GSM Scripting Manual

JavaScript support in iControl GSM builds on core JavaScript (ECMAScript). It further implements E4X (ECMAScript for XML), which makes XML a first-class data type in JavaScript. This document outlines the features that are specific to iControl GSM's implementation. The reader is expected to have a basic understanding of JavaScript. If you need to learn or refresh your memory, a good starting point is http://www.mozilla.org/js/scripting/.

Object hierarchy

- **`navigator`** - Navigator
- **`gsm`** - GSM

Classes

Reference

Navigator

The **Navigator** object represents the GSM application as a whole. There is only one **Navigator** object in the application, and it is accessible as the global variable **navigator**.

Property/Method	Description
appCodeName	Represents the internal code name of the application. Read-only.
appName	Read-only property used to get the name of the application, for instance **"iControl GSM"**.
appVersion	Read-only property used to get the application version string.
gsmAggregator	Read-only property of type **GSM** referencing all the currently discovered GSM services as one.
gsms[]	Array of **GSM** objects indexed by the IP address of the corresponding GSM server. The array also contains an entry for the **gsmAggregator** under the key **"aggregator"**.
language	Read-only property used to get the language of the application.
routers[]	An **Array** of **Router** objects representing the routers that have been discovered on the network, indexed by their router IDs (as **string**s).
securityDomain	Read-only property identifying the security domain path for the current application. Will be **null** if security hasn't been enabled for a site.
vendor	Read-only property used to get the vendor of the application.
connectCrosspoint()	This method switches a single router cross point. It takes four parameters: the router ID (**string**), and the level, source and destination (as **number**s). It is equivalent to **navigator.routers[*routerID*].connectCrosspoint(*level, source, destination*);**
resetAllLatches()	Resets all alarm latches for the application.
getAudioClip()	This method takes a single parameter representing the fully-qualified URL of a sound (WAV, AIFF, AU) file. The returned object can be controlled using its three methods: **play()**, **loop()** and **stop()**. Note that the sound will be played on the *server* running GSM; to play sounds on *client* computers, please refer to the iControlWeb scripting manual.
getService(longId*[,timeout]*)	This method returns a Jini service proxy which matches the supplied **longID**. It is possible that the service needs to be discovered on the network first, which could take some time. If the service can not be found within the allowed time (as specified by the optional **timeout** parameter, which is a number of milliseconds that defaults to 10000 (10 seconds)), **undefined** is returned. If a service is found but the service proxy is incompatible (and therefore unusable), the method will return **null**.
includeJS(script)	This function is used to load an external JavaScript file into the context of the navigator object. By using this method you can create reusable JavaScript libraries. The parameter is an absolute URL to load the script from, as a **string**.

`setTimeout()`	A utility method that schedules a task for execution after a given delay. This method takes two parameters: a task and a timeout delay (in milliseconds) before the task is executed. The task can be one of two things: either a JavaScript `Function` or a script string to be evaluated. In either case, `this` in the script refers to the `Navigator`. One advantage of using a `Function` over a `string` is that the environment where the function was defined remains accessible inside the function (this is technically called a *closure*). The method returns an object that can be saved and used to cancel the task later on using `clearTimeout()`.
`clearTimeout()`	This method can be used to cancel a task that was previously scheduled using `setTimeout()`. It takes a single parameter that must be the object that was returned when the task was initially scheduled.
`setInterval()`	A utility method that schedules a task for execution each time a given time interval has elapsed. This method takes two parameters: a task and an interval delay (in milliseconds) between task executions. The task can be one of two things: either a JavaScript `Function` or a script string to be evaluated. In either case, `this` in the script refers to the `Navigator`. One advantage of using a `Function` over a `string` is that the environment where the function was defined remains accessible inside the function (this is technically called a *closure*). The method returns an object that can be saved and used to cancel the task later on using `clearInterval()`.
`clearInterval()`	This method can be used to cancel a task that was previously scheduled using `setInterval()`. It takes a single parameter that must be the object that was returned when the task was initially scheduled.
`sendEmail()`	This method can be used to send e-mails. It takes up to seven parameters: the SMTP (mail) server host name or IP address, the list of recipient e-mail addresses (comma separated `string`), the sender's e-mail address, the subject, the message body, and optionally the username and password if authentication is required with the SMTP server. *(since 3.01)*

GSM

The `GSM` object represents the GSM service. It allows scripts to interact with the GSM service.
The local GSM service can be referenced using the global `gsm` object.

Property/Method	Description
`snmpPlugins[]`	An array of all the `SNMPManagerPlugin`s indexed by the IP address of the corresponding SNMP device.

addAlarm()	This method creates and publishes a new GSM alarm. The method takes six parameters, all of type **string**: the alarm URI, the alarm name, the alarm path (separated by **/** characters), the device class, the device URI and the type of alarm (**"status"** or **"text"**). Alternately and preferably, the alarm type (last parameter) can be specified as a combination of numeric flags from **GSMAlarm.TYPE_STATUS**, **GSMAlarm.TYPE_TEXT** and **GSMAlarm.TYPE_NOT_LOGGED** using the **|** operator. Refer to the GSM user manual for more information on each of these parameters. The return value is a **GSMAlarm**, which can be used to update the status of the alarm.
getAlarm()	This method takes a single parameter of type **string**, representing the alarm URI of the alarm you need, and it returns a **GSMAlarm** object which contains the current information about the alarm (including its current status). Note that it provides only a snapshot of the alarm, and will not update in real time.

addVirtualAlarm()	This method takes two parameters.
	
	The first one, of type **Object**, describes the parameters of the virtual alarm. This object must provide the necessary fields that describe the new virtual alarm:
	
	• **name** (**string**),
	• **path** (**"/"**-delimited **string**),
	• **mode** (**string**: **"AND"**, **"OR"** or **"XOR"**),
	• **textLogic** (optional **string**):
	  ○ **"null"**: no text logic (default)
	  ○ **"concatenate"**: concatenate texts from sub-alarms
	  ○ **"errors"**: list the sub-alarms that are in error
	• **forIncident** (optional **boolean**)
	  ○ **true**: create as an incident template
	  ○ **false**: create as a regular virtual alarm (default)
	
	The second parameter can be simply an **Array** of alarm URIs (as **string**s) representing the sub-alarms of the virtual alarm, in which case their contribution will be all be **PASSTHROUGH**, or, if you want to specify other contributions, it can be an associative array with the alarm URIs as **string** keys and the contribution as numeric values (from the possible values found in **GSMAlarm**: **PASSTHROUGH**, **INVERT**, **CRITICAL**, **MAJOR**, **MINOR**, **DISABLED**, **FAULTS_ONLY**).

