

BeIA JavaScript

This document is confidential and may not be distributed without the permission of Be Inc.

copyright © 2000 Be Incorporated

Bela JavaScript

Table of Contents

beos	3
1 Properties	3
binder	3
globals	3
2 Functions	3
dprintf()	4
exec()	4
sendMessage(), sendMessageSync(), reply[]	4
beos.exec	7
1 Functions	7
beos.binder	9
1 Properties	9
application	9
service	9
user	10
2 Functions	10
observe(), observeContents()	10
valueOf()	11

BeIA JavaScript

beos

Genus: JavaScript object

Description: Entry point for Be-defined JavaScript elements

`beos` is a global Javascript object that's the entry-point for all Be-defined JavaScript elements. You can access `beos` from any HTML page within your BeIA configuration. There's only one instance of the `beos` object; it's created by BeIA when the system is booted, and destroyed when the system is halted.

1 Properties

binder

- Object that represents the root of the Binder tree.

`binder`

The `binder` property represents the root of the Binder tree. See the argument *text* to standard output. You use this function while you're debugging your code. You should remove (or disable) all `dprintf` calls before you ship.

globals

- Miscellaneous storage object

`globals.field`

The `globals` property is a BMessage that's persistent and global to the browser. Provided as a convenience, you can add whatever fields you want to the property. The property contains no Be-defined fields.

2 Functions

All arguments to the `beos` functions are strings. To pass a number, simply put it in double quotes—the conversion from string to numeric value will be done for you.

dprintf()

- Prints to standard output

```
dprintf ( text )
```

`dprintf` prints the argument *text* to standard output. You use this function while you're debugging your code. You should remove (or disable) all `dprintf` calls before you ship.

exec()

- Executes an application or program

```
exec ( pathname, arg1, arg2, arg3, ... )
```

- Closes the application's input and output streams

```
exec().close ( )
```

- Kills the launched application

```
exec().kill ( )
```

- Reads and returns a line from standard input

```
exec().readln ( )
```

- Waits for the application to die and returns its exit code

```
exec().wait ( )
```

- Reads and returns a line from standard input

```
exec().write ( string )
```

`exec()` executes the application (binary) located at *pathname*, passing it as many as 63 arguments. The *pathname* must be an absolute path. The function returns an object that can be used to control the application, through the following functions:

- `exec().close()` closes the application's input and output streams. This is useful for applications that block on the input stream; you have to call `close()` before the application will exit.
- `exec().kill()` abruptly terminates the application. You should only `kill()` an application as a last resort. The function doesn't return a reliable exit code; if you need an exit code, call `wait()` instead.
- `exec().readln()` reads a line from the application's standard input stream and returns it as a string.
- `exec().wait()` waits for the application to die and then returns the application's exit code. Note that `wait()` doesn't induce the application to die, it simply waits for it to die naturally. If the application is already dead, the exit code may be invalid.
- `exec().write()` writes *string* to the application's standard input stream.

sendMessage(), sendMessageSync(), reply[]

- Sends an asynchronous message

```
sendMessage ( recipient, command,  
              name1, type1, value1,  
              name2, type2, value2,
```

```
name3, type3, value3
... )
```

- Sends a synchronous message

```
sendMessageSync ( recipient, command,
                  name1, type1, value1,
                  name2, type2, value2,
                  name3, type3, value3
                  ... )
```

- Reads a value from a message field

```
reply [ field ]
```

`sendMessage()`/`sendMessageSync()` sends a message to a running application or server. The arguments are:

- *recipient* is the **application signature** of the app or server you want to send the message to. For example, to send a message to the Sound Server, you would use “application/x-vnd.be-snd_server”.
- *command* is a four-character **command constant** that symbolizes the type (or intent) of the message you’re sending. The recipient must understand what the symbol means; the string is otherwise arbitrary. For example, the Sound Server understands “MAVS” to mean “set the volume.”
- Each *nameN*, *valueTypeN*, and *valueN* triplet adds a single data field to the message. *nameN* is the field name, *typeN* is its type (“string”, “float”, or “int32”), and *valueN* is its value. The number and type of fields that are expected and how their values are applied is determined by the message’s receiver. To set the volume of the built-in speaker, for example, you would add the “be:speaker.volume” field to the message:

```
beos.sendMessage( "application/x-vnd.be-snd_server", "MAVS",
                  "be:speaker.volume", "float", 0.5 );
```

All messages must have *recipient* and *command* arguments. Some messages don’t require the additional data fields (*nameN/typeN/valueN*).

`sendMessage()` sends the message and returns immediately (it doesn’t wait for a reply).

`sendMessageSync()` doesn’t return until a reply is sent back. To read the reply message, look at `beos.reply[]` immediately after calling `sendMessageSync()`.

