Customizing your IRC Experience

jIRCii is a fully scriptable using the sleep language. Sleep is a simple language that slightly resembles perl. This document covers the jIRCii specific additions to sleep. The document "Sleep Language Fundamentals" provides full coverage of the features in sleep itself. You may want to read it before you read this. This document is written as a reference, it is not meant to serve as a full tutorial for scripting with jIRCii.

A slight table of contents:

Loading and Unloading Scripts

The following commands are at your disposal for script loading and unloading.

/load filename.irc
Loads the specified script.

/unload filename.irc
Unloads the specified script, restoring any aliases/sets that were overridden by the specified script.

/reload filename.irc
Unloads and then reloads the specified script.

You can also use the script manager dialog. Click the View menu, Options, and then select Script Manager. The script manager dialog has options for loading and unloading scripts.

The script manager dialog has an option to "ignore script warnings". If you are developing a script you should never have this option checked. Often times script errors won't be caught when they are loaded. They are usually announced when the script is running.

Aliases

An alias in jIRCii is a way of specifying a /command that when called it will execute a bunch of statements.

```
alias jf
{
   call("/join #mIRCScripts");
   call("/join #java");
}
```

The alias jf is for joining favorite channels. The keyword alias followed by jf declares jf as an alias. This means /jf is now a valid command in jIRCii

Inside of jf the function call() is called with a string each time. Call executes the specific jIRCii command within the double quotes. Using call() is just like typing a command in the editbox.

Custom aliases can also receive parameters.

```
alias j
{
   call("/join $1");
}
```

Aliases in jIRCii can receive parameters. The first parameter is the scalar $1. The second parameter is the scalar $2. The third is $3 etc.. The name of the alias is $0. The above alias is a shortcut for the join command. If you type /j #floods, $1 in the above alias would have the value of #floods.

The " double quoted string inside of call is known as a parsed literal. It is basically a string where $scalar values are evaluated. See the sleep language fundamentals document for more about parsed literals and scalars.

Built in Variables

jIRCii supports a number of built in variables that are always ready to use. Whenever an alias is typed in you immediately have everything broken down for you. For example say a user types: /*blah hello ice cream!*

Immediately you have the **$n** variables available to you were n is an integer or a range of integers. For example:

$n var	value
$0	/blah
$1	hello
$2	ice
$3	cream!
$1-	hello ice cream!
$2-	ice cream!
$1-2	hello ice
$-2	/blah hello ice

This is different than sleep subroutines where the arguments are **numbered** from 1..n These tokenized ranges do not work in sleep subroutines. In general parameters to events, menus, aliases, and sets are passed in the tokenized form described above.

jIRCii also supports some other built in variables. These are available all the time.

$active	the current active query/target
$lag	your lag
$me	your nickname
$mymode	your mode on the server i.e. +i
$null	the empty scalar with a string value of "" and a numerical value of 0
$time	the current time in hh:mm format

%localData	local data in jIRCii consists of function parameters, event specific variables, set specific variables, basically any kind of temporary variables. This read-only hash will contain all the local variable information for the current function call, alias execution, event etc..
%GLOBAL	the %GLOBAL hash is the same variable visible to all jIRCii server connections. This can be used as a place to share information between server sessions.

Keyboard Shortcuts

In jIRCii nearly any key combination can be bound to execute a script when pressed. This is accomplished using the bind keyword.

```
bind Escape
{
   call("/part $active");
}
```

The above binds the "Escape" key to part the active channel when pressed. Whenever someone presses escape call("/part $active") will be executed.

The format for specifying binds is:

```
bind Modifier+Key { commands }
```

The Modifier+ part is optional. You can just bind a Key.

Key Constants

Modifiers: Shift, Ctrl, Alt, Meta

Keys:

Accept	F2	Help	Pause
Back_Quote	F3	Home	Period
Backspace	F4	Insert	Print_Screen
Caps_Lock	F5	Left	Quote
Clear	F6	Num_Lock	Right
Convert	F7	NumPad_*	Scroll_Lock
Delete	F8	NumPad_+	Space
Down	F9	NumPad_,	Tab
End	F10	NumPad_-	Up
Enter	F11	NumPad_/	
Escape	F12	Page_Down	
F1	Final	Page_Up	

Also acceptable is A-Z, a-z, 0-9, and anything else. The above table is just a list of special self-explanatory constants for common keys you might want to bind.

Responding to Events

An event is a specifically defined occurrence. Examples of events include a user joining a

channel, a dcc send completing, and even a window closing. It is possible to write scripts that respond to events when they occur. This is accomplished using the on keyword.

```
on event { commands }
```

The syntax is pretty simple: on the occurrence of some event execute these commands. Whenever the specified event occurs the commands in the { } curly braces will be executed.

When an event finishes executing you can use the halt keyword to stop jIRCii scripts from further processing that event. The halt keyword works for most events. Halt has the same syntax as return except you can't return anything with a halt.

```
# taken from "save the children script"
on public
{
   if ("*swear*word* iswm $parms)
   {
       halt;
   }
}
```

The above example halts any public channel text that is a wildcard match for *swear*word*. When halt is used the event will not be processed any further and no sets will be fired.

List of Events

* = event has a special variable associated with it.
+ = event has a note associated with it.