	This method returns a `GSMAlarm` object referring to the newly created (or modified) virtual alarm.
`logEvent()`	This method takes two parameters: a `GSMAlarm` that the event relates to, and a message `string` to attach to the event. Given those informations, it will log an event that can later be seen in the GSM log viewer.
`resetAllLatches()`	Resets all alarm latches for the application.
`setFaultSeverities()`	Defines the severities that will trigger incidents. This method takes as parameter an array of strings representing the statuses. The possible values are defined in `GSMAlarm`. The default "fault" severities are: `MINOR`, `MAJOR`, `CRITICAL`, `UNKNOWN`.
`eval()`	The `eval()` method takes a single `string` parameter representing JavaScript code to be evaluated in the context of the remote GSM. It returns the value returned by the evaluated script element. An example would be: `eval("navigator.appVersion;");`

GSMAlarm

The `GSMAlarm` object encapsulates the concept of both an alarm descriptor and an alarm in a single entity. It can be used to create a new GSM alarm descriptor and manipulate its status over time, or to wrap alarm notifications received from other modules (in which case you won't be able to modify most of the fields).

Property/Method	Description
Constructor	The constructor creates and publishes a new GSM alarm. It takes six parameters, all of type `string`: the alarm URI, the alarm name, the alarm path (separated by `/` characters), the device class, the device URI and the type of alarm (`"status"` or `"text"`). Alternately and preferably, the alarm type (last parameter) can be specified as a combination of flags from `TYPE_STATUS`, `TYPE_TEXT` and `TYPE_NOT_LOGGED`. Refer to the GSM user manual for more information on each of these parameters.
`status`	Read/write dynamic property referencing the current status of the alarm in GSM.
`previousStatus`	Read/write dynamic property referencing the previous status of the alarm in GSM.

latchedStatus	Read-only dynamic property referencing the current latched status of the alarm in GSM.
previousLatchedStatus	Read-only dynamic property referencing the previous latched status of the alarm in GSM.
acknowledgementStatus	Read-only dynamic property referencing the current acknowledgement status of the alarm in GSM.
previousAcknowledgementStatus	Read-only dynamic property referencing the previous acknowledgement status of the alarm in GSM.

operationalMode	Read-only dynamic property referencing the current operational mode(s) of the alarm in GSM (**OFFLINE** or **MAINTENANCE** or a combination of both).
	
	This property is a bit-field, so to check if a given operational mode (say, **OFFLINE**) is set, do something like:
	
	`if (alarm.operationalMode & GSMAlarm.OFFLINE)`
	`{ // Deal with offline alarm here... }`

previousOperationalMode	Read-only dynamic property referencing the previous operational mode(s) of the alarm in GSM (**OFFLINE** or **MAINTENANCE** or a combination of both).
text	The value of the text of the alarm, for a text alarm. This is a read/write property.
uri	This is a read/write property.
name	This is a read/write property.
pathString	This is a read/write property.
deviceClass	A **String** that identifies the type of device this alarm is attached to. This is a read-only property.
deviceURI	A **String** that identifies the URI of the device to which this alarm belongs. The device URI serves as a unique identifier for a device, and all alarms that come from the same device have the same URI. This is a read-only property.
timestamp	A read-only **Date** property indicating the time at which the alarm occured, as determined by the alarm publisher.
timecode	A read-only **string** property indicating the timecode at which the alarm occured, when known. The returned **string** is in the form: **"hh:mm:ss:ff"**. The value of the property might be **null** when no timecode information is available.
type	The type of the alarm, as a combination of flags from **TYPE_STATUS**, **TYPE_TEXT** and **TYPE_NOT_LOGGED** (read-only).

live	This is a read/write property. If you created this alarm and then set this property to **false**, the alarm will be destroyed and removed from GSM. You can set this to **true** later to publish it again.
resetLatch()	This method resets the latch on the alarm in GSM.
acknowledge()	This method acknowledges the alarm in GSM.
setOperationalMode()	This method sets or unsets the specified operational mode on the alarm in GSM. It takes two parameters of types **int** and **boolean**. The first parameter specifies the mode, and the second specifies whether to set or unset (**true** or **false**). Note: **NO_MODE** cannot be used.
addAction()	This method allows attaching of an alarm consumer **GSMPlugin** to a given alarm. Whenever the alarm changes after that, the plug-in will be notified.
removeAction()	This method allows detaching of an alarm consumer **GSMPlugin** from a given alarm. Whenever the alarm changes after that, the plug-in will no longer be notified.
NORMAL	A possible value for **status**.
MINOR	A possible value for **status**.
MAJOR	A possible value for **status**.
CRITICAL	A possible value for **status**.
UNKNOWN	A possible value for **status**.
DISABLED	A possible value for **status**.
NON_EXISTENT	A possible value for **status**.
PASSTHROUGH	A possible contribution value for a sub-alarm. The sub-alarm's status will be contributed as is to the overall status of a virtual alarm.

INVERT	A possible contribution value for a sub-alarm. The inverted value of the sub-alarm's status will be contributed to the overall status of a virtual alarm, based on the correspondences below.
	
	`NORMAL -> ERROR`
	`MINOR -> NORMAL`
	`MAJOR -> NORMAL`
	`CRITICAL -> NORMAL`
	`NON_EXISTENT -> UNKNOWN`
	`PENDING -> PENDING`
	`DISABLED -> DISABLED`
	`UNKNOWN -> UNKNOWN`

FAULTS_ONLY	A possible contribution value for a sub-alarm. The contributed status will be **NORMAL** unless the sub-alarm's status is a fault (**CRITICAL, MAJOR, MINOR**).
NO_MODE	The value of **operationalMode** when no mode is set.
OFFLINE	A possible value for **operationalMode**.
MAINTENANCE	A possible value for **operationalMode**.
TYPE_STATUS	A possible value for the alarm type parameter of the constructor and of the **GSM.addAlarm()** method. Can be combined with the other types using the **|** operator.
TYPE_TEXT	A possible value for the alarm type parameter of the constructor and of the **GSM.addAlarm()** method. Can be combined with the other types using the **|** operator. For instance, to create an alarm that will have text and status, you can specify **TYPE_TEXT | TYPE_STATUS**.
TYPE_NOT_LOGGED	A possible value for the alarm type parameter of the constructor and of the **GSM.addAlarm()** method. Can be combined with the other types using the **|** operator. For instance, to create a text alarm that should not be logged (say, because it changes too often), you can specify **TYPE_TEXT | TYPE_NOT_LOGGED**.

GSMPlugin

The **GSMPlugin** object represents a GSM alarm provider or consumer plug-in.

