Pawn

embedded scripting language

H0420 Programming Guide & Reference

Version 1.10

November 2011

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

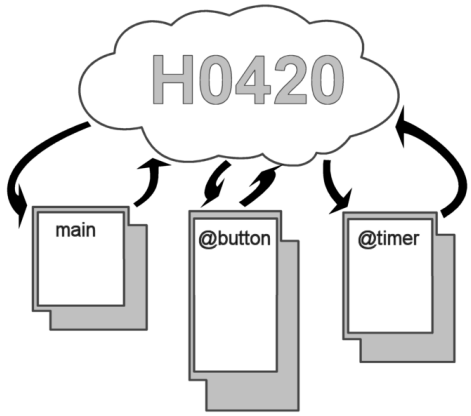
Overview

The "PAWN" programming language is a general purpose scripting language, and it is currently in use on a large variety of systems: from servers to embedded devices. Its small footprint, high performance and flexible interface to the "native" functionality of the host application/system, make PAWN well suited for embedded use.

This reference assumes that the reader understands the PAWN language. For more information on PAWN, please read the manual "The PAWN booklet — The Language" which comes with the H0420. For an introduction of the H0420 and its programming interface, please see the H0420 manual.

Event-driven programming

The H0420 follows an "event-driven" programming model. In this model, your script does not poll for events, but instead an event "fires" a function in your script. This function then runs to completion and then returns. In absence of events, the script is idle: no function runs.



When the pins of one of the 16 switches are shorted, this fires a "button down" event and the `@button` function in your script will run.* The `@button` function then handles the event, perhaps by starting to play another track, or changing volume or tone settings. After it is done, `@button` simply returns

@button: 32

* Provided that the script contains and `@button` function; if the script lacks the `@button` function, the "button down" and "button up" events would be discarded.

or exits the script. The script is now idle, but another event may wake it up. The event-driven programming model thereby creates reactive and/or interactive programs. The general manual "The PAWN booklet — The Language" has more details on the event-driven model.

The following script is a first, simple, example for scripting the H0420. In this script, the first seven switches are "linked" to playing seven tracks, with hard-coded names. Simplicity is the goal for this first example: later examples will remove the limitations of this script. For the syntax of the programming language, please see the general manual "The PAWN booklet — The Language".

Listing: **switches1.p**

```
/*  switches1
 *
 *  Play a track that is attached to a switch; there are seven tracks
 *  associated with 7 switches. The tracks have predefined names.
 *
 *  When pressing a switch for a track that is already playing, the
 *  track restarts.
 */

@button(pin, status)
    {
    /* act only on button-down */
    if (status == 1)
        {
        switch (pin)
            {
            case 0: play !"track1.mp3"
            case 1: play !"track2.mp3"
            case 2: play !"track3.mp3"
            case 3: play !"track4.mp3"
            case 4: play !"track5.mp3"
            case 5: play !"track6.mp3"
            case 6: play !"track7.mp3"
            }
        }
    }
```

When a function in the script is running, no other event can be handled. That is, while the script is busy inside, say, the `@timer` function, a button press or release event is queued. Only after the pending function has completed and has returned, will the button press/release event be handled. Functions do *not* interrupt or *pre-empt* each other.

On power-up, the first function that will run is `@reset`.[2] In this function, you

[2] `@reset` is an alias for `main`.

set up the devices that you need: LCD, RS232, I/O ports, or other. In most programming systems/languages, the program is *over* as soon as the function `@reset` (or another primary entry point) returns —this is the traditional "flow-driven" programming model. With the event-driven model in PAWN and the H0420, the script continues to be *active* after `@reset` returns. In fact, as the `switches1.p` script presented above demonstrates, function `@reset` is optional: you do not need to include it in your script if you have no particular initializations to make.

The event-driven programming model becomes convenient when the number of "events" grows. Each event has a separate "handler" (a *public function* in the PAWN environment) and it is processed individually. As an example, the next script also sets an I/O output line for the duration of the track. That is, while the H0420 is playing MP3 audio, the I/O pin will be high, and when *not* playing, it will be low. To toggle the pin, the script uses a second event: the status of the audio decoder. For testing the script, you can branch a LED directly on I/O pin 15, for visual feedback. The I/O pin can also drive an opto-coupler.

Listing: **switches2.p**

```
/*  switches2
 *
 *  Play a track that is attached to a switch; there are seven tracks
 *  associated with 7 switches. The tracks have predefined names.
 *  I/O pin 15 is high when audio is playing and low when it is
 *  silent.
 *
 *  When pressing a switch for a track that is already playing, the
 *  track restarts.
 */
const Busy = 15         /* the selected I/O pin */

@reset()
    {
    /* configure the I/O pin as output and set it low */
    configiopin Busy, Output
    setiopin Busy, 0
    }

@button(pin, status)
    {
    /* act only on button-down */
    if (status == 1)
        {
        switch (pin)
```

```
            {
            case 0: play !"track1.mp3"
            case 1: play !"track2.mp3"
            case 2: play !"track3.mp3"
            case 3: play !"track4.mp3"
            case 4: play !"track5.mp3"
            case 5: play !"track6.mp3"
            case 6: play !"track7.mp3"
            }
        }
    }

@audiostatus(AudioStat: status)
    {
    if (status == Playing)
        setiopin Busy, 1
    else
        setiopin Busy, 0
    }
```

As is apparent from this second example, function `@reset` serves for one-time initialization. Here, it is required, because the I/O pin needs to be configured for output. On power-on, all I/O pins are pre-configured as inputs.

@audiostatus: 31

Function `@audiostatus` is another event function, that runs when the status of the audio decoder changes; the parameter holds the new status, which can be `Stopped`, `Playing` or `Paused`.

button: 39

Apart from the "event" functions `@button` and `@audiostatus` mentioned earlier, the H0420 programming environment also contains a native functions `button` and `audiostatus` (without the "`@`" prefix). The `button` function returns the *current status* of a button. With it, you can check the status of each button at any convenient time. Likewise, the `audiostatus` function returns the active status of the audio decoder. With these functions in hand, you could create a polling loop inside `@reset` and skip the entire event-driven paradigm. For illustration, the next sample does this.

Listing: **switches2a.p**

```
/*  switches2a
 *
 *  The same program as switches2, but now implemented as a non-event
 *  driven program.
 */
const Busy = 15         /* the selected I/O pin */
```

```
@reset()
    {
    /* configure the I/O pin as output and set it low */
    configiopin Busy, Output
    setiopin Busy, 0

    /* we have to keep the status of all switches (in order to detect
     * the changes)
     */
    new curpin[7]

    /* we need an extra variable outside the loop to detect changes
     * in status
     */
    new AudioStat: curstatus = Stopped

    /* this loop should never end */
    for ( ;; )
        {
        /* test all switches */
        new pin, status
        for (pin = 0; pin < 7; pin++)
            {
            status = button(pin)
            if (status != curpin[pin])
                {
                /* status changed, save new status */
                curpin[pin] = status
                /* ignore button-up, act on button down */
                if (status == 1)
                    {
                    switch (pin)
                        {
                        case 0: play !"track1.mp3"
                        case 1: play !"track2.mp3"
                        case 2: play !"track3.mp3"
                        case 3: play !"track4.mp3"
                        case 4: play !"track5.mp3"
                        case 5: play !"track6.mp3"
                        case 6: play !"track7.mp3"
                        }
                    }
                }
            }

        /* test the audio status */
        new AudioStat: status = audiostatus()
        if (status != curstatus)
            {
            curstatus = status
            if (status == Playing)
                setiopin Busy, 1
```

```
            else
                setiopin Busy, 0

            }
        }
    }
```

In the flow-driven programming model, you have to *poll* for events, rather than respond to them. In programming methodologies, the flow-driven and event-driven models are reciprocal: the flow-driven model *queries* for events, the event-driven model *responds* to events. Especially in the situations where the number of events grows, the event-driven model produces neater and more compact scripts, that require less memory and in addition respond to the events quicker.

Modules

As a programming tool, PAWN consists of the "language" and a "library". The language is standardized and common for all applications. The library gives access to all the functionality that the host application/device provides. That being the case, the library is typically highly specific to the system into which PAWN is embedded. In other words, PAWN lacks something like a *standard* library.

On the other hand, it quickly proved convenient to let applications and devices provide *similar* functionality in a common way. This led to the library to be split up in several independent modules (which are also documented independently). An application/device, then, takes its choice of "modules", in addition to the application-specific interface functions.

This reference documents the functions that are specific to the H0420 and the essentials from the several modules that it uses. These modules are:

Core — The set of "core" functions (which support the language) is documented in this book, as well as in the main book on PAWN: "The PAWN booklet — The Language".

Console — The H0420 provides the console module for output to an optional LCD. See the "User Manual" of the H0420 on how to attach an LCD.

As a side remark: the original console as described in "The PAWN booklet — The Language" also has describes functions for getting *user input*; these are *stubbed out* in the H0420.

File I/O	General purpose file reading and writing functions, for both text and binary files.
Fixed-point	Fixed-point rational arithmetic is supported. Details on the fixed-point interface is in a separate application note "Fixed Point Support Library".
String functions	PAWN uses arrays for strings, and the H0420 provides a general set of string functions.
Time functions	The interface to the "date & time of the day", as well as the event timer (with a millisecond resolution).

Timers, synchronization and alarms

The H0420 provides various ways to react on timed events. These may be used in combination, as they run independently of each other.

For activities that must run at a constant interval, the `@timer` is usually the most convenient. This timer is set with function `settimer` to "go off" each time an specific interval has elapsed. This interval is in milliseconds —however, the timer resolution is not necessarily one millisecond. Due to the event-driven nature of the H0420, the precision of the timer depends on the activity of other public functions in the script. Nevertheless, the `@timer` function is the quick and precise general purpose timer.

The `@timer` function can also be set up as a single-shot timer. A single shot timer fires are the specified number of milliseconds "from now" and fires *only once*. This may be useful for time-out checking, for example.

The second timer is the `@alarm` function, which is set through the `setalarm` function. The primary purpose of this timer is to set a callback that fires at a specific "wall-clock" time. This timer may also be set to fire only at a specific date (in addition to a time). The `@alarm` timer is a repeating timer, but if you include the date and the year in the alarm specification, it has effectively become a single-shot timer ("year" numbers in dates do not wrap around, so they occur only once).

If you use the `@alarm` function, it may be needed to synchronize the internal clock of the H0420 to the actual time. This can be done with the functions `setdate` and `settime`. Note that the real-time clock of the H0420 has no battery backup, so when the power falls out, the current time is lost. At power-on, the H0420 starts at midnight, 1 January 1970.

For some purposes, you do not need absolute time, and you can use the `@alarm` function simply as a second timer. In comparison with the `@timer` function, `@alarm` as a low resolution.

When events must be synchronized with audio that is playing, the appropriate function is the `@synch` "timer" that works together with an ID3 tag, and specifically the `SYLT` frame in this tag. An ID3 tag is a block of information that is stored *inside* the MP3 file; it usually contains artist and album information, and it may contains other information as well. By adding time-stamped text to an MP3 file (in its ID3 tag), the `@synch` function will "fire" at the appropriate times and holding the line of text in its parameter. The script can then interpret the text and act appropriately.

The example below plays an MP3 file* that was prepared with a `SYLT` frame in its ID3 tag. The `SYLT` tag contains time-stamp strings in the form of:

+14 −15

where:

- ◇ the operator ("+" or "−") indicates a "toggle-on" or "toggle-off" command for an I/O pin
- ◇ the number following the operator indicates the I/O pin

Any number of codes may be on single time-stamped line, so you can turn on several I/O pins and turn off others all in the same command. For testing purposes, it is convenient to connect four LEDs to the I/O pins 12 to 15 (inclusive) —this will give you visual feedback of the command execution.

Listing: **sylt.p**

```
/* Plays an audio track and turns on and off LEDs based on the
 * commands stored in the ID3 tag (the SYLT frame).
 *
 * The commands have the form "+14 -12", where the numbers stand
 * for the I/O pins, and "+" and "-" mean "turn on" and "turn
 * off" respectively. So in this example, I/O pin 14 is turned
 * on and I/O pin 12 is turned off.
 */

@reset()
    {
    /* configure 4 I/O pins as output */
    for (new i = 12; i <= 15; i++)
        configiopin i, Output
```

* The original MP3 file was created by Orb Gettarr, and placed under the "Creative Commons" license.

```
    /* Orb Gettarr: "From The Machine World"
     * See: http://www.opsound.org/opsound/pool/gettarr.html
     */
    play !"From-The-Machine-World.mp3"
    }

@synch(const event[])
    {
    for (new index = 0; /* test is in the middle */ ; index++)
        {
        /* find first '+' or '-' */
        new c
        while ((c = event{index}) != '-' && c != '+' && c != EOS)
            index++
        if (c == EOS)
            break       /* exit the loop on an End-Of-String */

        /* get the value behind the operator ('+' or '-') */
        new pin = strval(event, index + 1)

        /* turn on or off the led (based on the operator) */
        setiopin pin, (c == '+')
        }
    }
```

LCD, or other displays

The console functions used throughout the main PAWN manual "The PAWN booklet — The Language" output to an LCD that is optionally connected to one of the connector blocks on the H0420. The H0420 is directly compatible with a wide range of character LCDs: those using the standard HD44780 controller, and a positive LC-driving voltage. With a simple voltage inversion circuit, the H0420 can also be made compatible with those LCDs that need a negative LC-driving voltage —though you may need to adjust the contrast setting to a non-default value (function `setattr`). Similarly, the H0420 can also use OLEDs, PLEDs or VFDs that use a command set that is compatible with the HD44780 controller.

In addition to character LCDs, the H0420 supports graphic displays that are based on the KS0108 controller. The graphic LCDs use a built-in monospaced font of 6×8 pixels with the Latin-1 character set and a few additional characters.

Before sending character data to the LCD, the LCD must be configured. This is done through the `console` function, also described in "The pawn

console: 44

booklet — The Language". The first two parameters of the function are the numbers of columns and lines of the display. The columns and lines are measured in characters for a character display and in pixels for a graphic display. Typical column sizes for character LCDs are 16, 20 and 40 and typical line counts are 1, 2 and 4. For a graphic LCD, typical resolutions are 64×64 and 128×64.

The third parameter for the console function holds the cursor type (a graphic display may not support a cursor). This value must be one of the following:

`CursorNone` No cursor at all, this is the default.

`CursorStable` A non-blinking cursor.

`CursorBlink` A blinking cursor.

After having set up the LCD, the standard console output functions, like `print`, `printf`, `gotoxy`, and `clrscr` are available. The console input functions, such as `getvalue` and `getstring`, are not available (and they are not documented in this manual). A further limitation of the console support is that the standard character LCDs do not support colours. Instead of changing the colour, the `setattr` function adjusts the contrast of the LCD. Graphic LCDs support inverse video; most character LCDs do not.

When an LCD is used, the first 11 I/O pins (pins numbered from 0 to 10) are unavailable for general input and output. The analogue pin is also unavailable for output, as it regulates the contrast of the LCD. A graphic LCD may require one or two extra I/O pins, depending on the configuration of the display and how it is wired to the H0420.

The following example script performs various functions, in addition to displaying information on the LCD, and it is therefore fairly large. This script assumes a character display (only the configuration would be different for a graphic LCD). Its functionality is:

- Start tracks on a switch press. When a switch is pressed during play-back, the old track is aborted and the new track plays. This script acts only on four of the sixteen possible switches.
- Initializes and prints file information on an LCD. In this particular example, the LCD is one that has four lines and twenty columns per line. For other LCD lay-outs, you need to modify the call to the `console` function (near the top of the `@reset` function).
- Reads data from the MP3 frame headers as well as from the ID3 tag (if any).

◇ Sets up and uses a timer, to refresh the LCD. This function also detects whether an audio stream is still playing. The timer is set up in function `@reset`, and the functionality itself is in the public function `@timer`.

