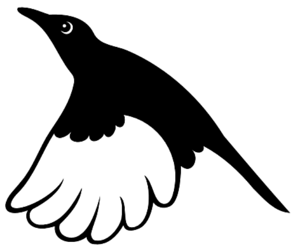
Starling



Model H0440

Programming Guide & Reference

Version 1.0

November 2012

CompuPhase

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

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

Contents

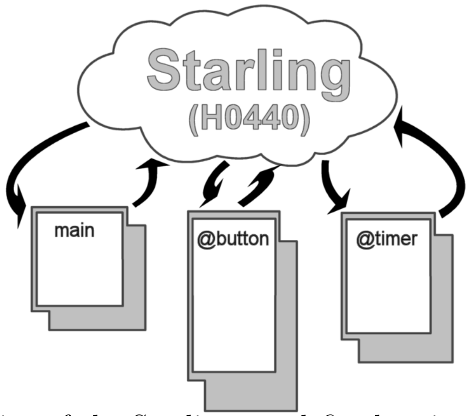
Overview

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 Starling. For an introduction of the Starling model H0440 and its programming interface, please see the Starling manual.

Event-driven programming

The Starling 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.



@input: 45

The general I/O pins of the Starling are defined as inputs on start-up and each pin has an internal pull-up. When an I/O pin is shorted to the ground, this fires a "input status changed" event and the `@input` function in your script will run.* The `@input` function then handles the event, perhaps by

* Provided that the script contains an `@input` function; if the script lacks the `@input` function, the "input status changed" events would be discarded.

starting to play another track, or changing volume or tone settings. After it is done, @input simply returns 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 Starling. In this script, the eight inputs are "linked" to playing eight 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 an input; there are eight tracks
 *  associated with eight inputs. The tracks have predefined names.
 *  The inputs have an internal pull-up, so their default state is
 *  high (1).
 *
 *  When pressing a switch for a track that is already playing, the
 *  track restarts.
 */

@input(pin, status)
    {
    /* act only on high-to-low edge (switch press) */
    if (status == 0)
        {
        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"
            case 7: play "track8.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 change of an input is queued. Only after the pending function has completed and has returned, will the "input change" 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 set up the peripherals that you need: RS232, I/O ports, SPI, 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 Starling, 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 turns the green LED on for the duration of the track. That is, while the Starling is playing audio, the LED will be on, and when *not* playing, it will be off. To toggle the LED, the script uses a second event: the status of the audio decoder.

Listing: **switches2.p**

```
/*  switches2
 *
 *  Play a track that is attached to an input; there are eight tracks
 *  associated with eight inputs. The tracks have predefined names.
 *  The inputs have an internal pull-up, so their default state is
 *  high (1).
 *
 *  The green LED is on when audio is playing and off when it is
 *  silent.
 *
 *  When pressing a switch for a track that is already playing, the
 *  track restarts.
 */

@reset()
    {
    /* turn green LED off on start-up (no track is playing yet) */
    setled LED_Green, false
    }
```

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

```
@input(pin, status)
    {
    /* act only on high-to-low edge (switch press) */
    if (status == 0)
        {
        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"
            case 7: play "track8.mp3"
            }
        }
    }

@audiostatus(AudioStat: status, decoder)
    {
    if (status == Playing)
        setled LED_Green, true
    else
        setled LED_Green, false
    }
```

As is apparent from this second example, function `@reset` serves for one-time initialization. Here, it merely switches the green LED off, because on start-up, no track is playing yet.

@audiostatus: 44

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`.

getiopin: 75

Apart from the "event" functions `@input` and `@audiostatus` mentioned earlier, the Starling programming environment also contains native functions `getiopin` and `audiostatus` (without the "`@`" prefix). The `getiopin` function returns the *current status* of an input pin. With it, you can check the status of each pin at any convenient time. Likewise, the `audiostatus` function returns the active status of (one of) the audio decoders. 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.
 */

@reset()
    {
    /* turn green LED off on start-up (no track is playing yet) */
    setled LED_Green, false

    /* we have to keep the status of all switches (in order to detect
     * the changes)
     */
    new curpin[8]

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

    /* this loop should never end */
    for ( ;; )
        {
        /* test all inputs */
        new pin, status
        for (pin = 0; pin < 8; pin++)
            {
            status = getiopin(pin)
            if (status != curpin[pin])
                {
                /* status changed, save new status */
                curpin[pin] = status
                /* ignore low-to-high edge, act on high-to-low only */
                if (status == 0)
                    {
                    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"
                        case 7: play "track8.mp3"
                        }
                    }
                }
            }

        /* test the audio status */
        new AudioStat: audiostat = audiostatus()
```

```
            if (audiostat != curstatus)
                {
                curstatus = audiostat
                if (audiostat == Playing)
                    setled LED_Green, true
                else
                    setled LED_Green, false

                }
            }
        }
```

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 Starling 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".
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 Starling 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 Starling 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 Starling, 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 Starling to the actual time. This can be done with the functions `setdate` and `settime`. When exchanging the backup battery, the Starling resets its clock to 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 audio file —typically an MP3 file. The tag 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:

$+1\ -2$

where:

- ◇ the operator ("+" or "−") indicates a "toggle-on" or "toggle-off" command for one of the on-board LEDs
- ◇ the number following the operator indicates which LED (1 for red, 2 for green)

Any number of codes may be on single time-stamped line, so you can turn on both LEDs in the same command —or turn on one LED while simultaneously turning of the other.

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 "+1 -2", where the numbers stand
 * for the LEDs (red and green), and "+" and "-" mean "turn on" and
 * "turn off" respectively. So in this example, the red LED is turned
 * on and the green LED is turned off.
 */

@reset()
    {
    /* turn both LEDs off */
    setled LED_Red, false
    setled LED_Green, false
```