Property/Method	Description
Constructor	The constructor instantiates a new GSM plug-in. The fully-qualified type of the plug-in must be specified as the sole **String** parameter to the constructor.
The actual fields of the plug-in are defined by the plug-in itself, and therefore vary depending on the type of plug-in instantiated. Refer to the plug-in documentation to know which properties are available and what they do. Some of the most commonly used are listed below.	
com.miranda.icontrol.gsm.server.plugins.consumers.ScriptedAlarmConsumerPlugin	
onAlarm()	Event handler that is called when one of the alarms this plug-in is attached to changes. The method has a single parameter, the alarm object (a **GSMAlarm**).
All SNMP provider plug-ins	

`parameters[]`	A write-only property used to configure the plug-in. Each plug-in can define its own properties to define its behavior, but all plug-ins support at least the following properties: `host`, `name` and `path`, all of type `string`.
`asyncParameters[]`	This property works just like `parameters`, except it executes asynchronously. Setting `parameters` re-executes the plug-in, which in some instances may take a long time (especially if the target device is momentarily down). If you don't need to wait for the results of setting the `parameters` (you usually don't), using `asyncParameters` will normally result in better perceived performance.

SNMPManagerPlugin

`SNMPManagerPlugin` is simply the type of the `this` object in the context of an SNMP manager plug-in initialization script.

Property/Method	Description
`alarm`	References a `GSMAlarm` used to represent the status of the current plug-in. It's name and path will be the ones specified by the user in the SNMP script dialog. This field is read-only but can be used to modify the status of the plug-in as required.
`alarms`	References an `Array` of all the alarms (as `GSMAlarm`s that have been created by the script, indexed by alarm URIs. This field is read-only.
`host`	References the host name or IP address of the target SNMP agent as a `String`, as it was entered by the user in the SNMP script dialog.
`poller`	References an `SNMPPoller` object that automatically connects to the plug-in's target agent host when you add specific object IDs to its list of SNMP variables to poll (read-only).
`snmp`	References an `SNMPAgent` object that automatically references the plug-in's target agent host for SNMP operations (read-only).
`onInit()`	The JavaScript function that initialized this plug-in instance, as specified by the user in the SNMP script dialog.
`NORMAL`	A possible value for alarm statuses, here for convenience.
`WARNING`	A possible value for alarm statuses, here for convenience.
`ERROR`	A possible value for alarm statuses, here for convenience.
`UNKNOWN`	A possible value for alarm statuses, here for convenience.
`DISABLED`	A possible value for alarm statuses, here for convenience.
`NON_EXISTENT`	A possible value for alarm statuses, here for convenience.

SNMPAgent

The **SNMPAgent** object allows you to communicate with a single SNMP agent on a specific host.

Some parameters are used for many methods. The ***OID*** parameter is a **String** that looks like **".1.3.6.1.2.1.1.3.0"**. The ***MIB*** parameter is a **String** representing the logical name of a MIB as it is specified in the MIB file itself, for instance **"RFC1213-MIB"**. The ***variable*** parameter is a **String** representing the name of a MIB variable within the MIB, for instance **"sysName"**. The ***index*** parameter is a **String** that references a specific instance of a MIB variable on the target host. For most variables, this is the default **".0"**; however, for SNMP tables the index can be a longer **String** with multiple keys. Finally the ***value*** parameter is a **String** represents the new value of a given variable on the target host, and it should conform to the type described in the MIB for that variable.

Property/Method	Description
targetHost	References the hostname or IP address of the target SNMP agent's host (read/write **String**).
targetPort	References the IP port of the SNMP agent host (read/write **int**).
retries	References the maximum number of times a failed SNMP operation should be retried before it fails (read/write **number**, default is no retry).
timeout	References the maximum time (in seconds) to wait for a reply before the attempt is considered a failure (read/write **number**, default is 5 seconds).
readCommunity	The value of the read community string that will be used for read operations on the remote agent. The default value is **"public"**
writeCommunity	The value of the write community **string** that will be used when **set()** is called. The default value is **"private"**. When set to **null**, the value of **readCommunity** will be used in place of **writeCommunity**. Agents typically use the default value of **"private"** for the write community string.
get()	Gets the value of an SNMP variable on the target host. There are three variations: • **get(*OID*)** • **get(*MIB*, *variable*)** • **get(*MIB*, *variable*, *index*)**
getVB()	Gets the variable binding (**SNMPVarbind**) representing an SNMP variable on the target host. There are three variations: • **getVB(*OID*)** • **getVB(*MIB*, *variable*)** • **getVB(*MIB*, *variable*, *index*)**

getNext()	Gets the value of the next SNMP variable on the target host. There are three variations: • **getNext(*OID*)** • **getNext(*MIB, variable*)** • **getNext(*MIB, variable, index*)**
getNextVB()	Gets the variable binding (**SNMPVarbind**) representing the next SNMP variable on the target host. There are three variations: • **getNextVB(*OID*)** • **getNextVB(*MIB, variable*)** • **getNextVB(*MIB, variable, index*)**
getOID()	Looks up a OID in an SNMP MIB. There are three variations: • **getOID(*OID*)** • **getOID(*MIB, variable*)** • **getOID(*MIB, variable, index*)**
set()	Sets the value of an SNMP variable on the target host. There are three variations: • **set(*OID, value*)** • **set(*MIB, variable, value*)** • **set(*MIB, variable, index, value*)** Note that this requires that the MIB containing the variable definition be loaded for this **SNMPAgent** already, even for the version that takes only the OID. This is achieved by having performed any previous name look-up using the MIB. The MIB is required to know what the data type of the variable is. In the future, another version of the method will enable you to explicitely specify the data type, for times when the MIB cannot be referenced.
sendNotification()	This method is used to send an SNMP version 2c notification to the target host. This method takes three parameters: the up time of the sending agent (which is probably the localhost), the trap OID (as a **string**), and the variable bindings. The variable bindings parameter should be a JavaScript **Array** containing zero or more instances of **SNMPVarbind**.

sendTrap()	This method is used to send an SNMP version 1 trap to the target host. The method takes five parameters: the up time of the sending agent (which is probably the localhost), the generic trap type (use the **trapType** constants from **SNMPTrapEvent**), the specific trap type (a **number** defined by your application), the enterprise OID (as a **String**) and finally the variable bindings. The variable bindings parameter should be a JavaScript **Array** containing zero or more instances of **SNMPVarbind**.

SNMPVarbind

The **SNMPVarbind** object is used to encapsulate SNMP variable bindings, essentially typed key/value pairs. Variable bindings are commonly used, for instance, to send extra information in the payload of an SNMP trap or notification.

