

# Boundary Devices

## Javascript Environment

(a.k.a. bdScript)

November 21, 2009

### 1 Revision History

| <b>Date</b> | <b>Revision</b> | <b>Description</b>  |
|-------------|-----------------|---|
| 2007-05-11  | 1.3             | Fourth incomplete draft. (Added <code>usb1p</code> and Postscript examples) |
| 2006-04-02  | 1.2             | Third incomplete draft. (Added TCP networking and Audio input examples)     |
| 2005-11-28  | 1.1             | Second incomplete draft. (to section 4.16)                                  |
| 2005-11-17  | 1.0             | First incomplete draft. (to section 4.10, Example 23)                       |

## Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Revision History</b>                         | <b>1</b> |
| <b>2</b> | <b>Introduction</b>                             | <b>2</b> |
| <b>3</b> | <b>Goals</b>                                    | <b>2</b> |
| <b>4</b> | <b>Examples</b>                                 | <b>3</b> |
| 4.1      | Hello, Javascript . . . . .                     | 3        |
| 4.2      | Invocation, argc and argv . . . . .             | 4        |
| 4.2.1    | shebang . . . . .                               | 4        |
| 4.2.2    | Direct invocation . . . . .                     | 5        |
| 4.3      | The Curl Cache . . . . .                        | 6        |
| 4.4      | Image display . . . . .                         | 8        |
| 4.5      | Other events for URL objects . . . . .          | 12       |
| 4.6      | Garbage collection . . . . .                    | 14       |
| 4.7      | MP3 Playback . . . . .                          | 15       |
| 4.8      | MPEG Playback . . . . .                         | 16       |
| 4.9      | Flash Playback . . . . .                        | 18       |
| 4.10     | The screen . . . . .                            | 19       |
| 4.10.1   | screen.clear() . . . . .                        | 19       |
| 4.10.2   | screen.line() . . . . .                         | 20       |
| 4.10.3   | screen.getPixel(), screen.setPixel() . . . . .  | 21       |
| 4.10.4   | screen.getRect(), screen.invertRect() . . . . . | 23       |
| 4.10.5   | screen.rect(), screen.box() . . . . .           | 24       |
| 4.11     | Image transparency . . . . .                    | 25       |
| 4.12     | Text rendering . . . . .                        | 26       |
| 4.13     | Touch Screen Calibration . . . . .              | 27       |
| 4.13.1   | Storage . . . . .                               | 27       |
| 4.13.2   | Raw and Cooked . . . . .                        | 28       |
| 4.13.3   | Calibration script . . . . .                    | 29       |
| 4.13.4   | Calibrated readings . . . . .                   | 30       |
| 4.14     | Buttons . . . . .                               | 30       |
| 4.14.1   | Bare buttons . . . . .                          | 30       |
| 4.14.2   | image buttons . . . . .                         | 33       |
| 4.14.3   | text buttons . . . . .                          | 34       |
| 4.15     | Webcam input . . . . .                          | 35       |
| 4.16     | Image manipulation . . . . .                    | 37       |
| 4.17     | Printer support . . . . .                       | 38       |
| 4.17.1   | CBM and Star printers . . . . .                 | 38       |
| 4.17.2   | usb1p . . . . .                                 | 38       |
| 4.17.3   | Postscript support . . . . .                    | 39       |
| 4.17.4   | Swecoin graphics support . . . . .              | 41       |

---

|  |           |
|--|-----------|
| 4.17.5 Quick image parsing . . . . .                 | 41        |
| 4.18 Networking with TCP . . . . .                   | 42        |
| 4.18.1 A first example: HTTP HEAD . . . . .          | 44        |
| 4.18.2 Another example: Streaming Web Data . . . . . | 46        |
| 4.19 Audio input . . . . .                           | 47        |
| 4.20 Filesystem access . . . . .                     | 50        |
| 4.20.1 Read/Write complete files . . . . .           | 50        |
| 4.20.2 Device driver and pipe access . . . . .       | 50        |
| 4.20.3 Watchdog support . . . . .                    | 51        |
| <b>5 Class and Function Reference</b>                | <b>53</b> |
| 5.1 'library' and 'use' . . . . .                    | 53        |
| 5.2 alphaMap . . . . .                               | 53        |
| 5.3 barcodeReader . . . . .                          | 53        |
| 5.4 bitmap . . . . .                                 | 53        |
| 5.5 button . . . . .                                 | 53        |
| 5.6 Camera . . . . .                                 | 53        |
| 5.7 CBM . . . . .                                    | 53        |
| 5.8 childProcess . . . . .                           | 53        |
| 5.9 curlFile . . . . .                               | 53        |
| 5.10 dir . . . . .                                   | 53        |
| 5.11 environ, getenv and setenv . . . . .            | 53        |
| 5.12 exit . . . . .                                  | 53        |
| 5.13 garbageCollect . . . . .                        | 53        |
| 5.14 gotoURL . . . . .                               | 53        |
| 5.15 fade . . . . .                                  | 53        |
| 5.16 file . . . . .                                  | 53        |
| 5.17 FileSys . . . . .                               | 53        |
| 5.18 flashMovie . . . . .                            | 53        |
| 5.19 flashVariable . . . . .                         | 53        |
| 5.20 font . . . . .                                  | 53        |
| 5.21 image . . . . .                                 | 53        |
| 5.22 Kernel . . . . .                                | 53        |
| 5.23 monitorWLAN . . . . .                           | 53        |
| 5.24 mp3Cancel . . . . .                             | 53        |
| 5.25 mp3File . . . . .                               | 53        |
| 5.26 mpegFile . . . . .                              | 53        |
| 5.27 nanosleep . . . . .                             | 53        |
| 5.28 pinger . . . . .                                | 53        |
| 5.29 printer . . . . .                               | 53        |
| 5.30 queueCode . . . . .                             | 53        |
| 5.31 recordBuffer . . . . .                          | 54        |
| 5.32 Screen . . . . .                                | 54        |
| 5.33 serialPort . . . . .                            | 54        |

|          |                                  |           |
|----------|----------------------------------|-----------|
| 5.34     | starStatus . . . . .             | 54        |
| 5.35     | starUSB . . . . .                | 54        |
| 5.36     | stat . . . . .                   | 54        |
| 5.37     | tcpClient . . . . .              | 54        |
| 5.38     | touchScreen . . . . .            | 54        |
| 5.39     | TTY . . . . .                    | 54        |
| 5.40     | udpSocket . . . . .              | 54        |
| 5.41     | usblp . . . . .                  | 54        |
| 5.42     | use and modularity . . . . .     | 54        |
| 5.43     | waitFor . . . . .                | 54        |
| 5.44     | waveFile . . . . .               | 54        |
| <b>6</b> | <b>Useful Javascript modules</b> | <b>54</b> |
| 6.1      | dump.js . . . . .                | 55        |
| 6.2      | staticText.js . . . . .          | 56        |

## 2 Introduction

Boundary Devices has wrapped many native Linux and library system calls into a convenient scripting environment based on the Javascript Library from Mozilla Project.

## 3 Goals

1. All content via URL
2. Keep it small

## 4 Examples

Before describing the details of each and every class and function, a number of examples will help show the types of things for which the Boundary Devices Javascript runtime is useful.

### 4.1 Hello, Javascript

In the K&R tradition, the following shows the simplest case of a Javascript applet.

Example 0

```
exit(0);
print( "Hello, Javascript\n" );
```

Short as it is, this example still shows a number of useful concepts.

The first thing you may notice is that the `exit()` call comes before the `print` call. This may lead you to ask “does the `print()` call execute?”.

The answer is *yes*. This is done because the `exit()` call sets a flag to the interpreter which is acted upon by the `jsExec` executable after the current Javascript frame finishes execution. In this simple case, the rationale is lost, but more complex examples may have multiple processes, pipes, etc opened. Deferring process `exit()` until the execution pipeline has completed allows proper cleanup of these resources.

For those new to Javascript, you may also note the C-style escaping of strings in the `print()` call. If you know the “C”, “C++”, or “Java” languages, you should feel at home here.

The `print()` routine itself is worthy of some mention at this point as well. Like the “C” `printf()` call, it accepts a variable number of arguments. Unlike `printf`, no format string is used. Rather, each argument is converted into a string through the Javascript `toString()` built-in.

The following example illustrates this in more detail.

Example 1

```
exit(0);
var i = 3 ;
var a = [ 1, 2, 3, 4 ];
var o = { m1: 1, m2: "string" };

print( "Hello, Javascript\n",
      'i == ', i, '\n',
      "a == '", a, "'\n",
      "o == '", o, "'\n" );
```

The output generated from this applet looks like the following.

Example 1 output

```
Hello, Javascript
i == 3
a == '1,2,3,4'
o == '[object Object]'
```

This example also illustrates a handful of new concepts.

- **Javascript allows single or double quotes.** In the example above, the array and object output values are quoted, and the rationale for switching quotes is shown. Use singly quoted strings to contain embedded double-quotes. Use double-quoted strings to embed single quotes. “C”-style escaping is also supported in case your keyboard’s broken or you have a fondness for backslashes.
- **Array.toString() produces readable output.**
- **Object.toString() doesn’t.** In order to display the content of an object, you’ll need to either override the `toString()` method, or iterate the member variables as done in `dumpObj()` (See 6.1).

## 4.2 Invocation, argc and argv

If you have access to a Boundary Devices board with `jsExec` installed, you’re probably asking how you can run these examples. In the style of most other \*nix scripting languages, there are two ways to invoke the Javascript interpreter, either through *shebang* or direct invocation of `jsExec`.