[*] The original MP3 file was recorded from a music box by Thea from the Klankbeeld group, and placed under the "Creative Commons" license.

```
    /* The "Tea for Two" theme recorded from a music box by Thea from
     * the Klankbeeld group. Published on the freesound.org site.
     */
    play "teafortwo.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)

        /* the pins are numbered 1, 2,..., but the LEDs start at zero */
        new LED:led = LED:(pin - 1)

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

RS232

The Starling has a standard serial RS232 interface, with two ports. All common Baud rates and data word lay-outs are supported. The interface optionally supports software handshaking, but no hardware handshaking. When using a single port, the DTR and DSR lines are available for handshaking and testing device status.

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 first serial port. It understands the basic commands to start playing files, to query which files are on the SD/MMC card, and to set volume and balance.

Listing: **serial.p**

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

@receive(const data{}, length, port)
    {
    static buf{40}
    strcat buf, data
    if (strfind(buf, "\r") >= 0 || strfind(buf, "\n") >= 0)
        {
        parse buf
        buf = ""    /* prepare for next buffer */
        }
    }

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

    /* strip trailing whitespace */
    for (idx = strlen(string); idx > 0 && string{idx-1} <= ' '; idx--)
        {}
    if (idx >= 0)
        string{idx} = 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))
            transmit "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 = fexist(string)
            new filename{100}
            for (new index = 0; index < count; index++)
                {
                fmatch filename, string, index
                transmit filename
                transmit "\n"
                }
            }
        else
            transmit "Unknown command or syntax error\n"

        transmit "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 Starling 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 SD/MMC card are listed (i.e. the command “list” is short for “list *”).

uudecode: 130

For transferring binary data over RS232, you may choose to convert the binary stream to UU-encode and transfer it as text, or to explicitly use the `length` parameter in the public function `@receive` to determine how many bytes have been received in binary mode. When receiving data in binary mode, 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.

The Starling 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.

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`).

File system

The Starling accepts SD/MMC cards that are formatted as FAT16 or FAT32. Most SD/MMC 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 Starling 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 Starling 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 Starling. 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’’`	*unpacked string*

• General file I/O

Apart from "playing" audio files, the Starling 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.

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 Starling can only hold two files open at any time for scripting. The file I/O needed for playing audio files are handled separately. That is, the script can open two files and still play 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.

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 charac-

	ters. 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.

Network

The Starling Ethernet interface allows the audio controller to be connected in a standard Ethernet network, using the TCP/IP protocol suite. The firmware contains a set of network functions that you can use from the script.

Apart from a few basic network control messages, no network functionality is hard-coded in the Ethernet interface. All network functionality is under control of the script. In its current release, the network interface supports the TCP/IP protocol suite with the following functionality:

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

Usage

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

Listing: **Initializing the network interface**

```
#include <tcpip>
```

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

Listing: **Initializing the network interface**

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

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

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

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

Low-level interface

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

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

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

Listing: **Telnet server skeleton**

```
#include <tcpip>

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

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

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

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

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

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

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

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

High-level interface

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

◇ HTTP client	`netdownload`
◇ HTTP server	`@nettransfer`
◇ FTP client	`netdownload` or `netupload`
◇ FTP server	`@nettransfer`

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

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

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

Audio streams

The Starling can play audio that is streamed to the device. There are three protocols for streaming: direct streaming via RTP, buffered streaming with a progressive HTTP protocol (e.g. Shoutcast), and buffered streaming via standard HTTP.

• Progressive HTTP versus standard HTTP

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

There are also important differences. To begin with, the server set-up is different: you need a HTTP server for standard HTTP streaming and a Shoutcast/Icecast server for progressive HTTP. Standard HTTP streams play MP3 *files* over the network, from start to finish —you do not have the option start at an arbitrary position in the file. The “standard” HTTP streaming is therefore not suitable for live streaming.

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

• Streaming with progressive HTTP

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

Progressive HTTP is more suitable for streaming over a WAN or the Internet because it is buffered in a “stream queue”. You can optionally also monitor the queue status to decide when to start playing the stream.

Like standard HTTP, progressive HTTP is a “pull” protocol: the Starling initiates the connection to a stream server.

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

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

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

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

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

Listing: **Streaming with HTTP**

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

The Starling supports meta-data in the stream. This meta-data is textual data, usually containing the title of the song and the name of the artist or the band, that the streaming server inserts into the audio stream at regular intervals. When a stream is playing, a script can retrieve that data from the function `trackinfo`.

• Restarting a HTTP stream

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

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

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

The following code snippet illustrates a the concept:

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

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

@main()
    {
    netsetup
    settimer 1000
    }

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

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

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

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

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

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

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

[*] an alternative would be to implement the `@netstatus` function and wait for the `NetAddrSet` event, see page 23.

[†] Since `StreamBufferLimit` is defined at 128 KiB, `StreamBufferLimit` divided by 4 is 32 KiB.

• Tips for progressive HTTP streaming

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

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

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

◇ To signal a failed connection to a stream:
  a) check the return value of `netstream`; it returns `false` if it cannot connect to the server;
  b) `@netstatus` gets the event `NetStreamBuffer` with status 0 (stream queue 0% full), which means that the remote stream server replied with an error or reset the connection.

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

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

• Streaming with RTP

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

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

RTP is a non-buffered "push" protocol. To play an RTP stream, the script can call the standard function `play` with an RTP URL instead of a filename. Alternatively, the script can call `netstream` for more control. For example, the following snippet starts playing the stream from the server at "224.82.71.81" on port 56952:

Listing: **Streaming with RTP**

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

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

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

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

Transferring files

The script supports the HTTP protocol for downloading files from a web server and the FTP and TFTP protocols for downloading and uploading files from and to a FTP/TFTP servers. Authenticated file transfers are only supported on the FTP protocol. The TFTP client in the Starling requires a server that supports TFTP options, notably the "transfer size" option. Modern TFTP servers support options.

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

These functions initiate the file transfer and thereby act as a "client". The script can also wait for an incoming request (from a remote host) to transfer a file, by setting up a server. See the section “HTTP, FTP and TFTP servers” on page 35 for this functionality.

