcfg(const readonly_community[],
           const readwrite_community[])
```

`readonly_community`
: The password that allows reading (but not modifying) device values. The default string for this community is "public".

`readwrite_community`
: The password that allows modifying device values. The default string for this community is "private".

Returns: This function currently always returns 0.

Notes: See the section on SNMP on page 13 for more information on SNMP authentication and access rules.

See also: `@netsnmp`, `netsnmptrap`

**netsnmptrap** Send an SNMP trap

Syntax: `bool: netsnmptrap(const remote_addr[], trap, item=0, const value[]=!"")`

`remote_addr` The IP address or the domain name of the host to send the trap to.

`trap` The code for the trap. Predefined (standardized) trap numbers are:

`ColdStart` (0)
: Device power-up.

`WarmStart` (1)
: Device reset.

`LinkDown` (2)
: Network link is down.

`LinkUp` (3)
: Network link is up.

`AuthenticationFailed` (4)
: Authentication failed.

`EGPNeighborLoss` (5)
: Neighbour in the Exterior Gateway Protocol was lost.

See the SNMP standard for details on the standard traps.

Instead of a predefined trap number, you can also send a device-specific trap (this is called an "enterprise-specific" trap in the SNMP documentation.

`item` Parameter to which the trap relates (see the MIB file).

`value` New value of the `item` parameter, which caused the trap.

Returns: `true` on success, `false` on failure (trap could not be sent).

Notes: The MIB file must define all "enterprise-specific" traps with trap numbers 6 and higher. The SNMP implementation of the H0420 does not support enterprise-specific traps with numbers 0 to 5, because these are reserved for the standard traps (see parameter `trap`).

See also: `@netsnmp`, `netsyslog`

---

**netsocket** Make a socket string from a socket number

Syntax: `netsocket(value)`

`value` The socket number.

Returns: A string containing the character "`#`" followed by the text representation of the parameter `value`. For example, if parameter `value` is 5, this function returns the string `"#5"`.

See also: `netsend`

---

**netstream** Start buffering an audio stream

Syntax:
```
netstream(const url[], buffermark=128,
          bool: autoplay=true)
```

`url` The full network path of the file to download, preferably including the protocol prefix. The protocol prefix is "`icy://`" for Shoutcast and Icecast servers that are on the default port 8000. If the server uses port 80 instead, you may use the protocol prefix "`http://`", or add a port number explicitly.

`buffermark` The criterion for the fill level of the stream queue before starting playing the stream, in kilobytes. This must be at least 8. The stream queue file (a non-fragmented file called "stream.swp") must be at least 64 kiB larger than this `buffermark` value. See page 6 for details on the stream queue.

`autoplay` If `true`, the stream starts to play (output audio) as soon as the level in parameter `buffermark` is reached. When set to `false`, the public function `@netstatus` is still called with code `NetStreamQueue`, but no audio is output.

Returns: The socket number opened for the stream, or 0 on failure.

Notes: Many Shoutcast and Icecast servers publish only an URL to a playlist, which then in turn contains the URL to the audio stream. This function needs the latter: the URL to the audio stream. If you wish to use the playlist approach, your script can download it via `netdownload` and then parse through it with the file functions (the playlist is a standard text file).

When the stream queue reaches the indicated level, the event function `@netstatus` receives the `NetStreamQueue` event. By default, the stream also starts playing automatically (possibly interrupting a track that may be playing at the time). However, if parameter `autoplay` is set to `false`, the script must explicitly call function `play` with parameter `"stream:"` to start playing the stream.

To close a stream, call `netstream` with the `url` parameter set to an empty string.

Example: See the code snippet on page 8.

See also: `@netstatus`, `play`

---

**netsynctime** Request network time synchronization

Syntax: `bool: netsynctime(const remote_addr[])`

`remote_addr` The IP address or the domain name of the remote host to send the network time request to. No port number may be attached to the IP address or domain name.

Returns: `true` if the request for the network time could be sent, `false` if sending the request failed.

Notes: The function returns immediately after sending the request; it does not wait for a reply. If the remote host responds to the network time request, the returned answer will fire the event `@netstatus` with code `NetTimeSync`. The internal clock of the MP3 controller will also be set to the time that the remote host returns.

This function uses the protocol SNTP to synchronize the clock. This protocol returns the time in UTC (the current name for "Greenwich Mean Time"). To obtain the accurate local time, you need to intercept the `NetTimeSync` event (function `@netstatus`) and add the time zone offset to the time. With this procedure, you can also adjust for daylight saving time.

See also: `@netstatus`

---

**netsyslog** Send a system log message

Syntax:
```
bool: netsyslog(const remote_addr[], const mes-
                sage[], severity=0)
```

`remote_addr` The IP address or the domain name of the remote host to send the log message to. No port number may be attached to the IP address or domain name.

`message` The message to send to the Syslog server.

`severity` By convention, a value between 0 and 7, with the following meanings:
0 = emergency (system is unusable)
1 = alert (immediate action required)
2 = critical
3 = error
4 = warning
5 = notice (normal, but significant condition)
6 = informational
7 = debug

Returns: `true` on success, `false` if sending the message failed.

Notes: Syslog is an industry standard protocol used for capturing log information for devices on a network, usually via UDP Port 514. Syslog support is included in Unix and Linux based systems, but is not included in Microsoft Windows and MacOS. However, there are third-party applications available to add this capability to your system.

The function uses "local0" as the facility code in the Syslog message.

See also: `netsnmptrap`

---

**netupload** Download a file

Syntax: `netupload(const url[], const filename[]="")`

| | |
|---|---|
| `url` | The full network path where the file will be uploaded, preferably including the protocol prefix. To upload a file, with the name "loops.mp3", on the remote host at address 195.200.2.66, and using TFTP, the URL would be: "tftp://195.200.2.66/loops.mp3". |
| `filename` | The path to the local filename upload file in. |

Returns: The function returns 0 on error (unable to connect to the host, or file not found) and a socket number on success.

Notes: In the current version of the firmware, only TFTP is available as a protocol for upload data to an external server. However, for compatibility with future revisions, it is best to specify the protocol designator "tftp://" for all uploads.

The function returns *before* the upload is complete. When the upload completes, you will receive the event `@netstatus` with code `NetHttpDone` or `NetTftpDone`. You can abort a transfer by calling `netclose` on the returned socket number.

See also: `@netstatus`, `netclose`, `netdownload`

# Index

◇ Names of persons (not products) are in *italics*.
◇ Function names, constants and compiler reserved words are in `typewriter font`.