Listing: **mp3info.p**

```
/* This example program demonstrates:
 *  o  The use of an LCD and printing information of the MP3 file
 *     that is currently playing.
 *  o  Browsing through the MP3 tracks on the CD-ROM
 *  o  Getting information from the MP3 decoder (bitrate and others)
 *  o  Getting information from the ID3 tag.
 *
 * At every switch press, go to the next track or the previous
 * track (depending on the switch). The filenames of the tracks
 * are read from the card.
 *
 * The information from the MP3 decoder is dynamic, that of the
 * ID3 tag is static.
 */

new TrackCount
new CurrentTrack

@reset()
    {
    /* the setting below is for an LCD with 4 rows and 20 columns;
     * you need to adjust it for other LCDs
     */
    console 20, 4
    consctrl 1, 0       /* disable auto-wrap */

    /* the updating of the LCD occurs on a timer */
    settimer 2000

    TrackCount = fexist(!"*.mp3")
    CurrentTrack = 0
    }

@button(pin, status)
    {
    /* act only on button-down, and only on pins 0 or 1 */
    if (status == 1 && pin < 2)
        {
        if (pin == 1)
            {
            CurrentTrack++
            if (CurrentTrack >= TrackCount)
                CurrentTrack = 0
            }
        else
            {
            CurrentTrack--
            if (CurrentTrack < 0)
                CurrentTrack = TrackCount - 1
```

```
            }

        /* find the new filename */
        new filename[100 char]
        fmatch filename, !"*.mp3", CurrentTrack
        play filename

        clrscr          /* also resets the cursor to (1,1) */
        print filename
        }
    }

@timer()
    {
    new buffer[128 char]
    static Info

    if (audiostatus() == Stopped)
        {
        if (Info >= 0)
            {
            Info = -1
            clrscr
            print !"Druk op een knop..."
            }
        return
        }
    if (Info < 0)
        Info = 0

    gotoxy 1, 2
    switch (Info)
        {
        case 0:
            printf !"Bitrate: %d kb/s", headerinfo(MP3_Bitrate)
        case 1:
            printf !"Freq: %d kHz", headerinfo(MP3_SampleFreq)/1000
        case 2:
            printf !"Avg.rate: %d kb/s", headerinfo(MP3_AvgBitrate)
        case 3:
            {
            taginfo ID3_Title, buffer
            printf !"Title: %s", buffer
            }
        case 4:
            {
            taginfo ID3_Artist, buffer
            printf !"Artist: %s", buffer
            }
        case 5:
            {
            taginfo ID3_Album, buffer
            printf !"Album: %s", buffer
            }
```

```
            case 6:
                {
                taginfo ID3_Comment, buffer
                printf !"Comment: %s", buffer
                }
            case 7:
                {
                taginfo ID3_Copyright, buffer
                printf !"Copyright: %s", buffer
                }
            case 8:
                {
                taginfo ID3_Year, buffer
                printf !"Year: %s", buffer
                }
            case 9:
                {
                taginfo ID3_Track, buffer
                printf !"Track: %s", buffer
                }
            case 10:
                {
                taginfo ID3_Length, buffer
                printf !"Length: %s ms", buffer
                }
            }
        clreol
        if (++Info > 10)
            Info = 0
        }
```

RS232

The H0420 has a standard serial line, using the RS232 protocol. All common Baud rates and data word lay-outs are supported. The interface optionally supports software handshaking, but no hardware handshaking. The RTS and CTS lines are linked (connected) at the connector; the DTR and DSR lines are connected too. Most programs/devices that use hardware handshaking will see the H0420 as a transparent unit: it accepts data when the other device/terminal does so too.

Software handshaking is optional. When set up, software handshaking uses the characters XOFF (ASCII 19, Ctrl-S) to request that the other side stops sending data and XON (ASCII 17, Ctrl-Q) to request that it resumes sending data. These characters can therefore not be part of the normal data stream (as they would be misinterpreted as control codes). Software handshaking is

therefore not suitable to transfer binary data directly (since two byte values are "reserved"). Instead, binary data should be transferred using a protocol like UU-encode.

The example script below functions as a simple terminal. It accepts a few commands that it receives over the serial port. It understands the basic commands to start playing files, to query which files are on the Compact Flash card, and to set volume and balance.

Listing: **serial.p**

```
@reset()
    {
    setserial 57600, 8, 1, 0, 0
    sendstring !"READY: "
    }

@receive(const string[])
    {
    static buf[40 char]
    strcat buf, string
    if (strfind(buf, "\r") >= 0 || strfind(buf, "\n") >= 0)
        {
        parse buf
        buf[0] = '\0'   /* prepare for next buffer */
        }
    }

stripline(string[])
    {
    /* strip leading whitespace */
    new span
    for (span = 0; string{span} != EOS && string{span} <= ' '; span++)
        {}
    strdel(string, 0, span)

    /* strip trailing whitespace */
    span = strlen(string)
    while (span > 0 && string{span-1} <= ' ')
        span--
    if (span >= 0)
        string{span} = EOS
    }

parse(string[], size=sizeof string)
    {
    stripline string

    new mark = strfind(string, " ")
    if (mark < 0)
        mark = strlen(string)
```

```
if (strcmp(string, !"PLAY", true, mark) == 0)
    {
    /* remainder of the string is the filename */
    strdel string, 0, mark
    stripline string
    if (!play(string))
        sendstring !"Error playing file (file not found?)"
    }
else if (strcmp(string, !"STOP", true, mark) == 0)
    stop
else if (strcmp(string, !"VOLUME", true, mark) == 0)
    {
    strdel string, 0, mark
    stripline string
    setvolume .volume=strval(string)
    }
else if (strcmp(string, !"BALANCE", true, mark) == 0)
    {
    strdel string, 0, mark
    stripline string
    setvolume .balance=strval(string)
    }
else if (strcmp(string, !"LIST", true, mark) == 0)
    {
    strdel string, 0, mark
    stripline string

    if (strlen(string) == 0)
        strpack string, "*", size

    new count = exist(string)
    new filename[100 char]
    for (new index = 0; index < count; index++)
        {
        selectfile filename, string, index
        sendstring filename
        sendstring !"\n"
        }
    }
else
    sendstring !"Unknown command or syntax error\n"

sendstring !"READY: "
}
```

Incoming data may be received character by character or in "chunks". Especially when the data is typed in by a user, it is likely that each invocation of `@receive` will only hold a single character. These characters or string segments must be assembled into whole commands. This script assumes that there is a single command per line.

When `@receive` sees a line terminator (a "newline" or CR character), it sends the complete line to the function `parse` that decodes it using a few string manipulation functions. The function `stripline` is a custom function that removes leading and trailing "white space" characters (spaces, TAB characters and others). The command "play" takes a parameter that follows the keyword "play" after a space separator. To play the file "TRACK1.MP3" (and assuming that you are connected to the H0420 through a simple terminal), you would type:

```
play track1.mp3
```

The commands "volume" and "balance" also take a parameter (a number, in this case). The command "list" optionally takes a file pattern as a parameter; if the pattern is absent, all files on the Compact Flash card are listed (i.e. the command "list" is short for "list *").

uudecode: 104

For transferring binary data over RS232, you may choose to convert the binary stream to UU-encode and transfer it as text, or to replace the public function `@receive` by `@receivebyte` and get bytes individually. When receiving bytes through `@receivebyte`, you should set up the serial port to use *no* software handshaking —otherwise the bytes that represent the XON & XOFF codes will still be gobbled internally. In addition, as the bytes of a stream are passed individually the the script, there is quite some overhead and the effective transfer rate is not very high. Passing binary data as UU-encoded strings (via `@receive`) has the advantage that you can *still* use software handshaking to transfer the data and throughput is likely higher as well. The drawback is that the sender must UU-encode the data before transfer.

When the incoming data has a structure in the form of well defined packets, you can use the function `packetfilter` to set up the definition of the packet format, after which the event function `@receivepacket` receives complete packets. This relieves your script from assembling packets from individual bytes and filter/handle them manually, plus that it has a better performance (i.e. better suited for high Baud rates).

The H0420 software toolkit also comes with a few ready-to-run scripts, among which is a script that implements a full serial protocol, similar to that of professional DVD players. These scripts come with commented source code and documentation in HTML format, and may therefore serve as (advanced) programming examples.

File system, file and path names

The H0420 accepts Compact Flash cards that are formatted as FAT16 or FAT32. Most Compact Flash cards will already have been formatted in either of these file systems. FAT16 is more suitable for smaller capacities (less than 256 MB) while FAT32 is more appropriate for larger capacities.

The H0420 supports subdirectories. It does *not* support relative paths, however, as it has no concept of a "working directory". All paths are relative to the root. The H0420 does not use a drive letter either —it only supports a single drive with a single partition.

The path separator may either be a backslash ("\", used in Microsoft Windows) or a forward slash ("/", used in Linux and other variants of UNIX). These may also be used interchangeably. Note that the backslash is also the default "control character" in PAWN, so you need to double it in a standard PAWN string; alternatively, you can use "raw strings". See the PAWN "Language Guide" for details on the control character and (raw) strings.

Paths and filenames are case insensitive for the H0420. This is similar to Windows and unlike Linux and UNIX.

As an example, the following PAWN strings all refer to the same file (in the same directory):

`"/media/classical.mp3"`	
`"media/classical.mp3"`	*initial slash is optional*
`"\\Media\\Classical.MP3"`	*double backslashes (normal string)*
`\"\MEDIA\CLASSICAL.MP3"`	*raw string*
`!"/media/classical.mp3"`	*packed string*

- **General file I/O**

Apart from "playing" audio files, the H0420 can read and write text and binary files. This allows capabilities such as writing usage information to a "LOG" file, storing settings and/or play files according to playlists. If the H0420 is connected to a computer, e.g. via RS232, such configuration files or playlist files can also be updated through this connection —without needing to extract the Compact Flash card.

Typically, the files that you wish to read or write are text files, and these files are probably created or analysed on software running on desktop computers. Operating systems differ in their conventions for file/path names (as was

discussed earlier), as well as the encoding of text files. The file I/O interface addresses the encoding difference to some extent, in order to be compatible with a wide range of files and hosts.

Due to memory restraints, the H0420 can only hold two files open at any time for scripting. The file I/O needed for playing MP3 files are handled separately. That is, the script can open two files and still play MP3 audio. You can manipulate more than two files in a single script, but only two files can be open at any time —before accessing a third file, you must close one of the earlier two files.

UNIX uses a single "line feed" character to end a text line (ASCII 10), the Apple Macintosh uses a "carriage return" character (ASCII 13) and Microsoft DOS/Windows use the pair of carriage return and line feed characters. Many high-level protocols of the TCP/IP protocol suite also require both a carriage return and a line feed character to end a line —examples are RFC 854 for Telnet, RFC 821 for SMTP and RFC 2616 for HTTP.

The file I/O support library provides functions for reading lines and blocks from a file, and for writing lines/blocks to a file. The line reading functions are for text files and the block reading functions for binary files. Additional functions allow you to read through a file character by character, or byte by byte, and to write a file character by character. The character reading/writing functions are indifferent for text versus binary files.

The line reading functions, `fread` and `fwrite`, check for all three common line ending specifications: CR, LF and CR–LF. If a LF character follows a CR character, it is read and considered part of a CR–LF sequence; when any other character follows CR, the line is assumed to have ended on the CR character. This implies that you cannot embed single CR characters in a DOS/Windows or UNIX file, and neither use LF characters in lines in a Macintosh file. It is uncommon, though, that such characters appear. The pair LF–CR (CR–LF in the inverted order) is *not* supported as a valid line-ending combination.

The line writing function writes the characters as they are stored in the string. If you wish to end lines with a CR–LF pair, you should end the string to write with `\r\n`.

The line reading and writing functions support UTF-8 encoding when the string to read/write is in *unpacked* format. When the source or destination string is a *packed* string, the line functions assume ASCII or another 8-bit encoding —such as one of the ISO/IEC 8859 character sets (ISO/IEC 8859-

1 is informally known as "Latin-1"). Please see the manual "The PAWN booklet — The Language" for details on packed and unpacked strings.

The block reading and writing functions, `fblockread` and `fblockwrite`, transfer the specified number of cells as a binary block. The file is assumed to be in Little Endian format (Intel byte order). On a Big Endian microprocessor, the block reading/writing functions translate the data from Big Endian to Little Endian on the flight.

The character reading and writing functions, `fgetchar` and `fputchar`, read and write a single byte respectively. Byte order considerations are irrelevant. These functions apply UTF-8 encoding by default, but they can also read/write raw bytes.

Next to data transfer functions, the library contains file support functions for opening and closing files (`fopen`, `fclose`), checking whether a file exists, (`fexist`), browsing through files (`fexist` and `fmatch`), deleting a file (`fremove`), and modifying the current position in the file (`fseek`).

Filename matching

The filename matching functions `fmatch` and `fexist` support filenames with "wild-card" characters —also known as filename patterns. The concept of these patterns exists in all contemporary operating systems (such as Microsoft Windows and UNIX/Linux), but they differ in minor ways in which characters they use for the wild-cards.

Pattern matching is not only done for filename selection; the function `packetfilter` sets up two patterns for automatically discovering packet boundaries and automatically accepting packets for forwarding to the public function `@receivepacket`. The matching of serial packets follows the same general rules as the matching of filenames.

The patterns described here are a simplified kind of "regular expressions" found in compiler technology and some developer's tools. The patterns do not have the power or flexibility of full regular expressions, but they are simpler to use.

Patterns are composed of normal and special characters. Normal characters are letters, digits, and other a set of other characters; actually, everything that is not a *special* character is "normal". The special characters are discussed further below. Each normal character matches one and only one

character —the character itself. For example, the normal character "a" in a pattern matches the letter "a" in a name or string. A pattern composed entirely of normal characters is a special case since it matches only one exactly one name/string: all characters must match exactly. The empty string is also a special case, which matches only empty names or strings.

Depending on the context, patterns may match in a case-sensitive or a case-insensitive way. Filename matching is case-insensitive, but packet matching is case-sensitive.

Special pattern characters are characters that have special meanings in the way they match characters in filenames. They may match a single instance or multiple occurrences of *any* character, or only a selected set of characters —or they may change the sense of the matching of the rest of the pattern. The special pattern characters are:

`?` Any
The *any* pattern `?` matches any single character.

`*` Closure
The *closure* pattern `*` matches zero or more non-specific characters.

`[abc]` Set
The *set* pattern `[abc]` matches a single character in the set (`a`, `b`, `c`). On case-insensitive matches, this will also match any character in the set (`A`, `B`, `C`). If the set contains the `]` character, it must be quoted (see below). If the set contains the hyphen character `-`, it must be the first character in the set, be quoted, or be specified as the range `---`.

`[a-z]` Range set
The *range* pattern `[a-z]` matches a single character in the range `a` through `z`. On case-insensitive matches, this will also match any character in the range `A` through `Z`. The character before the hyphen must sort lexicographically before the character after the hyphen. Sets and ranges can be combined within the same set of brackets; e.g. the pattern `[a-c123]` matches any character in the set (`a`, `b`, `c`, `1`, `2`, `3`).

`[!abc]` Excluded set
The *excluded set* pattern `[!abc]` matches any single character not in the set (`a`, `b`, `c`). Case-insensitive systems also exclude characters in the set (`A`, `B`, `C`). If the set contains the hyphen character, it must immediately follow the `!` character, be quoted, or be specified as

the range `---`. In any case, the `!` must immediately follow the `[` character.

`{abc}` Repeated set
The *repeated set* is similar to the normal set, `[abc]`, except that it matches zero or more occurrences of the characters in the set. It is similar to a *closure*, but matching only a subset of all characters. Similar to single character sets, the repeated set also supports ranges, as in `{a-z}`, and exclusions, as in `{!abc}`.

`‘x` Quoted (literal) character
A *back-quote* character ‘ removes any special meaning from the next character. To match the quote character itself, it must be quoted itself, as in ‘‘. The back-quote followed by two hexadecimal digits gives the character with the byte value of the hexadecimal number. This can be used to insert any character value in the string, including the binary zero. The back-quote character is also called the *grave accent*.

Some patterns, such as `*`, would match empty names or strings. This is generally undesirable, so empty names are handled as a special case, and they can be matched only by an empty pattern.

PAWN uses the zero character as a string terminator. To match a zero byte, you must use `‘00` in the pattern. For example, the pattern `a[‘00-‘1f]` matches a string that starts with the letter “a” followed by a byte with a value between 0 and 31.

INI files

Many programs need to store settings between sessions. For this reason, the library provides a set of high-level functions for storing the configuration in an “INI” file. An INI file is a plain text file where fields are stored as name/value pairs. The name (called the “key” in the function descriptions) and the value are separated by an equal sign (“`=`”) or a colon (“`:`”) —the colon separator is an extension of this library.

INI files are optionally divided into sections. A section starts with a section name between square brackets.

INI files are best known from Microsoft Windows, but several UNIX and Linux programs also use this format (although the file extension is sometimes

"`.cfg`" instead of "`.ini`"). Playlist files in Shoutcast/Icecast format also use the syntax of INI files.

Packed and unpacked strings

The PAWN language does not have variable types. All variables are "cells" which are typically 32-bit wide (there exist implementations of PAWN that use 64-bit cells). A string is basically an array of cells that holds characters and that is terminated with the special character '\0'.

However, in most character sets a character typically takes only a single byte and a cell typically is a four-byte entity: storing a single character per cell is then a 75% waste. For the sake of compactness, PAWN supports *packed* strings, where each cell holds as many characters as fit. In our example, one cell would contain four characters, and there is no space wasted.