Monitoring and configuration with SNMP

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

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

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

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

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

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

• The MIB file

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

Listing: **Template MIB file**

```
--
-- A template SNMP MIB file for use with the Starling
--
-- Copyright (c) 2007-2012 ITB CompuPhase
--

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

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


compuphase    OBJECT IDENTIFIER ::= { enterprises 28388 }
products      OBJECT IDENTIFIER ::= { compuphase 1 }
starling      OBJECT IDENTIFIER ::= { products 21 }

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


-- Add your definitions here...


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

END
```

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

When writing the MIB file, please note that the implementation of the SNMP agent in the Starling only supports whole numbers and (octet/character) strings. The Starling does not support "sequence" types for user data. In the MIB file, you may also use derived types as `Counter`, `Gauge`, `TimeTicks` and `IpAddress`, which are basically different representations of integer values.

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

Listing: **Minimal SNMP agent**

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

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

        default:
            return false
        }
    return true
    }
```

The definitions to put in the MIB file are below (these definitions must be merged in the template MIB file, see page 33):

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

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

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

HTTP, FTP and TFTP servers

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

To have the script initiate the file transfer itself, rather than wait for an incoming request, see section "Transferring files" on page 31.

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

• TFTP server

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

Listing: **Handling TFTP requests**

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

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

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

The TFTP server in the Starling requires a client that supports TFTP options, notably the "transfer size" option. Modern TFTP clients support options, but the command line "`tftp`" utility that comes with Microsoft Windows does not. For a free alternative (which supports options), see the TFTP command line client by WinAgents.

• HTTP server

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

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

Listing: **Handling HTTP requests**

```
bool: @nettransfer(path[], NetRequest:code, socket)
    {
    if (code != NetHttpGet)
        return false    /* deny non-HTTP transfers */
/* get and save any parameters */
new idx = strfind(path, "?");
if (idx >= 0)
    {
    new params[100 char]
    strmid params, path, idx + 1
    /* write the parameter in a file (without further processing) */
    new File: handle = fopen("params.txt", io_write)
    if (handle)
        {
        fwrite handle, params
        fclose handle
        }
    }

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

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

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

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

• FTP server

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

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

Listing: **Handling FTP requests**

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

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

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

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

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

Development and debugging

Reducing memory requirements

The Starling has 16 KiB of memory available to scripting. This limit is declared in the `h0440.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 "3"; the PAWN compiler generates more compact code. The relevant option is "`-O3`". 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 you have an RS232 cable attached to the Starling, you can send messages and values of variables over the serial line. These messages can then be inspected while the program is running. See the functions `setserial` and `transmit` in this reference for setting up a serial connection on pages 107 and 129 respectively.

The PAWN toolkit comes with a source level debugger that supports "remote debugging", meaning that the debugger controls the script running on the Starling 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 the serial port to use (e.g. COM1:).

◇ 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 SD/MMC 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 compiled script (the "`.amx`" file) is then transferred to the SD/MMC 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`". Assuming that the compiled script is called "STARTUP.AMX", the command line is:

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

  on Microsoft Windows and

```
pawndbg startup.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 SD/MMC card in the Starling and optionally reset (or power-cycle) the device. The debugger should now display the first line of function `@reset`.

When the Starling is reset and the script that it loads has debug information, it waits for a debugger to connect, with a timeout of one second. If no debugger connects, the Starling runs the script *without* debugger support. This is why it is advised to start the debugger *before* resetting the Starling.

After the script has been fully debugged, you will want to recompile it without debugging support: it avoids the start-up delay (when the Starling 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 Starling must reside on the SD/MMC card (in the root directory). For simple scripts, it is easy to write the script, compile it and copy the resulting "`.amx`" onto the SD/MMC 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 SD/MMC card between the Starling audio player and the card reader on the PC may become a nuisance. An alternative is to transfer the `.amx` file 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 Starling audio 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, as described in the previous section. Then, you need to reset the Starling, 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 `.amx` file to the Starling, which will then write it onto the SD/MMC card. After the copy is complete, the Starling will automatically restart, and the debugger restarts too.

If transferring the compiled script 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 transfer.amx -rs232=1 -transfer -quit
```

Especially for purposes of uploading compiled scripts, it can be useful to have the Starling reset on a command that comes over the same RS232 line — because the Starling audio player only picks up a debugger synchronization attempt within a second 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 `@receive` function responds to the '¡' character (ASCII 161). 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 Starling audio player, the PAWN debugger will reset the audio player if it was *not* polling for the debugger.