Property/Method	Description
constructor	The constructor for **SNMPVarbind** takes up to three parameters: an OID (as a **String**), a value (as a **String**, which will be converted appropriately depending on the value of **type**) and a type (one of the available type constants, **STRING** being the default if left unspecified).
oid	The OID for this variable binding, as a read/write **String**.
value	The value associated to the OID in this variable binding, as a read/write **String**. The value will be appropriately converted to the correct SNMP type for types other than **STRING**.
type	The SNMP type of the variable, as one of the constants defined in **SNMPVarbind**. This is a read/write property.
typeAsString	A read-only **String** representation of the SNMP type of the variable.
BITSTRING	A possible value for **type**.
COUNTER	A possible value for **type**.
COUNTER64	A possible value for **type**.
GAUGE	A possible value for **type**.
INTEGER	A possible value for **type**.
IPADDRESS	A possible value for **type**.
NSAP	A possible value for **type**.
NULLOBJ	A possible value for **type**.
OBJID	A possible value for **type**.
OPAQUE	A possible value for **type**.

`STRING`	The default value for `type`.
`TIMETICKS`	A possible value for `type`.
`UINTEGER32`	A possible value for `type`.
`UNSIGNED32`	A possible value for `type`.

SNMPPoller

The `SNMPPoller` object is used to poll SNMP variables from an SNMP agent.

Property/Method	Description
`autoActive`	Normally, the poller will automatically start polling as soon as it has enough information as to what needs to be polled from where. This property allows you to override that behavior and not start polling until `restartPolling()` is called explicitely. This is a read/write property.
`debug`	This `boolean` field can be used to log all the SNMP PDUs received from the remote agent. This is only useful to help diagnose problems with devices that send PDUs that are not handled properly. This is a read/write property.
`objectID`	References the SNMP object ID to poll, when there is only one. This is a read/write property. See also `objectIDs[]`.
`objectIDs[]`	An `Array` referencing all the SNMP object IDs to poll. This is a read/write property. See also `objectID`.
`plugin`	If this poller was created by a plug-in, the `plugin` field refers to that plug-in. This can be extremely useful when you need to have script variables that are visible both to the plug-in initialization script and to the poller result handler.
`pollInterval`	This references the interval (in seconds) between each polling attempt. This is a read/write property. The default value is one (1) second, which in most cases is probably too aggressive. Poller users should therefore normally always set a value explicitely.
`readCommunity`	The value of the read community string that will be used for read operations on the remote agent. The default value is `"public"`
`retries`	References the maximum number of times a failed SNMP operation should be retried before it fails (read/write `number`, default is no retry).

sendTimeoutEvents	A read/write boolean property specifying whether we should be notified of timeout events, in order to handle errors explicitely. This option is enabled (**true**) by default. The events will be delivered via the **onResult()** event handler, just like polling result. Use the **event.success** property in order to determine whether the event is a polled value or an error report.
targetHost	This references the target's host name or IP address, as a **String**. This is a read/write property.
targetPort	References the IP port of the SNMP agent host (read/write **int**).
timeout	References the maximum time (in seconds) to wait for a reply before the attempt is considered a failure (read/write **number**, default is 5 seconds).
addObjectID()	This adds a single SNMP object ID (the **String** parameter) to the list of polled objects.
removeObjectID()	This removes a single SNMP object ID (the **String** parameter) from the list of polled objects.
destroy()	This method should be called when the poller is no longer required, in order to release resources that it might be holding. In general, do not destroy a poller unless you have created it or you have inherited its management explicitely.
restartPolling()	This method is used to start the polling if it was stopped or never started (see **autoActive** property).
NORMAL	A possible value for alarm statuses, here for convenience.
WARNING	A possible value for alarm statuses, here for convenience.
ERROR	A possible value for alarm statuses, here for convenience.
UNKNOWN	A possible value for alarm statuses, here for convenience.
DISABLED	A possible value for alarm statuses, here for convenience.
NON_EXISTENT	A possible value for alarm statuses, here for convenience.
onResult()	The event handler that is invoked with poll results. It takes a single parameter, of type **SNMPResultEvent**.

SNMPResultEvent

The **SNMPResultEvent** object represents a polling result obtained from an **SNMPPoller** object, and passed as a parameter to its **onResult()** event handler.

Property/Method	Description

`error`	In case of a failure (`success == false`), this references a read-only error message `String`.
`OIDs[]`	An `Array` object referencing the list of SNMP object IDs (as `String`s) for the variable bindings in the result.
`success`	Indicate whether the request succeeded (`true`) or failed (`false`). Timeouts and other failure events are generated if the `sendTimeoutEvents` parameter is set to `true` in the `SNMPPoller`.
`values[]`	An `Array` object referencing the list of SNMP values (as `String`s) for the variable bindings in the result.
`variables[]`	A JavaScript `Array` representing the SNMP variable bindings from the response PDU. The array is indexed by the OID of the variables (as `String`s), and the values in the array are `String`s.

SNMPTrapReceiver

The `SNMPTrapReceiver` object is used to receive traps from an SNMP agent host.

Property/Method	Description
`community`	References the SNMP community for this trap receiver, which is a read/write `String`. The default value is `null`.
`communityEnabled`	Indicates whether community name authentication should be enabled or not for SNMP v1/v2 traps. This is a read/write `boolean` value with a default value of `true`.
`debug`	This `boolean` field can be used to log all the SNMP PDUs received from the remote agent. This is only useful to help diagnose problems with devices that send PDUs that are not handled properly. This is a write-only property.
`port`	References the local IP port to bind on to receive SNMP traps, as a read/write `int`. The default value is `162`. This will throw an exception if the port is already in use.
`destroy()`	This method should be called when the trap receiver is no longer required, in order to release resources that it might be holding. In general, do not destroy a trap receiver unless you have created it or you have inherited its management explicitely.
`stop()`	Stops the reception of traps. Use `start()` to start receiving traps again.
`start()`	This method is used to restart the trap reception after a call to `stop()`.
`onTrap()`	This is the event handler that is invoked when a trap is received. It takes a single parameter of type `SNMPTrapEvent`.

SNMPTrapEvent

The **SNMPTrapEvent** object represents a trap that is received by an **SNMPTrapReceiver** object, and passed as a parameter to its **onTrap()** event handler.