The `reply[]` function lets you access a *field* of the message that’s sent back to your code. You use `reply[]` immediately after calling `sendMessageSync()`. For example, if you want to retrieve the volume of the internal speaker, you would send a volume-querying message to the Sound Server, and then immediately examine the “be:speaker_volume” field by accessing it through `reply[]`:

```
<!--Ask for the volume settings. -->
beos.sendMessageSync( "application/x-vnd.be-snd_server", "MAVG" );
<!--Look at the speaker setting. -->
beos.reply[ "be:speaker_volume" ];
```


BeIA JavaScript

beos.exec

Genus: JavaScript object
Description: Executes an application or program

```
exec ( pathname, arg1, arg2, arg3, ... )
```

`beos.exec()` executes the application (binary) located at *pathname*, passing it as many as 63 arguments. The *pathname* must be an absolute path. The function returns an object that can be used to control the application, through the following functions:

1 Functions

close()

:: Closes the standard input stream

```
close( )
```

`close()` closes the application's standard input stream. If your application is waiting for input from the stream, you have to call `close()` before the application will exit.

kill()

:: Kills the launched application

```
kill( )
```

`kill()` abruptly terminates the application. You should only `kill()` an application as a last resort. The function doesn't return a reliable exit code; if you need an exit code, call `wait()` instead.

readln()

```
readln( )
```

`readln()` reads a line from the application's standard input stream and returns it as a string.

wait()

```
wait( )
```

`wait()` waits for the application to die and returns the application's exit code. The exit code is legitimate, even if the application has already died when you call `wait()`. Note that `wait()` doesn't induce the application to die, it simply waits for it to die naturally. Keep in mind that for applications that read from standard input, you may need to call `close()` first.

write()

```
write( "input" )
```

`write()` writes *input* to the application standard input.

Bela JavaScript

beos.binder

Genus: JavaScript object

Description: Represents the root of the Binder tree

`beos.binder` represents the root node of the Binder tree. All other nodes in the tree are given, in JavaScript, as properties of `beos.binder`. For example, the `user` node is represented by the `beos.binder.user` object.

This document lists the top level `beos.binder` properties. For more information on the Binder tree (and, thus, the JavaScript objects that represent the Binder nodes and properties), see the *The Binder* chapter.

1 Properties

application

- Used by applications to store data

The `application` node is used by applications (or plug-ins) to store app-specific data. An application registers itself within `application` by creating a node that's named for the application's MIME signature. For example, this node...

```
beos.binder.application.x-vnd-MoviePlayer
```

...would contain data for the (fictitious) plug-in with the MIME signature "application/x-vnd-MoviePlayer".

It's anticipated that new `application` nodes will only be created programmatically (through C++), when a new application or plug-in is first launched.

service

- System wide services and settings

The `service` node provides system-wide information (device ID numbers, network configuration information, etc.), and software capabilities (lists of supported languages, printers, fonts, etc.). As examples, the ISP's telephone number is given in the `service.network` node; the various resolutions that a printer can handle for a given page are listed in the `service.printing` node.

user

- User account settings

The `user` node contains a node for every user account, each of which contains user-specific settings and data—language, locale, bookmarks, email info, and so on. The XML template for new user accounts is stored in `/boot/home/config/settings/binder/user-record.skel`. When a new user account is added to the system, the `user-record.skel` definition is automatically copied into the account.

2 Functions

Unless otherwise noted, the following JavaScript functions can be invoked on any Binder property or node.

`observe()`, `observeContents()`

- Monitor changes to a property
`observe (callbackFunction , userData)`
- Monitor changes to the properties in a node
`observeContents (callbackFunction , userData)`
- Binder observer callback function syntax

```
function callbackFunction ( userData ,
                             nodeObject ,
                             propertyName ,
                             eventCode )
```
- Binder observer event codes
`"added"` , `"changed"` , `"removed"` , `"unknown"`

`observe()/observeContents()` registers a callback function that's invoked when an observed property changes. `observe()` is used to monitor (and is invoked upon) a specific property; `observeContents()` monitors changes to any of the properties in the invoked-upon node. The first argument names the callback function that will be invoked; the `userData` argument is passed to the callback function when it's invoked.

The arguments to the callback function are:

- `userData` is the data that was passed to the `observe()/observeContents()`
- `nodeObject` is a JavaScript object that represents the node that contains the changed property.
- `propertyName` is the name of the property that changed.
- `eventCode` describes what happened to the property: it was `added`, its value `changed`, it was `removed`, or some other change occurred (`unknown`).

Reporting a change to a property is completely in the hands of the property's node—the system doesn't compel a node to report any changes.

valueOf()

- Returns the value of a property

`valueOf ()`

`valueOf()` returns the value of a property or node.