### 4.2.1 shebang

Invocation through *shebang* is performed by as follows.

Example 2

```
#!/bin/jsExec -
print( "Hello, Javascript\n" );
for( var arg = 0 ; arg < argc ; arg++ )
    print( "arg[,arg,] == '", argv[arg], "'\n" );
exit(0);
```

This form is most useful when you need to pass command line arguments to the script or want to call out to a Javascript applet from within another script (e.g. a shell script).

Note that the hyphen (-) on the shebang line is required. It tells the `jsExec` program to read the script from stdin.

The following screen scrape ties it all together.

Example 3

```
/ # cat >test
#!/bin/jsExec -
print( "Hello, Javascript\n" );
for( var arg = 0 ; arg < argc ; arg++ )
    print( "arg[" ,arg, "] == '", argv[arg], "'\n" );
exit(0);
/ # chmod a+x test
/ # ./test
Hello, Javascript
/ # ./test This is a test
Hello, Javascript
arg[0] == 'This'
arg[1] == 'is'
arg[2] == 'a'
arg[3] == 'test'
```

Note the usage of the implicitly defined `argc` and `argv` variables. They are similar to the “C” variables except that `argv[0]` does not include the path to the executable.

#### 4.2.2 Direct invocation

The second form of invocation is more direct and involves running `jsExec` with a command-line parameter containing the script as shown below.

Example 4

```
/ # cat >test.js
print( "Hello, Javascript\n" );
for( var arg = 0 ; arg < argc ; arg++ )
    print( "arg[" ,arg, "] == '", argv[arg], "'\n" );
exit(0);
/ # jsExec file:///test.js
Hello, Javascript
/ # jsExec file:///test.js This is a test
Hello, Javascript
arg[0] == 'This'
arg[1] == 'is'
arg[2] == 'a'
arg[3] == 'test'
```

This example also illustrates the use of URLs to refer to content. As you may have guessed, the `'http:'` URL method is also supported. If you place the `'test.js'` file on a web server, you can execute the script as shown below.

Example 5

```
/ # jsExec http://192.168.0.126/ericn/test.js This is a test
Hello, Javascript
arg[0] == 'This'
arg[1] == 'is'
arg[2] == 'a'
arg[3] == 'test'
```

If your system has name resolution configured or a `hosts` file, and your server is named, you can also invoke this through the use of a host name. The `test.js` file created above is available on the Boundary Devices website to illustrate.

Example 6

```
/ # cat >/etc/hosts
66.113.228.134 boundarydevices.com
^D
/ # jsExec http://boundarydevices.com/test.js 1 2 3
No touch screen settings, using raw input
Hello, Javascript 222
arg[0] == '1'
arg[1] == '2'
arg[2] == '3'
```

As stated in the Goals section, most content within the Javascript runtime may be retrieved by URL. This includes:

- scripts (`.js` files)
- fonts (True-type or `.pfb` bitmap)
- audio (MP3 or WAV)
- images (GIF, PNG, JPEG)
- movies (MPEG, Flash)

This discussion of URL execution leads directly into the next section.

### 4.3 The Curl Cache

As mentioned in the Neon user's manual, the `bdScript` environment uses `libCURL` to provide HTTP support. This decision was so fundamental to the Boundary Devices Javascript environment that the library built from the scripting directory is named `libCurlCache`.

As its' name implies, this library provides support for caching data retrieved via `CURL`. It does this in a manner that allows very tight restrictions on the total size used (for memory-constrained systems).

By default, the CURL cache is placed in `/tmp/curl/` and the total space is limited to 3MB. Since the `/tmp` directory is typically a RAM disk, this means that the cache will not persist across boots.

The default directory and size can be overridden through the use of the `CURLTMPDIR` and `CURLTMPsize` environment variables.

Example 7

```

/ # export CURLTMPDIR=/mmc/curlcache
/ # export CURLTMPsize=16000000

/ # jsExec http://boundarydevices.com/test.js 1 2 3
No touch screen settings, using raw input
Hello, Javascript
arg[0] == '1'
arg[1] == '2'
arg[2] == '3'

/ # ls /mmc/curlcache/ -l
-rwxr-xr-x    1 0          0                181 Jan  1 02:29 00000000

/ # cat /mmc/curlcache/00000000
#http://boundarydevices.com/test.jsprint( "Hello, Javascript\n" );
for( var arg = 0 ; arg < argc ; arg++ )
    print( "arg[" ,arg, "] == '", argv[arg], "'\n" );
exit(0);

```

As shown above, files in the Curl cache are not directly named, but have their URLs encoded within the file. Additional information such as the last-access time are also encoded within the file.

When the cache size is exceeded, URL data is removed from the Curl cache in a strictly *least recently used* order.

What's not apparent from the above is that this caching does not use either “Last-modified” headers or ETag mechanisms for refreshing data from the cache.

This conscious decision prevents the need for a round-trip with a Web Server upon repeated access to a URL. It also has a ramification on usage of `jsExec` during development.

To illustrate this, let's modify the `test.js` script used above on the 192.168.0.126 web server.

Example 8

```

ericn@linuxbox ericn $ cat test.js
print( "Hello, Javascript\n" );
for( var arg = 0 ; arg < argc ; arg++ )
    print( "arg[" ,arg, "] == '", argv[arg], "'\n" );
exit(0);

```

```
ericn@linuxbox ericn $ cat > test.js
print( "Hello, again\n" );
exit(0);
```

If we run this script again without a restart, we see the same results as before:

```
/ # jsExec http://192.168.0.126/ericn/test.js This is a test
Hello, Javascript
arg[0] == 'This'
arg[1] == 'is'
arg[2] == 'a'
arg[3] == 'test'
```

In order to counteract this, use a command-line like this to allow your command-line recall buffer to work for you. This way, multiple runs of a file are a simple matter of <up-arrow><Enter>.

```
/ # rm -f $CURLTMPDIR/* && \
> jsExec http://192.168.0.126/ericn/test.js
Hello, again
```

#### 4.4 Image display

If you've been following this document in order, you may be getting bored by now. We're more than a half-dozen pages into the document, and haven't done anything more interesting than print stuff to the console (generally a serial port).

Let's fix that. The following small applet `showImage.js` shows how to display an image on the screen at X/Y coordinate [0,0].

##### Example 9

```
var logo = image( {url: 'neonSplash.png'});
logo.draw(0,0);
exit(0);
```

Because the simplest way to develop and execute Javascript applets is via a web-server for which you can edit the content, all of the examples below will assume that environment.

If you happened to notice that you need a 'neonSplash.png' file and placed it in the same directory as the Javascript file, you'll get no output on the console, but should see an image on the LCD panel.

```
/ # rm -f $CURLTMPDIR/* && \  
> jsExec http://192.168.0.126/ericn/showImage.js  
/ #
```

More likely, since it wasn't mentioned, you won't have a 'neonSplash.png' file in the appropriate directory and you'll see the following output:

```
/ # rm -f $CURLTMPDIR/* && \  
> jsExec http://192.168.0.126/ericn/showImage.js  
Error Object not initialized, can't draw  
file http://192.168.0.126/ericn/showImage.js, line 2  
/ #
```

This brings up another point: Javascript errors, including syntax errors and the like will display on stderr, and will include references back to the source (Javascript) file and line number.

Another thing you may have noticed is the weird curly-brace {} syntax in the `image()` call.

The syntax itself is used to define an anonymous object literal. As shown in the earlier example, Javascript allows definition of an object through the use of curly braces enclosing a set of member-variable declarations.

This is used in the `image()` call as a means of passing a variable set of parameters by name, rather than by position. This convention is used throughout the `bdScript` environment when constructing an object. The reason is to allow completion handlers to run upon completion of downloads.

Did you notice that downloading and decoding all occurred within line 1 of the example above? It did.

That's appropriate in those cases when an image is expected to be immediately available (e.g. when using 'file://' URLs). When a media file is downloaded from a server, it is often more useful to allow other operations to continue during the download, and to defer processing until end-of-download.

The following is a minimal example of just that.

Example 10: minAsync.js

```
var arg = 0 ;

function showFile( file ){
  print( 'file:', file.initializer.url, ' loaded\n' );
  file.draw(0,0);
  loadFile();
}

function fileError( file ){
  print( 'loadError ', file.initializer.url, '\n' );
  print( 'msg:', file.loadErrorMsg, '\n' );
  loadFile();
}

function loadFile(){
  if( arg < argc ){
    logo = new image( {
      url: argv[arg++]
      , onLoad: 'showFile(this);'
      , onLoadError: 'fileError(this);'
    }
  );
  }
  else {
    print( 'all files loaded\n' );
    exit(0);
  }
}

if( 0 < argc )
  loadFile();
else
  print( 'usage: minAsync.js imgFile [imgFile]\n' );
```

A large number of concepts are introduced in the previous example. Let's go through some of them here.