At the same time, PAWN also supports *unpacked* strings where each cell holds only a single character, with the purpose of supporting Unicode or other wide-character sets. The Unicode character set is usually represented as a 16-bit character set holding the 60,000 characters of the Basic Multilingual Plane (BMP), and access to other "planes" through escape codes. A PAWN script can hold all characters of all planes in a cell, since a cell is typically at least 32-bit, without needing escape codes.

Many programming language solve handling of ASCII/Ansi character sets versus Unicode with their typing system. A function will then work either on one or on the other type of string, but the types cannot be mixed. PAWN, on the other hand, does not have types or a typing system, but it can check, at run time, whether a string a packed or unpacked. This also enables you to write a single function that operates on both packed and unpacked strings.

The functions in the H0420 firmware have been constructed so that they work on packed and unpacked strings.

UU-encoding

For transmitting binary data over communication lines/channels or protocols that do not support 8-bit transfers, or that reserve some byte values for special "control characters", a 6-bit data encoding scheme was devised that uses only the standard ASCII range. This encoding is called "UU-encoding".

This daemon can encode a stream of binary data into ASCII strings that can be transmitted over all networks that support ASCII.

The basic scheme is to break groups of 3 eight bit bytes (24 bits) into 4 six bit characters and then add 32 (a space) to each six bit character which maps it into the readily transmittable character. As some transmission mechanisms compress or remove spaces, spaces are changed into back-quote characters (ASCII 96) —this is a modification of the scheme that is not present in the original versions of the UU-encode algorithm.

Another way of phrasing this is to say that the encoded 6 bit characters are mapped into the set:

```
‘!"#$%&’()*+,-./012356789:;<=>?@ABC...XYZ[\]^_
```

for transmission over communications lines.

A small number of eight bit bytes are encoded into a single line and a count is put at the start of the line. Most lines in an encoded file have 45 encoded bytes. When you look at a UU-encoded file note that most lines start with the letter “M”. “M” is decimal 77 which, minus the 32 bias, is 45. The purpose of this further chopping of the byte stream is to allow for handshaking. Each chunk of 45 bytes (61 encoded characters, plus optionally a newline) is transferred individually and the remote host typically acknowledges the receipt of each chunk.

Some encode programs put a check character at the end of each line. The check is the sum of all the encoded characters, before adding the mapping, modulo 64. Some encode programs have bugs in this line check routine; some use alternative methods such as putting another line count character at the end of a line or always ending a line with an “M”. The functions in this module encode byte arrays without line check characters, and the decoder routine ignores any “check” characters behind the data stream.

To determine the end of a stream of UU-encoded data, there are two common conventions:

- ◇ When receiving a line with less that 45 encoded bytes, it signals the last line. If the last line contains 45 bytes exactly, another line with zero bytes must follow. A line with zero encoded bytes is a line with only a back-quote.
- ◇ A stream must always be ended with a line with 0 (zero) encoded bytes. Receiving a line with less than 45 encoded bytes does not signal the end of the stream — it may indicate that further data is only delayed.

Rational numbers

The PAWN programming language supports only one data type: the 32-bit integer, called a *cell*. With special operators and a strong tag, the PAWN language can also do rational arithmetic, with three decimal digits. To use the "fixed-point arithmetic", your script must include the file `rational.inc`, for example by using the following directive:

```
#include <rational>
```

The fixed point format used in this library uses three decimal digits and stores the values in two's complement. This gives a range of -2147483 to +2147482 with 3 digits behind the decimal point. Fixed point arithmetic also goes by the name "scaled integer" arithmetic. Basically, a fixed point number is the numerator of a fraction where the denominator is implied. For this library, the denominator is 1000 —therefore, the integer value 12345 stands for $\frac{12345}{1000}$ or 12.345.

In rounding behaviour, however, there is a subtle difference between fixed point arithmetic and straight-forward scaled integer arithmetic: in fixed point arithmetic, it is usually intended that the least significant digit should be rounded before any subsequent digits are discarded; but many scaled integer arithmetic implementations just "drop" any excess digits. In other words, $^2/_3$ in fixed point arithmetic results in 0.667, which is more accurate than the scaled integer result of 0.666.

To convert from integers to fixed point values, use one of the functions `fixed` or `strfixed`. The function `fixed` creates a fixed point number with the same integral value as the input value and a fractional part of zero. Function `strfixed` makes a fixed point number from a string, which can include a fractional part.

A user-defined assignment operator is implemented to automatically coerce integer values on the right hand to a fixed point format on the left hand. That is, the lines:

```
new a = 10
new Fixed: b = a
```

are equivalent to:

```
new a = 10
new Fixed: b = fixed(a)
```

To convert back from fixed point numbers to integers, use the functions `fround` and `ffract`. Function `fround` is able to round upwards, to round

downwards, to "truncate" and to round to the nearest integer. Function `ffract` gives the fractional part of a fixed point number, but still stores this as a fixed point number.

The common arithmetic operators: `+`, `-`, `*` and `/` are all valid on fixed point numbers, as are the comparison operators and the `++` and `--` operators. The modulus operator `%` is forbidden on fixed point values.

The arithmetic operators also allow integer operands on either left/right hand. Therefore, you can add an integer to a fixed point number (the result will be a fixed point number). This also holds for the comparison operators: you can compare a fixed point number directly to an integer number (the return value will be `true` or `false`).

Reducing memory requirements

The H0420 has 16 kiB of memory available to scripting. This limit is declared in the `h0420.inc` file, so that the PAWN compiler is aware of this limit and can (hopefully) verify that the script fits into the memory. If the PAWN compiler complains that the script is too large, you must find a way to reduce the size of the script after compilation.

- ◇ If performance is not critical, switch on code overlays. Overlays set a maximum size of 4 kiB *per function*, but the number of functions is unlimited. To enable code overlays, set the option "`-V`" on the command line for the PAWN compiler, or check the "overlay code generation" option in the Quincy IDE.
- ◇ Some space will be gained if you compiled *without run-time checks*. To do so, add the option "`-d0`" on the command line for the PAWN compiler, or set the "debug level" option to zero in the Quincy IDE. This removes array bounds checks and assertions.
- ◇ Make sure that the optimization level is set to "2"; the PAWN compiler generates more compact code. The relevant option is "`-O2`". Note that this option is set by default.
- ◇ See if there is similar code repeated several times in the script. Such code could then be put in a separate function, and this function is then re-used for every routine needing the code.

- At a smaller scale, if the same value gets calculated several times in a function, declare instead a new variable that holds this calculated value. The academic terminology for replacing common sub-expressions with helper variables is *strength reduction*.
- Verify the stack usage (use the option "`-v`" of the compiler; optionally use "`-r`" to get a detailed report). If the compiler reports that there is ample unused stack space, you may reduce the size of the stack with the compiler option "`-S`" or adding a "`#pragma dynamic`" in your script —the latter is probably more convenient, as you do not have to remember to add an option to the command line at each compile.
- If you use strings, make sure that these are packed strings. Packed strings take less space on the stack and/or heap. Literal strings also take less space in the "literal pool" of the script.
- When a function has an array parameter (such as a string) with a default value, declare the parameter as "`const`" if possible. With a non-const parameter, a copy of the default value of the parameter must be made on the stack, because the function should not be able to change the default parameter. Declaring the parameter as `const` allows the compiler to avoid this copy.

If a script still does not fit in the available memory, it must be split into separate scripts, where each script performs a different task. The scripts can switch to other scripts (and thereby to other tasks) through the `exec` function.

Finding errors (debugging)

If a script behaves in an unexpected (or undesired) way, there are various methods to see which code is responsible for the behaviour.

If there is already an LCD attached to the H0420, a simple method is to print messages and values of variables at critical points, so that these can be inspected while the program is running. Even if you do not need an LCD for the H0420 in its "production use", it may be convenient to have an LCD for the specific purpose of debugging the script. For setting up and using an LCD, see page 9 and the example program `mp3info.p` on page 11.

If no LCD is available, or if the I/O pins for the LCD are already in use for other purposes, an alternative is to send these "trace" strings over the

serial line. This is not as flexible, as the serial interface lacks the equivalent of `printf`, a "formatted print" function, but with the companion functions of the string module, it provides adequate tracing facilities. See the functions `setserial` and `sendstring` in this reference for setting up a serial connection on pages 85 and 81 respectively.

The PAWN toolkit comes with a source level debugger that supports "remote debugging", meaning that the debugger controls the script running on the H0420 from a host PC. The remote debugging facility also uses the serial line, but it sets it up automatically. To use remote debugging, follow these steps:

- ◇ If you are using the Quincy IDE, make sure that the IDE is configured for remote debugging. In the "Options. . . " dialog (under the "Tools" menu), choose the TAB-page "Debugger" and select either COM1: or COM2:.

- ◇ Compile the script with full debug information (compiler option "`-d2`" or select "debug level" 2 from the Quincy IDE) and store the compiled script on the Compact Flash card.

- ◇ Also keep the compiled script and its source code on the local PC. It is assumed that the script resides on a local hard disk of your PC while you edit and build it, and that the resulting "AUTORUN.AMX" file is then transferred to the Compact Flash card.

- ◇ If you are using the Quincy IDE, you have to set a breakpoint in the source code, otherwise the IDE will not launch the debugger. Once the breakpoint is set, select the option "Run" from the menu/toolbar (or press `F5`).

  If not using the Quincy IDE, launch the PAWN debugger separately, with the filename of the compiled script and the option "`-rs232`". The filename is always "AUTORUN.AMX". The command line is therefore:

  ```
  pawndbg autorun.amx -rs232
  ```

  This assumes that you are using the first serial port ("COM1:") on the host PC. If you use the second serial port, use:

  ```
  pawndbg autorun.amx -rs232=2
  ```

  on Microsoft Windows and

  ```
  pawndbg autorun.amx -rs232=1
  ```

  on Linux or UNIX.Note that the serial ports are numbered from zero in Linux —ttyS1 is what Microsoft Windows would call COM2:.

◇ Insert the Compact Flash card in the H0420 and optionally reset (or power-cycle) the device. The debugger should now display the first line of function `@reset`.

When the H0420 is reset and the script that it loads has debug information, it waits up to 2 seconds for a debugger to connect. If no debugger connects, the H0420 runs the script *without* debugger support. This is why it is advised to start the debugger *before* resetting the H0420.

After the script has been fully debugged, you will want to recompile it without debugging support: avoids the start-up delay (when the H0420 polls for a debugger to connect), and it reduces the size of the script and increases performance.

Transferring scripts over RS232

The script for the H0420 must reside on the Compact Flash card (in the root directory). For simple scripts, it is easy to write the script, compile it and copy the resulting "AUTORUN.AMX" onto the Compact Flash card. To copy the file, you can use a common "card reader" that branches on an USB port.

During development and debugging, with many "write/compile/copy/test" cycles, constantly swapping the Compact Flash card between the H0420 MP3 player and the card reader on the PC may become a nuisance. An alternative is to transfer the AUTORUN.AMX over a serial line. The function to transfer files over the serial line works through the debugger or from inside the Quincy IDE. The debugger/IDE is able to synchronize with the H0420 MP3 player if the compiled script contains debugging information, or after a reset.

The first step is to compile the script as usual. If you are using the Quincy IDE, choose then option Transfer to remote host from the Debug menu. If not using the Quincy IDE, launch the debugger with the compiled script name ("AUTORUN.AMX"), as described in the previous section. Then, you need to reset the H0420, either by pressing the "RESET" switch on the board or by power-cycling the device.

With the Quincy IDE, the transfer will now proceed automatically, but with the stand-alone debugger, you will need to give the command "`transfer`" to send the latest release of the AUTORUN.AMX file to the H0420, which will then write it onto the Compact Flash card. After the copy is complete, the H0420 will automatically restart, and the debugger restarts too.

If transferring the AUTORUN.AMX is the only purpose of launching the debugger, you may also give the `transfer` command as a command line option. For instance, the line below starts the debugger, transfers the file and then exits:

```
pawndbg autorun.amx -rs232=1 -transfer -quit
```

There is also a DOS/Windows "batch" file, called `upload.bat`, that contains the above command. Again, this batch file and the debugger commands described above do not apply if you use the Quincy IDE.

Especially for purposes of uploading compiled scripts, it can be useful to have the H0420 reset on a command that comes over the same RS232 line

—because the H0420 MP3 player only picks up a debugger synchronization attempt within 2 seconds after a reset. A convenient hook is in the example below: the `@reset` function sets up the serial port with a Baud rate of 57600 bps and the `@receivebyte` function responds to the '!' character. These Baud rate and *synchronization command* are the same as used by the PAWN debugger, meaning that in attempting to synchronize with the debugger support in the H0420 MP3 player, `pawndbg` will reset the MP3 player if it was *not* polling for the debugger.

Listing: **Reset the MP3 player on receiving a '!' on the RS232**

```
@reset()
    {
    setserial 57600
    }

@receivebyte(value)
    {
    if (value == '!')
        reset
    }
```

Public functions

@alarm — The timer alarm went off

Syntax: `@alarm()`

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

Notes: The alarm must have been set with `setalarm`.

After firing, the alarm is automatically reset.

See also: `@timer`, `setalarm`

@audiostatus — The audio status changed

Syntax: `@audiostatus(AudioStat: status)`

`status` The new audio status.

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

Notes: The status is one of the following:

`Stopped` (0) The audio is stopped.
`Paused` (1) The audio is paused and can be resumed.
`Playing` (2) The audio is currently playing.
`FadeCompleted` (5) The volume fade (started with `setvolume`) has completed.

In special circumstances, you may receive a "`Stopped`" notification without receiving a "`Playing`" signal earlier. This happens in particular when a file that was passed to function `play` did not contain valid MP3 audio data.

See also: `audiostatus`, `play`, `pause`, `resume`

@button A switch was pressed or released

Syntax: `@button(pin, status)`

`pin`	The switch number, between 0 and 15.
`status`	The new status: 1 for "down" and 0 for "up".

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

Example: See `mp3info.p` on page 11.

See also: `@input`, `button`

@eject The card is removed

Syntax: `@eject()`

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

Notes: This function is called when the device detects that the CompactFlash card is removed ("ejected"). After completion of the `@eject` function, the H04x0 series of MP3 controllers do an implicit reset in approximately one second

If you need to store data or status information on eject, you need to store such information in the configuration area of the of the H0420 itself —see `storeconfig`. You cannot write device data or status information to the CompactFlash card (because it is "ejected"...)

See also: `storeconfig`

@input A digital pin changed

Syntax: `@input(pin, status)`

`pin`	The pin number, between 0 and 15.
`status`	The new logical level (0 or 1).

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

Notes: Only the pins that are configured as "input" can cause this event function to execute. See `configiopin` for configuration.

This function is invoked when the logical level of an input pin changes. The function `getiopin` may be used to read the active status of a pin.

The function `inputlapse` can be used to measure time intervals between changes on an input pin.

See also: `@button`, `configiopin`, `getiopin`, `inputlapse`

---

main Script entry point

Syntax: `main()`

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

Notes: `main` is an alternative name for function `@reset`.

See also: `@reset`

@receive Data from RS232 is received

Syntax: `@receive(const string[])`

`string` The data received as a zero-terminated string. The string may contain one or more characters.

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

Notes: The H0420 optionally uses software handshaking (XON/OFF) —see `setserial`. Due to this design, bytes with the values 17 (`0x11`, Ctrl-Q), 19 (`0x13`, Ctrl-S) and zero cannot be received with this function. When you need to transfer binary data, you should encode it using a protocol like UU-encode.

Example: See `serial.p` on page 14.

See also: `@receivebyte`, `@receivepacket`, `sendbyte`, `sendstring`, `setserial`

@receivebyte A single byte is received from RS232

Syntax: `@receivebyte(value)`

`value` The data received. This may be any value between 0 and 255.

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

Notes: The function `@receive` can receive all byte values except zero. If software handshaking is on, bytes with the values 17 (`0x11`, Ctrl-Q), 19 (`0x13`, Ctrl-S) and zero cannot be received either, as these are the XON & XOFF characters. The function `@receivebyte` allows to receive all bytes, including the zero byte. Note that software handshaking should still be turned off for receiving bytes with values 17 and 19.

If a script implements both this function and `@receive`, this function is only called for the "zero" bytes. All bytes with other values are collected in strings and passed through `@receive`.

Example: See the debugger support function on page 30.

See also: `@receive`, `@receivepacket`, `sendbyte`, `sendstring`, `setserial`

@receivepacket A data packet is received from RS232

Syntax: `@receivepacket(const packet[], numbytes)`

`packet` The data received.

`numbytes` The number of elements in the packet.