Property/Method	Description
agentAddress	References the host name or IP address of the agent host, as a **String** (read-only).
community	The SNMP community the trap was sent to, as a read-only **String** (typically **"public"**).
enterprise	The trap enterprise OID, as a read-only **String**.
remoteHost	The host name or IP address of the remote agent host, as a read-only **String**.
remotePort	The IP port of the remote agent host, as a read-only **int**.
specificType	The specific type number for the trap received, as a read-only **int**.
trapOID	References the trap OID present (as a variable binding) in an SNMPv2 trap, as a read-only **String**. The value will be **null** for traps that are not SNMPv2. As the information is sent as a variable binding, this is equivalent to **variables['.1.3.6.1.6.3.1.1.4.1.0']**.
trapType	The trap type of the received trap, as a read-only **int**. The value will be one of the generic trap types (**coldStart**, **warmStart**, **linkDown**, **linkUp**, **authenticationFailure** or **egpNeighbourLoss**), or **enterpriseSpecific**, in which case the **specificType** field becomes relevant.
coldStart	A possible value for **trapType**.
warmStart	A possible value for **trapType**.
linkDown	A possible value for **trapType**.
linkUp	A possible value for **trapType**.
authenticationFailure	A possible value for **trapType**.
egpNeighbourLoss	A possible value for **trapType**.
enterpriseSpecific	A possible value for **trapType**.
upTime	The uptime value for a trap, as a read-only **long**.
variables[]	A JavaScript Array representing the SNMP variable bindings from the trap PDU. The array is indexed by the OID of the variables (as **String**s), and the values in the array are **String**s.
version	The SNMP version number from the trap PDU, as a read-only **int**.

ScriptedAlarmConsumerPlugin

The **ScriptedAlarmConsumerPlugin** represents the equivalent GSM plug-in and is the **this**

object within a script in the plug-in.

Property/Method	Description
`NORMAL`	A possible value for `alarm.status`.
`WARNING`	A possible value for `alarm.status`.
`ERROR`	A possible value for `alarm.status`.
`UNKNOWN`	A possible value for `alarm.status`.
`DISABLED`	A possible value for `alarm.status`.
`NON_EXISTENT`	A possible value for `alarm.status`.
`PENDING`	A possible value for `alarm.status`.
`onAlarm()`	Event handler that is called when one of the alarms this plug-in is attached to changes. The method has a single parameter, the alarm object (a `GSMAlarm`).
`includeJS(script)`	This function is used to load an external JavaScript file into the context of the scripted plug-in object. By using this method you can create reusable JavaScript libraries. The parameter is an absolute URL to load the script from, as a `string`.

Router

The `Router` represents an A/V router (or switcher) that has been discovered on the network.

Property/Method	Description
`name`	A read-only `string` property containing the router name.
`type`	A read-only `string` property describing the type of router.
`numLevels`	A read-only `number` property indicating the number of levels defined for the router.
`numInputs`	A read-only `number` property indicating the number of inputs/sources defined for the router.
`numOutputs`	A read-only `number` property indicating the number of outputs/ destinations defined for the router.
`service`	Provides direct access to the router service.
`connectCrosspoint()`	This method switches a single router cross point. It takes three `number` parameters: the level, the source and the destination to connect (as `number`s).

`getConnectedSource()`	This method returns (as a `number`) the currently connected source for a given combination of level and destination (the two `number` parameters to this method).
`getSourceLabel()`	This method returns (as a `string`) the label of the specified source on a given level. It takes two `number` parameters: the level index and the source index.
`getSourceIndex()`	This method returns (as a `int`) the one-based index of the specified source label on a given level. It returns -1 if the label is not found. It takes two parameters: the level index and the source label.
`getDestinationLabel()`	This method returns (as a `string`) the label of the specified destination on a given level. It takes two `number` parameters: the level index and the destination index.
`getDestinationIndex()`	This method returns (as a `int`) the one-based index of the specified destination on a given level. It returns -1 if the label is not found. It takes two parameters: the level index and the destination label.
`lockCrosspoints()`	Locks a router destination at a given level. It takes three parameters: the level and the destination (as `number`s) and the lock message.
`unlockCrosspoints()`	Unlocks a router destination at a given level. It takes two parameters: the level and the destination.
`isLocked()`	Tells whether a router destination at a given level is locked. It takes two numeric parameters: the level and the destination.
`takeSalvo()`	Execute salvo. It takes only one parameter: the salvo name as defined in the salvo editor. If at least one crosspoint cannot be taken, none is taken.

KaleidoX

The `KaleidoX` object is used to communicate with a Kaleido X multi-image display device from Miranda. The user is responsible for creating instances of this class, using the constructor explicitly. The constructor requires a single argument `string`, representing the host name or IP address of the Kaleido X you wish to talk to.

Property/Method	Description
constructor	The `KaleidoX()` constructor is used to specify the host name or IP address of the remote Kaleido X unit to control. It takes a single `string` parameter representing this address.
`room`	This is the name of the room currently selected, or `null` when in the global context.
`rooms[]`	This `Array` contains the list of all rooms available on the KX, each one represented as a `string` object containing the name of the room.

audioOut	A read/write property which can take four types of values: **null** To indicate that no audio source is currently being routed to the audio output on the Kaleido. a number To indicate which audio card input is routed to the audio output on the Kaleido. an object with **input**, **group** and **aes** properties To indicate that an embedded audio is routed to the audio output on the Kaleido. The object has three numeric properties named **input**, **group** (from 1 to 4) and **aes** (either 1 or 2) to indicate which embedded audio is used. an object with **host** and **feed** properties To indicate that an audio stream on the network is routed to the audio output on the Kaleido. The object has two **string** properties named **host** (an IP address) and **feed** (the ID of the feed to use from the specified host).
layout	This is the name of the layout currently being displayed for the current room on the KX, without the layout file extension (**.kg2**). Accordingly, it will only work in the context of a specific room, i.e. when **room** is not **null**. It is a read/write **string** value which can also be used to load a different layout for the current on the KX.
layouts[]	This **Array** contains the list of all layouts available for the current room on the KX, each one represented as a **string** object containing the name of the layout without its extension. When in the global context, this will list all layouts in all rooms using the following format: ***room/layout***.
open()	This method is used to open the connection with the KX gateway whose IP address was specified in the constructor.
closeID()	This method is used to close the connection with the KX gateway.
setText()	This method is used to set any dynamic text component on the KX display. It takes two **string** arguments representing the address or ID of the text component and the value to be set.
setStatus()	This method is used to set any status component on the KX display. It takes three arguments representing the ID of the status component (as a **string**), the status (**"OK"**, **"DISABLE"**, **"WARNING"**, **"ERROR"**, or one of the **GSMAlarm** status constants) and a status message (as a **string**).
setAssignment()	This method is used to assign a channel to a monitor on the KX display. It only works in the context of a specific room, i.e. when **room** is not **null**. It takes two **string** arguments representing the full path to the channel (ex.: **/Input A/Channel 1**) and the monitor name (which is available in the XEdit layout under **Properties | Assignments | Name**).