- **You can define functions.** You probably already guessed this even if you're new to Javascript, but here it is in action.
- **You can use functions prior to declaration.** Notice that the `loadFile()` function is used in `showFile()` prior to its' definition and without declaration. Since Javascript utilizes late binding, it isn't necessary to have the `loadFile()` routine defined until it is used at run-time.
- **There's code in quotes!** If you're an old-hand at Javascript, you might be used to seeing this. One of the primary features of Javascript is its' ability to execute code from string values. The `eval()` routine exposes this to Javascript. The `jsExec` program also uses this internally.
- **How do we know when it's done?** A former colleague of mine used to ask this question on a regular basis, and it applies in so many areas... All of the prior examples showed the `exit()` routine called in the main-line (unindented) code. That is actually fairly rare for `bdScript`, and used only when the script is strictly sequential. Most uses of `bdScript` are event-driven, and terminate in response to some event. This example shows a typical case, exiting when all files have been handled.
- **The stringified code uses 'this'.** As is typical when using Javascript in HTML documents, event handlers generally execute in the context of an object. That is, they act as if they were method calls to the object to which the event is generated. Because the `showFile()` and `fileError()` functions need access to the object, it is resolved and passed as a parameter by the event handler string.
- **The file variable has an initializer.** Remember the anonymous object passed to the `image()` routine? By convention, all of the objects which can be referred to by URL use constructor functions that accept a single object as a parameter. They also retain a reference to this object in the `initializer` member variable. What this means for you is that not only can you refer to the pieces used by the constructor (e.g. `url`), but you can also add your own fields into the initializer for reference later in an event handler! Javascript is a loosely typed language. You can use this to your advantage.
- **The example uses `new image()`, not `image()`.** The use of the `new` keyword is used by `bdScript` to make the distinction between asynchronous operation and synchronous operation. If the `new` keyword is

not used, the script will block on the `image()` call until the transfer is complete or fails.

- **logo is undeclared and mostly unused.** Good catch! All of the previous examples showed variable declarations using the `var` keyword. Definition of a variable without `var` creates a global variable. In this example, we actually need the 'logo' variable, and it is appropriately global to keep the garbage collector from deciding that the file being transferred is un-referenced. To be precise, we could have used the `var` keyword, since local variables are persistent in Javascript (similar to *static* local variables in "C").

#### 4.5 Other events for URL objects

A couple of other events are available for use with URL-supporting objects. You can register a handler in the *initializer* object through the `onSize` and `onProgress` member variables.

Any code attached to the `onSize` is executed when (if) we know the size of the file to be downloaded (from the HTTP header *Content-Length*). Note that if the file being downloaded is generated dynamically on the server (as is sometimes done with images or scripts), the handler will not execute.

Code associated with the `onProgress` initializer member is executed at intervals during the download process.

These events are illustrated in the following example. The new functions `showSize()` and `showProgress()` are added, and referred to in the `image()` initializer.

Example 11: imageAsync.js

```
var arg = 0 ;

function showFile( file ){
    print( 'file:', file.initializer.url, ' loaded\n' );
    file.draw(0,0);
    loadFile();
}

function fileError( file ){
    print( 'loadError ', file.initializer.url, '\n' );
    print( 'msg:', file.loadErrorMsg, '\n' );
    loadFile();
}

function showSize( file ){
    print( 'expecting ', file.expectedSize, ' bytes\n' );
}

function showProgress( file ){
    print( file.readSoFar, ' bytes read\n' );
}

function loadFile(){
    if( arg < argc ){
        logo = null ;
        garbageCollect();
        logo = new image( {
            url: argv\[arg++\]
            , onLoad: 'showFile(this);'
            , onLoadError: 'fileError(this);'
            , onSize: 'showSize(this);'
            , onProgress: 'showProgress(this);'
        }
        );
    }
    else {
        print( 'all files loaded\n' );
        exit(0);
    }
}

if( 0 < argc )
    loadFile();
else
    print( 'usage: minAsync.js imgFile \[imgFile\]\n' );
```

## 4.6 Garbage collection

Did you know that Javascript is a garbage-collected language?

It is. Refer to more definitive documents such as the O'Reilly Rhino book for a more complete description. What needs mention here is that the `jsExec` program provides you some control over garbage collection.

In particular, the `garbageCollect()` method may be called to free up resources at points in your application when you know that the time spent won't be noticed by a user. Even though garbage collection generally takes only a fraction of a second, there are times when you really don't want the user to notice that fraction of a second.

In general, your applications should call `garbageCollect()` when a slight delay might be appropriate, or when resources have become un-reachable. The `imageAsync.js` example above has a convenient point, for instance.

If a larger number of images were being displayed, the line before the `image()` call within `loadFile()` would be a good candidate. The application is about to connect to a web server, so a slight delay will be hidden. Also, the `logo` variable is about to be re-assigned to a new object. By releasing the reference first, we could reclaim the RAM associated with the previous file.

```
...
logo = null ;           // remove reference to logo
garbageCollect();      // reclaim resources
logo = new image( {
    ...
```

Actually, this isn't quite true. Note that after the first call, the `loadFile()` call is made from within a handler. Since the *this* reference in the handler is still around, the previous image cannot be reclaimed without some additional effort. There are ways to address this, but they're only necessary in very memory-constrained environments or with very large content (see `queueCode()` and `timer()`).

## 4.7 MP3 Playback

All this talk of garbage is getting me down. How about some music?

As with images, only a small number of lines are necessary.

```

Example 12: mi2.js
var tune = mp3File( {url: 'mi2.mp3'} );
if( tune.isLoaded )
    tune.play( { onComplete:'exit(0);'} );
else {
    print( 'error ', tune.loadErrorMsg, ' loading tune\n' );
    exit(1);
}

```

To run this example, you'll need an MP3 audio file. The sample 'mi2.mp3' file is available [here](#).

Whoah! Was that as loud for you as it was for me?

Let me tell you about the `setVolume()` call. It takes one parameter, a number between zero and 100. Larger is louder. For typical office use, a value around 80 is a good choice.

```

Example 13: notSoLoud.js
setVolume(80);
var tune = mp3File( {url: 'mi2.mp3'} );
if( tune.isLoaded )
    tune.play( { onComplete:'exit(0);'} );
else {
    print( 'error ', tune.loadErrorMsg, ' loading tune\n' );
    exit(1);
}

```

You'll note a lot of similarity here to the `image()` examples. The *initializer* is present again, and handlers for `onLoad`, `onLoadError`, `onSize`, and `onProgress` can be defined for `mp3Files` as well. The comment earlier about the use of *new* also applies here:

`new` means asynchronous.

Another global method is also available for cancelling audio output. The `mp3Cancel()` routine will cancel any audio queued to the speaker. It will also call a handler for each `mp3File()` object for which playback has been requested.

The following example illustrates usage of `mp3Cancel()` and the `onCancel` handler by only allowing 1 second of audio to play.

```

Example 14: oneSec.js
var tune = mp3File( {url: 'mi2.mp3'} );
if( tune.isLoaded ) {

```

```

tune.play( {
    onComplete:'exit(0);',
    onCancel: 'print( "cancelled\\n" ); exit(0);'
} );
oneShot( 1000, 'mp3Cancel();' );
}
else {
    print( 'error ', tune.loadErrorMsg, ' loading tune\\n' );
    exit(1);
}

```

This example introduces the `oneShot()` function, which executes a code snippet after a specified number of milliseconds.

It also shows that sometimes you need to double-escape backslashes when you have code within a Javascript string. The first one is eaten by the Javascript compiler when parsing the string, leaving a single backslash for when the Javascript runtime executes the statement.

## 4.8 MPEG Playback

Now for some real fun! Let's play a movie.

```

Example 15: ajacks.js
var movie = mpegFile( {url: 'ajacks.mpg'} );
if( movie.isLoading ) {
    movie.play( {
        x:0, y:0,
        width: movie.width,
        height: movie.height,
        onComplete:'exit(0);'
    } );
}
else {
    print( 'error ', movie.loadErrorMsg, ' loading movie\\n' );
    exit(1);
}

```

I hope that very little of this is surprising. By now, I think you will expect to see extra playback parameters to control the image placement. The width and height members may be a surprise. These are present because the SM-501 on the Neon board allows mpeg scaling.

If you want to see this in action, change the width and height parameters to the following.

```

Example 16: fullScreen.js
var movie = mpegFile( {url: 'ajacks.mpg'} );
if( movie.isLoaded ) {
    movie.play( {
        x:0, y:0,
        width: screen.width,
        height: screen.height,
        onComplete:'exit(0);'
    } );
}
else {
    print( 'error ', movie.loadErrorMsg, ' loading movie\n' );
    exit(1);
}

```

Depending on the aspect ratio of your display, this may look pretty good, or horribly stretched (6.2 display, anyone?).

You may also notice that the display on a Neon board reverts to its' old content on completion of playback. This is because the SM-501 has a hardware YUV overlay. The overlay is disabled when a movie completes playback.

More detail will be given shortly regarding the `screen` variable (see 4.10 and 5.32).

Because the pacing of video files is driven by the same thread as audio output, the `mp3Cancel()` routine can be used to stop movie playback as well as audio. This is illustrated below through the use of a `oneShot()` timer and `onCancel` handler.

```

Example 17: oneSecFull.js
var movie = mpegFile( {url: 'ajacks.mpg'} );
if( movie.isLoaded ) {
    movie.play( {
        x:0, y:0,
        width: screen.width,
        height: screen.height,
        onComplete:'exit(0);',
        onCancel: 'print( "cancelled\n" ); exit(0);'
    } );
    oneShot( 1000, 'mp3Cancel();' );
}
else {
    print( 'error ', movie.loadErrorMsg, ' loading movie\n' );
    exit(1);
}

```

Note that the `ajacks.mpg` MPEG file is available on the Boundary Devices [website](#). Another good source of fun movies to play is [Pocket Movies](#). Don't forget to donate!

## 4.9 Flash Playback

Through the use of the [Swift Tools](#) Flash Library, the `bdScript` environment also supports playback of Macromedia Flash<sup>TM</sup> (`.swf`) files.