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

Notes: This function will only receive packets after a packet filter has been set up. Software handshaking should be turned off for receiving bytes with values 17 and 19 in a packet.

See also: `@receive`, `@receivebyte`, `packetfilter`, `setserial`

@reset Script entry point

Syntax: `@reset()`

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

Notes: On power-up or on reset of the device, this is the first function that is called. This function is therefore appropriate to initialize the settings needed for the script and other call-back functions.

Function `main` is an alternative name for the same function —you can use either `@reset` or `main` in a script, but not both.

After starting a new script with `exec`, the new script also starts with the `@reset` function.

See also: `exec`

@sample A burst of samples arrived

Syntax: `@sample(const Fixed:stamps[], numsamples)`

`stamps` An array containing time-stamps in milliseconds. As the values are in fixed-point format with three decimals, the time-stamps have a resolution of a microsecond.

`numsamples` The number of time-stamps in parameter `stamps`

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

Notes: After a pin has been set up for sampling (see `configiopin`, the MP3 player starts sampling data as soon as the state of that input pin changes, either from high to low, or from low to high. What it passes to the `@sample()` function are only the time-stamps of these changes, not whether they go up or down. However, you only need to know the direction of the first state change; since each time-stamp signals a toggle of the pin level, you can derive the pin level at any moment in time from the initial state. For the H0420 MP3 player, the initial state is defined as "high", so the first state change that

is recorded is a transition from high-level to low-level. This occurs at time-stamp zero, because this change also starts the sampling and all subsequent time-stamps are relative to the start.

As it is always present, the zero time-stamp that starts the sampling is *not* in the `stamps` array passed to the function. That is, when the first element in the `stamps` array is 1.000, the signal at the input pin is low between 0.000 ms and 1.000 ms (relative to the start of the sampling); at 1.000 ms, the signal toggled high.

If the pin is low-level at rest and the first change of the pin goes high, the `stamps` array contains a zero time-stamp as its first element —i.e. `stamps[0]` is 0.000 in this case.

See also: `configiopin`

@synch Synchronized lyrics/cue arrived

Syntax: `@synch(const event[])`

`event` The text of the synchronized event, as read from the ID3 tag.

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

Notes: The buffer for storing synchronized events is shared with the buffer for the script. When the script is large, less memory is available for storing the events. See the section "Reducing memory requirements" on page 25 for details.

Example: See `sylt.p` on page 8

See also: `play`

@timer A timer event occurred

Syntax: `@timer()`

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

Notes: This function executes after the delay/interval set with `settimer`. Depending on the timing precision of the host, the call may occur later than the delay that was set.

If the timer was set as a "single-shot", it must be explicitly set again for a next execution for the `@timer` function. If the timer is set to be repetitive, `@timer` will continue to be called with the set interval until it is disabled with another call to `settimer`.

See also: `delay`, `settimer`

audiostatus — Get the current audio status

Syntax: `AudioStat: audiostatus()`

Returns: One of the following values:

`Stopped`	(0) The audio is stopped.
`Paused`	(1) The audio is paused and can be resumed.
`Playing`	(2) The audio is currently playing.

Notes: This function always returns the active status; it does not rely on the presence of the event function `@audiostatus`.

Example: See `mp3info.p` on page 11.

See also: `@audiostatus`

bass — Tone adjust (bass)

Syntax: `bass(gain, frequency=200)`

`gain`	The gain in the range −12 to +12. Each step is in 1.5 dB (so the range of gain is −18. . . +18 dB.
`frequency`	The frequency at which the attenuation/enhancement starts. The suggested range is 50 Hz to 750 Hz; a typical value is 200 Hz. This parameter is clamped between 20 Hz and 1000 Hz.

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

Notes: The volume level is downward adjusted to allow for the maximum enhancement of bass or treble, while avoiding clipping. That is, when enhancing bass frequencies, the overall volume may decrease.

See also: `setvolume`, `treble`

button Read the status of a button

Syntax: `button(pin)`

`pin` The switch number, between 0 and 15; or -1 to read the state of all 16 switches as a bit mask.

Returns: If parameter `pin` is in the range 0...15, the return value is the status of the specified switch: 1 if the switch is down and 0 if the switch is up. If parameter `pin` is -1, the return value is a value where the first 16 bits represent the state of the respective switches.

See also: `@button`, `getiopin`

channelselect Set mono/stereo, or invert channels

Syntax: `channelselect(ChannelType: code=Stereo)`

`Code` The channel, one of the following:

`Stereo` (0)
: Stereo: left channel on left output and right channel on right output.

`LeftChannel` (1)
: Left channel on both outputs.

`RightChannel` (2)
: Right channel on both outputs.

`Inverted` (3)
: Inverted stereo (left channel on right output and vice versa).

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

Notes: To adjust the balance between the channels, use the function `setvolume`.

See also: `setvolume`

clearioqueue Remove switch or input events from the queue

Syntax: `clearioqueue(queue=3)`

`queue`	The queue to clear: set to 1 to clear the queue for the switch events, to 2 to clear the queue for I/O pin state changes, and to 3 to clear both queues.

Returns: This function always returns 0.

Notes: During lengthy processing (by the script), any switch (button) or I/O events are queued. These events will then be handled as soon as the lengthy processing terminates. If this is undesired, the script may clear either queue (or both). immediately after finishing the process. All events that have happened in the mean time will then have been "forgotten".

See also: `@button`, `@input`

clamp Force a value inside a range

Syntax: `clamp(value, min=cellmin, max=cellmax)`

`value`	The value to force in a range.
`min`	The low bound of the range.
`max`	The high bound of the range.

Returns: `value` if it is in the range `min` – `max`; `min` if `value` is lower than `min`; and `max` if `value` is higher than `max`.

See also: `max`, `min`

clreol Clear up to the end of the line

Syntax: `clreol()`

Returns: Always returns 0.

Notes: Clears the line on the LCD from the position of the cursor to the right margin of the console. This function does not move the cursor.

The LCD must be configured with function `console` before calling this function.

See also: `clrscr`, `console`

clrscr Clear the LCD

Syntax: `clrscr()`

Returns: Always returns 0.

Notes: Clears the console and sets the cursor in the upper left corner.

The LCD must be configured with function `console` before calling this function.

See also: `clreol`, `console`

configiopin Configure an I/O pin

Syntax: `configiopin(pin, PinConfig: type, timeout=0)`

`pin` The pin number, between 0 and 15.

`type` The type, one of the following:

`Output` (0)
The pin is configured as output, and it can be set with `setiopin`.

`Input` (1)
: The pin is configured as input and it can be read with `getiopin`; a change of the pin also invokes public function `@input`.

`Sample` (2)
: The pin is configured as input and for collecting time-stamped data; when a change of the value of the pin is detected, all subsequent changes of the pin within the configured time-out are passed to the public function `@sample`, with precision time-stamps.

`timeout`
: This parameter is relevant only when the pin type is `Sample`; it indicates the duration of the sampling period, in milliseconds, starting from the first detected change in the level of the pin.

Returns: This function always returns 0.

Notes: After reset, all pins are configured as inputs (high-impedance).

When an LCD is configured, pins 0 to 10 are unavailable. See function `console` for configuring an LCD.

When configured as outputs, the pins can drive a LED or an opto-coupler directly (no intermediate "driver" IC is required).

Only a single pin may be configured as type `Sample`. The sampling period starts as soon as the logic level of the specified pin changes (low to high or high to low), and it has a duration specified in the `timeout` parameter.

Example: See `switches2.p` on page 3 and `sylt.p` on page 8.

See also: `@input`, `@sample`, `console`, `getiopin`, `setiopin`

consctrl Adjust console settings

Syntax: `consctrl(code, value)`

`code` The parameter to change, one of the following:

0 Console support check: parameter `value` is ignored; the return value is always 1 (the hardware is unable to verify whether a display is attached to the LCD connector).

1 Auto-wrap: if the `value` is 1 (the default), text wraps from the right margin of the display to the next line; if zero, text is cut off at the right margin.

2 Buffer swap: not supported.

3 Bold font: not supported.

4 Console "initialized" check: the return value is 1 if the display is initialized and 0 otherwise.

5 Reserved.

6 Wait for "busy" flag of the display: on slow displays, it may be required to wait for the signal of the display that it is ready for new commands. Most displays, though, accept commands at the speed that the controller sends them, and the check for the busy flag is superfluous.

7 Inverse video: if supported by the display, setting this value to 1 inverts the foreground and background on the display.

8 Display-dependent configuration: this option selects a hardware configuration that is appropriate for the display and its wiring.

`value` The new value for the console parameter.

Returns: The return value depends on the value of `code`.

Notes: The LCD must be configured with function `console` before calling this function.

See also: `console`, `setattr`

console Initialize the LCD

Syntax:
```
console(columns, rows,
        LCDcursor: cursor=CursorNone)
```

`columns` For a character display, the number of columns on the LCD, typically 16, 20 or 40. For a graphic display, the number of pixels horizontally.

`rows` For a character display, the number of rows on the LCD, typically 1, 2 or 4. For a graphic display, the number of pixels vertically.

`cursor` The cursor type, one of the following:

- `CursorNone` No cursor at all, this is the default.
- `CursorStable` A non-blinking cursor.
- `CursorBlink` A blinking cursor.

Graphic LCDs typically do not support a cursor. If the display does not support a cursor, this parameter is ignored.

Returns: Always returns 0.

Notes: This function initializes the LCD, and configures the I/O pins 0–10 accordingly. It sets the LCD contrast to a default setting that is suitable for most standard LCDs. If you use a display that requires a non-standard contrast tension, you should adjust it with function `setattr`. In particular, PLED and OLED modules often need a high contrast setting.

When the display is a graphic LCD or OLED, additional configuration settings (with function `consctrl`) may be needed. A graphic LCD may also use additional I/O pins, specifically pins 11 and 12.

Noritake graphical VFDs of the 7800 series are a special case: these displays are compatible with the character LCD modules (i.e. they emulate the HD44780 command set), but they also provide graphical commands. To use a VFD, you must set it up with character column and row numbers, like a character

LCD. However, you can still use the `image` function on a VFD to display graphical data.

Example: See `mp3info.p` on page 11.

See also: `consctrl`, `image`, `print`, `printf`, `setattr`

---

cvttimestamp Convert a timestamp into a date and time

Syntax:
```
cvttimestamp(seconds1970, &year=0, &month=0,
             &day=0, &hour=0, &minute=0, &second=0)
```

`year`	This will hold the year upon return.
`month`	This will hold the month (1–12) upon return.
`day`	This will hold the day of (1–31) the month upon return.
`hour`	This will hold the hour (0–23) upon return.
`minute`	This will hold the minute (0–59) upon return.
`second`	This will hold the second (0–59) upon return.

Returns: This function always returns 0.

Notes: Some file and system functions return timestamps as the number of seconds since midnight, 1 January 1970, which is the start of the UNIX system epoch. This function allows to convert these time stamps into date and time fields.

See also: `gettime`, `getdate`, `settimestamp`

---

delay Halts execution a number of milliseconds

Syntax:
```
delay(milliseconds)
```

`milliseconds`
The delay, in milliseconds.

Returns: This function currently always returns zero.

Notes: On some platforms, the `sleep` instruction also delays for a given number of milliseconds. The difference between the `sleep` instruction and the `delay` function is that the `delay` function does not yield events and the `sleep` instruction typically yields. When yielding events is, any pending events are handled. As a result, the `delay` function waits *without* handling any pending events and the `sleep` instruction waits and deals with events.

See also: `tickcount`

---

deletecfg Deletes a key or a section from an INI file

Syntax:
```
bool: deletecfg(const filename[]="", const
                section[]="", const key[]="")
```

`filename` The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".

`section` The section from which to delete the `key` under. If this parameter is not set, the function stores the key/value pair outside any section.

`key` The key to delete. If this parameter is not set, the function deletes the entire section.

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

Notes: If both `section` and `key` are not set, the function deletes all keys that are outside any sections.

See also: `readcfg`, `writecfg`

exec Chain to another script

Syntax: `bool: exec(const filename[])`

`filename` The full name of the new script, including the extension and path.

Returns: `false` if there was an error in loading of the script, or if its validation failed. If the function succeeds, it will not return, but instead start the new script.

See also: `@reset`

fabs Return the absolute value of a fixed point number

Syntax: `Fixed: fabs(Fixed: value)`

`value` The value to return the absolute value of.

Returns: The absolute value of the parameter.

fattrib Set the file attributes

Syntax: `bool: fattrib(const name[], timestamp=0, attrib=0x0f)`

`name` The name of the file.

`timestamp` Time of the last modification of the file. When this parameter is set to zero, the time stamp of the file is not changed.

`attrib` A bit mask with the new attributes of the file. When set to 0x0f, the attributes of the file are not changed.

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

Notes: The time is in number of seconds since midnight at 1 January 1970: the start of the UNIX system epoch.

The file attributes are a bit mask. The meaning of each bit depends on the underlying file system (e.g. FAT, NTFS, etx2 and others).

See also: `fstat`

---

fblockread Read an array from a file, without interpreting the data

Syntax:
```
fblockread(File: handle, buffer[],
           size=sizeof buffer)
```

`handle` The handle to an open file.

`buffer` The buffer to read the data into.

`size` The number of *cells* to read from the file. This value should not exceed the size of the `buffer` parameter.

Returns: The number of cells read from the file. This number may be zero if the end of file has been reached.

Notes: This function reads an array from the file, without encoding and ignoring line termination characters, i.e. in binary format. The number of bytes to read must be passed explicitly with the `size` parameter.

See also: `fblockwrite`, `fopen`, `fread`

---

fblockwrite Write an array to a file, without interpreting the data

Syntax:
```
fblockwrite(File: handle, const buffer[],
            size=sizeof buffer)
```

`handle` The handle to an open file.

`buffer` The buffer that contains the data to write to the file.

	`size`	The number of *cells* to write to the file. This value should not exceed the size of the `buffer` parameter.

Returns: The number of cells written to the file.

Notes: This function writes an array to the file, without encoding, i.e. in binary format. The buffer need not be zero-terminated, and a zero cell does not indicate the end of the buffer.

See also: `fblockread`, `fopen`, `fwrite`

---

fclose — Close an open file

Syntax: `bool: fclose(File: handle)`

	`handle`	The handle to an open file.

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

See also: `fopen`

---

fcopy — Copy a file

Syntax: `bool: fcopy(const source[], const target[])`

	`source`	The name of the (existing) file that must be copied, optionally including a full path.
	`target`	The name of the new file, optionally including a full path.

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

Notes: If the target file already exists, it is overwritten.

See also: `frename`

fdiv Divide a fixed point number

Syntax: `Fixed: fdiv(Fixed: oper1, Fixed: oper2)`

`oper1` The numerator of the quotient.

`oper2` The denominator of the quotient.

Returns: The result: $^{\texttt{oper1}}/_{\texttt{oper2}}$.

Notes: The user-defined `/` operator forwards to this function.

See also: `fmul`

fexist Count matching files, check file existence

Syntax: `fexist(const pattern[])`

`pattern` The name of the file, optionally containing wild-card characters.

Returns: The number of files that match the pattern

Notes: In the pattern, the characters "`?`" and "`*`" are wild-cards: "`?`" matches any character —but only exactly one character, and "`*`" matches zero or more characters. Only the final part of the path (the portion behind the last slash or backslash) may contain wild-cards; the names of the directories must be fully specified.

If no wild-cards are present, the function returns 1 if the file exists and 0 if the file cannot be found. As such, you can use the function to verify whether a file exists.

See also: `fmatch`

ffract Return the fractional part of a number

Syntax: `Fixed: ffract(Fixed: value)`

`value` The number to extract the fractional part of.

Returns: The fractional part of the parameter, in fixed point format. For example, if the input value is "3.14", `ffract` returns "0.14".

See also: `fround`

fgetchar Read a single character (byte)

Syntax: `fgetchar(File: handle)`

`handle` The handle to an open file.

Returns: The character that was read, or `EOF` on failure.

See also: `fopen`, `fputchar`

filecrc Return the 32-bit CRC value of a file

Syntax: `filecrc(const name[])`

`name` The name of the file.

Returns: The 32-bit CRC value of the file, or zero if the file cannot be opened.

Notes: The CRC value is a useful measure to check whether the contents of a file has changed during transmission or whether it has been edited (provided that the CRC value of the original file was saved). The CRC value returned by this function is the same as the one used in ZIP archives (PKZip, WinZip) and the "SFV" utilities and file formats.

See also: `fstat`

fixed Convert integer to fixed point

Syntax: `Fixed: fixed(value)`

`value` the input value.

Returns: A fixed point number with the same (integral) value as the parameter (provided that the integral value is in range).

See also: `fround`, `strfixed`

flength Return the length of an open file

Syntax: `flength(File: handle)`

`handle` The handle to an open file.