KaleidoK2

The **KaleidoK2** object is used to communicate with a Kaleido K2 multi-image display device from Miranda. The user is responsible for creating instances of this class, using the

constructor explicitly. The constructor requires a single argument **string**, representing the host name or IP address of the Kaleido K2 you wish to talk to.

Property/Method	Description
constructor	The **KaleidoK2()** constructor is used to specify the host name or IP address of the remote K2 unit to control. It takes a single **string** parameter representing this address.
audioOut	A read/write property which can take four types of values: **null** To indicate that no audio source is currently being routed to the audio output on the Kaleido. a number To indicate which audio card input is routed to the audio output on the Kaleido. an object with **input**, **group** and **aes** properties To indicate that an embedded audio is routed to the audio output on the Kaleido. The object has three numeric properties named **input**, **group** (from 1 to 4) and **aes** (either 1 or 2) to indicate which embedded audio is used. an object with **host** and **feed** properties To indicate that an audio stream on the network is routed to the audio output on the Kaleido. The object has two **string** properties named **host** (an IP address) and **feed** (the ID of the feed to use from the specified host).
audioOutMode	A read/write property specifying the current audio output mode. Valid values are **NORMAL**, **MUTE** and **-20dB**.
audioOutVolume	The audio monitoring volume, as a read/write numeric property ranging between -90dB and 0dB.
layout	This is the name of the layout currently being displayed on the K2, without the layout file extension (**.kg2**). It is a read/write **string** value which can be used to load a different layout on the K2. This replaces the old **getLayout()** and **loadLayout()** methods, which remain for backwards compatibility.
layouts[]	This **Array** contains the list of all layouts available on the K2, each one represented as a **string** object containing the name of the layout without its extension. This is also accessible using the **getLayouts()** method for backwards compatibility.
openID()	This method is used to open the connection with the K2 gateway. It takes one single **string** argument representing the IP address of the K2.
closeID()	This method is used to close the connection with the K2 gateway. It takes a single **string** argument representing the IP address of the K2.

getMetadata()	This method takes a single parameter, the name (as a **string**) of a multidata component configured with metadata (for instance XDS data) on the K2. It returns an **XML** object representing the metadata published by the component, as documented in the K2 gateway manual. The object can be accessed using the E4X syntax. For example, to read the network name of the station on input #5, you can write:
	`var metadata = k2.getMetadata(dataName);`
	`var networkName = metadata.input.(@id==5).@networkName;`

saveLayout()	This method saves the current layout being displayed on disk. It takes a single **string** parameter representing the name of the saved layout (no extension).
setText()	This method is used to set any dynamic text component on the K2 display. It takes two **string** arguments representing the address or ID of the text component and the value to be set.
setStatus()	This method is used to set any status component on the K2 display. It takes three arguments representing the ID of the status component (as a **string**), the status (**"OK"**, **"DISABLE"**, **"WARNING"**, **"ERROR"**, or one of the **GSMAlarm** status constants) and a status message (as a **string**).
setAssignment()	This method is used to assign a channel to a monitor on the K2 display. It takes two **string** arguments representing the channel name and the monitor name.
setCountdownTimer()	This method is used to configure countdown timers on the K2 display. It takes four arguments representing the timer ID (as a **string**), the preset time (a **string** in the **"HH:MM:SS"** format), the direction (a **boolean** value of **true** for **"DOWN"** or **false** for **"UP"**), and the loop (a **boolean** value of **true** for **"ON"** or **false** for **"OFF"**).
resetCountdownTimer()	This method is used to reset a countdown timer on the K2 display to its original state. It takes a single **string** argument, representing the ID of the countdown timer to reset.
startCountdownTimer()	This method is used to start a countdown timer on the K2 display. It takes a single **string** argument, representing the ID of the countdown timer to start.
stopCountdownTimer()	This method is used to stop countdown timers on the K2 display. It takes a single **string** argument, representing the ID of the countdown timer to stop.
fireAction()	This method is used to fire any programmed actions on the K2. It takes a single **string** argument representing the action name saved on the K2.
enableAlarmGroup()	This method is used to enable any alarm group programmed on the K2. It takes a single **string** argument representing the alarm group name saved on the K2.

`disableAlarmGroup()`	This method is used to disable any alarm group programmed on the K2. It takes a single `string` argument representing the alarm group name saved on the K2.

KaleidoAlto

The `KaleidoAlto` object is used to communicate with a Kaleido Alto or Quad multi-image display device from Miranda. The user is responsible for creating instances of this class, using the constructor explicitly. The constructor requires a single argument string, representing the host name or IP address of the Kaleido Alto or Quad you wish to talk to.

Property/Method	Description
constructor	The `KaleidoAlto()` constructor is used to specify the host name or IP address of the remote Alto unit to control. It takes a single `string` parameter representing this address.
`audioOut`	A read/write property which can take three types of values: `null` To indicate that no audio source is currently being routed to the audio output on the Kaleido. a number To indicate which audio card input is routed to the audio output on the Kaleido. an object To indicate that an embedded audio is routed to the audio output on the Kaleido. The object has three numeric properties named `input`, `group` (from 1 to 4) and `aes` (either 1 or 2) to indicate which embedded audio is used. *Note that this requires version 3.01 or above of the Alto software.*
`audioOutMode`	A read/write property specifying the current audio output mode. Valid values are `NORMAL`, `MUTE` and `-20dB`.
`audioOutVolume`	The audio monitoring volume, as a read/write numeric property ranging between -90dB and 0dB.
`layout`	This is the name of the layout currently being displayed on the Kaleido, without the layout file extension. It is a read/write `string` value which can be used to load a different layout on the Kaleido. This replaces the old `loadLayout()` method, which remains for backwards compatibility.
`layouts[]`	The (read-only) list of all layouts available on the Alto, as an `Array` of `string`s (the names of the layouts without the extensions).