Usage is almost identical to playback of MPEG files, so this example should just confirm what you already know.

Example 18: `bdLogo.js`

```
var anim = flashMovie( {url: 'bdLogo.swf'} );
if( anim.isLoaded ) {
    anim.play( {
        x:0, y:0,
        width: screen.width,
        height: screen.height,
        onComplete:'exit(0);'
    } );
}
else {
    print( 'error ', anim.loadErrorMsg, ' loading file\n' );
    exit(1);
}
```

It should also confirm what you may have guessed (I'm not an artist).

There's another difference between MPEG movies and SWF animations. Since Flash animations are timed separately from their audio, and may not even contain audio, the `flashMovie.stop()` method is used to cancel playback.

The following example illustrates.

Example 19: `oneSecBd.js`

```
var anim = flashMovie( {url: 'bdLogo.swf'} );
if( anim.isLoaded ) {
    anim.play( {
        x:0, y:0,
        width: screen.width,
        height: screen.height,
        onComplete:'exit(0);',
        onCancel: 'print( "cancelled\n" ); exit(0);'
    } );
    oneShot( 1000, 'movie.stop();' );
}
```

```
}  
else {  
    print( 'error ', anim.loadErrorMsg, ' loading file\n' );  
    exit(1);  
}
```

You've also probably noticed that Flash animations remain on the screen after playback completes. Flash animations do not use the YUV overlay, so clearing or restoring the screen is the responsibility of the application.

Convenient methods for doing this are available through the `screen` variable mentioned earlier.

## 4.10 The screen

The `Screen` class and `screen` variable have two primary variables (`width` and `height`) and a number of methods for capture and rendering to the system's frame buffer.

### 4.10.1 `screen.clear()`

The first of these is the `clear()` method. It takes one optional argument with the RGB value in typical fashion (one byte of `Red`, one byte of `Green`, and one byte of `Blue`). The parameter defaults to black if not specified.

Example 20: `blink.js`

```
screen.clear();  
screen.clear(0xFFFFFFFF);  
screen.clear();  
exit(0);
```

Depending on your screen resolution, you may or may not be able to see the blink. The next example will introduce a delay so you can see the blank.

Example 21: `rgb.js`

```
var red    = 0xFF0000 ;  
var green  = 0x00FF00 ;  
var blue   = 0x0000FF ;  
screen.clear();  
screen.clear(red);  
nanosleep(1.0);  
screen.clear(green);  
nanosleep(1.0);  
screen.clear(blue);  
nanosleep(1.0);  
screen.clear();  
exit(0);
```

### 4.10.2 screen.line()

The following example shows how you can draw horizontal lines through `screen.line()`.

Example 22: `grayVert.js`

```
for( var i = 0 ; i < screen.height ; i++ )
{
    var byteVal = i & 255 ;
    var rgb = byteVal << 16 | byteVal << 8 | byteVal ;
    screen.line( 0, i, screen.width-1, i, 1, rgb );
}
exit(0);
```

The following example shows how you can set pixel values directly.

Example 23: `shades.js`

```
var maxDist = Math.sqrt(
    screen.width*screen.width
    +screen.height*screen.height
);

for( var x = 0 ; x < screen.width ; x++ ) {
    for( var y = 0 ; y < screen.height ; y++ ) {
        var d = Math.sqrt( x*x+y*y );
        var lightness = d/maxDist ;
        var byteVal = Math.floor(255-(lightness*255));
        var rgb = byteVal << 16 | byteVal << 8 | byteVal ;
        screen.setPixel( x, y, rgb );
    }
}
exit(0);
```

### 4.10.3 screen.getPixel(), screen.setPixel()

The `screen.getPixel()` and `screen.setPixel()` routines allow access to individual pixels on the screen.

This example shows how you might use this to draw a circle of points on the screen.

Example 24: circles.js

```
screen.clear( 0xFFFFFFFF );
function degToRad( deg ) {
    return deg*(Math.PI/180);
}
var centerX = screen.width/2 ;
var centerY = screen.height/2 ;

//
// radius 10..height/2
//
for( var r = 10 ; r < centerY ; r += 20 )
{
    // each angle in degrees
    for( var a = 0 ; a < 360 ; a++ )
    {
        var rad = degToRad(a);
        var x = centerX + Math.floor(r*Math.sin(rad));
        var y = centerY + Math.floor(r*Math.cos(rad));
        screen.setPixel( x, y, 0 );
    }
}
exit(0);
```

The following example shows how to retrieve an RGB value using `screen.getPixel()`, transform the colors and redisplay using `screen.setPixel()`.

Example 25: rotColors.js

```
// rotate colors
function transform(rgb)
{
    var colors = [
        rgb>>16,          // red
        (rgb>>8) & 0xff,  // green
        rgb & 0xff ];    // blue
    return colors[1] << 16
        |colors[2] << 8
        |colors[0];
}

var logo = image( {
    url: 'boundaryDevices.png' }
);
logo.draw(0,0);
for( var x = 0 ; x < logo.width ; x++ )
{
    for( var y = 0 ; y < logo.height ; y++ )
    {
        var rgb = screen.getPixel( x, y );
        screen.setPixel(
            x, y,
            transform(rgb) );
    }
}
exit(0);
```

#### 4.10.4 screen.getRect(), screen.invertRect()

The `screen.getRect()` method returns an `image` object from a rectangle on the screen. Parameters are `x`, `y`, `width`, and `height`.

The `screen.invertRect()` method performs a binary inverse on the Red, Green, and Blue bits for every pixel in a screen rectangle. This is useful during development to animate buttons or other on-screen objects.

The following example shows how this can be used to tile an image across the screen.

Example 26: `tileLogo.js`

```
var logo = image( {
    url: 'boundaryDevices.png' }
);
logo.draw(0,0);
var r = screen.getRect( 0, 0, logo.width, logo.height );
var inv = false ;
for( var x = 0 ; x < screen.width; x += r.width )
{
    for( var y = 0; y < screen.height ; y += r.height )
    {
        r.draw(x,y);
        if( inv )
            screen.invertRect(
                x, y,
                x+logo.width-1,
                y+logo.height-1 );
        inv = !inv ;
    }
}

exit(0);
```

#### 4.10.5 `screen.rect()`, `screen.box()`

The `screen.rect()` routine will fill an on-screen rectangle with a specified color. Parameters are `x1`, `y1`, `x2`, `y2`, and `color` (as RGB).

The `screen.box()` routine will draw lines of the specified width within the rectangle specified as in `screen.rect`. Parameters are `x1`, `y1`, `x2`, `y2`, `penWidth` (*in pixels*), and `color` (*as RGB*).

Example 27: `rectBox.js`

```
var xLeft = screen.width/2 - screen.width/4 ;
var xRight = xLeft + screen.width/2 ;
var yTop = screen.height/2 - screen.height/4 ;
var yBottom = yTop + screen.height/2 ;

while( ( xLeft < xRight )
      &&
      ( yTop < yBottom ) )
{
  screen.rect( xLeft, yTop, xRight, yBottom, 0 );
  screen.box(
    xLeft, yTop,
    xRight, yBottom,
    8, 0x982c32 );
  xLeft += 16 ;
  xRight -= 16 ;
  yTop += 16 ;
  yBottom -= 16 ;
}

exit(0);
```

### 4.11 Image transparency

Note that transparency is supported for the PNG image file format. It is retained as a byte-map (range 0..255) in the `.alpha` member variable of the image if present. Otherwise `image.alpha` is left undefined.

Example 28: `alpha.js`

```
function showImage( path )
{
  var img = image( {url: path } );
  print( img.width, 'x', img.height, ' --> ' );
  if( null != img.alpha )
    print( img.alpha.length, ' bytes of alpha\n' );
  else
    print( 'no alpha\n' );
  img.draw( Math.floor(screen.width/2 - img.width/2),
           Math.floor( screen.height/2 - img.height/2) );
}
showImage( 'boundaryDevices.png' );
showImage( 'x24.png' );
exit(0);
```

The `Math.floor()` routine is used to convert a possibly fractional number to an integer.

## 4.12 Text rendering

The bytemap form of storage is also used when rendering text for display. The Boundary Devices Javascript application uses the FreeType library to generate byte-maps of a text string's transparency with a specified font.

The `font` object class is used to represent the font itself. `font.render( pointSize, stringValue )` will generate an alpha (or opacity) bytemap using the font. A value of 255 means fully opaque, and 0 means fully transparent. `alphamap.draw(x,y,color)` may be used to display the `alphamap` onto the display. Partially transparent values will be blended with the background, or currently displayed pixel color.

Example 29: `textRender.js`

```
var labelFont = font( { url: "Vera.ttf" } );
var labelSize = 12 ;
var labelImg = labelFont.render(
    labelSize, 'Hello, bdScript' );
var black = 0 ;
labelImg.draw(
    Math.floor(screen.width/2-labelImg.width/2),
    Math.floor(screen.height/2-labelImg.height/2),
    black );
exit(0);
```

This differs in a couple of ways from the alpha or opacity values used with PNG images. The first is that the `alphamap` returned from `font.render()` is an object with a width and height.

Variable width fonts are inherently supported, so the width is normally not computable without rendering the string. Also, the height is clamped to the height needed for any opaque pixels. A couple of other member variables are also present to indicate the offset from the *baseline* and the number of pixels for each row in the Y or vertical direction <sup>1</sup>.

To recap, the following are the member variables of the `alphamap` class.