Returns: The length of the file, in bytes.

See also: `fopen`, `fstat`

fmatch Find a filename matching a pattern

Syntax:
```
bool: fmatch(name[], const pattern[], index=0,
             maxlength=sizeof name)
```

`name` If the function is successful, this parameter will hold a n^{th} filename matching the pattern. The name is always returned as a packed string.

`pattern` The name of the file, optionally containing wildcard characters.

`index` The number of the file in case there are multiple files matching the pattern. Setting this parameter to 0 returns the first matching file, setting it to 1 returns the second matching file, etc.

`size` The maximum size of parameter `name` in cells.

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

Notes: In the pattern, the characters "`?`" and "`*`" are wild-cards: "`?`" matches any character —but only exactly one character, and "`*`" matches zero or more characters. Only the final part of the path (the portion behind the last slash or backslash) may contain wild-cards; the names of the directories must be fully specified.

The name that is returned in parameter `name` always contains the full path to the file, starting from the root.

See also: `fexist`

---

fmkdir Create a directory

Syntax: `bool: fmkdir(const name[])`

`name` The name of the directory to create, optionally including a full path.

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

Notes: To delete the directory again, use `fremove`. The directory must be empty before it can be removed.

See also: `fremove`

---

fmul Multiply two fixed point numbers

Syntax: `Fixed: fmul(Fixed: oper1, Fixed: oper2)`

`oper1`
`oper2` The two operands to multiply.

Returns: The result: `oper1` × `oper2`.

Notes: The user-defined `*` operator forwards to this function.

See also: `fdiv`

fmuldiv Fixed point multiply followed by a divide

Syntax:
```
Fixed: fmuldiv(Fixed: oper1, Fixed: oper2,
               Fixed: divisor)
```

`oper1`
`oper2` The two operands to multiply (before the divide).

`divisor` The value to divide `oper1` × `oper2` by.

Returns: The result: $\frac{oper1 \times oper2}{divisor}$.

Notes: This function multiplies two fixed point numbers, then divides it by a third number ("`divisor`"). It avoids rounding the intermediate result (the multiplication) to a fixed number of decimals halfway. Therefore, the result of `fmuldiv(a, b, c)` may have higher precision than "`(a * b) / c`".

See also: `fdiv`, `fmul`

fopen Open a file for reading or writing

Syntax:
```
File: fopen(const name[],
            filemode: mode=io_readwrite)
```

`name` The name of the file, including the path.

`mode` The intended operations on the file. It must be one of the following constants:

`io_read`
: opens an existing file for reading only (the file must already exist)

`io_write`
: creates a new file (or truncates an existing file) and opens it for writing only

`io_readwrite`
: opens a file for both reading and writing; if the file does not exist, a new file is created

`io_append`
: opens a file for writing only, where all (new) information is appended behind the existing contents of the file; if the file does not exist, a new file is created

Returns: A "handle" or "magic cookie" that refers to the file. If the return value is zero, the function failed to open the file.

Notes:

See also: `fclose`

---

fpower Raise a fixed point number to a power

Syntax: `Fixed: fpower(Fixed: value, exponent)`

`value` The value to raise to a power; this is a fixed point number.

`exponent` The exponent is a whole number (integer). The exponent may be zero or negative.

Returns: The result: $\texttt{value}^{exponent}$; this is a fixed point value.

Notes: For exponents higher than 2 and fractional values, the `fpower` function may have higher precision than repeated multiplication, because the function tries to calculate with an extra digit. That is, the result of `fpower(3.142, 4)` is probably more accurate than `3.142 * 3.142 * 3.142 * 3.142`.

See also: `fsqroot`

---

fputchar Write a single character to the file

Syntax: `bool: fputchar(File: handle, value)`

`handle` The handle to an open file.

`value` The value to write (as a single character) to the file.

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

Notes:

Notes: The function writes a single byte to the file; values above 255 are not supported.

See also: `fgetchar`, `fopen`

---

fread Reads a line from a text file

Syntax:
```
fread(File: handle, string[], size=sizeof string,
      bool: pack=false)
```

`handle` The handle to an open file.

`string` The array to store the data in; this is assumed to be a text string.

`size` The (maximum) size of the array in cells. For a packed string, one cell holds multiple characters.

`pack` If the `pack` parameter is `false`, the text is stored as an *unpacked* string; otherwise a *packed* string is returned.

Returns: The number of characters read. If the end of file is reached, the return value is zero.

Notes: Reads a line of text, terminated by CR, LF or CR–LF characters, from to the file. Any line termination characters are stored in the string.

See also: `fblockread`, `fopen`, `fwrite`

---

fremove Delete a file or directory

Syntax:
```
bool: fremove(const name[])
```

`name` The name of the file or the directory.

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

Notes: A directory can only be removed if it is empty.

See also: `fmkdir`, `fexist`, `fopen`

frename Rename a file

Syntax: `bool: frename(const oldname[], const newname[])`

`oldname` The current name of the file, optionally including a full path.

`newname` The new name of the file, optionally including a full path.

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

Notes: In addition to changing the name of the file, this function can also move the file to a different directory.

See also: `fcopy`, `fremove`

fround Round a fixed point number to an integer value

Syntax:
```
fround(Fixed: value,
       fround_method: method=fround_round)
```

`value` The value to round.

`method` The rounding method may be one of:

`fround_round`
: round to the nearest integer; a fractional part of exactly 0.5 rounds upwards (this is the default);

`fround_floor`
: round downwards;

`fround_ceil`
: round upwards;

`fround_tozero`
: round downwards for positive values and upwards for negative values ("truncate");

`fround_unbiased`
: round to the nearest *even* integer number when the fractional part is exactly 0.5 (the values "1.5" and "2.5" both round to "2"). This is also known as "Banker's rounding".

Returns: The rounded value, as an integer (an untagged cell).

Notes: When rounding negative values upwards or downwards, note that −2 is considered smaller than −1.

See also: `ffract`

---

fseek Set the current position in a file

Syntax:
```
fseek(File: handle, position=0,
      seek_whence: whence=seek_start)
```

`handle` The handle to an open file.

`position` The new position in the file, relative to the parameter `whence`.

`whence` The starting position to which parameter `position` relates. It must be one of the following:

- `seek_start` Set the file position relative to the start of the file (the `position` parameter must be positive);
- `seek_current` Set the file position relative to the current file position: the `position` parameter is added to the current position;
- `seek_end` Set the file position relative to the end of the file (parameter `position` must be zero or negative).

Returns: The new position, relative to the start of the file.

Notes: You can either seek forward or backward through the file.

To get the current file position without changing it, set the `position` parameter to zero and `whence` to `seek_current`.

See also: `fopen`

fsqroot Return the square root of a value

Syntax: `Fixed: fsqroot(Fixed: value)`

`value`	The value to calculate the square root of.

Returns: The result: the square root of the input number.

Notes: This function raises a "domain" error is the input value is negative.

See also: `fpower`

fstat Return the size and the time stamp of a file

Syntax:
```
bool: fstat(const name[], &size=0, &timestamp=0,
            &attrib=0, &inode=0)
```

`name`	The name of the file.
`size`	If the function is successful, this parameter holds the size of the file on return.
`timestamp`	If the function is successful, this parameter holds the time of the last modification of the file on return.
`attrib`	If the function is successful, this parameter holds the file attributes.
`inode`	If the function is successful, this parameter holds inode number of the file. An inode number is a number that uniquely identifies a file, and it usually indicates the physical position of (the start of) the file on the disk or memory card.

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

Notes: In contrast to the function `flength`, this function does not need the file to be opened for querying its size.

The time is in number of seconds since midnight at 1 January 1970: the start of the UNIX system epoch.

The file attributes are a bit mask. The meaning of each bit depends on the underlying file system (e.g. FAT, NTFS, etx2 and others).

The inode number is useful for minimizing the gap between tracks when playing MP3 tracks sequentially. By storing the inode number and the file size of the next track in a "resource id" (while the H04x0 MP3 controller is still playing the current track), you avoid the time needed to search through the directory system of the FAT file system. See function `play` for details on resource ids.

See also: `fattrib`, `flength`

---

funcidx Return a public function index

Syntax: `funcidx(const name[])`

Returns: The index of the named public function. If no public function with the given name exists, `funcidx` returns −1.

amx_Exec: see the "Implementer's Guide"

Notes: A host application runs a public function from the script by passing the public function's index to `amx_Exec`. With this function, the script can query the index of a public function, and thereby return the "next function to call" to the application.

---

fwrite Write a string to a file

Syntax: `fwrite(File: handle, const string[])`

`handle` The handle to an open file.

`string` The string to write to the file.

Returns: The number of characters actually written; this may be a different value from the string length in case of a writing failure ("disk full", or quota exceeded).

Notes:

The function does not append line-ending characters to the line of text written to the file (line ending characters are CR, LF or CR–LF characters).

See also: `fblockwrite`, `fopen`, `fread`

---

getarg Get an argument

Syntax: `getarg(arg, index=0)`

`arg`	The argument sequence number, use 0 for first argument.
`index`	The index, in case `arg` refers to an array.

Returns: The value of the argument.

Notes: This function retrieves an argument from a variable argument list. When the argument is an array, the `index` parameter specifies the index into the array. The return value is the retrieved argument.

See also: `numargs`, `setarg`

---

getdate Return the current (local) date

Syntax: `getdate(&year=0, &month=0, &day=0)`

`year`	This will hold the year upon return.
`month`	This will hold the month (1–12) upon return.
`day`	This will hold the day of (1–31) the month upon return.

Returns: The return value is the number of days since the start of the year. January 1 is day 1 of the year.

See also: `gettime`, `setdate`

getiopin Read the indicated I/O pin

Syntax: `getiopin(pin)`

`pin` The pin number, between 0 and 15; or -1 to read the state of all 16 digital I/O pins as a bit mask.

Returns: If parameter `pin` is in the range 0...15, the return value is the logical value of the specified I/O pin: 0 or 1. If parameter `pin` is -1, the return value is a value where the first 16 bits represent the state of the respective I/O pins.

Notes: When a pin is defined as output, its latched value (usually the last value that the pin was set to) is returned. Pins that are reserved (for the LCD) always read back as zero. See function `configiopin` for configuring pins. After reset, all pins are configured as inputs (high-impedance).

This function always returns the current logical level of the pin, regardless of whether the public function `@input` is defined.

See also: `@input`, `@button`, `configiopin`, `@console`, `setiopin`

gettime Return the current (local) time

Syntax: `gettime(&hour=0, &minute=0, &second=0)`

`hour` This will hold the hour (0–23) upon return.

`minute` This will hold the minute (0–59) upon return.

`second` This will hold the second (0–59) upon return.

Returns: The return value is the number of seconds since midnight, 1 January 1970: the start of the UNIX system epoch.

See also: `getdate`, `settime`

getvolume Read the current volume and balance settings

Syntax: `getvolume(&volume=0, &balance=0)`

`volume` This (optional) parameter will hold the volume setting upon return. This is a value in the range 0–100.

`balance` This (optional) parameter will hold the balance setting upon return. This is a value in the range −100–100.

Returns: This function always returns `true` if a volume fade is currently in progress, and `false` if no fade was started or the fade has finished.

Notes: If the output channels are muted, the original volume settings will still be returned.

See also: `bass`, `setvolume`, `treble`

gotoxy Set the cursor position

Syntax: `gotoxy(x=1, y=1)`

`x` The horizontal position to move the cursor to.

`y` The vertical position to move the cursor to.

Returns: Always returns 0.

Notes: The upper left corner is at (1,1).

See also: `wherexy`

headerinfo Read frame header values

Syntax: `headerinfo(MP3Value: code)`

`code` The item from the frame header to read. One of the following:

`MP3_ID` (0)
: MP3 file format version, see the notes below.

`MP3_Layer` (1)
: MP3 file format layer.

`MP3_Bitrate` (2)
: The bit rate of the current frame, in kb/s.

`MP3_SampleFreq` (3)
: The sample frequency in Hz.

`MP3_Mode` (4)
: The audio mode (mono, stereo, ...), see the notes below.

`MP3_AvgBitrate` (5)
: The average bit rate as determined by the decoder, in kb/s.

`MP3_Length` (7)
: The track duration in milliseconds.

Returns: The value of the requested item.

Notes: The "ID" of the MP3 header gives the version of the format. This is one of the following values:

`ID_MPEG_2_5` (0) unofficial MPEG 2.5 extension (very low bit rates)

`ID_MPEG_2` (2) MPEG version 2

`ID_MPEG_1` (3) MPEG version 1

The "Layer" field indicates the layer of the format, which is a kind of "sub-version" —it is the "3" in the "MP3" identifier. The most common file type is MPEG version 1, layer 3, but versions 2 and 2.5 are supported too. The H0420 does not support layers 1 or 2.

An MPEG file consists of independent chunks, called "frames". Each frame has a frame header with the above information. Due to the frames being independent, changes in bit rate, or even sampling frequency, in the middle of a track are handled transparently. See the section "Resources" on page 112 for pointers to in-depth information on the MPEG audio file format.

The mode of a frame is one of the following values:

`MODE_Stereo`	(0) standard stereo
`MODE_JointStereo`	(1) single channel plus delta-signal (for the other channel)
`MODE_DualChannel`	(2) two independent channels (e.g. two languages)
`MODE_Mono`	(3) monaural sound

The average bit rate returned by this function is a average of the bit rates of the most recent MP3 frames that the audio processor on the H0420 has seen —it is not the average bit rate of the entire track. In a "constant bit rate" file, the average bit rate will be *constant* and have the same value as the bit rate of every frame. In a "variable bit rate" file, the bit rate of every frame may change and the average bit rate will smooth out these variations somewhat. The average bit rate will still fluctuate, however.

The duration of the track can be read from the header information as well as from the ID3 tag (see function `taginfo`). However, the length field is usually not present in the ID3 tag. The track duration can only be reliably calculated by this function for "variable bit rate" tracks (VBR) that have a "Xing" header, and for "constant bit rate" tracks (CBR). Some encoders create variable bit rate tracks without Xing header.

Example: See `mp3info.p` on page 11.

See also: `taginfo`

---

heapspace Return free heap space

Syntax: `heapspace()`

Returns: The free space on the heap. The stack and the heap occupy a shared memory area, so this value indicates the number of bytes that is left for either the stack or the heap.

Notes: In absence of recursion, the PAWN parser can also give an estimate of the required stack/heap space.

image Display an image

Syntax: `image(const filename[], x=0, y=0)`

`filename` The full name and path to the image file. See the notes for the supported image file format.

`x, y` The position in pixels for the upper left corner of the image. Depending on the display, the image may need to be aligned to a particular raster.

Returns: Always returns 0.

Notes: The image must be in "poly-raster image" format and be formatted for the appropriate display (LCD, VFD or other). For utilities to convert images to poly-raster format and a description of the format, see http://www.compuphase.com.

The display (LCD) must be configured with function `console` before calling this function. The display must furthermore support graphic operations.

See also: `consctrl`, `console`, `print`, `printf`

inputlapse Get precision time stamp of an I/O event

Syntax: `Fixed: inputlapse(Fixed: basestamp=0.0)`

`basestamp` The time stamp relative to which the returned value will be. This value is in units of 1 millisecond.

Returns: The interval in milliseconds between the most recent change on one of the I/O lines since the time stamp in `basestamp`

Notes: Because the values for the parameter `basestamp` and the function result have three decimal digits, the resolution of the timestamps are 1 microsecond.

The function returns the time interval between `basestamp` and the most recent change on any input pin. If multiple pins are configured as input, the return value be refer to a change on every pin.

See also: `@input`

---

ispacked Determine whether a string is packed or unpacked

Syntax: `bool: ispacked(const string[])`

`string` The string to verify the packed/unpacked status for.

Returns: `true` if the parameter refers to a packed string, and `false` otherwise.

---

max Return the highest of two numbers

Syntax: `max(value1, value2)`

`value1`
`value2` The two values for which to find the highest number.

Returns: The higher value of `value1` and `value2`.

See also: `clamp`, `min`

---

memcpy Copy bytes from one location to another

Syntax:
```
memcpy(dest[], const source[], index=0, numbytes,
       maxlength=sizeof dest)
```

`dest` An array into which the bytes from `source` are copied in.

`source` The source array.

`index` The index, in *bytes* in the source array starting from which the data should be copied.

`numbytes` The number of bytes (not cells) to copy.

`maxlength` The maximum number of *cells* that fit in the destination buffer.

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

Notes: This function can align byte strings in cell arrays, or concatenate two byte strings in two arrays. The parameter `index` is a byte offset and `numbytes` is the number of bytes to copy.

This function allows copying in-place, for aligning a byte region inside a cell array.

Endian issues (for multi-byte values in the data stream) are not handled.