DCC Events

chat	send_complete	receive_complete
chat_close	send_failed	receive_failed
chat_open	send_start	receive_start

GUI Events

active	exit	open
click +	input	session +
close	inactive	sclick +
dclick +	minimize	window +

click	fired when a word is clicked on in a jIRCii window
dclick	fired when a nickname in the channel listbox is double clicked on
sclick	fired when a text is double clicked on in a sorted window: $0 = row; $1- = source window
session	fired when the current session for this script becomes active again
windo w	fired every time text is echo'd to a window

IRC Events

error	*nnn* +	reply
invite	notice	request
join	part	signoff
kick	private_action	signon
mode	public	topic
msg	public_action	
nick	quit	

nnn refers to an irc numeric i.e. 001, or 363, basically an irc numeric reply.

Miscellaneous Events

connect	ident	unload +
disconnect	raw	

unload fired when a script is unloaded. Event is only fired within the unloaded script.

Event Variables

Variable	Notes
$address	Exists only when a user is the source of an event.. represents a users address
$channels	Available in the QUIT event; a comma separated list of channels the user was on.
$data	Available in most sets; the string jIRCii tokenizes into $0, $1, $2- etc.
$event	Available in most sets; a string describing the type of event
$host	The host part of user@host; exists only when $address exists
$item	Available on click event; the actual word the user clicked on.
$mouse	Available in on click/dclick/sclick events; contains mouse info
$nick	Available in most events where you would expect $nick to be available.
$parms	The parameters related to the event (available in most events)
$raw	Available in most irc related sets, the raw text at the root of the event
$server	the server that was the source of the event, exists only when the event is for an irc event and a server sent the event instead of a user. To be safe use $source to get the source of an event.
$source	the source of the event, available in most events that have a source
$target	Available in events that have a target; the target of the event
$this	Available in dcc events; a reference to that exact dcc... can be used to get the dcc session information with the function getDCCConnection($this)
$user	The user part of user@host, exists only when $address exists
$type	Available in request and reply events; represents the type of ctcp.
$window	Available in most gui events; the window the event occured in

The variables listed in the Aliases -> Built in Variables section also applies to sets as well. Generally $0 is the target/source/focus of the event. $1- is the rest of the parameters.

Temporary Events

Along with normal event listeners jIRCii also features the ability to bind temporary event listeners. Temporary event listeners are for responding to an irc event once and thats it. Temporary events are fired before normal event listeners. Temporary events can also be halted stopping the normal event listeners from processing the event.

Temporary events are bound using the wait keyword.

```
wait event { commands }
```

Temporary events can be declared nearly any where.

```
alias j
{
   if ($1 eq "#floods")
   {
      call("/msg JakieChan dude, what is the key to #floods?");

      wait privmsg
      {
         call("/join #floods $1");
      }
   }
   else
   {
      call("/join $1-");
   }
}
```

In the above example the alias j is declared. The alias checks to see if I'm trying to join the channel #floods. If I am it messages a friend of mine asking for the key. When he responds (presumably with the key), jIRCii joins the channel.

Event specific variables from when the temp listeners was declared are not carried over to temporary events. They have there own set of variables just like any other event. Nothing inside of the { } curly braces is evaluated until the event occurs. As such watch out for using variables outside of a wait command and then expecting it to be available or be the same thing inside of it.

Filtered Events

Generally when you declare an event listener it is not very descriptive. You state the event name and that is all the discretion you get. To get more specific you can specify a filter string with your events. i.e.

```
wait event "*source* *target* *parameters*" { commands }
```

An event filter string contains 3 wildcard strings separated by spaces. When an event occurs jIRCii will check the first wildcard against the source of the event meaning the nickname/server that originated the event. The second wildcard is checked against the target of the event i.e. who the event affects. The third wildcard is checked against the event parameters. If you don't care about a particular part of the filter string just specify a * which means match anything.

```
alias j
{
   if ($1 eq "#floods")
   {
      call("/msg JakieChan dude, what is the key to #floods?");

      wait msg "JakieChan $me *"
```

```
      {
         call("/join #floods $1");
      }
   }
   else
   {
      call("/join $1-");
   }
}
```

The above example is similar to our original example for temporary events. Except this time when a message happens our temporary listener will only be fired for a msg event originating from JakieChan sent to $me. The filter string is evaluated when you declare the event. Once the wait msg "filter" is declared the $variable's are immediately resolved. In the above example, if you change your nick before JakieChan messages you the temporary event won't trigger.

Descriptive Events

Descriptive events are similar to filtered events in the respect that they let you better describe the event you are looking for. To declare a descriptive event listener:

```
wait (comparison) { commands }
```

The above will declare a temporary listener that will be fired only when the comparison following the wait keyword is true. Think of the comparison part as being similar to an if statement. When an event occurs the event is checked against the specified comparison to see if the event matches that description or not. If it does then the event is fired. If it doesn't the event is not fired.

Descriptive events work with both normal events (on keyword) and temporary events (wait keyword). Descriptive events only work for events originating from the IRC server. Most GUI events and such won't fire under descriptive events.

```
alias j
{
   if ($1 eq "#floods")
   {
      call("/msg JakieChan dude, what is the key to #floods?");

      wait ($event eq "PRIVMSG" && $nick eq "JakieChan")
      {
         call("/join #floods $1");
      }
   }
   else
   {
      call("/join $1-");
   }
}
```

Beating an example to death this example is the same thing as the previous two examples. One difference though: the comparison is re-evaluated each time the event is checked. Variables specific to the original event do not carry over to the comparison.

Hacking Event Listeners

Note: *The mechanism described in this section is more or less a hack. It is only documented as it may be useful to scripter's working on more advanced functionality.*

One downfall of the temporary event mechanism is you don't get a lot of control. Basically you can register an event listener and once it is fired it is done. This is simple to use and good for some cases such as capturing the server reply of a /USERHOST request. However it is not usable for things like doing a /WHO, capturing all of the output from the server, and when the server sends the end of who reply removing the temporary event listener.