- `pixBuf` - Opacity of each pixel
- `width` - pixels of width
- `height` - pixels of height
- `baseline` - pixels above the baseline to begin rendering
- `yAdvance` - number of pixels to advance the baseline between rows

---

<sup>1</sup>This is computed from the point size and does not depend on the string

The following example shows how the *baseline* and *yAdvance* member variables are generally used. The *yPos* variable indicates the current baseline position, and the *baseline* value is subtracted before rendering.

Example 30: textMetrics.js

```

screen.clear(0xFFFFFFFF);
var labelFont = font( { url: "Vera.ttf" } );
var labelSize = 12 ;
var yPos = 20 ;
var black = 0 ;
function showLine( line )
{
    var labelImg = labelFont.render(
        labelSize, line );
    labelImg.draw(
        Math.floor(screen.width/2-labelImg.width/2),
        yPos-labelImg.baseline,
        black );
    print( 'string: <', line
        , '>, baseline: ', labelImg.baseline
        , ', yPos: ', yPos
        , ', advance: ', labelImg.yAdvance
        , '\n' );
    yPos += labelImg.yAdvance ;
}

showLine( 'Hello, Javascript' );
showLine( 'Hello, again' );
showLine( '_____' );
showLine( '^^^^^^' );
showLine( 'WWWWWjjjj----' );
exit(0);

```

## 4.13 Touch Screen Calibration

### 4.13.1 Storage

Before moving on to the subject of touch-screen access, we'll need to cover the process of calibrating the screen.

Calibration data is currently stored in the last sector of an MTD device defined by the `FLASHVARDEV` environment variable (or `/dev/mtd2` by default). It, along with other setup information is stored in a linked list of sorts, with duplicate entries appended to the end of the chain to perform wear-leveling. (When the end-of-sector is reached, the collapsed list is stored at the beginning of the sector)

The `flashVar` utility program (defined in `build/bdScript`) may be used to examine the contents of this parameter storage area. The following shows the content after the `tsCalibrate.js` applet has been run twice on a device. The second set of values will be used.

Think of this as a persistent set of environment variables.

```

flashVar usage
-----
/mmc # flashVar
tsCalibrate=-0.00003171245956402457,
           -0.8351079814810448,
           826.1304443952578,
           0.5506458834865502,
           -0.0023623390472401248,
           -43.20371650258457,
           15,
           8
tsCalibrate=-0.007830978303665637,
           -0.8334215554800619,
           831.1713006878666,
           0.5607566332050368,
           0.0011646925910092838,
           -50.08751187936246,
           19,
           14
used 139 of 262144 bytes

```

#### 4.13.2 Raw and Cooked

If no setting is present in the flash parameter area for the `tsCalibrate` variable, the touch screen will operate in `raw` mode, and the `touchScreen` object will report the raw A/D readings from the UCB1400 touch screen controller, rather than returning `cooked` pixel values.

The `touchScreen.setRaw()` and `touchScreen.setCooked()` methods may also be used to control whether the values are returned in pixels or A/D units. The following show how the `onTouch()` and `onRelease()` global methods can be used along with the `touchScreen` object's `getX()` and `getY()` may be used to perform actions.

```

Example 31: rawTouch.js
-----
function press() {
    screen.clear(0);
    print( "touch: ",
          touchscreen.getX(),
          ":", touchscreen.getY(), "\n" );
}

```

```
function release(){
    screen.clear( 0xFFFFFFFF );
    print( "release: ",
          touchscreen.getX(),
          ":", touchscreen.getY(), "\n" );
}

screen.clear( 0xFFFFFFFF );

onTouch( "press()" );
onRelease( "release()" );
```

The `touchScreen.getX()` and `touchScreen.getY()` routines are used to retrieve touch data directly. They each report the last known values.

The `tsCalibrate.js` script uses readings from five points in raw mode to produce the coefficients to the following equations. Variables  $i$  and  $j$  are the two A/D readings.  $x$  and  $y$  are the resulting pixel values.

$$x = a_1i + a_2j + a_3$$

$$y = b_1i + b_2j + b_3$$

### 4.13.3 Calibration script

A Javascript implementation of Touch-Screen calibration is available [here](#). As described earlier, it places the touch screen into raw mode, then prompts the user to touch five points on the screen, accumulating data to solve the equations above.

After running the calibration utility, you should see (another) set of calibration constants in the `tsCalibrate` flash variable.

```

                                     "tsCalibrate.js"
/mmc # jsExec file:///mmc/calibrate/tsCalibrate.js
...debug msgs go here
saving touch settings
/mmc # flashVar
tsCalibrate=-0.00003171245956402457,-0.8351079814810448,826.1304443952578,
0.5506458834865502,-0.0023623390472401248,-43.20371650258457,15,8
tsCalibrate=-0.007830978303665637,-0.8334215554800619,831.1713006878666,
0.5607566332050368,0.0011646925910092838,-50.08751187936246,19,14
tsCalibrate=0.00011994945569704114,-0.8386250196935172,832.2495249514159,
0.5792547477031031,0.012674838118714359,-63.474716968753995,15,11
used 417 of 262144 bytes
```

#### 4.13.4 Calibrated readings

Now that we can read calibrated points from the touch screen, we can make decisions based on them. The following example shows a simple script that uses the `touchScreen.getX()` and `touchScreen.getY()` routines to display a red cross at each touched point:

Example 32: redCross.js

```
var color = 0xFF0000 ;

function cross( x, y )
{
    if( x < 0 ) x = 0;
    if( x >= screen.width ) x = screen.width-1;
    if( y < 0 ) y = 0;
    if( y >= screen.height ) y = screen.height-1;
    screen.line(
x >= 10 ? x-10 : 0, y,
x+10, y, 1, color );
    screen.line(
x, y >= 10 ? y-10 : 0,
x, y+10, 1, color );
}

function press() {
    cross(
        touchScreen.getX(),
        touchScreen.getY(),
        color );
}

screen.clear( 0xFFFFFFFF );

onTouch( "press()" );
onMove( "press()" );
```

## 4.14 Buttons

The `button` Javascript class is used to represent a rectangular area of the screen for which touch, release, and *move off* actions should be associated.

### 4.14.1 Bare buttons

The following example shows the simplest usage of buttons as on-screen rectangles. It is a variation of the earlier `tileLogo.js` example program

that inverts the on-screen area for a button while pressed, and flips the Boundary Devices logo upon release.

Example 33: tileButton1.js

```
var logo = image( {
  url: 'boundaryDevices.png' }
);
var invert = logo.rotate90();
invert = invert.rotate90();

function press() {
  screen.invertRect(
    this.x, this.y,
    this.width, this.height
  );
}

function release() {
  print( "release: ", this.x, ":", this.y, "\n" );
  this.inv = !this.inv ;
  if( this.inv )
    invert.draw( this.x, this.y );
  else
    logo.draw( this.x, this.y );
}

var buttons = new Array();
var inv = false ;
var w = logo.width ;
var h = logo.height ;
for( var xPos = 0 ; xPos < screen.width; xPos += w )
{
  for( var yPos = 0; yPos < screen.height ; yPos += h )
  {
    var b = button( {
      x: xPos, y: yPos,
      width: w,
      height: h,
      onTouch: press,
      onMoveOff: release,
      onRelease: release } );
    b.inv = inv ;

    if( inv )
      invert.draw(xPos,yPos);
```

```
        else
            logo.draw(xPos,yPos);
        inv = !inv ;
        buttons.push( b );
    }
}
```

Note that the `button` call accepts a variable number of parameters in the same manner as URL-supporting classes. This allows optional, non-positional parameters. The minimum set of parameters is `x`, `y`, `width`, and `height`. All are expressed in pixels.

Event handlers are specified through the `onTouch`, `onRelease`, and `onMoveOff` parameters. Note that each of these can be either a string form of code or a function reference. In either case, the event handler operates in the context of the button, so the member variables of the button are available through the `this` keyword.

#### 4.14.2 image buttons

In the example above, all drawing is deferred to Javascript code. Because a two image button with one image referring to the *untouched* state, and another referring to the *touched* state is a very common use, the `button` function accepts two optional parameters `img` and `touchImg`. If specified, the `button` class will use these images directly before delivering `touch` and `release` events. The following examples shows how this reduces the amount of code.