See also: `strcopy`, `strpack`, `strunpack`, `uudecode`, `uuencode`

---

min Return the lowest of two numbers

Syntax: `min(value1, value2)`

`value1`
`value2` The two values for which to find the lowest number.

Returns: The lower value of `value1` and `value2`.

See also: `clamp`, `max`

---

mp3password Set the user password for encrypted tracks

Syntax: `mp3password(const password[])`

`password` A string containing your "user password" to use for the encrypted MP3 tracks.

Returns: This function currently always returns 0.

Notes: This function sets the "user password" for deciphering encrypted MP3 tracks. The user password must match the password that was used for encrypting the MP3 track. If the track was encrypted without user password, the `password` parameter should be an empty string.

The encryption algorithm uses both an internal, device-specific 128-bit "system key" and the user password to protect MP3 tracks. The user password is therefore an augmented protection. Even if the password "leaks out", the MP3 files can still only be played back on a hardware player with the appropriate system key. The system key is embedded in the firmware in a way that it cannot be read from the device even if a code breaker has full access to the device.

Unencrypted MP3 tracks will still play as before. Setting a user password has only effect on encrypted MP3 tracks.

---

mute Mute or unmute the audio

Syntax: `mute(bool: on)`

`on`	Set to `true` to silence the audio, or `false` to return to the previously set volume.

Returns: This function always returns 0.

Notes: This function does not change the volume and balance setting. When "unmuting", the device returns to the previously set volume.

When starting to play a new track (function `play`), the audio is unmuted implicitly.

See also: `play`, `setvolume`

numargs Return the number of arguments

Syntax: `numargs()`

Returns: The number of arguments passed to a function; `numargs` is useful inside functions with a variable argument list.

See also: `getarg`, `setarg`

packetfilter Filter received RS232 data

Syntax:
```
packetfilter(const format[]="",
             const filter[]="*", timeout=0)
```

`format` A string describing the format of the packets that are (by reasonable assumption) received on the RS232 line. The data is collected in an internal buffer until a complete packet is received or until the reception of the packet times out.

`filter` A string describing the contents of packets that are acceptable. Any packet that does not match the filter is rejected; i.e. the script will not receive a `@receivepacket` event for rejected packets. The default filter, a single asterisk ("`*`") matches *any* packet.

`timeout` The time in milliseconds to wait for a packet to be completed. When a packet is incomplete and no more data is received within this time period, the packet is assumed *invalid*.

Returns: This function always returns 0.

Notes: The strings for the `format` and `filter` may contain the wild card characters described for filename matching. See page 19 for details. A pattern may contain at most 64 characters (including the zero byte that terminates the string). The maximum size of a packet is also 64 bytes.

When a packet is received and the packet matches the filter, the script receives an `@receivepacket` event with the packet as its parameter. This relieves the script from finding the packet boundaries itself and do its filtering in the script.

Received bytes that do not form a valid packet (according to the definition in the `format` parameter) are directed to the event function `@receivebyte`. If the script does not contain a `@receivebyte` function, the non-conforming bytes are dropped. Any packets that matches the format, but fails the filter is dropped, even if a `@receivebyte` function is present.

The serial port must have been set up ("opened") before using this function. If software handshaking is enabled (see function `setserial`), bytes with the values 17 (`0x11`, Ctrl-Q), 19 (`0x13`, Ctrl-S) will be handled internally, and these bytes are then *not* received. These values denote the XON and XOFF signals.

See also: `@receivepacket`, `setserial`

---

pause Pauses playback

Syntax: `pause(fadeout=250)`

`fadeout`	The time to use for fading out the audio when a track is cut off by another, in milliseconds.

Returns: `true` on success, `false` on failure (no audio is currently playing).

Notes: The function waits until the fade is complete.

See also: `play`, `resume`, `stop`

play Start playing an audio file

Syntax:
```
bool: play(const filename[], repeats=0,
           fadeout=250, fadein=0)
```

`filename` The full filename and path of the file, or a *resource id* for the file. See the notes for the format of a resource id.
For firmware editions that include network support, the filename may also be an URL to a track on a HTTP server or an URL to a streaming server. See the function `netstream` in the network functions addendum for details.

`repeats` The number of times that the audio segment should be repeated. When set to zero (the default value), the audio file plays only once. When set to 255, the audio file is repeated indefinitely until it is explicitly stopped or until another file is scheduled to play.

`fadeout` The time to use for fading out the audio when a track is cut off by another, in milliseconds.

`fadein` The time to use for fading in an audio track as it starts, in milliseconds.

Returns: `true` on success, `false` on failure (file not found or invalid format).

Notes: Due to the format of "Layer 3" MPEG file, frames are not as independent as the MPEG standard implies. When you start playing a file while another file is already playing, an audible glitch may occur when the "bit reservoir" belonging to the preceding track cannot be synchronized with the starting frames of the new track. The only solution (apart from encoding the MP3 files to *not use* the bit reservoir) is to suppress this audible glitch with a fade-out. A fade in at the start of a track is optional.

Note that the `fadeout` parameter is to fade out the track that is playing *before* the new track starts. If the MP3 player is

not playing a track at the time function `play` is called, the parameter `fadeout` is ignored.

If the audio outputs were muted, the mute is turned off by the `play` command.

Instead of a path and filename of an MP3 track, you can also pass in a "resource id" of the track. The resource id is an array with three values:

- ◇ Array index 0 (the first cell of the array) must have the value 1.
- ◇ Array index 1 must have the "inode" number of the file, see `fstat`.
- ◇ Array index 2 must have the size of the file in bytes (also obtained with `fstat`).

The purpose of resource id's is that looking up a track in the directory structure may be a time-consuming operation if you have many MP3 tracks on the card. With `fstat`, the script can prepare the parameters of the *next* track to play and store it in a resource id —all while the device is playing another track. When that track ends, the script plays the resource id. Since no more "looking up" is necessary, the prepared track plays immediately. Thus, playing a resource id allows you to minimize the gap between tracks.

Example: See `mp3info.p` on page 11 and `serial.p` on page 14.

See also: `fstat`, `mute`, `stop`

---

print Display a (partial) string

Syntax: `print(const string[], start=0, end=cellmax)`

`string`	The string to display on the LCD.
`start`	The character to start printing with (the number of characters to skip at the start of the string).
`end`	An index in the string *behind* the last character that is printed. In other words, the number of characters printed is `end−start`.

Returns: Always returns 0.

Notes: This function displays a plain string on the LCD, without interpreting placeholders. Control characters *are* taken into account, though. By setting the optional parameters `start` and `end`, you can also display part of a string.

The LCD must be configured with function `console` before calling this function.

See also: `console`, `image`, `printf`

---

printf Display a formatted string

Syntax: `printf(const format[], Fixed,_:...)`

`format` The string to display, which may contain placeholders (see the notes below).

`...` The parameters for the placeholders. These values may be untagged, weakly tagged, or tagged as "Fixed" point values.

Returns: Always returns 0.

Notes: Prints a string with embedded *placeholder* codes:

- `%c` print a character at this position
- `%d` print a number at this position in decimal radix
- `%q` same a `%r` (for compatibility with other implementations of PAWN)
- `%r` print a fixed point number at this position
- `%s` print a character string at this position
- `%x` print a number at this position in hexadecimal radix

The values for the placeholders follow as parameters in the call.

You may optionally put a number between the "%" and the letter of the placeholder code. This number indicates the field width; if the size of the parameter to print at the position of the placeholder is smaller than the field width, the field is expanded with spaces.

The `printf` function works similarly to the `printf` function of the C language.

The LCD must be configured with function `console` before calling this function.

Example: See `mp3info.p` on page 11.

See also: `console`, `image`, `print`, `strformat`

---

random Return a pseudo-random number

Syntax:
```
random(max)
```

`max` The limit for the random number.

Returns: A pseudo-random number in the range `0` – `max-1`.

Notes: The random-number generator is based on a cryptographical function and it is considered to produce good quality pseudo-random numbers. The generator chooses its own seed at each power-up.

---

readcfg Reads a text field from an INI file

Syntax:
```
readcfg(const filename[]="", const section[]="",
        const key[], value[], size=sizeof value,
        const defvalue[]="", bool: pack=true)
```

`filename` The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".

`section`	The section to look for the `key`. If this parameter is not set, the function reads the key outside any section.
`key`	The key whose value must be looked up.
`value`	The buffer into which the field that is read is stored into. If the `key` cannot be found in the appropriate `section` of the INI file, this field will contain the `defvalue` parameter upon return.
`size`	The (maximum) size of the `value` array in cells. For a packed string, one cell holds multiple characters.
`defvalue`	The string to copy into parameter `value` in case that the function cannot read the field from the INI file.
`pack`	If the `pack` parameter is `false`, the text is stored as an *unpacked* string; otherwise a *packed* string is returned.

Returns: The number of characters stored in parameter `value`.

See also: `readcfgvalue`, `writecfg`

---

readcfgvalue Reads a numeric field from an INI file

Syntax:
```
readcfgvalue(const filename[]="",
             const section[]="", const key[],
             defvalue=0)
```

`filename`	The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".
`section`	The section to look for the `key`. If this parameter is not set, the function reads the key outside any section.
`key`	The key whose value must be looked up.

`defvalue` The value to return in case that the function cannot read the field from the INI file.

Returns: The numeric value if the field, or the value of `defvalue` if the field was not found in the section and/or at the key.

See also: `readcfg`, `writecfgvalue`

---

readconfig Read device configuration

Syntax: `readconfig(data[], size=sizeof data)`

`data` An array that will contain the data read from the configuration area upon return of this function.

`size` The number of cells to read in the array. The maximum size if 64 cells.

Returns: This function currently always returns 0.

Notes: The H04x0 series of MP3 controllers have an auxiliary non-volatile memory area into which the script can store data. Typically, device configurations that should be saved even when the CompactFlash card is exchanged, are stored in the configuration area. The data in the configuration area is saved even when the power is removed.

See also: `storeconfig`

---

receivebyte Receive a single byte over the serial line

Syntax: `receivebyte(timeout, port=1)`

`timeout` The number of milliseconds to wait for data. When this parameter is zero, the function returns the first byte in the queue if one is present, but does not wait if the queue is empty.

`port` For devices supporting multiple serial ports, this parameter specifies which port to receive from.

Returns: The byte read on success, or a value less then zero on failure (operation timed out).

Notes: The serial port must have been set up ("opened") before using this function.

If software handshaking is enabled (see function `setserial`), bytes with the values 17 (`0x11`, Ctrl-Q), 19 (`0x13`, Ctrl-S) will be handled internally, and these bytes are then *not* received. These values denote the XON and XOFF signals.

See also: `@receive`, `@receivebyte`, `sendbyte`, `setserial`

---

reset Causes a full reset

Syntax: `reset(bool: wait=true)`

`wait`	Whether to wait for the reset, which happens in approximately 1.5 seconds after calling this function. If true, the function does not return. If false, the function returns immediately, giving the script a chance to complete an operation before the reset occurs.

Returns: If parameter `wait` is true (or if the parameter is absent), the function does not return. If parameter `wait` is false, the function returns 0.

Notes: When this function is called, the H0420 goes into a reset. This also causes function `@reset` (in the script) to be invoked again.

The H0420 will poll for a debugger on the RS232 after a programmed reset, regardless of whether the script on the Compact Flash card was built with debug information. If no debugger is present, the polling causes a start-up delay of 2 seconds.

Example: See the debugger support function on page 30.

See also: `@reset`, `standby`

resume Resumes playback that was paused earlier

Syntax: `resume(fadein=0)`

`fadein` The time to use for fading in an audio track as it starts, in milliseconds.

Returns: `true` on success, `false` on failure (i.e. no audio is currently paused).

Notes: The difference between `resume` and `play` is that `resume` will resume playback from the position where the audio was paused earlier; `play` will always start playing from the beginning of the track.

See also: `pause`, `play`

seekto Set the position in the MP3 track

Syntax: `seekto(milliseconds,fade=250)`

`milliseconds` The position to move to, in milliseconds from the start of the track.

`fade` The time to use for fading out the audio track before changing its playback position, and to fade it in when resuming the track. The fade time is in milliseconds.

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

Notes: You must have started to play the track before you can seek to a position.

See function `headerinfo` to get the duration of the track. To get the current position into a playing track, you should obtain a time stamp (function `tickcount`) and subtract from this the time stamp at which the track started to play.

You cannot seek in encrypted tracks; function `seekto` will return failure if encryption is configured.

Seeking to a position is accurate for "constant bit rate" tracks (CBR); it is *fairly* accurate for "variable bit rate" tracks (VBR) that have a "Xing" header. When a variable bit rate track lacks a Xing header, the `seekto` function works, but the seek position may be inaccurate.

See also: `headerinfo`, `mp3password`, `play`

---

sendbyte Send a single byte over the serial line

Syntax: `sendbyte(value, port=1)`

`value`	The byte to send.
`port`	For devices supporting multiple serial ports, this parameter specifies which port to use.

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

Notes: The serial port must have been set up ("opened") before using this function.

To receive data from the serial port, the script must implement either the `@receive` public function or the `@receivebyte` function. See page 33 for details. Alternatively, one may use the `receivebyte` function to poll for serial input.

If software handshaking is enabled (see function `setserial`), bytes with the values 17 (`0x11`, Ctrl-Q), 19 (`0x13`, Ctrl-S) cannot be sent either, because these denote the XON and XOFF signals. When you need to transfer binary data, you should encode it using a protocol like UU-encode.

See also: `@receive`, `receivebyte`, `sendstring`, `setserial`

sendstring Send a string over the serial line

Syntax: `sendstring(const string[], port=1)`

`string` The string to send.

`port` For devices supporting multiple serial ports, this parameter specifies which port to use.

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

Notes: The serial port must have been set up ("opened") before using this function.

To receive data from the serial port, the script must implement either the `@receive` public function or the `@receivebyte` function. See page 33 for details. Alternatively, one may use the `receivebyte` function to poll for serial input.

The maximum string length that can be sent with this function is currently 256 characters.

If software handshaking is enabled (see function `setserial`), bytes with the values 17 (`0x11`, Ctrl-Q), 19 (`0x13`, Ctrl-S) cannot be sent either, because these denote the XON and XOFF signals. When you need to transfer binary data, you should encode it using a protocol like UU-encode.

Example: See `serial.p` on page 14.

See also: `@receive`, `sendbyte`, `setserial`

setalarm Set the timer alarm

Syntax:
```
setalarm(year=-1, month=-1, day=-1, weekday=-1,
         hour=-1, minute=-1, second=-1)
```

`year` The year to match for the alarm, or -1 for not matching the year for the alarm. This value must be in the range 1970–2099.

`month` The month to match for the alarm, or -1 for not matching the month for the alarm. This value must be in the range 1–12.

`day` The day to match for the alarm, or -1 for not matching the day for the alarm. This value must be in the range 1–31 (or the last valid day of the month).

`weekday` The "day of the week" to match for the alarm, or -1 for not matching the day of the week for the alarm. This value must be in the range 1–7, where Monday is day 1.

`hour` The hour to match for the alarm, or -1 for not matching the hour for the alarm. This value must be in the range 0–23.

`minute` The minute to match for the alarm, or -1 for not matching the minute for the alarm. This value must be in the range 0–59.

`second` The second to match for the alarm, or -1 for not matching the second for the alarm. This value must be in the range 0–59.

Returns: This function currently always returns 0.

Notes: This function sets the alarm to go off at a specific time. All parameters of this function are optional, and you can switch the alarm off by leaving all parameters at their default value when calling the function.

The alarm may be fully specified, with a day, month and year as well as a complete time with hour, minute and second. Such a timer will only go off once. Another usage is to set an alarm at a recurring event, such as every day at 7:15 o'clock. For this purpose, one would set only the `hour` and `minute` parameters (to 7 and 15 respectively) and leave the rest at −1.

The alarm function needs the current time and date to be set in the H0420 accordingly. On a power-on, the device starts at midnight 1 January 1970.

See also: `@alarm`, `setdate`, `settime`

setarg Set an argument

Syntax: `setarg(arg, index=0, value)`

`arg`	The argument sequence number, use 0 for first argument.
`index`	The index, in case `arg` refers to an array.
`value`	The value to set the argument to.

Returns: `true` on success and `false` if the argument or the index are invalid.

Notes: This function sets the value of an argument from a variable argument list. When the argument is an array, the `index` parameter specifies the index into the array.

See also: `getarg`, `numargs`

setattr Set LCD contrast

Syntax: `setattr(contrast)`

`contrast`	The new contrast value; a value between 0 and 255. Suitable values are usually between 20 and 60 for LCDs and between 100 and 200 for OLED or PLED modules.

Returns: Always returns 0.

Notes: The LCD must be configured with function `console` before calling this function.