The following example shows how to get this type of functionality:

```
alias tempevent
{
   on ($event eq "315" || $event eq "352")
   {
       if ($event eq "352")
       {
          # process /who output
          echo("Caught /who: $1-");
          halt;
       }

       if ($numeric eq "315")
       {
          return 6; # halt and remove this event listener
       }
   }

   call("/who $1");
}
```

The above snippet creates an alias tempevent. When /tempevent is executed a descriptive event is registered. This descriptive event fires when a numeric reply 352 (/who data) or a numeric reply 315 (end of /who data) is received from the server.

If the event is the /who output we just choose to process it. If the event is the end of the /who data we then return a value that tells jIRCii we want to deregister the listener (4) and halt the event (2).

Internally jIRCii processes event listeners based on their return value. A value of 4 means remove the event listener. A return value of 2 means halt the event. A return value of 1 means continue processing other events. Returning 5 (1 + 4) means continue processing other events but remove this listener. Returning 6 (2 + 4) means halt processing of other events and remove this listener.

This form of event hacking is only effective on irc server related events. Other events such as gui events are processed differently.

Scripted Sets

A popular feature of scripts in the past was a built in theme system. A theme system allowed scripters to customize any echo inside the script without altering the functionality of the scripts features. jIRCii has a built in theme system that is accomplished with scripted sets. Nearly every echo event that is echoed to the screen is defined in a scripted set. Altering these sets allows you

to change the way jIRCii looks completely without altering the functionality of jIRCii.

The basic format of a set is:

```
set SET_NAME { return "formatted echo"; }
```

It is of course possible to use if statements, call functions, and anything else within a set. The important thing is that the set returns a string that formats the data from an event into something presentable (preferably cool looking).

A set that returns nothing will simply not be echoed. It is possible to disable an echo for a set by having the set return nothing.

Time Stamping

If time stamping is enabled the set TIMESTAMP is automatically attached to the beginning of all of your set output. To disable time stamping for a specific set append an ! exclamation point to the set name.

```
set NOTIFY_SIGNON! { return "*** $nick has signed on at $time"; }
```

In the above set for a notify signon we already have the time in the echo. There is no point in adding a timestamp even if time stamping is enabled. Adding the ! exclamation point disables time stamping for that specific set.

List of Sets

* = set has a special variable associated with it.
+ = set has a note associated with it.

Channel Event Sets

CHANNEL_TOPIC_CHANGED
CHANNEL_MODE
CHANNEL_JOIN
CHANNEL_KICK
CHANNEL_PART
USER_QUIT

Channel Information Sets

CHANNEL_BANLIST
CHANNEL_BANLIST_END
CHANNEL_CREATED *
CHANNEL_MODE_IS
CHANNEL_NAMES
CHANNEL_TOPIC_IS
CHANNEL_TOPIC_SETBY *
FORMATTED_NAMES * +
FORMATTED_NAMES_HEADER *
JOIN_SYNC *

FORMATTED_NAMES	Called for each name in a /names reply. jIRCii provides special word wrapping for /names replies.

DCC Sets *

CHAT_CLOSED
CHAT_MSG
CHAT_OPEN
SEND_CHAT
SEND_CHAT_ERROR
DCC_LIST_INFORMATION +
DCC_LIST_NICK +
DCC_LIST_TYPE +

DCC_REQUEST
SEND_COMPLETE
SEND_DCC
SEND_FAILED
SEND_START
RESUME_FAILED
RESUME_REQUEST
RESUME_REQUEST_ERROR
RESUME_SUCCEEDED
RECEIVE_COMPLETE
RECEIVE_FAILED
RECEIVE_START
RESOLVED_LOCALINFO

DCC_LIST_* These sets are used for the /dcc stats window. Each set represents a column in the window.

Messages Received

ACTION
ACTION_INACTIVE
CHANNEL_TEXT
CHANNEL_TEXT_INACTIVE
CTCP_REPLY *
CTCP_REQUEST *
INVITE
NOTICE
PRIVACTION
PRIVMSG

Miscellaneous Sets

NICKLIST_FORMAT * +
ON_CHANNELS
SBAR_LEFT *
SBAR_RIGHT *
SET_IGNORE
SET_NOTIFY
TIMESTAMP

NICKLIST_FORMAT used in the channel listbox to format the list of names; $nick and $channel are available in this set.

Other Events

IDENT_REQUEST
IRC_ATTEMPT_CONNECT
IRC_CONNECT
IRC_DISCONNECT
IRC_RECONNECT
NUMERIC +
NOTIFY_SIGNON
NOTIFY_SIGNOFF
PROCESS_DATA +
REPL_*nnn* +
RESOLVED_HOST
SERVER_ERROR
USER_NICK
USER_MODE

NUMERIC fired for any numeric that does not have a set associated with it
PROCESS_DATA data printed out by a process launched using /run or /exec
REPL_*nnn* a set for a specific numeric where nnn is the number of the numeric.