Listing: **Reset the Starling player on receiving a '¡' on the RS232**

```
@reset()
    {
    setserial 57600
    }

@receive(const string{}, length, port)
    {
    if (string{0} == '\xa1')
        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, decoder)`

`status`	The new audio status.
`decoder`	The decoder that is the source of the event. On devices with only a single decoder, this parameter is always 1.

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 audio data.

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

@eject The card is removed

Syntax: `@eject()`

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

Notes: This function is called when the SD/MMC card is removed ("ejected"). After completion of the `@eject` function, the Starling controller performs a 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 Starling itself —see `storeconfig`. You cannot write device data or status information to the SD/MMC card (because it is "ejected"...)

See also: `storeconfig`

@input A digital pin changed

Syntax: `@input(pin, status)`

`pin`	The pin number, see the notes.
`status`	The new logical level (0 or 1).

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

Notes: Some models of the Starling controller have 8 I/O pins (e.g. model H0440), other models have 16 I/O pins. On models with 8 I/O pins, the `pin` parameter is a value between 0 and 7 for the normal I/O pins; on models with 16 I/O pins, the `pin` parameter is a value between 0 and 15. However, the "FUNC" switch is assigned pin number 16 (regardless of the model of the Starling).

Only the pins that are configured as "input" can cause this event function to execute. See `configiopin` for configuration. On start-up, all pins are pre-configured as inputs.

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.

See also: configiopin, getiopin

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

@netreceive

A data packet is received

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

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

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

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

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

When the script is listening on a TCP socket and a remote device connects to this socket (i.e., a passive connect), the `@netreceive` function is called with the `size` parameter set to zero. A script can use this special case to send a greeting message to the remote host on connect.

Example: See the Telnet server (skeleton) on page 23.

See also: `netlisten`

@netsnmp

An SNMP request is received

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

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

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

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

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

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

Example: See the code and associated MIB snippets on page 34.

See also: `netsnmptrap`

@netstatus — Network status changed/event occurred

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

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

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

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

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

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

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

`NetTftpDone` (5)
: TFTP transfer has finished; parameter `status` is 1 on success and 0 on failure. A failure may also indicate a transfer that has been aborted (by the user or by the remote server).

`NetHttpDone` (6)
: HTTP transfer has finished; parameter `status` is 1 on success and 0 on failure. A failure may also indicate a transfer that has been aborted (by the user or by the remote server).

`NetStreamQueue` (7)
: Stream queue mark reached; parameter `status` is the level (in kilobytes), it is zero if the remote server rejected the stream request.

`NetFtpDone` (8)
: FTP transfer has finished; parameter `status` is 1 on success and 0 on failure. A failure may also indicate a transfer that has been aborted (by the user or by the remote server).

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

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

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

Example: See the Telnet server (skeleton) on page 23.

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

@nettransfer

A file transfer request was received

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

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

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

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

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

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

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

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

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

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

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

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

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

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

If you do not implement this function, all TFTP, HTTP and FTP server requests are denied. The FTP server can only handle one user at a time. A login request while there is already a connection open is denied. Some modern FTP clients issue a second (or third...) login for every file transfer; this option must be disabled for the FTP server in the Starling.

Example: See the code snippets on page 35 and page 36.

See also: `netdownload`, `netupload`

@receive

Data from RS232 is received

Syntax: `@receive(const data{}, length, port)`

`data`	The data received. The array may contain one or more characters. A final zero-byte is appended to the end.
`length`	The number of bytes received. The zero-byte appended to the data array is not included in this count.
`port`	The port number of the serial port on which the data is received. On devices that support only a single port, this parameter is always 1.

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

Notes: If you are expecting to receive only text, the `length` parameter is the same as the string length of the `data` parameter.

The Starling can use software handshaking (XON/OFF) —see `setserial`. If software handshaking is enabled, bytes with the values 17 (`0x11`, Ctrl-Q) and 19 (`0x13`, Ctrl-S) and zero cannot be received with this function. As an alternative, when you need to transfer binary data in combination with software handshaking, you can encode it using a protocol like UU-encode.

Example: See `serial.p` on page 10.

See also: `setserial`, `transmit`

@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 `samplepin`, the Starling 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 Starling controller, 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: `samplepin`

@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 39 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 `set-timer`. 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`

Native functions

audiostatus

Get the current audio status

Syntax: `AudioStat: audiostatus(decoder=1)`

`decoder`	The decoder for which the status information is requested. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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`.

See also: `@audiostatus`

bass

Tone adjust (bass)

Syntax: `bass(gain, frequency=200, decoder=1)`

`gain`	The gain in the range of 0 dB to +12 dB (boost only).
`frequency`	The frequency at which the bass enhancement starts. This parameter is clamped between 20 Hz and 150 Hz.
`decoder`	The decoder to which the tone adjustment applies. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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

Notes:	The bass enhancer uses a DSP algorithm that improves the bass levels while avoiding clipping. The algorithm is most effective with dynamical music material, or when the playback volume is not set to maximum.
See also:	`setvolume`, `treble`

clearioqueue

Remove switch or input events from the queue

Syntax:	`clearioqueue()`
Returns:	This function always returns 0.
Notes:	During lengthy processing (by the script), any 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 the I/O event queue immediately after finishing the process. All I/O events that have happened in the mean time will then have been "forgotten".
See also:	`@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`

configiopin

Configure an I/O pin

Syntax:

```
configiopin(pin, PinConfig: type,
            bool: debounce=false)
```

`pin` The pin number, between 0 and 7 for models with 8 I/O pins, or between 0 and 15 for models with 16 I/O pins.

`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`.

`debounce` This parameter is only relevant when a pin has been declared as “input”. When debouncing for an input pin is `true`, a change in status of the pin is reported only after it has stabilized to a new level. Glitches with a duration less then 20 ms are ignored.

Returns: This function always returns 0.

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

When configured as outputs, the pins can drive a LED or an opto-coupler directly (no intermediate “driver” IC is required). The voltage of the output pins can be set with `setvoltage`.

For high-speed sampling of an input pin, see function `samplepin`.

Example: See `switches2.p` on page 3

See also: `@input`, `getiopin`, `samplepin` `setiopin`, `setvoltage`

cvttimestamp

Convert a time-stamp 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 time-stamps 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 wild-card 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.

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: 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 audio tracks sequentially. By storing the inode number and the file size of the next track in a "resource id" (while the Starling 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, or -1 to read the state of all 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 that specified I/O pin: 0 or 1. If parameter `pin` is -1, the return value is a value where the bits represent the state of the respective I/O pins.

Notes: On models with 8 I/O pins, the `pin` parameter must be in the range 0. . . 7, or -1 to read all 8 pins as a bit mask. On models with 16 I/O pins, the `pin` parameter must be in the range 0. . . 15, or -1 to read all 16 pins as a bit mask.

When a pin is defined as output, it reads 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`, `configiopin`, `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: `bool: getvolume(&volume=0, &balance=0, decoder=1)`

`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.
	`decoder`	The decoder whose volume must be queried. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

Returns: This function 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`, `mute`, `setvolume`, `treble`

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.

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`

mute

Mute or unmute the audio

Syntax: `mute(bool: on, decoder=1)`

`on`	Set to `true` to silence the audio, or `false` to return to the previously set volume.
`decoder`	The decoder that must be muted or unmuted. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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.

See also: `setvolume`

netarp

Refresh the ARP cache

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

`remote_addr`	The domain name or the IP address of the host whose hardware address (MAC address) should be refreshed in the ARP cache.

Returns: `true` if the remote MAC address is in the ARP cache, and `false` otherwise.

Notes: The ARP cache holds the hardware (MAC) address of the first hop to send a network packet to, in order to get the packet to the destination. This may either be the MAC address of the other host, or the MAC address of the relevant gateway.

When making a connection, or sending a packet to another host, if the MAC address is not already in the ARP cache, the network interface first needs to obtain the MAC address. It does this via a protocol named "ARP". Waiting for the ARP response may take several seconds, especially if the remote host is unresponsive (e.g. it is "down"). In situations where no delay in setting up a connection may be allowed, one option is to regularly refresh the MAC address in the ARP cache, and to communicate with the remote host only if the MAC address is indeed cached (and therefore the remote host is "up").

This function sends an ARP request, but returns immediately —before the response arrives. The first time that this function is called for a new host, it may therefore return `false`, even if the host is up. When `netarp` is called again, after a suitable delay, the ARP cache will have been updated.

See also: `netlookup`

netclose

Close a socket

Syntax: `bool: netclose(socket)`

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

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

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

See also: `netconnect`, `netlisten`

netconnect

Open a connection / socket

Syntax: `netconnect(const remote_addr[])`

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

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

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

See also: `netclose`, `netsend`

netctrl

Set connection options

Syntax: `netctrl(NetCtrl: option, value)`

`option` The connection option to set, it must be one of the following:

`MSS512` (1)
A value of 1 forces the TCP MSS to be 512 bytes and the TCP reception window to be twice the MSS (i/e/ 1024 bytes). A value of 0 sets the default value of the MTU minus 40 bytes, and a dynamic window size.

`FullDuplex` (2)
A value of 1 switches the interface to be full-duplex. A value of 0 sets the interface to half-duplex. The network interface starts up as half-duplex.

`UseChecksum` (3)

A value of 1 activates checksum verification on all received packets. Packets with an incorrect checksum are rejected. A value of 0 deactivates checksum verification. This option does not have any effect on transmitted packets: packets sent out always have a checksum set.

`value` The value to set the option to. See parameter `option` for details.

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

See also: `netsetup`

netdownload

Download a file

Syntax:

```
netdownload(const url[], const filename[]="",
                         File: handle=File:0)
```

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

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

`handle` An optional handle to a file that was has been explicitly opened by the script.

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

Notes: To download from a HTTP server, use the protocol designator "http://"; to download from an FTP server, the protocol designator is "ftp://". To download a file from a TFTP server, the protocol designator is "tftp://".

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

When passing in a file handle instead of a filename, the handle must be opened by the script before calling this function, but it is closed at the end of the download. Using a file handle allows you to explicitly reserve the file space on the memory card.

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

netinfo

Get network status information

Syntax:

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

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

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

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

`SubnetMask` (2)
: The subnet mask for the LAN.

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

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

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

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

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

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

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

`NetErrors` (10)
: The number of transmission errors that are detected by the Ethernet hardware. A high number of errors indicates a mismatch of full versus half duplex between the device and the switch or router to which it is connected. See the function `netctrl` for setting full/half duplex.

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

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

Returns: The requested value, or zero on error.

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

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

netlisten

Open a “listening” connection

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

`port`	The number of the port to listen to. The default port is 9930.
`protocol`	Must be either `TCP` or `UDP`.

Returns: The socket number, or zero on error.

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

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

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

You can only listen to *one* TCP socket and/or *one* UDP socket at a time. A UDP socket may receive incoming packets from multiple hosts (and reply to multiple hosts); a TCP socket is a point-to-point connection to a single host.

Example: See the Telnet server (skeleton) on page 23.

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

netlookup

Look up a domain name

Syntax:
```
bool: netlookup(const domainname[], ipaddress[],
                size=sizeof ipaddress)
```

`domainname`	The domain name of the host to get the IP address for.
`ipaddress`	The IP address will be stored in this parameter, as a packed string. For the maximum address length, the string should be able to contain at least 16 characters.
`size`	The size (in cells) of parameter `ipaddress`. If this value is less than four, the returned IP address may be truncated.

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

Notes: The purpose of this function is to convert a domain name to a dotted IP address. This allows a script to use the IP address to communicate with the remote host, and "forget" the domain name. There are two advantages in using IP addresses instead of domain names: IP addresses are usually shorter (and require less memory) and connecting to an IP address is quicker than to a domain name.

See also: `netconnect`

netping

"Ping" remote host

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

`remote_addr`	The IP address or the domain name of the remote host to send a ping request to. No port number may be attached to the IP address or domain name.
`sequence`	An arbitrary number that allows you to match the ping response to a request, in case you send multiple "pings".

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

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

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

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

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

See also: `@netstatus`, `netinfo`

netsend

Send a packet

Syntax:

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

`buffer`	The data to send to a remote host.
`size`	The size of the buffer in cells.
`remote_addr`	Either an IP address and a port, for sending an UDP datagram, or a socket number for sending a TCP message —see the notes for details.

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

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

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

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

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

Example: See the Telnet server (skeleton) on page 23.

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

netsetup

Initialize the network

Syntax:

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

`ip_address`	The IP address of this host (the MP3 controller), or empty to have it looked up from a DHCP server.
`gateway_address`	The IP address of the gateway, or empty to have it looked up from a DHCP server.
`dns_address`	The IP address of the DNS server, or empty to have it looked up from a DHCP server.
`subnet_mask`	The network mask in "dotted format" (see below), or empty to have it looked up from a DHCP server.
`hostname`	The name of this host. This name is used for the DHCP request and for the DNS and NetBIOS look-ups. If left empty, the standard name is "Starling".

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

Notes:	All IP addresses should be in "dotted format", meaning four decimal numbers in the range of 0 to 255 separated by periods. An example is `192.168.10.29`. You should avoid doing partial DHCP look-ups —either leave the first three parameters of this function empty, in order to have them provided by a DHCP server, or specify all three: the host IP address, the gateway address and the DNS server address. For common networks, the function can establish the network mask automatically, but if known, it is best to specify it as well. If no IP addresses are given, and DHCP fails too, the Starling assigns a "link-local" address to itself. Link-local addresses are only valid inside a LAN (the link-local address range is non-routable). The Starling will not have access to the Internet when it has a link-local address. The link-local address scheme is also known as "AutoIP" and "APIPA". The network interface starts up in half-duplex with an MTU of 1454 bytes (a safe value for based on Ethernet 2 frames tunneled over PPoE), and with an adaptive reception window. These options can be changed with `netctrl`.
Example:	See the code snippets on page 22 and page 28.
See also:	`netctrl`, `netshutdown`

netshutdown

Close the network interface

Syntax:	`netshutdown()`
Returns:	This function currently always returns 0.
Notes:	This function closes down the network support and frees all resources.
See also:	`netsetup`

netsnmpcfg

Set the communities (passwords) for SNMP

Syntax:

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

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

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

Returns: This function currently always returns 0.

Notes: See the section on SNMP on page 32 for more information on SNMP authentication and access rules.

See also: `@netsnmp`, `netsnmptrap`

netsnmptrap

Send an SNMP trap

Syntax:

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

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

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

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

`WarmStart` (1)
: Device reset.

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

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

`AuthenticationFailed` Authentication failed.	(4)
`EGPNeighborLoss` Neighbour in the Exterior Gateway Protocol was lost.	(5)

See the SNMP standard for details on the standard traps.

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

`item`	Parameter to which the trap relates (see the MIB file).
`value`	New value of the `item` parameter, which caused the trap.

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

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

See also: `@netsnmp`, `netsyslog`

netsocket — Make a socket string from a socket number

Syntax: `netsocket(value)`

`value`	The socket number.

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

See also: `netsend`

netsocket

Make a socket string from a socket number

Syntax:

```
bool: netsockstat(socket, &protocol=0, &sent=0,
                          &received=0)
```

`socket`	The socket number.
`protocol`	On return, this value will be set to 1 (ICMP), 6 (TCP) or 17 (UDP), depending on the protocol of the socket.
`sent`	On return, this value will be set to the current value of the TCP sequence number for the transmitted data.
`received`	On return, this value will be set to the current value of the TCP sequence number for the received data.

Returns: `true` on success, `false` on failure (invalid socket)

Notes: In the TCP/IP protocols, the sequence numbers represent the number of bytes being transmitted and received, including all bytes transmitted for data synchronization and acknowledgements. The sequence numbers do, however, *not* start at zero (due to protocol reasons). To get the true number of received and transmitted bytes, query the sequence numbers immediately after opening a connection, and subtract these "start values" from the sequence numbers obtained in subsequent calls to `netsockstat`.

netstream

Start buffering an audio stream

Syntax:

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

`url`	The full network path of the file to download, preferably including the protocol prefix. The protocol prefix is "`icy://`" for Shoutcast and Icecast servers that are on the default port 8000. If the server uses port 80 instead, you may use the protocol prefix "`http://`", or add a port number explicitly.
`buffermark`	The criterion for the fill level of the stream queue before starting playing the stream, in kilobytes. The minimum value is 8. See page 26 for details on the stream queue.
`autoplay`	If `true`, the stream starts to play (output audio) as soon as the level in parameter `buffermark` is reached. When set to `false`, the public function `@netstatus` is still called with code `NetStreamQueue`, but no audio is output.

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

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

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

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

Example: See the code snippet on page 28.

See also: `@netstatus`, `play`

netsynctime

Request network time synchronization

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

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

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

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

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

See also: `@netstatus`

netsyslog

Send a system log message

Syntax:
```
bool: netsyslog(const remote_addr[],
                const message[], severity=5)
```

`remote_addr`	The IP address or the domain name of the remote host to send the log message to. No port number may be attached to the IP address or domain name.
`message`	The message to send to the Syslog server.
`severity`	By convention, a value between 0 and 7, with the following meanings:

0 = emergency (system is unusable)
1 = alert (immediate action required)
2 = critical
3 = error
4 = warning
5 = notice (normal, but significant condition)
6 = informational
7 = debug

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

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

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

See also: `netsnmptrap`

netupload

Download a file

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

`url`	The full network path where the file will be uploaded, preferably including the protocol prefix. To upload a file, with the name "loops.mp3", on the remote host at address 195.200.2.66, and using TFTP, the URL would be: "tftp://195.200.2.66/loops.mp3".
`filename`	The full path and filename of the local file. If not provided, the file is downloaded to the root directory.

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

Notes: In the current version of the firmware, only FTP and TFTP are available as protocols for uploading data to an external server. To upload to an FTP server, use the protocol designator "ftp://"; for a TFTP server, the protocol designator is "tftp://".

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

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

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`

pause

Pauses playback

Syntax: `bool: pause(decoder=1)`

`decoder` The decoder that must be paused. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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

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

play

Start playing an audio file

Syntax:

```
bool: play(const filename[], repeats=0, bool:
                paused = false, decoder=1)
```

`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. The filename may also be an URL to a track on a HTTP server or an URL to a streaming server.
`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 `cellmax`, the audio file is repeated indefinitely until it is explicitly stopped or until another file is scheduled to play.
`paused`	When set to `true`, the track is prepared for playback in the specified decoder, but the decoder is put in “paused” mode. To play the track, you must call `resume`.
`decoder`	The decoder to play the track on. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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

Notes: Instead of a path and filename of an audio 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 audio 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.

Function `play` may also be used to start playing a network stream. However, the function `netstream` offers more control for streaming audio.

Example: See `serial.p` on page 10.

See also: `fstat`, `netstream`, `resume`, `stop`

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 cryptographic 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, area=0)`

`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.
`area`	The area to store the data; 0 = Flash ROM, 1 = battery backed RAM.

Returns: This function currently always returns 0.

Notes: The Starling controller has two areas of auxiliary non-volatile memory area into which the script can store data. Typically, device configurations that should be saved even when the SD/MMC card is exchanged, are stored in the configuration area.

See function `storeconfig` for the difference between the two configuration memory areas.

See also: `storeconfig`

reset

Causes a full reset

Syntax: `reset()`

Returns: This function does not return.

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

The Starling will poll for a debugger on the RS232 after a programmed reset, regardless of whether the script on the SD/MMC card was built with debug information. If no debugger is present, the polling causes a start-up delay of one second.

Example: See the debugger support function on page 42.

See also: `@reset`

resume

Resumes playback that was paused earlier

Syntax: `bool: resume(decoder=1)`

`decoder` The decoder that must be resumed. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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`

samplepin

Configure a pin for input sampling

Syntax: `samplepin(pin, timeout)`

`pin` The pin number, between 0 and 7 for models with 8 I/O pins, or between 0 and 15 for models with 16 I/O pins.

`timeout` The duration of the sampling period, in milliseconds, starting from the first detected change in the level of the pin (low to high, or high to low).

Returns: This function always returns 0.

Notes: The pin is configured as input (without debouncing) 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.

Only a single pin may be configured for sampling.

See also: `@sample`, `configiopin`

seekto

Set the position in the audio track

Syntax: `bool: seekto(milliseconds, decoder=1)`

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

`decoder` The decoder that must jump to a new position (in the track that it is playing). For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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

Notes: You must have started to play the track before you can seek to a position. The track may be in "paused" state, but it must be active in the decoder.

See function `trackinfo` 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.

For MP3 files, 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 MP3 file lacks a Xing header, the `seekto` function works, but the seek position may be inaccurate.

For Vorbis files, the seek position may be inaccurate.

See also: `trackinfo`, `play`

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 Starling accordingly. On a first start-up after inserting the battery (or in absence of a battery), 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`

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, 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. . . 7, or 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 bits represent the desired output state of the pins.

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.

See also: `configiopin`, `getiopin`

setled

Configure a pin for input sampling

Syntax: `setled(LED: led, bool: on)`

`led`	The LED, one of either: `LED_Red` (0) The red LED (normally indicating card access). `LED_Green` (1) The green LED (normally indicating power).
`on`	`true` to turn the LED on, `false` to turn it off.

Returns: This function always returns 0.

Notes: The LEDs on the Starling have a default function, but it can be overruled.

Example: See `sylt.p` on page 8.

See also: `setiopin`

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 Starling player, the RS232 interface of the Starling 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 10.

See also: `@receive`, `receive`, `transmit`

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`

setvoltage

Set and enable the I/O voltage level

Syntax: `setvoltage(voltage, interfaces)`

`voltage` The desired voltage on the I/O pins and the SPI pins. The parameter is in multiples of 0.1V, so 50 stands for 5.0V and 33 for 3.3V.

`interfaces` This parameter selects on which interfaces the power pins are enabled. It is one of the following:

0	disable the power pins on all interfaces
1	enable the power pin on the SPI bus only
2	enable the power pin on the I/O bus only
3	enable the power pins on the SPI and the I/O buses

Returns: This function always returns 0.

Notes: The SPI interface and the general I/O interface have power pins that can power external peripherals. The voltage setting applies to all pins (I/O and power), however, the power pins can be individually enabled and disabled.

On start-up, the voltage is set to 3.3V and the power pins on the interfaces are disabled.

See also: `setiopin`, `spi`

setvolume

Set the audio volume and balance

Syntax:

```
bool: setvolume(volume=cellmin, balance=cellmin,
                fadetime=0, decoder=1)
```

`volume`	This (optional) parameter holds the new volume level, a value in the range 0...100. When set to `cellmin`, the volume is not changed.
`balance`	This (optional) parameter holds the new balance setting, a value in the range −100...100. When set to `cellmin`, the balance is not changed.
`fadetime`	The duration in milliseconds to take for the volume or balance change.
`decoder`	The decoder to which the volume change applies. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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.

Fading the change in volume (or balance) happens in the background. The script continues running while the fading takes place. 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 10.

See also: `@audiostatus`, `bass`, `getvolume`, `mute`, `treble`

spi — Send SPI data

Syntax:
```
spi(const data{}, size, frequency=1, select=1,
    mode=1)
```

`data`	An array with the bytes to send. This must be a packed array.
`size`	The number of bytes in parameter `data` to send.
`frequency`	The SPI clock frequency in MHz. The default value of 1 means a 1 MHz clock. When this parameter is zero, the SPI clock is not changed from its start-up setting, which is 6 MHz.
`select`	The SPI "chip select" line (also called "slave select"). Two chip select pins are available on the SPI connector. 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 Starling has an SPI bus with two general-purpose chip select lines. To use the SPI bus, you must therefore connect the device to communicate with to the relevant pins on the connector. The data sheet documents the pins to use. Since there are two chip select lines, the function can communicate with two SPI devices.

The data that a device returns is stored in the `data` array. Some devices require additional time to process a command. In such a case, append one or more additional zero bytes to the `data` array.

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

The chip select pin is toggled after every byte for SPI modes 1 and 3, it stays low for the entire transfer for SPI modes 0 and 2. (This is conforming to the SPI specification.)

See also: `setvoltage`

stop

Stop playback

Syntax: `bool: stop(decoder=1)`

`decoder`	The decoder that must stop playing. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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.

Example: See `serial.p` on page 10.

See also: `pause`, `play`

storeconfig

Read device configuration

Syntax: `storeconfig(const data[], size=sizeof data, area=0)`

`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.

`area`	The area to store the data; 0 = Flash ROM, 1 = battery backed RAM.

Returns: This function currently always returns 0.

Notes: The Starling controller has two areas of auxiliary non-volatile memory area into which the script can store data. Typically, device configurations that should be saved even when the SD/MMC card is exchanged, are stored in the configuration area.

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.

Area 0 is the Flash ROM. Data stored in this area is kept if the both power and the battery are removed. The drawbacks are that writing to Flash ROM is slow and that Flash memory can be re-written 100,000 times on the average. Since the configuration area is internal to the Starling, you need to replace the board once the Flash ROM area becomes defective due to exceeding the number of re-writes. This area is intended to be updated only infrequently. (Reading from Flash ROM is quick and does not wear out the memory.)

Area 1 is SRAM that is backed up with a battery. Writing to this area is quick and frequent writes do not wear out the memory (unlike Flash ROM). However, the memory contents are lost when the battery is removed or when the battery has discharged below the minimum level required for memory backup.

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).

sysconfig

Set or return system configuration

Syntax: `sysconfig(SysConfig: code, value=0)`

`code` The item from the frame header to read, it is one of the following:

`SysXtalAdjust1` (0)
: The `value` parameter adjusts the crystal frequency of the first decoder.

`SysXtalAdjust2` (1)
: The `value` parameter adjusts the crystal frequency of the second decoder.

`SysResetID` (2)
: Returns the reason for the start-up or reset. If the `value` parameter is non-zero, the recorded reason is erased.

`SysCardID` (3)
: Returns the manufacturer ID of the memory card.

`SysCardSize` (4)
: Returns the size of the memory card, in MiB.

`value` For read/write parameters, this parameter holds the new value of the system parameter, if applicable.

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

Notes: The crystals of the decoders can be adjusted in increments of 2 parts-per-million (PPM). That is, setting the `SysXtalAdjust1` field to 1 will adjust the crystal of decoder 1 to tick 2 PPM quicker.

The “reset ID” returns how the Starling started up. The return value can be one of the following:

0	Reason of start-up is unknown.
1	External power was applied to the device.
2	The “RESET” switch was pressed.
4	The device was reset from software, either through detection of a fault or because the script called the `reset` function.
8	A value of 8 or higher means that a power glitch was detected.

temperature

Return the detected temperature

Syntax: `temperature()`

Returns: The temperature in a multiple of $^1/_{10}{}^{th}$ of a degree Celsius. For example, a value of 213 means a temperature of 21.3° Celsius.

Notes: To convert the temperature to Fahrenheit, use the equation

$$Fahrenheit = Celsius \times \frac{9}{5} + 32$$

The temperature sensor is mounted on the Starling PCB and measures mainly the temperature of the PCB itself. Some chips on the Starling PCB are warmer than the PCB temperature at the spot of the sensor.

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", 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`

trackinfo

Return track information

Syntax:
```
trackinfo(TrackCode: code, destination{}="",
          size=sizeof destination, decoder=1)
```

`code` The item from the frame header to read, it is one of the following:

`TrackTitle` (0)
The track title.

`TrackArtist` (1)
The name of the artist or band.

`TrackAlbum` (2)
The album title.

`TrackComment` (3)
A general-purpose comment.

`TrackCopyright` (4)
Copyright information on the track.

`TrackSourceID` (5)
The ISRC code or any other code that identifies the track.

`TrackFormat` (6)
The kind of track, see the notes.

`TrackLength` (7)
The track duration in milliseconds.

`TrackBitrate` (8)
The bit rate of the current frame, or the average bit rate of the track, in kb/s.

`TrackSampleFreq` (9)
The sampling frequency of the track, in Hz.

`TrackCue` (10)
The cue time in milliseconds (silence at the start of the track).

`TrackSegue` (11)
: The segue time in milliseconds from the start of the track (silence at the end of the track).

`destination`
: The buffer that will hold the returned field as a packed string. This will be set to an empty string if no ID3 or APE 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.

`decoder`
: On devices with multiple decoders, two tracks can play simultaneously. This parameter specifies which decoder to query.

Returns: The value of the requested item (or 0 if the requested item is not numeric).

Notes: Some of the track information is read from a “tag” that is optionally added to a track. MP3 files often have an ID3 tag or an APE tag, both of which are supported. Other fields are extracted from the headers or binary information of tracks. See section “Resources” on page 136 for details on the ID3 and APE tags.

The Starling supports version 2 of the ID3 tag. The support for Unicode frames in the ID3 tag is limited to the characters of the Basic Multilingual Plane.

The `TrackFormat` field is one of the following values:

`0`	MPEG version 1, layer 3 (MP3)
`1`	Vorbis
`2`	WAVE

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.

Depending on the format of the track, the bit rate that this function returns is either the average bit rate of the complete track, or the bit rate at the current position in the track. For constant bit rate files, the bit rate is of course the same at any position in the file.

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 136 for pointers to in-depth information on the MPEG audio file format.

The cue and segue time need to be read from an APE tag. See the section "Resources" on page 136 for information on the APE tag and cue/segue times.

The `SYLT` (Synchronized lyrics) frame in an ID3 tag is not returned by this function, but events or cues in the `SYLT` tag "fire" the public function `@synch` at the appropriate times.

See also: `@synch`, `play`

trackpassword

Set the user password for encrypted tracks

Syntax: `trackpassword(const password[])`

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

Returns: This function currently always returns 0.

Notes: Currently, only MP3 tracks can be encrypted.

This function sets the "user password" for deciphering encrypted audio tracks. The user password must match the password that was used for encrypting the 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 audio tracks. The user password is therefore an augmented protection. Even if the password "leaks out", the audio 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 audio tracks will still play as before. Setting a user password has only effect on encrypted tracks.

transmit

Transmit a string over the serial line

Syntax: `bool: transmit(const data[], length=-1, port=1)`

`data`	The array with data to send.
`length`	The number of bytes in the array (which must be a packed array). If set to -1, the `data` parameter must be a zero-terminated string.
`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 the public function `@receive` See page 52 for details. Alternatively, one may call function `receive` 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.

Example: See `serial.p` on page 10.

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

treble

Tone adjust (treble)

Syntax: `treble(gain, frequency=3000, decoder=1)`

`gain`	The gain in the range of −12 dB to +11 dB.
`frequency`	The frequency at which the attenuation/enhancement starts. The This parameter is clamped between 1 kHz and 15 kHz (1000 to 15.000 Hz).
`decoder`	The decoder to which the tone adjustment applies. For models with a dual decoder, this parameter can be 1 or 2. This parameter is ignored on models with a single decoder.

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

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, e.g. 1 for version 1.2 of the firmware.

`VersionMinor` (1)
The minor version number, e.g. 2 for version 1.2 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. This value is currently always zero.

Returns: This function returns the requested value, or zero on error. Note that the build number is never zero.

vumeter

Return the volume level

Syntax: `vumeter(channel=0, decoder=1)`

`channel`	The channel whose volume level to query; it must be 1 for the left channel and 2 for the right channel. When setting this value to 0, the function returns the weighted average of both channels.
`decoder`	The decoder whose volume level to return.

Returns: This function returns the VU value.

Notes: The return value pertains to the level of the audio source. The values of this function do not change if you adjust the volume with function `setvolume`.

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

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 Starling 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 Starling.

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 Starling 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.

The APE tag is described at **http://wiki.hydrogenaudio.org**. In contrast to the ID3 tag, the APE tag contents are free format, with no mandated field names. The Starling supports a set of the more common fields.

Since the Starling 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`.