`verticalOffset`	This write-only property specifies the number of lines (between 0 and 175) to offset the graphics vertically on the display.
`closeID()`	Closes the connection with the Alto gateway. It takes a single `string` argument representing the IP address of the Alto.
`fireAction()`	This method is used to fire any programmed actions on the Kaleido. It takes a single `string` argument representing the action name saved on the Kaleido.
`ftpLayout()`	The `ftpLayout()` method is used to change the layout being displayed on the Alto. It takes a single `string` parameter representing the name of the layout as defined on the Alto. This uses the FTP interface on the Alto (not the gateway). The other methods all use the gateway interface.
`openID()`	Opens the connection with the Alto gateway. It takes a single `string` argument representing the IP address of the Alto.
`resetCountdownTimer()`	This method is used to reset a countdown timer on the Kaleido display to its original state. It takes a single `string` argument, representing the ID of the countdown timer to reset.
`saveLayout()`	This method saves the current layout being displayed to disk. It takes a single `string` parameter representing the name of the saved layout (without an extension).
`setAssignment()`	This method is used to assign a channel to a monitor on the Kaleido display. It takes two `string` arguments representing the channel name and the monitor name.
`setCountdownTimer()`	This method is used to configure countdown timers on the Kaleido display. It takes four arguments representing: the timer ID (as a `string`), the preset time (a `string` in the `"HH:MM:SS"` format), the direction (a `boolean` value of `true` for `"DOWN"` or `false` for `"UP"`), and the loop (a `boolean` value of `true` for `"ON"` or `false` for `"OFF"`).
`setStatus()`	This method is used to set any status component on the Kaleido display. It takes three arguments representing the ID of the status component (as a `string`), the status (`"OK"`, `"DISABLE"`, `"WARNING"`, `"ERROR"`, or one of the `GSMAlarm` status constants) and a status message (as a `string`).
`setText()`	This method is used to set any dynamic text component on the Kaleido display. It takes two `string` arguments representing the address or ID of the text component and the value to be set.
`startCountdownTimer()`	This method is used to start a countdown timer on the Kaleido display. It takes a single `string` argument, representing the ID of the countdown timer to start.
`stopCountdownTimer()`	This method is used to stop countdown timers on the Kaleido display. It takes a single `string` argument, representing the ID of the countdown timer to stop.

RCP100

The **RCP100** object is used to communicate by an iControl RCP-100 control panel from Miranda. The user is responsible for creating instances of this class, using the constructor explicitly. Note that the RCP-100 unit must be placed in *Router Follow* mode for this to work.

Property/Method	Description
constructor	The **RCP100()** constructor is used to specify the host name or IP address of the remote RCP-100 unit to control. It takes a single string parameter representating this address.
close()	The **close()** method is used to place the RCP-100 in a mode where no card is selected and it is awaiting another command. It will normally show **NO CARD ASSIGNED** on the LCD.
quit()	The **quit()** method is used to release resources (sockets) that are used by this object, when it is not required anymore.
show()	The **show()** method is used to change the card currently under control in the RCP-100. It takes a single string parameter representing the **longID** of the card to load.
showGroup()	The **showGroup()** method is used to change the set of cards currently under control in the RCP-100. The first argument to this method is an **Array** of **longID**s (as **String**s) of the cards to load for selection. If a second **String** argument is added to the method, it will be the **longID** of the card that should be pre-selected from the set defined in the first argument.

Gateway

The **Gateway** object represents an iControl XML Gateway, which is used to communicate with remote gateway-enabled processes using an XML-based protocol.

Property/Method	Description
constructor	The constructor takes up to two arguments: a **string** representing the host name or IP address of the remote gateway you wish to control, and an optional port number as a **number** (the default port is 14000).
sendCommand()	Sends an XML command to the gateway (as a **string**). An exception will be thrown if the gateway sends a NACK reply.

XMLHttpRequest

XMLHttpRequest is a JavaScript object that was initially created by Microsoft and later adopted

by Mozilla. You can use it to easily retrieve data via HTTP. Despite its name, it can be used for more than just XML documents.

Property/Method	Description
pollInterval	This read/write numeric field indicates the interval (in milliseconds) to use between successive requests for the HTTP page when using the asynchronous mode of operation. If set to a positive non-zero value prior to calling **send()**, the page will be requested repeatedly at the specified interval until the value is set back to zero. The default value is **0**, and if not modified it will result in a single HTTP request.
readyState	Represents the state of the request. Read-only. The following values are defined: 0 (**UNINITIALIZED**) The object has been created, but not initialized (the **open** method has not been called). 1 (**LOADING**) The object has been created, but the **send** method has not been called. 2 (**LOADED**) The **send** method has been called, but the status and headers are not yet available. 3 (**INTERACTIVE**) Some data has been received. Calling the **responseText** property at this state to obtain partial results will return an error, because status and response headers are not fully available. 4 (**COMPLETED**) All the data has been received, and the complete data is available in the **responseText** property.
responseText	Represents the response entity body as a string. Read-only.
responseXML	If you load an XML document, the **responseXML** property will contain the document as an **XMLDocument** object that you can manipulate using DOM methods. For loaded non-XML documents (whose **Content-Type** header does not indicate XML content), the **responseXML** property is **null**.
status	Represents the HTTP status code returned by a request. Read-only.
statusText	Represents the HTTP response line status. Read-only.
onreadystatechange()	Specifies the event handler to be called when the **readyState** property changes. Read/write.
abort()	Cancels the current HTTP request.
getAllResponseHeaders()	Retrieves the values of all the HTTP headers.

`getResponseHeader()`	This method can be used to get an HTTP header from the server response. It takes a single `string` parameter containing the case-insensitive header name.
`open()`	Initializes an HTTP request, and specifies the method, URL, and authentication information for the request. It takes up to five parameters, the last three of which are optional. The first parameter is the HTTP method to use as a `string`, such as `GET`, `POST`, `PUT`, or `PROPFIND`. The second argument represents the requested URL, either relative or absolute. The third (optional) argument is a `boolean` indicating whether the call is asynchronous, the default being `true` (the method returns immediately). If set to `true`, attach an `onreadystatechange` property callback so that you can tell when the `send()` call has completed. The (optional) fourth and fifth arguments are the username and password to use for authentication, if required.
`send()`	Sends an HTTP request to the server and receives a response.
`setRequestHeader()`	This method can be used to set an HTTP header on the request before you send it.

WorkBook

The `WorkBook` object is used to represents an Excel workbook (file). Any of the sheets in the workbook can be accessed as an array element with the spreadsheet index as an array index.

Property/Method	Description
constructor	The `WorkBook` constructor is used to build a parser on a given Excel (`.xls`) file. It takes a single `String` parameter which represents the URL where the Excel file can be found.
`url`	References the URL of the currently loaded file.

Spreadsheet

The `Spreadsheet` object represents an Excel spreadsheet. Any of the rows can be accessed as an array element with the row index as an array index.

Row

The `Row` object is used to represent a row in an Excel spreadsheet. Any of the cells can be accessed as an array element with the cell column index as an array index.

Cell

The **Cell** object is used to represents a cell in an Excel spreadsheet.

Property/Method	Description
value	References the cell value. The value is read-only. The actual type of the value being returned depends on the Excel cell format for that specific cell.