Send Messages

SEND_ACTION
SEND_ACTION_INACTIVE
SEND_CTCP
SEND_MSG
SEND_NOTICE
SEND_TEXT
SEND_TEXT_INACTIVE
SEND_WALL
SEND_WALLEX

Set Variables

Variable	Notes
$channel	Available in NICKLIST_FORMAT, for sets based on channel events use $target.
$created	Available in CHANNEL_CREATED set, channel creation time formatted
$line	Available in SBAR_LEFT, SBAR_RIGHT, equal to the line number of the statusbar starting at 0 for the first line.
$parms	The parameters related to the event (available in most sets)
$pt	Available in CTCP_REPLY set when $0 is PING, set to the ping time
$query	Available in SBAR_LEFT, SBAR_RIGHT - equal to the $active for the window the statusbar is in. Use instead of $active in SBAR sets..
$seton	Available in CHANNEL_TOPIC_SETBY set to time date stamp of when channel topic was set
$sync	Available in JOIN_SYNC set, set to the join sync time
$this	Available in DCC Related sets, used to reference the exact dcc connection using $dcc = getDCCConnection($this);
$total	Available in FORMATTED_NAMES and FORMATTED_NAMES_HEADER, set to the total number of users in the / names reply.
$window	Available in SBAR_LEFT, SBAR_RIGHT - the window the statusbar set is being fired for

If a set is in response to an irc event then $variable's available to the event will also be available to the set

The variables listed in the Aliases -> Built in Variables section also applies to sets as well. Generally $0 is the target/source of the set event. $1- are the rest of the parameters.

Menus

In jIRCii it is possible to script menus for most places in the client. Popup menus are made visible by right clicking in a popup menu hotspot. The popup menu hotspots in jIRCii are *background*, *channel, dcc*, *input, list, nicklist*, *query*, *status, switchbar*, and *tab*.

The *background* hotspot refers to the background when jIRCii is using an MDI based interface. The *tab* hotspot refers to the server tab for a specific server connection. The *input* hotspot refers to the editbox in jIRCii windows. The *list* hotspot refers to the /list dialog. The *switchbar* hotspot refers to the buttons in the window switchbar.

To setup a popup menu for a hotspot you use the menu keyword:

```
menu hotspot
{
   item "item title" { commands }
   menu "menu title" { commands | more item/menu declarations }
}
```

Within the menu keyword you can use the menu keyword again to create a submenu. You can also use the item keyword followed by a title to add an item to the menu. Menus in jIRCii are all created dynamically. That is each time a popup menu is to be shown the menu scripts are executed. Submenu scripts are not executed until they are clicked on. Item scripts are not executed until they are clicked on.

The function addSeparator() can be used inside of the menu keyword to insert a separator in the

menu.

The function addItem("item title", "/command args") can be used to quickly add an item to a menu.

To add a menu to the nicklist:

```
menu nicklist
{
    item "Whois" { call("/whois $1"); }
    item "Query" { call("/query $1"); }
}
```

In the above example two menu items are added to the nicklist hotspot. These are "Whois" and "Query". Whenever "Whois" is clicked on the script inside the { } curly braces will be evaluated. In this case it is call("/whois $1"). In nicklist menus $1 and $snick refer to the nickname that is currently highlighted. $0 represents the current channel. $2- represents the nicknames of the other highlighted nicknames.

When a menu item declared with the keyword item is executed the local variable $command is available. $command contains the item label for the executed menu item. The commands declared inside of a menu or item block are only executed when that menu or item is activated. Local variables declared outside of a menu or item block are not available when they are executed.

Using the menu keyword with a menu hotspot does not erase the menus currently in place. It just adds your menu items to the hotspot along with other menu items already there.

Menubar Menus

Menubar menus can be scripted in jIRCii as well. In fact menubar menus work almost the same as popup menus. The difference is you use the menubar keyword to add an item to the menubar.

```
menubar "menu title"
{
   item "item title" { commands }
   menu "menu title" { commands | more item/menu declarations }
}
```

If a menu title already exists in the menubar then the menu will be combined with the already existing menu. If a menu title does not exist then the menu is added to the menubar right before the Window and Help menu items. You can add to the Window and Help menus using "&Window" or "&Help" for the menu title.

Menubar menus are dynamic just like popup menus. Every time the "menu title" is shown the items and submenus are dynamically added.

The function refreshMenubar() can be used to refresh the top-level menubar.

Adding a Mnemonic

You can use the & ampersand character in "item title" and "menu title" to set a mnemonic for the menu item. The focus accelerated character will be underlined. Users of your script can then use

their platform specific keyboard combination to access your menu item when the item's parent menu is showing. In Windows this combination is Alt+(focus accelerator).

Built-in Client Resources

jIRCii has a number of bundled "resource" files that make up the jIRCii experience. These include images that define the toolbar, the window icons, and even the jIRCii logo.

Copies of all of these files are contained within the jerk.jar file (which is just a ZIP compressed archive).

To override a jIRCii built-in "resource" with one of your own, simply create your resource and copy it to the .jIRC directory on your computer where jIRCii is keeping all of your settings.

For example default.irc is the default script the drives jIRCii. To override the internal version of this resource:

1. create your own version or edits to default.irc
2. copy it to your .jIRC folder (on Linux, OS X /home/<your user>/.jIRC, on Windows c:\documents and settings\<your user>\.jIRC)
3. restart jIRCii

When loading a resource jIRCii looks in the .jIRC folder first before using its own internal version of the resource.

Editing the Default Scripts

Internally jIRCii has two scripts loaded. default.irc and menus.irc default.irc contains all of the default sets and a few features for jIRCii. menus.irc contains all of the popup and menubar menus within the client.
Copies of these two files are distributed with jIRCii under the extras/ directory in the jIRCii archive. The real versions of the default scripts are contained inside of the jerk.jar file.

It is possible to force jIRCii to load modified versions of menus.irc and/or default.irc. Simply modify the copies of default.irc and menus.irc and save the modifications to the jIRCii settings directory. On UNIX the jIRCii settings directory is located in your home directory under .jIRC. In Windows the jIRCii settings directory is located in something like c:\documents and settings\your-username\.jIRC\.

When jIRCii tries to load default.irc or menus.irc it will first look in the .jIRC directory before looking in the jerk.jar file.