Example 34: tileButton2.js

```
var logo = image( {
    url: 'boundaryDevices.png' }
);
var invert = logo.rotate90();
invert = invert.rotate90();

var buttons = new Array();
var w = logo.width ;
var h = logo.height ;
for( var xPos = 0 ; xPos < screen.width; xPos += w )
{
    for( var yPos = 0; yPos < screen.height ; yPos += h )
    {
        var b = button( {
            x: xPos, y: yPos,
            width: w,
            height: h,
            img: logo,
            touchImg: invert } );
        buttons.push( b );
    }
}
```

### 4.14.3 text buttons

Although they're not very pretty, sometimes having a set of text buttons is useful. The `button` function accepts a set of six parameters for generating rectangular text buttons.

|                          |   |
|--------------------------|---|
| <code>borderWidth</code> | Defines the edge width in pixels. Text buttons are drawn with a highlight along the top and left edges, to give a 3D appearance with a light source in the northwest corner of the screen. Since the edges are not joined in a graceful manner, choose a small number here. |
| <code>text</code>        | Text label of the button.   |
| <code>font</code>        | Font to use when rendering text.  |
| <code>pointSize</code>   | Point size of the label text. Choosing a value of 1/2 the height of the button generally gives good results. Be careful about clipping the width, though.   |
| <code>bgColor</code>     | Background color of the button as an RGB triplet.   |
| <code>textColor</code>   | Color of the label text.  |

Example 35: textButtons.js

```

var screenHeight = screen.height ; // 240 ;
var screenWidth  = screen.width  ; // 320 ;
var bg = 0xc0c0c0 ;
var buttonBack = 0x828FA3 ;
screen.rect( 0, 0, screenWidth, screenHeight, bg );

var font = font( { url: "Vera.ttf" } );

var numButtons = 6 ;
var margin     = 2 ;
var buttonHeight = Math.floor(screenHeight / numButtons) - margin ;
var buttonWidth  = Math.floor( screenWidth / 3 ) ;
var pointSize   = buttonHeight/2 ;
var buttons = new Array();
var yPos = 0 ;

function release( button )
{
  screen.rect( buttonWidth+margin, 0,
              screenWidth-1, screenHeight, bg );
  var alpha = font.render( pointSize, button.text );
  var xPos = (screenWidth-buttonWidth)
            -
            (alpha.width >> 1 );
  alpha.draw( xPos, screenHeight>>1, 0 );
}

var bParams = {
  x:0, y:0,
  width: buttonWidth,
  height: buttonHeight,
  borderWidth: 1,
  text: '',
  font: font,
  pointSize: pointSize,
  bgColor: buttonBack,
  textColor: 0,
  onRelease: 'release(this)'
};

for( var b = 0 ; b < numButtons - 1 ; b++ )
{
  bParams.text = 'button ' + (b+1);
  buttons.push( button( bParams ) );
  bParams.y += buttonHeight + margin ;
}

bParams.text = 'Exit' ;
bParams.onRelease = 'screen.clear(0); exit();' ;

```

```
buttons.push( button( bParams ) );
```

The example also shows that the parameter object for things like buttons and URL-supporting classes may be pre-defined and altered between calls.

## 4.15 Webcam input

The Boundary Devices Javascript package contains support for the Logitech® QuickCam driver. Refer to [SourceForge](#) for details on the driver (it is included in the Boundary Devices kernel patches). The Javascript support uses only the standard [Video For Linux](#) interface calls, so it should work with other cameras as well.

The following shows how the Camera class can be used to display video onto the screen.

Example 36: camera.js

```
screen.clear( 0 );

function saveJPEG( cam )
{
    var jpegString = imgToJPEG( cam.grab() );
    if( jpegString )
    {
        writeFile( "/tmp/camera.jpg", jpegString );
        print( "wrote ", jpegString.length, " bytes\n" );
    }
}

var cam = new Camera( "/dev/video0" );

if( 'undefined' != typeof( cam.type ) )
{
    var xPos = ( screen.width > cam.width )
                ? (screen.width-cam.width) >> 1
                : 0 ;
    var yPos = ( screen.height > cam.height )
                ? (screen.height-cam.height) >> 1
                : 0 ;

    cam.display( xPos, yPos, cam.width, cam.height );
    onRelease( "saveJPEG( cam )" );
}
```

The `Camera` class contains a number of properties that you can query:

|                        |   |
|------------------------|---|
| <code>device</code>    | string. Normally <code>"/dev/video0"</code>           |
| <code>name</code>      | string. Normally <code>"Logitech QuickCam USB"</code> |
| <code>type</code>      | number. Identifies the model number.                  |
| <code>minWidth</code>  | minimum display width in pixels.                      |
| <code>maxWidth</code>  | maximum display width in pixels.                      |
| <code>minHeight</code> | minimum display height in pixels.                     |
| <code>maxHeight</code> | maximum display height in pixels.                     |
| <code>displayX</code>  | Horizontal display offset in pixels.                  |
| <code>displayY</code>  | Vertical display offset in pixels.                    |
| <code>width</code>     | Current width of camera input in pixels.              |
| <code>height</code>    | Current height of camera input in pixels.             |
| <code>depth</code>     | Color depth in bits per pixel.                        |
| <code>color</code>     | Boolean. True if color is supported.                  |

The `display` and `capture` methods start streaming capture of data with and without display respectively. Each of them takes parameters of `x, y, w, h`. The `stop()` method can be used to stop capture. The `grab()` method can be used to grab an image from the display (returned as a 16-bit image as with the `image()` call described earlier).

You may be asking yourself why you would want to capture without display. The reason is that the Quickcam driver auto-adjusts the brightness (*aperture?*) of the camera during the read process.

Also shown in the example above is the `imageToJPEG()` routine, which encodes an `image` object using the JPEG compression method, returning a `String`. The `writeFile()` routine saves it to file.

## 4.16 Image manipulation

The `image` class has a number of useful methods for manipulating and querying image data.

|                       |   |
|-----------------------|---|
| <code>draw</code>     | Display an image on the screen. Parameters are <code>x,y</code> .   |
| <code>dither</code>   | Dither an image to gray scale using the Floyd-Steinberg dithering algorithm. Returns an alpha map where the opacity indicates the darkness level.   |
| <code>scale</code>    | Scale all or part of an image to a specified size. Parameters are <code>desiredWidth</code> , <code>desiredHeight</code> , <code>srcLeft</code> , <code>srcTop</code> , <code>srcWidth</code> , <code>srcHeight</code> expressed in pixels. All of the parameters except <code>desiredWidth</code> and <code>desiredHeight</code> are optional. |
| <code>rotate90</code> | Returns an image rotated 90 degrees counter-clockwise.  |
| <code>getPixel</code> | Returns the RGB value of the specified pixel. Parameters are <code>x,y</code> .   |
| <code>setPixel</code> | Sets the RGB value of the specified pixel. Parameters are <code>x,y,rgb</code> .  |
| <code>line</code>     | See <code>screen.line()</code> .  |
| <code>rect</code>     | See <code>screen.rect()</code> .  |
| <code>box</code>      | See <code>screen.box()</code> .   |

To illustrate, I'll use a variation of the `rotColors.js` example from earlier. This time, I'll alter the pixels in the image instead of on the screen, invert the image, and scale it to the size of the display.

Example 37: `imageMan.js`

```
// rotate colors
function transform(rgb)
{
    var colors = [
        rgb>>16,          // red
        (rgb>>8) & 0xff,  // green
        rgb & 0xff ];    // blue
    return colors[1] << 16
        |colors[2] << 8
        |colors[0];
}

var logo = image( {
    url: 'boundaryDevices.png' }
);
logo.draw(0,0);
```

```
logo = logo.rotate90().rotate90(); // flip 180

for( var x = 0 ; x < logo.width ; x++ )
{
  for( var y = 0 ; y < logo.height ; y++ )
  {
    var rgb = logo.getPixel( x, y );
    logo.setPixel(
      x, y,
      transform(rgb) );
  }
}

logo.scale(screen.width,screen.height).draw(0,0);

exit(0);
```

It may not be apparent in the example above, but the `scale()` method does not modify the object on which it is called. Rather, it returns a new image with the desired width and height.

## 4.17 Printer support

### 4.17.1 CBM and Star printers

Define interfaces for CBM and Star printers

### 4.17.2 usblp

The `usblp` Javascript class allows an application to communicate with a device attached through the USB *line printer* (`usblp`) device driver. The `usblp` kernel driver act in many ways like a serial port, in that it provides a bidirectional byte stream over a BULK-IN and a BULK-OUT endpoint.

It's primary interfaces are the same as `serialPort`:

- `read()`
- `write()`

The interfaces to these are the same as with `serialPort`. `read()` returns a string or `false`. `write()` accepts a string parameter.

Other methods are as follows:

| Method                          | Description   |
|---------------------------------|---|
| <code>isOpen()]]</code>         | returns <code>true]]</code> if the device was opened successfully.  |
| <code>startLog(fileName)</code> | This method allows logging of output data to a printer. It is useful for capturing print data as a <code>.ps</code> file for later rendering into PDF as well as as a debugging aid. After a successful call to this method, the <code>usb1p</code> object will |
| <code>stopLog()</code>          | Stops logging output to a file.   |

The following is the proverbial "Hello, world" example.

Example 38: `hello_usb1p.js`

```
var lp = new usb1p();
if( !lp || !lp.isOpen() ){
    throw( "Error connecting to printer" );
}

lp.write( 'Hello, printer\x0c' ); // '\x0c' is a form-feed
```

To read data from the printer, you'll normally want to install an `onLineIn()` method. If defined, this method will be invoked whenever data arrives from the printer, and the method will need to read until no more data is returned from `read()`.

Example 38: `usb1p_listen.js`

```
lp.onDataIn = function(){
    var s ;
    while( false !== ( s = this.read() ) ){
        print( "usb1p_rx<", escape(s), ">\n" );
    }
}
```

### 4.17.3 Postscript support

When used with a printer that supports the Adobe **Postscript** language, most **Postscript** output is generated directly by Javascript through the `usb1p.write()` method. Since **Postscript** is a textual language, this works well for text, lines, cursor positioning and the like.

`usb1p.write()` doesn't work well for images, however. Various interpreters differ in their image-handling abilities, but most **Postscript** printers

support at least `zlib` and `JPEG` compression. Each of these forms still requires a bit of processing to put them in the right form, however.

The method `usblp.imageToPS(imgData)` is used to encode and output the data from an image. The `imgData` parameter should be the encoded content of an image file in either the `PNG` or `JPEG` form. If the image data is determined to be `JPEG`, it is translated almost directly into the equivalent `Postscript`. If not, it is decoded and re-coded using `zlib` compression. `JPEG` is faster, although it is lossy and may generate bigger output.