Quick reference

Connecting a router crosspoint:

```
navigator.connectCrosspoint("routerLongID", level, source, dest);
```

Performing different actions based on the alarm status when it changes (in a scripted action):

```
switch (alarm.status) {
   case alarm.NORMAL:
  // Handle normal status
      break;
   case alarm.WARNING:
  // Handle warning status
      break;
   case alarm.ERROR:
  // Handle error status
      break;
   case alarm.UNKNOWN:
  // Handle unknown status
      break;
   case alarm.DISABLED:
  // Handle disabled alarm
      break;
   case alarm.NON_EXISTENT:
  // Handle missing alarm
      break;
   default:
  // Some other (future?) value
      break;
}
```

To add a method to an object, you can use the following syntax:

```
myObject.myMethodName = function myMethodName() {
   java.lang.System.out.println("myMethodName called on object " + this);
}
```

Exception handling:

```
try {
 // This is where you put the code that could fail
} catch (exception) {
 // This is where you handle any exception that might occur
 // If no exception occurs, this code will be skipped
   java.lang.System.err.println("Exception caught: " + exception);
}
```

If your script fails in an unexpected way, where you didn't handle the generated exception, a default error message will be shown. You can use explicit exception handling to show a different message or otherwise handle the exception differently. It can also be used to gain more fine-grained control over the handling of exceptions in your scripts. For instance, if your script needs to perform multiple actions, you can decide how the failure of the first should affect the execution of the second. For instance, this will not attempt to execute action #2 if action #1 fails:

```
try {
 // Action #1
 // Action #2
} catch (exception) {
   java.lang.System.err.println("Exception caught in action #1 or #2: " +
exception);
}
```

Whereas this will attempt to execute action #2 whether action #1 fails or not:

```
try {
 // Action #1
} catch (exception) {
   java.lang.System.err.println("Exception caught in action #1: " + exception);
}
try {
 // Action #2
} catch (exception) {
   java.lang.System.err.println("Exception caught in action #2: " + exception);
}
```

A sample SNMP **sysUpTime** script:

```
this.sysUpTimeAlarm = gsm.addAlarm("snmp://" + host + "/RFC1213-MIB/sysUpTime",
                                   "Test SNMP poller",
                                   "SNMP/Pollers",
```

```
                                        "Generic SNMP device",
                                        "snmp://" + host,
                                        "text");

poller.objectID = snmp.getOID('RFC1213-MIB','sysUpTime'); // '.1.3.6.1.2.1.1.3.0';
poller.pollInterval = 1; //second
poller.onResult = function onResult(event) {
   if (event.success) {
      alarm.status = alarm.NORMAL;
      this.plugin.sysUpTimeAlarm.text = event.values[0];
   } else {
      alarm.status = alarm.ERROR;
      this.plugin.sysUpTimeAlarm.text = event.error;
      java.lang.System.err.println('Error: ' + event.error);
   }
};
```

Setting up an SNMP trap receiver:

```
trapper = new SNMPTrapReceiver();
trapper.onTrap = function onTrap(event) {
   java.lang.System.err.println("TRAP! agent: " + event.agentAddress
                              + "  community: " + event.community
                              + "  enterprise: " + event.enterprise
                              + "  uptime: " + event.upTime
                              + "  remote host: " + event.remoteHost + ":" + event.
remotePort
                              + "  trap type: " + event.trapType + " (" + event.
specificType + ")"
                              + "  version: " + event.version);
   for (i in event.variables) {
      java.lang.System.err.println("      " + i + " = " + event.variables[i]);
   }
};
```

Attaching a scripted action to scripted alarms:

```
// Assume virtualAlarms is an Array of GSMAlarms.  These could have
// been generated any number of ways, for instance using calls to
// GSM.addVirtualAlarm(...)

// Create the alarm consumer (action) plug-in
var action = new GSMPlugin("com.miranda.icontrol.gsm.server.plugins.consumers.
ScriptedAlarmConsumerPlugin");

// Define the alarm handler of the action plug-in
// Parameter alarm is a GSMAlarm
```

```
action.onAlarm = function onAlarm(alarm) {
   java.lang.System.err.println("Alarm received: " + alarm.name
                                                + "=" + alarm.text);
};

// Attach the action to each virtual alarm; function onAlarm() will
// be called every time the alarm changes
for (i in virtualAlarms) {
   virtualAlarms[i].addAction(action);
}
```

Example of a channel assignment where channel information is obtained by parsing an Excel spreadsheet. Note that in this example, the **Channel** class is an application-specific class.

```
var wb = new WorkBook("http://10.2.0.254/MBEInfo_4.xls");
this.allChannels = new Array();

function cellValue(cell) {
   return (cell == undefined) ? "" : cell.value;
}

for (var i = 0; i < 256; i++) {
   var channel = new Channel();
   channel.receiverID          = cellValue(wb[0][i][0]);
   channel.ptcID               = cellValue(wb[0][i][1]);
   channel.callLetter          = cellValue(wb[0][i][2]);
   channel.cityID              = cellValue(wb[0][i][3]);
   channel.affID               = cellValue(wb[0][i][4]);
   channel.channelNumber       = cellValue(wb[0][i][5]);
   channel.controlPhoneNumber  = cellValue(wb[0][i][6]);
   channel.enginerName         = cellValue(wb[0][i][7]);
   channel.engineerPhoneNumber = cellValue(wb[0][i][8]);
   channel.stationPhoneNumber  = cellValue(wb[0][i][9]);
}
```

XMLHttpRequest examples

Basic usage

Using **XMLHttpRequest** is very simple. You create an instance of the object, open a URL, and send the request. The HTTP status code of the result, as well as the result document are available in the request object afterwards.

Example

```
var req = new XMLHttpRequest();
req.open('GET', 'http://www.mozilla.org/', false);
req.send(null);
if (req.status == 200) {
   dump(req.responseText);
}
```

Note that this example works synchronously, so it will block the user interface if you call this from your JavaScript. You should not use this in practice.

Asynchronous usage

If you intend to use **XMLHttpRequest** from an extension, you should let it load asynchronously. In asynchronous usage, you get a callback when the data has been received, which lets the application continue to work as normal while your request is happening.

Example

```
req = new XMLHttpRequest();
req.open('GET', 'http://www.mozilla.org/', true);
req.onreadystatechange = function () {
   if (req.readyState == 4) {
      if (req.status == 200) {
         dump(req.responseText);
      } else {
         dump("Error loading page\n");
      }
   }
};
req.send(null);
```

$Id: GSMScriptingManual.html,v 1.97 2008/08/25 17:13:12 mcormier Exp $