If you are creating a "full script" and want to disable the loading of default.irc and menus.irc there are two internal properties "load.default" and "load.menus". These two internal properties can be set to true or false to enable or disable the loading of the default.irc or menus.irc file. To set internal properties look up the setProperty() function.

Generally any .irc file loaded takes precedence over anything in the default.irc and menus.irc. It should be possible to write full featured scripts without disabling the two internal scripts. However if you need to do it now you know how.

Built in Functions and Operators

Conventions

$ add(@array, $scalar, [index])

The $ to the left of the function name represents the return type of the function. Some functions will return an array (@), some a hash (%), and others a scalar ($). Functions marked with (?) will return 1 if the value is true or the empty scalar if the value is false. Functions that have no return value will have nothing to the left.

Optional parameters will be enclosed in [ ] square brackets.

Channel Functions

jIRCii supports the following comparison operators for use in if statements with regards to channels:

$nick hasmode “#channel”	true if the specified $nick has a mode on “#channel”
$nick ison "#channel"	true if the specified $nick is on the "#channel"
$nick isop "#channel"	true if the specified $nick has operator status on "#channel"
$nick isvoice "#channel"	true if $nick has voice (+v) status on "#channel"
$nick ishalfop "#channel"	true if $nick has halfop (+h) status on "#channel"
$nick isnormal "#channel"	true if $nick has no status on "#channel"
"m" ismode "#channel"	true if "m" is a set mode on "#channel"
-ischannel "string"	true if "string" is a valid channel string i.e. #string

jIRCii provides the following functions for retrieving channel information:

@	getHalfOps("#channel")	returns an array of all +h users on #channel
@	getNormal("#channel")	returns an array of all no status users on #channel
@	getOps("#channel")	returns an array of all +o users +o on #channel
@	getUsers("#channel")	returns an array of all the users on #channel
@	getVoiced("#channel")	returns an array of all +v users on #channel
$	getModeFor($nick, “#channel”)	returns the mode character of $nick on #channel
$	getKey("#channel")	returns the key for #channel
$	getLimit("#channel")	returns the user limit for #channel
$	getMode("#channel")	returns the mode string for #channel i.e. "+stn"
$	getTopic("#channel")	returns the topic for #channel

Colormap Functions

These functions are provided for manipulating the jIRCii colormap. jIRCii allows colors 17-99 to be user defined as any color the user wants. These colors are saved in the colormap.

The "aarrggbb" format is a hexadecimal format for specifying a color. Similar to what you would use for specifying a color on a webpage. The one caveat is rather than saving this number in hex form jIRCii deals with colors as integers.

To convert a number from a hex triplet to the “aarrggbb” format try:

```
formatNumber('hex triplet', 16, 10);
```

	generateThemeScript("file.thm")	exports all jIRCii color information and some configuration stuff to a importable theme script
$	getMappedColor(index)	returns the "aarrggbb" value as an integer for the color at the specified index.
	saveColorMap()	forces a save of the color map.
	setMappedColor(index, "aarrggbb")	sets the color at index to the specified "aarrggbb" integer value.

Config System Functions

The following functions are used to interface with jIRCii's user preferences sub-system. All user preferences are stored using this system. These functions make it possible to test for any user preference or change any user preference. jIRCii preferences are generally stored in a file called jirc.prop in the .jIRC folder located in the users home folder.

The config system supports the following comparisons for user within if statements

-isSetT "propery.key"	Checks if the property.key is flagged to true. Using true as a default value if property.key does not exist.
-isSetF "property.key"	Checks if the property.key is flagged to false. Using false as a default value if property.key does not exist.

The following functions are provided for interfacing with the config system:

$	baseDirectory()	returns the base directory where all of the jIRCii preferences and files are stored.
$	getProperty("key", ["default value"])	retrieves the specified key from the config system
@	getPropertyList("key")	returns a property list (a way of storing an array).
	setProperty("key", "value")	sets the specified key to the specified value
	setPropertyList("key", @array)	stores @array as a property list
$	versionString()	returns the client version string

Setting a property via the setProperty() or setPropertyList() methods will generally take immediate effect in the client. The stuff being configured in jIRCii is wired to know when its property gets changed. For a list of *some* of the properties you can change see Appendix A - Client Configuration Variables.

Date/Time Functions

In general dates are represented using a scalar long that holds the number of milliseconds since the epoch. The epoch is January 1st, 1970. Some of the functions for date/time manipulation require a scalar long as a parameter. That parameter is represented as *date* without any " quotes.

jIRCii provides the following functions for date/time manipulation:

$	ctime()	returns the number of seconds since the epoch
$	duration(seconds)	converts the seconds into a string describing exactly how long the time has been i.e. 3 hours 20 minutes 15 seconds

$	formatTime(seconds)	converts the seconds into a string describing *roughly* how long the time has been i.e. 3 hours **or** 20 minutes **or** 15 seconds.
$	formatTime2(seconds)	returns a string with seconds reduced to dd:hh:mm:ss form
$	timeDateStamp(date)	quick convienence function to turn the specified date into a time/date string.
$	timeStamp()	returns a timestamp for the current time. same value as $time.