See also: `consctrl`, `console`

setdate Set the system date

Syntax: `setdate(year=cellmin, month=cellmin, day=cellmin)`

`year` The year to set; if this parameter is kept at its default value ("`cellmin`") it is ignored.

`month` The month to set; if this parameter is kept at its default value ("`cellmin`") it is ignored.

`day` The month to set; if this parameter is kept at its default value ("`cellmin`") it is ignored.

Returns: This function always returns 0.

The date fields are kept in a valid range. For example, when setting the month to 13, it wraps back to 1.

See also: `getdate`, `settime`, `settimestamp`

setiopin Set the indicated I/O pin

Syntax: `setiopin(pin, status)`

`pin` The pin number, between 0 and 16; or -1 to set the status of all digital I/O pins using a bit mask in `status`.

`status` The new status for the pin. This is a logical value (0 or 1) for the digital pins 0...15 and a value between 0 and 1023 for the analogue pin 16. If `pin` is -1, this parameter is interpreted as a bit mask where the first 16 bits represent the desired output state of pins 0...15.

Returns: The previous state of the pin; this may either be a logical value (0 or 1) or a bit mask, depending on parameter `pin`.

Notes: Only pins that are configured as outputs can be set; see the function `configiopin` for configuring pins. After reset, all pins are configured as inputs.

Pin 16 is an analogue pin. It is hard-wired as an output pin and it cannot be read. If a wave generator has been set up on pin 16, you should not set the pin to a value with `setiopin`. The analogue pin is not available when an LCD is set up —but see function `setattr`.

Example: See `switches2.p` on page 3 and `sylt.p` on page 8.

See also: `configiopin`, `getiopin`, `wavegenerator`

---

setserial Configure the serial port

Syntax:
```
setserial(baud=57600, databits=8, stopbits=1,
          parity=0, handshake=0, port=1)
```

`baud` The Baud rate, up to 115200. The standard Baud rates are 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 and 115200. The serial port also supports non-standard Baud rates. When this parameter is zero, the serial port is closed.

`databits` The number of data bits, a value between 5 and 8.

`stopbits` The number of stop bits, 1 or 2.

`parity` The parity options, one of the following:
0 disable
1 odd
2 even
3 mark (force 1)
4 space (force 0)

`handshake` The handshaking options; 0 for no handshaking and 1 for software handshaking.

`port` For devices supporting multiple serial ports, this parameter specifies which port to set up.

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

Notes: Software handshaking uses the characters XOFF (ASCII 19, Ctrl-S) to request that the other side stops sending data and XON (ASCII 17, Ctrl-Q) to request that it resumes sending data. These characters can therefore not be part of the normal data stream (as they would be misinterpreted as control codes).

In a data transfer both sides must agree on the protocol. As the settings are sometimes fixed on the apparatus that you wish to attach to the H0420 player, the RS232 interface of the H0420 is designed to fit a wide range of protocols.

The Baud rate is a trade-off between transfer speed and reliability of the connection: in noisy environments or with long cables, you may need to reduce the Baud rate.

The number of data bits is usually 8, occasionally 7 and rarely 6 or 5. With 8 databits, the number of stop bits is typically 1.

Mark and space parity codes are rarely used.

Example: See `serial.p` on page 14.

See also: `@receive`, `@receivebyte`, `sendbyte`, `sendstring`

---

settime — Set the system time

Syntax:
```
settime(hour=cellmin, minute=cellmin,
        second=cellmin)
```

`hour` — The hour to set, in the range 0–23; if this parameter is kept at its default value ("`cellmin`") it is ignored.

`minute` — The minute to set, in the range 0–59; if this parameter is kept at its default value ("`cellmin`") it is ignored.

`second` — The second to set, in the range 0–59; if this parameter is kept at its default value ("`cellmin`") it is ignored.

Returns: This function always returns 0.

The time fields are kept in a valid range. For example, when setting the hour to 24, it wraps back to 23.

See also: `gettime`, `setdate`, `settimestamp`

---

settimer Configure the event timer

Syntax: `settimer(milliseconds, bool: singleshot=false)`

`milliseconds`
: The number of milliseconds to wait before calling the `@timer` callback function. Of the timer is repetitive, this is the interval. When this parameter is 0 (zero), the timer is shut off.

`singleshot`
: If `false`, the timer is a repetitive timer; if `true` the timer is shut off after invoking the `@timer` event once.

Returns: This function always returns 0.

Notes: See the chapter "Usage" for an example of this function, and the `@timer` event function.

See also: `@timer`, `tickcount`

---

settimestamp Sets the date and time with a single value

Syntax: `settimestamp(seconds1970)`

`seconds1970`
: The number of seconds that have elapsed since midnight, 1 January 1970. This particular date, 1 January 1970, is the "UNIX system epoch".

Returns: This function always returns 0.

Notes: The function `getdate` returns the number of seconds since 1 January 1970.

See also: `getdate`, `setdate`, `settime`

setvolume Set the audio volume and balance

Syntax:
```
setvolume(volume=cellmin, balance=cellmin,
          fadetime=0)
```

`volume` This (optional) parameter holds the new volume level, a value in the range 0...100.

`balance` This (optional) parameter holds the new balance setting, a value in the range −100...100.

`fadetime` The duration in milliseconds to take for the volume or balance change.

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

Notes: If the output channels are muted, the new settings take effect as soon as the audio is unmuted.

The value for the volume level is relative to the range set with `volumebounds`.

Fading the change in volume (or balance) happens in the background. The script continues running while the fading takes place (this is in contrast with the "fade" parameters of functions like `play` and `stop`, that wait until the fade is complete). When fading is complete, the script receives an `@audiostatus` event with the code `FadeCompleted`. Function `getvolume` can also be used to check whether a fade is in progress.

Example: See `serial.p` on page 14.

See also: `bass`, `getvolume`, `mute`, `treble`, `volumebounds`

spi Send SPI data

Syntax:
```
spi(const data[], size=sizeof data, frequency=1,
    select=1, mode=1)
```

`data` An array with the bytes to send. Each element (cell) holds a byte to send. To read data from a device, one typically sends a zero byte.

`size` The number of elements (in parameter `data`) to send.

`frequency` The SPI clock frequency in MHz. The default value of 1 means a 1 MHz clock.

`select` The SPI "chip select" line (also called "slave select"). The SPI hardware reserves two chip select pins on the extension connector; see the notes for the details. When this parameter is set to zero, no chip select is issued at all.

`mode` The SPI mode to use; valid values are in the range 0...3. See the notes for details.

Returns: The last value returned by the remote device.

Notes: The H0420 has an SPI bus on its extension connector. To use the SPI bus, you must therefore connect the device to communicate with to the relvant pins on the extension connector. The data sheet documents the pins to use. Briefly, the pins are:

- ◇ pin 44: clock
- ◇ pin 21: serial out (MOSI)
- ◇ pin 45: serial in (MISO)
- ◇ pin 18: chip select line 1
- ◇ pin 16: chip select line 2

Since there are two chip select lines, the function can communicate with two SPI devices.

The `data` array must hold a byte in each cell element. In other words, it must be an unpacked array.

When reading data from a device, write a byte with the `spi` function and use the return value. That is, call `spi` with `size` set to 1 (and a valid `data` parameter) and the value returned by the `spi` function is the value that the device returned. Some devices require additional time to process a read command. In such a case, send two bytes with the `spi` function instead of one.

SPI is flexible in its specification of the clock polarity and the sampling flank (the "phase"). The SPI "mode" selects one of

the four possible configurations. Another method that is often used is to specify the polarity and phase separately (these are denoted as "cpol" and "cphase"). The relation between these values is:

- ◇ mode 0: cpol = 0, cphase = 0
- ◇ mode 1: cpol = 0, cphase = 1
- ◇ mode 2: cpol = 1, cphase = 0
- ◇ mode 3: cpol = 1, cphase = 1

---

standby Puts the device in "low power" mode

Syntax: `standby()`

Returns: This function only returns after de device has come out of stand-by mode; the return value is always zero.

Notes: In low-power mode, the CPU and many peripheral components of the H0420 are shut down to conserve power. The device resumes from stand-by upon detection of a change in the switch status or the signal level of an I/O pin. The script will handle the event (on the switch or I/O pin) after resuming —the event is not lost). Only events on the switch and I/O pin inputs will take the device out of low-power mode. Activity at the RS232 port and internal events like timer ticks will *not* power-up the device.

Typical current consumption values or the H0420 are:

- ◇ 140 mA when playing audio
- ◇ 105 mA when idle
- ◇ 50 mA in stand-by (low power) mode

These current consumption values exclude current consumption of attached components or peripherals, like LEDs, optocouplers or an LCD. Current consumption of the CompactFlash card is included, but variance in the specifications of CompactFlash cards of different brands may cause the above current consumption values to be off by 10%.

If low-power mode is used in combinations with the `exec` function, all scripts *must* include the low-power functionality. The

reason for this requirement is the presence of a special monitoring circuit on the H0420, which verifies the correct functioning of that the CPU and the embedded operating system. This circuit interferes with low-power mode because low-power mode halts the CPU and many peripheral functions. Therefore, this circuit must not be started if the script needs to switch to low-power mode (once started, the circuit *cannot be disabled*).

The software programmable watchdog functionality is disabled in low-power mode.

See also: `exec`, `reset`, `watchdog`

---

stop Stop playback

Syntax: `stop(fadeout=0)`

`fadeout`	This (optional) parameter is the time used for fading out the audio prior to stopping.

Returns: `true` on success, `false` on failure (no audio is currently playing).

Notes: The difference between this function and function `pause` is that a paused track may be resumed. The `stop` function releases the resources for the track and resets the audio hardware.

The `fadeout` parameter is only taken into account when the device was playing a track; when not playing audio, the function returns immediately.

Example: See `serial.p` on page 14.

See also: `pause`, `play`

storeconfig Read device configuration

Syntax: `storeconfig(const data[], size=sizeof data)`

`data` An array that contains the data to be stored in the configuration area.

`size` The number of cells to store in the configuration area. The maximum size if 64 cells.

Returns: This function currently always returns 0.

Notes: The H04x0 series of MP3 controllers have an auxiliary non-volatile memory area into which the script can store data. Typically, device configurations that should be saved even when the CompactFlash card is exchanged, are stored in the configuration area. The data in the configuration area is saved even when the power is removed.

The size of the configuration area is small: only 64 cells. Large amounts of data should be stored on the memory card via the file functions.

Although reading from the configuration area is fast, writing to it is slow. In addition, the configuration area can be re-written 100,000 times on the average. Since the configuration area is internal to the H04x0 MP3 controller, you need to replace the board once the configuration area becomes defective due to exceeding the number of re-writes. The configuration area is intended to be updated only infrequently.

See also: `readconfig`

strcat Concatenate two strings

Syntax:
```
strcat(dest[], const source[],
       maxlength=sizeof dest)
```

`dest` The buffer in which the result will be stored. This buffer already contains the first part of the string.

`source` The string to append to the string in `dest`.

`maxlength` If the length of `dest` would exceed `maxlength` cells after the string concatenation, the result is truncated to `maxlength` cells.

Returns: The string length of `dest` after concatenation.

Notes: During concatenation, the `source` string may be converted from packed to unpacked, or vice versa, in order to match `dest`. If `dest` is an empty string, the function makes a plain copy of `source`, meaning that the result (in `dest`) will be a packed string if `source` is packed too, and unpacked otherwise.

See also: `strcopy`, `strins`, `strpack`, `strunpack`

---

strcmp Compare two strings

Syntax:
```
strcmp(const string1[], const string2[],
       bool: ignorecase=false, length=cellmax)
```

`string1` The first string in the comparison.

`string2` The first string in the comparison.

`ignorecase` If logically "true", case is ignored during the comparison.

`length` The maximum number of characters to consider for comparison.

Returns: The return value is:
−1 if `string1` comes *before* `string2`,
1 if `string1` comes *after* `string2`, or
0 if the strings are equal (for the matched length).

Notes: Packed and unpacked strings may be mixed in the comparison.

This function does *not* take the sort order of non-ASCII character sets into account. That is, no Unicode "Collation Algorithm" is used.

See also: `strequal`, `strfind`

strcopy Create a copy of a string

Syntax:
```
strcopy(dest[], const source[],
        maxlength=sizeof dest)
```

`dest` The buffer to store the copy of the string string in.

`source` The string to copy, this may be a packed or an unpacked string.

`maxlength` If the length of `dest` would exceed `maxlength` cells, the result is truncated to `maxlength` cells. Note that several packed characters fit in each cell.

Returns: The number of characters copied.

Notes: This function copies a string from `source` to `dest`. If the source string is a packed string, the destination will be packed too; likewise, if the source string is unpacked, the destination will be unpacked too. See functions `strpack` and `strunpack` to convert between packed and unpacked strings.

See also: `strcat`, `strpack`, `strunpack`

---

strdel Delete characters from the string

Syntax:
```
bool: strdel(string[], start, end)
```

`string` The string from which to remove a range characters.

`start` The parameter `start` must point at the first character to remove (starting at zero).

`end` The parameter `end` must point *behind* the last character to remove.

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

Notes: For example, to remove the letters "ber" from the string "Jabberwocky", set `start` to 3 and `end` to 6.

See also: `strins`

strequal Compare two strings

Syntax:
```
bool: strequal(const string1[], const string2[],
               bool: ignorecase=false,
               length=cellmax)
```

`string1` The first string in the comparison.

`string2` The first string in the comparison.

`ignorecase` If logically "true", case is ignored during the comparison.

`length` The maximum number of characters to consider for

Returns: `true` if the strings are equal, `false` if they are different.

See also: `strcmp`

strfind Search for a sub-string in a string

Syntax:
```
strfind(const string[], const sub[],
        bool: ignorecase=false, index=0)
```

`string` The string in which you wish to search for sub-strings.

`sub` The sub-string to search for.

`ignorecase` If logically "true", case is ignored during the comparison.

`index` The character position in `string` to start searching. Set to 0 to start from the beginning of the string.

Returns: The function returns the character index of the first occurrence of the string `sub` in `string`, or −1 if no occurrence was found. If an occurrence was found, you can search for the next occurrence by calling `strfind` again and set the parameter `offset` to the returned value plus one.

Notes: This function searches for the presence of a sub-string in a string, optionally ignoring the character case and optionally starting at an offset in the string.

See also: `strcmp`

---

strfixed Convert from text (string) to fixed point

Syntax: `Fixed: strfixed(const string[])`

`string` The string containing a fixed point number in characters. This may be either a packed or unpacked string. The string may specify a fractional part, e.g., "`123.45`".

Returns: The value in the string, or zero if the string did not start with a valid number.

---

strformat Convert values to text

Syntax:
```
strformat(dest[], size=sizeof dest,
          bool: pack=false, const format[], ...)
```

`dest` The string that will contain the formatted result.

`size` The maximum number of *cells* that the `dest` parameter can hold. This value includes the zero terminator.

`pack` If `true`, the string in `dest` will become a packed string. Otherwise, the string in `dest` will be unpacked.

`format` The string to store in `dest`, which may contain placeholders (see the notes below).

`...` The parameters for the placeholders. These values may be untagged, weakly tagged, or tagged as rational values.

Returns: This function always returns 0.

Notes: The `format` parameter is a string that may contain embedded *placeholder* codes:

- `%c` store a character at this position
- `%d` store a number at this position in decimal radix
- `%q` store a fixed point number at this position
- `%r` same as `%q` (for compatibility with other implementations of PAWN)
- `%s` store a character string at this position
- `%x` store a number at this position in hexadecimal radix

The values for the placeholders follow as parameters in the call.

You may optionally put a number between the "`%`" and the letter of the placeholder code. This number indicates the field width; if the size of the parameter to print at the position of the placeholder is smaller than the field width, the field is expanded with spaces.

The `strformat` function works similarly to the `sprintf` function of the C language.

See also: `valstr`

---

strins Insert a sub-string in a string

Syntax:
```
bool: strins(string[], const substr[], index,
             maxlength=sizeof string)
```

`string` The source and destination string.

`substr` The string to insert in parameter `string`.

`index` The character position of `string` where `substr` is inserted. When 0, `substr` is prepended to `string`.

`maxlength` If the length of `dest` would exceed `maxlength` cells after insertion, the result is truncated to `maxlength` cells.

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

Notes: During insertion, the `substr` parameter may be converted from a packed string to an unpacked string, or vice versa, in order to match `string`.

If the total length of `string` would exceed `maxlength` cells after inserting `substr`, the function raises an error.

See also: `strcat`, `strdel`

---

strlen Return the length of a string

Syntax:
```
strlen(const string[])
```

`string` The string to get the length from.

Returns: The length of the string in characters (not the number of cells). The string length *excludes* the terminating "\0" character.