Example 38: `usblp_psimage.js`

```
var lp = new usblp();
if( !lp || !lp.isOpen() ){
    throw( "Error connecting to printer" );
}

var img = curlFile( { url: 'file:///tmp/myImg.jpg' } );
if( !img.isLoaded )
    throw( "Error loading myImg.jpg" );

var info = imageInfo( img.data );
if( false == info )
    throw( "Unknown image format" );

lp.imageToPS( {
    image: img.data,
    x: 0, y:0,
    w:info.width,
    h:info.height } );

lp.write( "showpage\n" );
```

Note that the `imageToPS()` routine takes an anonymous object as input to specify not only the encoded image data, but also the image location and scale. The location and scale are in `Postscript` coordinates (1/72 inches) where the lower left corner of the page is `[0,0]`.

Also note that the image location defines the position of the bottom left corner of the image, so the image in the example will be placed at the bottom of the page.

The `imageInfo()` routine above is a utility routine to perform a quick check for the image size of a known image format (`PNG` or `JPEG`). Since

this information is present in the header of each of these file formats, processing each pixel is not needed. The object returned from `imageInfo()` contains the width and height in pixels, so the image will be scaled up to 1 pixel==1/72 inches (612 pixels for a letter-sized page).

#### 4.17.4 Swecoin graphics support

| Method                         | Description                  |
|--------------------------------|------------------------------|
| <code>bitmapToSwecoin()</code> | <a href="#">Document me.</a> |

#### 4.17.5 Quick image parsing

When rendering images into Postscript, it is not necessary or helpful to use the `image` Javascript class to decode the image. If you know beforehand the image size, you can render an image directly from `file` or `curlFile` into the corresponding Postscript.

If you don't know the image size, you may be tempted to use the `image()` class to decode the image so you can put a nice border around the image or `somesuch`.

Don't do it. It's slow and wasteful.

Use the `imageInfo()` routine instead. If it understands the image type (GIF, PNG, or JPEG), it will return an object with `width` and `height` methods as shown in the following example.

## 4.18 Networking with TCP

The `tcpClient` class allows connection to a server process using the TCP protocol. This is useful both for connecting to native processes on the same machine as well as for communication with other machines.

The `tcpClient` class is defined in the C++ module `bdScript/jsTCP.cpp`.

A number of key design choices were made to interface nicely to Javascript:

- Input is line-oriented, providing a callback whenever more lines are available.
- Both send and receive interfaces use Javascript strings.
- Uses an anonymous object to specify constructor parameters.

A number of other design choices were made for expediency, and may change in the future:

- Only dotted-decimal notation for the server is allowed.
- Connection is synchronous (i.e. done within the constructor)
- `onLineIn` and `onClose` callbacks are treated as global code, and do not contain a reference to the object for whom the callback is made.

The general form of the `tcpClient` constructor is like so:

```
var mySock = new tcpClient( {
    serverPort: 80,
    ,serverIP:   "127.0.0.1"
    ,onLineIn:  "myLineInHandler()"
    ,onClose:   "myCloseHandler()"
} );
```

Note the use of the anonymous object for the parameter set. All parameters are optional except `serverPort`, which must be the TCP port number. Upon construction, the `tcpClient` object performs a TCP connect. The `.isConnected` member variable can be tested to see if the connection was established. The following code snippet illustrates how it may be used.

```
var mySock = new tcpClient({serverPort:8080});
if( !mySock.isConnected )
    print( "connection failed\n" );
```

I'm sure this is no surprise, but the `send()` method is used to send data to peer over a TCP connection. It expects a single, `String` parameter that

is sent without any preamble or trailer. The parameter may or may not contain delimiters and such. Also note that no conversions of line ends and whatnot are performed.

```
mySock.send( "Hello, world\n" );
```

As mentioned above, input over a `tcpClient` connection *is* line-oriented. The `tcpClient` class will internally queue any non-empty set of characters which are not either the carriage-return `'\r'` or line-feed `'\n'` for consumption by the Javascript application. It will *not queue empty lines* to allow lines ending in a carriage-return/line-feed pair to be seen as a single line. Data is read from the queue by a Javascript application through the `readln()` method. If no data is available to be read, `readln()` will return `false`. It is suggested that you use the strict comparison operators `===` or `!==` to prevent type conversion of the input string.

```
var s ;
while( false !== ( s = mySock.readln() ) )
{
    print( "rx: <", s, ">\n" );
}
```

### 4.18.1 A first example: HTTP HEAD

The following example illustrates how the `tcpClient` class may be used to implement the HTTP HEAD method.

```

httpHead.js
var dump = use({url:'dump.js'});
var connect = null ;
var headers = null ;

function headClose() {
    dump.dumpObj( 'headers', headers );
    exit();
}

function headLineIn() {
    var s ;
    while( false != ( s = connect.readLine() ) ) {
        var colonPos = s.indexOf(':');
        if( 0 > colonPos )
        {
            var parts = /HTTP\/(\d+)\.(\d+)\s+(\d+)\s+(.*)/.exec(s);
            if( parts ) {
                headers.httpVer    = parts.slice(1,3).join('.');
                headers.status     = parseInt(parts[3]);
                headers.statusMsg  = parts[4];
            }
            else {
                if (!header.others)
                    header.others = new Array();
                header.others.push(s);
            }
        }
        else
            headers[s.substr(0,colonPos)] = s.substr(colonPos+2);
    }
}

if( 1 == argc ) {
    var url = argv[0];
    if( 'http://' == url.substr(0,7) )
        url = url.substr(7);
    var parts = url.split('/');
    var hostPort = parts[0];
    var hpParts = hostPort.split(':');
    var port = 80;
    if( hpParts.length == 2 )
        port = parseInt(hpParts[1]);
    var host = hpParts[0];
    var file = '/' + parts.slice(1).join('/');
    headers = new Object();
    connect = new tcpClient( {
        serverPort: port,
        serverIP: host,
        onLineIn:"headLineIn()",
        onClose: "headClose()"
    } );

    if( connect.isConnected ) {
        var sent = connect.send( "HEAD " + file + " HTTP/1.0\r\n\r\n" );
    }
    else {
        print( "connect failed\n" );
        exit();
    }
}
else {
    print( "Usage: httpHead.js url\n" );
    exit();
}

```

This application creates an object with each of the HTTP headers as members, then uses the `dump.dumpObj()` method to display the results (see section 6.1).

Note the use of the `connect` global variable (the connection is not inherently available to the handlers).

You may be asking yourself why the example above used the `tcpClient` class for something dealing with a Web Server. Isn't that what `curlFile` is used for? I'm glad you asked.

While `curlFile` provides a convenient mechanism to controlled file transfer of HTTP objects, and a good basis for Remote Procedure Call, there are a few advantages to the use of `tcpClient`.

1. **Connection control** - `curlFile` will cache connections to a server, but the lifetime of the connections is unknown to the Javascript application.
2. **Intermediate results** - `curlFile` does not allow results to be retrieved from a Javascript application prior to the end of the HTTP transaction. Because applications using `tcpClient` pull data (lines) from the server, intermediate results can be acted on before the end. This allows `tcpClient` to implement a robust form of HTTP streaming (see [this article](#) for a browser equivalent of this web server pattern).
3. **Limit memory usage** - `curlFile` requires that memory be allocated for all of an HTTP request. `tcpClient` only buffers unread lines.

### 4.18.2 Another example: Streaming Web Data

The following sample PHP script and Javascript applet illustrate the use of `tcpClient` for continuous input from a Web Server application. The `timeOut.php` script simply outputs a formatted date and time once every second. Note the use of the `<PRE>` tag in each to allow the HTTP headers to be skipped in the Javascript case, while simultaneously allowing the script to be viewed in a browser.

```

timeOut.php
<PRE>
<?
  while (true) {
    echo date('Y-m-d H:i:s') . "\n";
    flush();
    sleep(1);
  }
?>

```

```

httpRead.js
var connect = null ;
var reading = false ;

function headClose() {
  exit();
}

function headLineIn() {
  var s ;
  while( false !== ( s = connect.readLine() ) ) {
    if( !reading )
      reading = ( '<PRE>' == s );
    else
      print( "rx:", s, "\n" );
  }
}

if( 1 == argc ) {
  var url = argv[0];
  if( 'http://' == url.substr(0,7) )
    url = url.substr(7);
  var parts = url.split('/');
  var hostPort = parts[0];
  var hpParts = hostPort.split(':');
  var port = 80;
  if( hpParts.length == 2 )
    port = parseInt(hpParts[1]);
  var host = hpParts[0];
  var file = '/' + parts.slice(1).join('/');
  connect = new tcpClient( {
    serverPort: port,
    serverIP: host,
    onLineIn: "headLineIn()",
    onClose: "headClose()"
  } );

  if( connect.isConnected ) {
    var sent = connect.send( "GET " + file + " HTTP/1.0\r\n\r\n" );
  }
  else {
    print( "connect failed\n" );
    exit();
  }
}
else {
  print( "Usage: httpRead.js url\n" );
  exit();
}

```

Who says Web servers can't push data?

### 4.19 Audio input

The `recordBuffer` Javascript class allows simple access to microphone input on a Boundary Devices™ board. Usage normally involves four steps:

1. Declare a `recordBuffer` variable to hold the input samples.
2. Start recording through `recordBuffer.record()`.
3. Stop recording using `recordBuffer.stop()`.

Construction of a `recordBuffer` also involves an anonymous object like this:

```
var rb = recordBuffer( {sampleRate:11025, maxSeconds:10.0} );
```

The `sampleRate` member is specified in Hertz. The `maxSeconds` variable declares the maximum size of the input buffer. Samples are always 16-bit PCM in little-endian byte order.