DCC Functions

Many of the dcc functions return a $scalar that contains a reference to the dcc connection information. This reference can be queried using functions that take a $dcc_connection as a parameter. Its helpful to know that DCC's come in three varieties: CHAT, RECEIVE, SEND. DCC's can be in one of three states: CLOSED, OPEN, or WAIT.

jIRCii provides the following comparison operator for use in if statements with regards to DCC's:

-isdccopen $dcc_conn true if the dcc connection specified is in fact open.

jIRCii provides the following functions for getting information about the current DCC's:

@	getActiveConnections()	returns all of the active connections
@	getAllConnections()	returns all of the connections
@	getInactiveConnections()	returns all of the inactive connections
@	getWaitingConnections()	returns all of the waiting connections
$	getDCCConnection($this)	dcc events/sets have $this set within them. You can use this function to resolve a $dcc_conn from a $this.
$	getSpecificConnection("nick", type)	
	closeDCC($dcc_conn)	closes the specified dcc
$	getConnectionState($dcc_conn)	returns the state for the specified connection
$	getConnectionType($dcc_conn)	returns the type of the specified connection
$	getLocalPort($dcc_conn)	returns the local port of the dcc connection
$	getRemotePort($dcc_conn)	returns the remote port of the dcc connection
$	getDCCAddress($dcc_conn)	returns the remote ip address of the dcc.
$	getDCCIdleTime($dcc_conn)	returns the idle time of the dcc in milliseconds
$	getDCCNickname($dcc_conn)	returns the nickname for the dcc
$	getDCCStartTime($dcc_conn)	returns the start time as a scalar long
$	getDCCTotalTime($dcc_conn)	returns the total time the dcc has been active in seconds
$	getNextPort()	returns the next dcc port that will be used
$	localip()	returns your localip as determined by jIRCii

Send and Get Specific

$	getDCCFileName($dcc_conn)	returns the filename associated with tihs dcc
$	getDCCFilePath($dcc_conn)	returns the full path to the file for this dcc
$	getFileSizeOffset($dcc_conn)	returns the file size offset in case the file was resumed.
$	getTimeRemaining($dcc_conn)	returns the estimated time left for this transfer (in seconds)
$	getTransferRate($dcc_conn)	returns the total transfer rate in bytes/second

Get Specific

$	getBytesReceived($dcc_conn)	returns the total number of bytes received
$	getExpectedSize($dcc_con)	returns the total expected size for the file

Send Specific

$	getAcknowledgedSize($dcc_conn)	returns the number of bytes acknowledged
$	getBytesSent($dcc_conn)	returns the total number of bytes sent

Dialog Functions

	showAboutDialog()	shows the about dialog
$	showDirectoryDialog("title", ["initial"], ["ok text"])	shows a directory chooser dialog
$	showFileDialog("title", ["initial"], ["ok text"])	shows a file chooser dialog
	showHelpDialog()	shows the help dialog
$	showInputDialog("title", "text")	shows an input dialog
	showOptionDialog(["option item"])	shows the option dialog
	showSearchDialog([“window”])	opens up the search dialog for the specified window
	refreshData($dialog)	refreshes the contents of the specified sorted $dialog.
$	showSortedList("title", "__menu", @data, "col", ...)	returns a $dialog

opens a sorted list window similiar to the /list -gui or /dcc stats windows. the "__menu" option specifies the toplevel popup hook name. it must begin with "__". @data is an array containing all of the data inside of the sorted list. each row is represented by one element of @data. each column is separated by the "\t" special character. it is important that each row has column data separated by "\t" for each specified column. the parameters after @data are the columns in the sorted list window. one argument for each column. if the contents of @data change use refreshData to tell the window that it's contents have changed.

Echo Functions

echo("text")	echoes text to the active window
echo("target", "text")	echoes text to the specified target
echo("target", "text", 1|2)	specifying a 1 echoes text to all relevant windows for the specified target. specifying a 2 does the same thing but guarantees the text will also go to the status window
echoAll("text")	echoes text to all the windows

echoColumns("target", "cols", 0.5)	echoes text as formatted columns. The "cols" parameter is a string of columns separated by the \t (tab) character. The double parameter at the end specifies a percentage of the screen the wordwrapped columns can take up. Think of this as a function that lets you control the wordwrapping. Wrapping happens at the specified percentage of the window. The tabbed columns each represent one wrappable unit. Rather than breaking the string up by spaces as normal.
echoRaw("target", "text")	echoes text to the specified target except there is no event processing (i.e. on window is not called). Use this if you want to echo text from the on window event.
echoStatus("text")	echoes text to the status window

GUI Functions

The string "window" refers to the name of a window you want to access. The Status window has a special name: %STATUS%.

jIRCii provides the following comparison operator for use in if statements with regards to its UI:

-isspecial "window"	true if the specified window is considered special i.e. the /list window or the dcc sessions window.
-iswindow "target"	true if the specified target has a window associated with it.

jIRCii provides the following functions for manipulating its user interface:

GUI General

@	getClipboardText()	returns the clipboard text
	setClipboardText("string")	puts "string" into the system clipboard
	openCommand("some file/url")	uses the open command for the users platform to open the specified url or file with the app meant to handle it.
	loadFont("/path-to/fontfile.ttf")	loads the specified font file into the java virtual machine. Unable to verify that this works so it might go away one day.
	refreshMenubar()	forces jIRCii to refresh the top level menus for the menubar