Notes: Like all functions in this library, the function handles both packed and unpacked strings.

To get the number of *cells* held by a packed string of a given length, you can use the predefined constants `charbits` and `cellbits`.

See also: `ispacked`

---

strmid Extract a range of characters from a string

Syntax:
```
strmid(dest[], const source[],
       start=0, end=cellmax,
       maxlength=sizeof dest)
```

`dest` The string to store the extracted characters in.

`source` The string from which to extract characters.

`start` The parameter `start` must point at the first character to extract (starting at zero).

`end` The parameter `end` must point *behind* the last character to extract.

`maxlength` If the length of `dest` would exceed `maxlength` cells, the result is truncated to `maxlength` cells.

Returns: The number of characters stored in `dest`.

Notes: The parameter `start` must point at the first character to extract (starting at zero) and the parameter `end` must point *behind* the last character to extract. For example, when the source string contains "Jabberwocky", `start` is 1 and `end` is `5`, parameter `dest` will contain "abbe" upon return.

See also: `strdel`

---

strpack Create a "packed" copy of a string

Syntax:
```
strpack(dest[], const source[],
        maxlength=sizeof dest)
```

`dest` The buffer to store the packed string in.

`source` The string to copy, this may be a packed or an unpacked string.

`maxlength` If the length of `dest` would exceed `maxlength` cells, the result is truncated to `maxlength` cells. Note that several packed characters fit in each cell.

Returns: The number of characters copied.

Notes: This function copies a string from `source` to `dest` where the destination string will be in packed format. The source string may either be a packed or an unpacked string.

See also: `strcat`, `strunpack`

strunpack Create an "unpacked" copy of a string

Syntax:
```
strunpack(dest[], const source[],
          maxlength=sizeof dest)
```

`dest` The buffer to store the unpacked string in.

`source` The string to copy, this may be a packed or an unpacked string.

`maxlength` If the length of `dest` would exceed `maxlength` cells, the result is truncated to `maxlength` cells.

Returns: The number of characters copied.

Notes: This function copies a string from `source` to `dest` where the destination string will be in unpacked format. The source string may either be a packed or an unpacked string.

See also: `strcat`, `strpack`

strval Convert from text (string) to numbers

Syntax:
```
strval(const string[], index=0)
```

`string` The string containing a number in characters. This may be either a packed or unpacked string.

`index` The position in the string where to start looking for a number. This parameter allows to skip an initial part of a string, and extract numbers from the middle of a string.

Returns: The value in the string, or zero if the string did not start with a valid number (starting at `index`).

See also: `valstr`

swapchars Swap bytes in a cell

Syntax: `swapchars(c)`

`c` The value for which to swap the bytes.

Returns: A value where the bytes in parameter "`c`" are swapped (the lowest byte becomes the highest byte).

taginfo Return ID3 tag information

Syntax:
```
taginfo(ID3String: code, destination[],
        size=sizeof destination)
```

`code` The code for the requested field, one of the following:

`ID3_Title` (0)
: Track title.

`ID3_Artist` (1)
: Name of the artist or band.

`ID3_Album` (2)
: Album title.

`ID3_Comment` (3)
: General comment.

`ID3_Copyright` (4)
: Copyright information.

`ID3_Year` (5)
: Year of the album.

`ID3_Track` (6)
: The track number on the original CD.

`ID3_Length` (7)
: Duration of the track in milliseconds.

`destination`
: The buffer that will hold the returned tag field as a packed string. This will be an empty string if no tag is present or if the requested field is not in the tag.

`size` The size of the destination buffer in cells. Since the field is stored as a packed string, the number of characters that fit in the buffer is 4 times the value of this parameter.

Returns: This function currently always returns 0.

Notes: See section "Resources" on page 112 for details on the ID3 tag and pointers to software to create the tags.

The `SYLT` (Synchronized lyrics) tag is not returned by this function, but events or cues in the `SYLT` tag "fire" the public function `@synch` at the appropriate times.

The H0420 supports version 2 of the ID3 tag. The presence of an ID3 tag in an MP3 file is entirely optional. If it is included, all fields in the tag are each optional as well. The field for the duration of the track is frequently absent from tracks extracted from audio CDs, for example, whereas the track number is typically set only for tracks extracted from an audio CD.

The support for Unicode frames in the ID3 tag is limited to the characters of the Basic Multilingual Plane.

The duration of the track can also be read from the header information, see function `headerinfo`.

Example: See `mp3info.p` on page 11.

See also: `@synch`, `headerinfo`

---

tickcount Return the current tick count

Syntax: `tickcount(&granularity=0)`

`granularity` Upon return, this value contains the number of ticks that the internal system time will tick per second. This value therefore indicates the accuracy of the return value of this function.

Returns: The number of milliseconds since start-up of the system. For a 32-bit cell, this count overflows after approximately 24 days of continuous operation.

Notes: If the granularity of the system timer is "100" (a typical value for UNIX systems), the return value will still be in milliseconds, but the value will change only every 10 milliseconds (100 "ticks" per second is 10 milliseconds per tick).

This function will return the time stamp regardless of whether a timer was set up with `settimer`.

See also: `settimer`

---

tolower Convert a character to lower case

Syntax: `tolower(c)`

`c` The character to convert to lower case.

Returns: The upper case variant of the input character, if one exists, or the unchanged character code of "`c`" if the letter "`c`" has no lower case equivalent.

Notes: Support for accented characters is platform-dependent.

See also: `toupper`

---

toupper Convert a character to upper case

Syntax: `toupper(c)`

`c` The character to convert to upper case.

Returns: The lower case variant of the input character, if one exists, or the unchanged character code of "`c`" if the letter "`c`" has no upper case equivalent.

Notes: Support for accented characters is platform-dependent.

See also: `tolower`

treble Tone adjust (treble)

Syntax: `treble(gain, frequency=3000)`

`gain`	The gain in the range
`gain`	The gain in the range −12 to +12. Each step is in 1.5 dB (so the range of gain is −18. . . +18 dB.
`frequency`	The frequency at which the attenuation/enhancement starts. The suggested range is 1.5 kHz to 5 kHz; a typical value is 3000 Hz. This parameter is clamped between 1 kHz and 10 kHz (1000 to 10.000 Hz).

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

Notes: The volume level is downward adjusted to allow for the maximum enhancement of bass or treble, while avoiding clipping. That is, when enhancing treble frequencies, the overall volume may decrease.

See also: `bass`, `setvolume`

uudecode Decode an UU-encoded stream

Syntax:
```
uudecode(dest[], const source[],
         maxlength=sizeof dest)
```

`dest`	The array that will hold the decoded byte array.
`source`	The UU-encoded source string.
`maxlength`	If the length of `dest` would exceed `maxlength` cells, the result is truncated to `maxlength` cells. Note that several bytes fit in each cell.

Returns: The number of *bytes* decoded and stored in `dest`.

Notes: Since the UU-encoding scheme is used for binary data, the decoded data is always "packed". The data is unlikely to be a string (the zero-terminator may not be present, or it may be in the middle of the data).

A buffer may be decoded "in-place"; the destination size is always smaller than the source size. Endian issues (for multi-byte values in the data stream) are not handled.

Binary data is encoded in chunks of 45 bytes. To assemble these chunks into a complete stream, function `memcpy` allows you to concatenate buffers at byte-aligned boundaries.

See also: `memcpy`, `uuencode`

---

uuencode Encode an UU-encoded stream

Syntax:
```
uuencode(dest[], const source[], numbytes,
         maxlength=sizeof dest)
```

`dest` The array that will hold the encoded string.

`source` The UU-encoded byte array.

`numbytes` The number of bytes (in the `source` array) to encode. This should not exceed 45.

`maxlength` If the length of `dest` would exceed `maxlength` cells, the result is truncated to `maxlength` cells. Note that several bytes fit in each cell.

Returns: Returns the number of characters encoded, excluding the zero string terminator; if the dest buffer is too small, not all bytes are stored.

Notes: This function always creates a packed string. The string has a newline character at the end.

Binary data is encoded in chunks of 45 bytes. To extract 45 bytes from an array with data, possibly from a byte-aligned address, you can use the function `memcpy`.

A buffer may be encoded "in-place" if the destination buffer is large enough. Endian issues (for multi-byte values in the data stream) are not handled.

See also: `memcpy`, `uudecode`

---

valstr Convert a number to text (string)

Syntax: `valstr(dest[], value, bool: pack=false)`

`dest` The string to store the text representation of the number in.

`value` The number to put in the string `dest`.

`pack` If `true`, `dest` will become a packed string, otherwise it will be an unpacked string.

Returns: The number of characters stored in `dest`, excluding the terminating "\0" character.

Notes: Parameter `dest` should be of sufficient size to hold the converted number. The function does not check this.

See also: `strval`

---

version Return the firmware version

Syntax: `version(FirmwareVersion: code)`

`code` The code for the requested field, one of the following:

`VersionMajor` (0)
The major version number, always 1 for the H0420.

`VersionMinor` (1)
The minor version number, e.g. 6 for version 1.6 of the firmware.

`VersionBuild` (2)
The build number, which is a unique number for a particular revision of the firmware.

`VersionOptions` (3)
: A bit mask with the options that are compiled into the firmware. Currently, only bit 0 is defined: when set, the firmware includes a TCP/IP stack for a network interface.

Returns: This function returns the requested value, or zero on error. Note that the build number is never zero.

---

volumebounds Set range for the volume

Syntax: `volumebounds(low=0, high=100)`

`low` The lower limit of the volume range.

`high` The upper limit of the volume range.

Returns: This function currently always returns 0.

Notes: The `setvolume` function adjusts the volume between the lower and and upper limits that are defined by this function. By default, the range is 0...100 (full range).

The relation between a "volume setting" and the perceived "loudness" of and audio signal is a complex one. Audio volume is normally measured in "decibels" (dB). A decibel is a ratio, like a percentage. If you set the volume of the MP3 player to 100, this does not mean that it will produce 100 dB, but rather that it is at full volume.

Relative audio levels also have another impact: whether we can hear some audible signal also depends on the level of the environmental sound. This changes per application of the MP3 player, of course. The purpose of the `volumebounds` function is to set the maximum volume that is deemed useful and the minimum volume level that is audible (given the typical environmental noise). Once the bounds are set, the desired volume can be set with function `setvolume`, with a range 0...100.

If the volume range is *not* set (or set to the full range), it may happen (e.g. in a noisy environment) that the effective

range of the `setvolume` function is 70...100 —meaning that at any volume level below 70, the audio from the MP3 player "drowns" in the environmental noise.

The decibel range is a logarithmic scale: a difference of 10 dB between two sound signals means that one signal is twice as loud as the other. A difference of 20 dB gives a factor of four in relative loudness (2×2) and 30 dB is a factor of eight ($2 \times 2 \times 2$).

See also: `setvolume`

---

watchdog Watchdog timer

Syntax: `watchdog(seconds)`

`seconds` The number of seconds that the script may use for handling an event before a full reset is activated.

Returns: This function currently always returns zero.

Notes: A watchdog timer is a guard against an infinite loop in the script or other activity that causes the device to hang (and become non-responsive). When setting the watchdog, you specify the maximum time that the script is allowed to take for handling an event. If the script takes longer than this, the watchdog timer assumes that the script is "stuck" and it issues a full reset of the device.

The time-out that you allow for the watchdog should be long enough to be confident that something has gone awry in the script. For example, if the script typically handles an event within a second, but may take up to 5 seconds on rare occasions, a good value for the watchdog time-out would be 10 seconds (twice the longest latency).

See also: `reset`

wavegenerator Produce a waveform on the analogue output

Syntax: `wavegenerator(Fixed: frequency=0.0, WaveType: wavetype=WaveNone, range=8)`

`frequency` The desired frequency of the wave signal. This must be a value between 0.001 and 5000, for a range of 0.001 Hz to 5 kHz. If this parameter is 0.0, the wave generator is shut off. If parameter `wavetype` is `WaveBitStream`, the frequency is in bits per second.

`wavetype` The shape of the wave; one of the following:

`WaveNone`	(0) Disable wave output (shut off wave generation).
`WaveSine`	(1) Sine wave
`WaveTriangle`	(2) Triangle wave
`WaveSquare`	(3) Square wave
`WaveSawTooth`	(4) Saw tooth, ramp up
`WaveInvSawTooth`	(5) Saw tooth, ramp down
`WaveBitStream`	(6) Pulse train; the `range` parameter holds up to 32 bits that are send out at the requested frequency (see the notes).

`range` For all wave type except `WaveBitStream`, this parameter is the amplitude scale for the wave signal. It is one of the following:

`0`	0–0.02 V
`1`	0–0.04 V
`2`	0–0.08 V
`3`	0–0.15 V
`4`	0–0.3 V
`5`	0–0.6 V
`6`	0–1.2 V
`7`	0–2.5 V
`8`	0–5 V

If `wavetype` is `WaveBitStream`, the `range` parameter is the stream of bits to be sent (see the notes).

Returns: `true` on success and `false` on failure. This function fails if a console (LCD) has been set up.

Notes: This function set up a wave generator in I/O pin 16, the analogue output pin. No console/LCD may be set up, because the LCD uses the same pin to adjust the contrast.

If either the parameter `frequency` is 0.0 or the `wavetype` is `WaveNone`, the wave generation is shut off.

The wave signal is digitally sampled with 256 amplitude steps and at a frequency of maximally 10 kHz. When the value of parameter `frequency` is set at 5 kHz (the maximum), there are only two samples for a complete wave cycle. As a result, the triangle and sine wave types are indistinguishable from the square wave type at this maximum frequency.

When `wavetype` is `WaveBitStream`, the `frequency` parameter is in bits per second and the `range` parameter holds the 32 bits that are sent out with the lowest bit transmitted first. The amplitude of the bit stream is always 0–5 V. The function always sends out 32 bits; if your application requires shorted bit stream, make sure that the trailing bits in the `range` parameter are set to the "idle" level.

See also: `configiopin`, `console`, `setiopin`

---

wherexy Return the cursor position

Syntax: `wherexy(&x, &y)`

`x`	Will hold the horizontal cursor position on return.
`y`	Will hold the vertical cursor position on return.

Returns: Always return 0.

Notes: The upper left corner is at (1,1).

See also: `gotoxy`

writecfg Writes a text field to an INI file

Syntax:
```
bool: writecfg(const filename[]="",
               const section[]="", const key[],
               const value[])
```

`filename` The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".

`section` The section to store the `key` under. If this parameter is not set, the function stores the key/value pair outside any section.

`key` The key for the field.

`value` The value for the field.

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

See also: `deletecfg`, `readcfg`, `writecfgvalue`

---

writecfgvalue Writes a numeric field to an INI file

Syntax:
```
bool: writecfgvalue(const filename[]="",
                    const section[]="",
                    const key[], value)
```

`filename` The name and path of the INI file. If this parameter is not set, the function uses the default name "`config.ini`".

`section` The section to store the `key` under. If this parameter is not set, the function stores the key/value pair outside any section.

`key` The key for the field.

`value` The value for the field, as a signed (decimal) number.

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

See also: `readcfgvalue`, `writecfg`

Resources

The PAWN toolkit can be obtained from **www.compuphase.com/pawn/** in various formats (binaries and source code archives).

Note that the downloadable version is a general-purpose release, whereas the one that comes with the H0420 is configured for the device. If you wish to update the PAWN tool chain, back up the configuration files "pawn.cfg" and "default.inc". These two files contain settings specific for the H0420.

The anatomy of the MPEG files is broadly described on several places on the web and in books. For example, see:

- **http://www.mp3-tech.org/**
- "**MP3: The Definitive Guide**" by Scot Hacker; First Edition March 2000; O'Reilly; ISBN: 1-56592-661-7.

Various "application notes" on how to prepare audio fragments for looping playback and chaining tracks are available on the compuphase web site, at the above mentioned address. The number of applications notes will grow over time, so you are invited to visit on **www.compuphase.com/mp3/** a regular basis.

The MPEG file format is a collection of ISO standards. A detailed specification can therefore be obtained from the ISO offices. That said, the description of the "layer 3" audio sub-format consists basically of the source code of the encode/decoder programs that were developed at Fraunhofer IIS.

The (informal) standard of the ID3 tag is on the site **http://www.id3.org** together with links to software that reads and writes these tags. The H0420 only supports version 2 of this tag —version 1 is not supported. Many tag editors exist, both commercial and freeware, but only few can generate the `SYLT` (Synchronized Lyrics) tag.

Since the H0420 MP3 player/controller is an *audio* device, it helps to know a bit about audio and sound. A good start is the description of "decibels" and how that measure relates to volume, energy and loudness. For more information, see **http://en.wikipedia.org/wiki/Decibel**.

Index

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

R

S

T