The `record()` method begins recording into the start of the buffer. It takes an anonymous object as input to allow specification of two handlers.

```
rb.record( {  
  onComplete: "endRecord();",  
  onCancel:   "cancelRecord();"  
} );
```

Each of these handlers executes as code in the global context, so the `recordBuffer` variable must be a well-known global.

The `stop()` method cancels recording, calling the `onCancel` method if specified in the call to `record()`.

After recording, `recordBuffer` objects contain a `.data String` member variable that contains the actual PCM audio samples. They also contain a `playback()` method which allows immediate playback of the captured audio.

The following small script shows how all of this can be used together:

```

----- testMic.js -----
var rb ;

function endPlayback(){ exit(); }

function endRecord() {
  print( rb.secsRecorded, " seconds recorded\n" );
  rb.playback( {onComplete:"endPlayback()"} );
}

if( 1 <= argc ) {
  var secs = parseInt( argv[0] );
  if( 0 < secs )
  {
    rb = recordBuffer( { sampleRate:11025, maxSeconds:secs } );
    rb.record( { onComplete:"endRecord()", onCancel:"endRecord();" } );
  }
  else
    exit();
} else {
  print( "Usage: testMic.js #seconds\n" );
  exit();
}

```

This application simply records a buffer specified in seconds on the command-line, then plays it back and exits.

Note the use of the global variable `rb` and the altered meaning of the `onComplete` member in `rb.record()` versus `rb.playback()`.

This is pretty fun, but this application still amounts to a toy. This application model *can* be used by real applications as well, though with a somewhat limited scope.

The following application shows that you can access the data as well.

```

----- micSamples.js -----
var rb ;

function endRecord() {
  print( rb.secsRecorded, " seconds recorded\n",
        "md5 == ", md5(rb.data), "\n" );
  exit();
}

if( 1 <= argc ) {
  var secs = parseInt( argv[0] );
  if( 0 < secs )
  {
    rb = recordBuffer( { sampleRate:11025, maxSeconds:secs } );
    rb.record( { onComplete:"endRecord();" } );
  }
  else
    exit();
} else {
  print( "Usage: micSamples.js #seconds\n" );
  exit();
}

```

So what, you say? Yes, you can grab the raw samples. What might not be clear is that if you can grab them, you can send them somewhere. To a file through the `writeFile()` method. To a server using either `curlFile` or `tcpClient`. Choices abound.

The key design note is that this is *not streaming audio*. This is a *record and send* model. Think walkie-talkie instead of telephone and you'll be on track.

I'll wrap up the audio input section with somewhat more of a real application. The following example uses the `tcpClient` class to post data to a web server.

The following tiny script reads and produces an MD5 hash of the input.

```
'md5Post.php'
<PRE>
<? echo "server: " . md5( $HTTP_RAW_POST_DATA ) . "\n" ; ?>
```

The following Javascript records some number of seconds of audio, produces a local MD5 hash, then sends it in an HTTP POST to the specified server, displaying the response.

```
postAudio.js
var rb ;
var connect ;
var sent = false ;

function postClose(){ exit(); }
function postLineIn() {
  var s ;
  while( false !== ( s = connect.readln() ) ) {
    if( reading ) print( s, "\n" );
    reading = reading || ( '<PRE>' == s );
  }
}

function endRecord() {
  print( rb.secsRecorded, " seconds recorded\n",
        "local md5 == ", md5(rb.data), "\n" );
  connect.send( "Content-type: application/octet-stream\r\n"
              + "Content-Length: " + rb.data.length + "\r\n"
              + "\r\n" );
  connect.send( rb.data );
}

if( 2 <= argc ) {
  var secs = parseInt( argv[0] );
  if( 0 < secs ) {
    var url = argv[1];
    if( 'http://' == url.substr(0,7) )
      url = url.substr(7);
    var parts = url.split('/');
    var hostPort = parts[0];
    var hpParts = hostPort.split(':');
    var host = hpParts[0];
    var port = 80;
    if( hpParts.length == 2 )
      port = parseInt(hpParts[1]);
    var file = '/' + parts.slice(1).join('/');
    connect = new tcpClient( {
      serverPort: port,
      serverIP: host,
      onLineIn:"postLineIn()",
      onClose: "postClose()"
    } );

    if( connect.isConnected ) {
      sent = connect.send( "POST " + file + " HTTP/1.0\r\n" );
      rb = recordBuffer( { sampleRate:11025, maxSeconds:secs } );
      rb.record( { onComplete:"endRecord();" } );
    }
  }
}

if( !sent ) exit();
```

I had to omit some error checking to keep this on a single page, but I think this shows that a useful application *can* be built using these parts. Pipe this data into the [LAME](#) MP3 encoder and you can play them back elsewhere. Pull the raw samples into [Audacity](#) and you can convert to most any sample format.

## 4.20 Filesystem access

The Javascript application supports a couple of different methods of accessing the file-system. The first and easiest are direct file read and write routines `readFile()` and `writeFile()`.

### 4.20.1 Read/Write complete files

The `readFile()` routine takes a path as input and returns either a string with the complete file content or `false` to indicate failure.

```
myVar = readFile( '/mmc/myFile.in' );
if( myVar ){
    print( "read ", myVar.length, " bytes\n" );
}
```

In a similar fashion, the `writeFile()` routine accepts a path and string value as input, and writes or creates a complete file with the content. It returns `true` to indicate success or `false` to indicate failure.

```
if( writeFile( '/mmc/myFile.out',
              'data data data...\n' ) ){
    print( "file written successfully\n" );
}
```

### 4.20.2 Device driver and pipe access

The Javascript `file` class allows access to character device drivers or pipes in a manner similar to the `serialPort` class.

A method `onCharIn()` must be supplied by the Javascript application and will be called whenever data arrives. The `file` class provides `read()` and `write()` methods to allow the application to read input and send output to the device driver.

```
var myDev = file( '/dev/ttyS0' );
myDev.onCharIn = function()
{
    var sInput ;
    while( false !== ( sInput = this.read() ) ){
        print( "rx: ", sInput, "\n" );
        this.write( 'got it\n' );
    }
}
```

### 4.20.3 Watchdog support

The global `wdEnable()` routine can be used to enable the platform-specific watchdog timer. Once enabled, the `jsExec` program will keep the watchdog alive as long as it is operating its' main loop.

If configured to include it, the Linux kernel provides a device named `/dev/watchdog` that requires a `write()` every so often, or will perform a hard reset of the machine.

Another way to accomplish this is to use the `file` class, and write to the file handle from Javascript code. The benefit of this approach is that the Javascript code can use additional logic to determine if the machine is operating properly. The downside is that the additional logic may break down, and you may end up with hard resets where a more graceful fix is possible.



## 5 Class and Function Reference

5.1 'library' and 'use'

5.2 alphaMap

5.3 barcodeReader

5.4 bitmap

5.5 button

5.6 Camera

5.7 CBM

5.8 childProcess

5.9 curlFile

5.10 dir

5.11 environ, getenv and setenv

5.12 exit

5.13 garbageCollect

5.14 gotoURL

5.15 fade

5.16 file

5.17 FileSys

5.18 flashMovie

5.19 flashVariable

5.20 font

5.21 image

5.22 Kernel

5.23 monitorWLAN

5.24 mp3Cancel

5.25 mp3File

5.26 mpegFile

5.27 nanosleep

5.28 pinger

5.29 printer

55

5.30 queueCode

**5.31 recordBuffer****5.32 Screen**

Describe the Screen class, data and methods.

**5.33 serialPort****5.34 starStatus****5.35 starUSB****5.36 stat****5.37 tcpClient****5.38 touchScreen****5.39 TTY****5.40 udpSocket****5.41 usblp****5.42 use and modularity****5.43 waitFor****5.44 waveFile****6 Useful Javascript modules**

## 6.1 dump.js

The following script is used in various sample scripts through the `use` method. It provides pretty-printed output of Javascript variables, including objects and arrays.

```
function indent( depth ) {
  for( var i = 0 ; i < depth ; i++ )
    print( '  ' );
}

function dumpObj( name, obj, depth ) // depth is optional
{
  if( 2 == arguments.length )
    depth = 0 ;
  if( 2 <= arguments.length ) {
    if( 0 < name.length ){
      if( 0 == depth )
        print( name, ' = ' );
      else
        print( name, " : " );
    }
    switch( typeof( obj ) ) {
      case 'array' : {
        print( '[\n' );
        depth++ ;
        for( var property in obj ){
          indent( depth );
          if( 0 != property )
            print( ', ' );
          dumpObj( ' ', obj[property], depth );
        }
        depth-- ;

        indent( depth );
        print( ']\n' );
        break ;
      }
      case 'object' : {
        print( '{\n' );

        depth++ ;
        var first = true ;
        for( var property in obj ) {
          indent( depth );
          if( !first )
            print( ', ' );
          else
            first = false ;

          dumpObj( property, obj[property], depth );
        }
        depth-- ;

        indent( depth );
        if( 0 == depth )
          print( '};\n' );
        else
          print( '}\n' );
        break ;
      }
      default: {
        if( obj ) {
          var s = obj.toString();
          if( s.length < 256 )
            print( s, '\n' );
          else
            print( s, 'length ', s.length, '\n' );
        }
        else
          print( 'null\n' );
      }
    }
  }
  else
    print( 'usage: dumpObj( "name", object )\n' );
}
```

## 6.2 **staticText.js**