Individual Windows

	copySelectedText()	copies selected text from the editbox in current window to the clipboard
	cutSelectedText()	cuts and copies selected text in the current window editbox
$	getSelectedText("target")	returns the selected text from the editbox in the specified window
	removeSelectedText()	clears selected text from the editbox in the current window

	pasteText()	pastes text from clipboard
$	getSelectedUser("target")	returns the selected nickname in the target channel window listbox
@	getSelectedUsers("target")	returns all of the selected users in the target channel window listbox.
$	getButtonColor(“target”)	returns the color of the switchbar button text
$	setButtonColor(“target”, aarrggbb)	sets the color of the switchbar button text
$	getCursorPosition(“target”)	returns the position of the editbox cursor
	setCursorPosition(“target”, n)	sets the position of the editbox cursor
$	getInputText(“target”)	returns the text in the editbox for the target
	setInputText(“target”, “text”)	sets the editbox text for the specified target
$	getWindowPrompt(“target”)	returns the text for the targets window prompt
	setWindowPrompt(“target”, “text”)	sets the window prompt for the specified window
$	getWindowTitle([“target”])	returns the titlebar text for the specified window. If “target” is not specified returns the text of the jIRCii titlebar.
	setWindowTitle([“target”], “text”)	sets the titlebar text for the specified window. Defaults to setting text for the jIRCii titlebar.
	renameWindow(“target”, “title”)	renames target to specified title, right now this works only on query windows
	refreshWindow(“target”)	updates the statusbar and repaint the window immediately.
	scrollWindow(“target”, +/-n)	scrolls the text up (-n) or down (+n)

Window Management

	activateWindow(“target”)	makes the target window active
	cascadeWindows()	cascades all the windows (works in MDI only)
	closeWindow(“target”)	closes the specified window
$	getActiveWindow()	returns the title of the real active window
$	getCurrentWindow()	returns the title of the current window that is safe to echo text too. Uses the active window if the active window isn’t a special window, returns the status window otherwise.
@	getWindows()	returns an array of all the windows
	openWindow(“target”, [1])	opens a window for the specified target, the optional second parameter forces jIRCii to open a non-channel window as inactive.
	tileWindows()	tiles all the windows (works in MDI only)
$	getWindowState([“target”])	returns the state of the target window. Not specifying a target window will return the state of the main jIRCii window

	setWindowState([“target”], “state”)	sets the target windows state to either MINIMIZED, MAXIMIZED, or NORMAL. Not specifying a target will set the state of the main jIRCii window

IRC Command Functions

	call(“/command parms”, [1])	Tells jIRCii to execute the specified command. Specifying a 1 for the second parameter forces jIRCii to use the built in command and avoid the scripted aliases.
	Exit()	perhaps the only command you will ever need.
	fireEvent(“irc data string”)	processes “irc data string” as if it came from the irc server. You can use this to fire custom events.
	fireEvent(“EVENT”, %data)	fires the custom event named “EVENT” with %data as the variables for the event. Each “$key” inside of %data will be available to the listening events.
@	getAliasList()	returns a list of all aliases in the client, lowercase means a built in alias, UPPERCASE means a scripted alias.
$	getServerHost()	returns the hostname of the current server
$	getServerNetwork()	returns the network of the current server
$	getServerPassword()	returns the password used on the current server
$	getServerPort()	returns the port number for the current server
%	getSupportHints()	returns a hash of all 005 ISUPPORT hints specified by the current server
?	isServerConnected()	returns 1 if the server is connected, 0 otherwise
?	isServerSecure()	returns 1 is the connection is SSL, 0 otherwise
	processInput(“text”)	processes “text” as if it was typed in the active window
	sendRaw(“raw text”)	sends “raw text” directly to the server
	say(“message”)	sends “message” to the active target
	sendAction(“target”, “message”)	sends an action to the specified target
	sendMessage(“target”, “message”)	sends a message to the specified target
	sendNotice(“target”, “message”)	sends a notice to the specified target
	sendReply(“target”, “type”, “message”)	sends a ctcp reply to the specified target
	sendRequest(“target”, “type”, “message”)	sends a ctcp request to the specified target

cycleQuery()	forces jIRCii to cycle the /query for the status window to the next non /window'd channel
setQuery("target")	sets the target as the active query for the status window

Internal Data List

The internal data list in jIRCii stores all of the channel and address information for users.

jIRCii supports the following comparison operator for use in if statements with regards to the internal data list:

-isidle "nick" true if the specified nickname has been idle for 5 minutes or more

The following functions are also available:

$	getAddress(["nick"])	returns the address of the specified nick in nick@address format. Blue-elf is a punk.
@	getChannels(["nick"])	returns all of the channels the specified nick is on that you are also on
$	getIdleTime(["nick"])	returns the total idle time of the specified nick in seconds
$	nickComplete("pnick", '#channel')	uses built-in nick completion routine to resolve p(artial)nick to a full nickname from "#channel"
@	nickCompleteAll("pnick", "#channel")	same as above function except returns an array of all matches in order of relevance
@	searchAddressList("*!*@*")	returns an array of nicknames that have addresses that match the specified wildcard string.

Logger Functions

The following functions are provided for querying and accessing the jIRCii built-in logging functionality.

$	getLogFile("window")	returns the full path to the file where messages for the specified "window" are being logged.
	logMessage("window", "text")	writes a message to the log file for the specified window if and only if the user has logging enabled

Multiserver Session Manipulation

The following functions are provided for manipulating the multiserver aspect of jIRCii. Each session in jIRCii has a session id. The first session is 0, the second is 1, etc. Most of these functions are related to manipulating the "tab" for the session in jIRCii's UI.

	activateSession(n)	activate the specified session number
	callInSession(n, "/cmd")	executes the specified command as if it was typed in session number *n*
$	getActiveSessionId()	returns the session id of the current active session

$	getSessionId()	returns the session id of the session executing this script.
$	getTotalSessions()	returns the total number of sessions
$	getSessionColor([n])	returns the session color for session n in "aarrggbb" format, defaults to this session if n isn't specified.
	setSessionColor("aarrggbb")	sets the session color for this session to the specified color.
$	getSessionText([n])	returns the session text for session n or this session
	setSessionText("text")	sets the session text for this session

Notify Functions

The notify feature in jIRCii works much like an instant messaging buddy list. It tells you when someone is online or not.

jIRCii provides the following comparison operators for use in if statements with regard to the notify list.

-isnotify "nick"	true if the specified nick is in the notify list
-issignedoff "nick"	true if the specified nick is signed off
-issignedon "nick"	true if the specified nick is signed on

jIRCii provides the following functions for accessing the notify list:

$	getAddressFromNotify("nick")	returns address of the user from the notify address list
@	getNotifyUsers()	returns all of the users in the notify list
@	getSignedOnUsers()	returns all of the signed on users in the notify list
@	getSignedOffUsers()	returns all of the signed off users in the notify list
$	onlineFor("nick")	returns the number of milliseconds the user has been online for.

Servers.ini Functions

The file servers.ini is used by jIRCii to keep track of irc servers the user may want to connect to. The following functions are used for accessing this list of servers. The functions that return arrays return read-only arrays.

@	getAllServers()	returns an array of "data" strings representing all of the irc servers in the servers.ini file
@	getAllNetworks()	returns an array of "network" names
@	getServersForNetwork("network")	returns all of the server "data" strings for the specified irc network
$	getServerInfo("irc.server.com")	returns the "data" string for the specified server
$	serverInfoCommand("data")	formats the specified server data into a ready to use /server command reflecting the server settings
$	serverInfoHost("data")	extracts the hostname from the "data" string
$	serverInfoPortRange("data")	extracts the port range from the "data" string
$	serverInfoNetwork("data")	extracts the network from the "data" string

?	serverInfoIsSecure("data")	extracts wether or not the specified server is specified as an SSL enabled server
$	serverInfoPassword("data")	extracts the server password from the "data"
$	serverInfoDescription("data")	extracts the description from the "data" string
$	serverInfoConnectPort("data")	extracts the port jIRCii would use to connect to the server specified in the "data" string

Sound Functions

$	loadSound("filename")	loads and returns an audio clip $sound, accepteable sound file types include: .wav, .au, and .mid
	soundLoop($sound)	plays $sound continuously
	soundPlay($sound)	plays $sound once
	soundStop($sound)	stops the playing of $sound

String Formatting Codes

Sleep has a feature called parsed literals. Parsed literals are strings contained inside of " double quotes. A feature of parsed literals is that certain escapes can be applied. An escape is started with a \ backslash and followed by a character. Sleep replaces some escapes with a specific type of text. jIRCii uses this feature of sleep to make it easy to insert text formatting codes into your scripts. A \ backslash followed by a \ backslash is just a backslash.

Escape	Code	Example	Result
\b	bold	"\bthis text is bold\b"	**this text is bold**
\c	color	"\c15,5this text is colored"	this text is colored
\u	underline	"this \uword\u is underlined"	this word is underlined
\o	cancel	"\\o \b\ucancels\o \\b and \\u"	\o **cancels** \b and \u
\r	reverse	"this \rword\r is reversed"	this word is reversed

The \b escape is equivalent to using Ctrl+B, \c is equivalent to using Ctrl+K etc. Also in jIRCii bold is a brighter version of the current color while reverse is a darker version of the current color.

Remember these escapes only work within " double quoted strings.

String Functions (Miscellaneous)

$	buildCP437String("text")	takes specified text and remaps characters 128-255 to characters specified in the CP437 charset. The CP437 charset is the default mapping for fonts like Terminal, Lucida Console, and other "ansi" fonts.
@	fileCompleteAll("partial-file-name")	returns an array of all files matching partial-file-name in partial-file-name's directory.
$	formatBytes(n)	returns a string with n reduced to the appropriate units i.e. 3mb or 4kb
$	formatDecimal(n)	returns a string with a decimal number going to the thousandths place.
$	getScriptPath("script.irc")	returns the location of the specified loaded script.
$	getScriptResource("script.irc", "file")	returns the full-path to a file in the same directory as script.irc.

@	groupNicks(@array, n)	breaks @array into an array of comma separated nickname groups of n size.
$	longip("ip")	if input is an ip address in ip.ip.ip.ip format this will convert it to a long ip. If input is an ip address in long ip format this will covert it to an ip address in ip.ip.ip.ip format.

$	mask("nick!user@host.domain", 0-9)	converts the passed in address into a masked format. The type of mask depends on the second parameter:
		`0: *!user@host.domain`
		`1: *!*user@host.domain`
		`2: *!*@host.domain`
		`3: *!*u@*.domain`
		`4: *!*@*.domain`
		`5: nick!user@host.domain`
		`6: nick!*user@host.domain`
		`7: nick!*@host.domain`
		`8: nick!*user@*.domain`
		`9: nick!*@*.domain`

$	parseSet("SET", "target", "parms")	fires the custom "SET" with $0 = to the target and $1- = to the parameters.
$	parseSet("SET", %data)	fires the custom/real "SET" with $variables being specified in the %data hashmap.
$	strip("text")	returns a copy of "text" with all of the formatting codes stripped out.

String Functions (Tokenizing)

jIRCii features a powerful built in string tokenizer. Strings are tokenized using the tokenize() function. A tokenized string is a string that is split up into parts (called tokens) by a specified delimiter. The return value of the tokenize function is referred to as $tokens.

jIRCii supports the following comparison operator for working with tokenized strings:

"string" istoken $tokens true if "string" is a token inside of the tokenized string

The following functions exist for working with tokenized strings:

$	findToken($tokens, "string")	returns the index of token "string"
@	getAllTokens($tokens)	returns an array of all the tokens
$	getToken($tokens, n)	returns the token at index n
$	getTokenFrom($tokens, n)	returns all tokens starting at index n (preserving the delimeter)
$	getTokenRange($tokens, n, m)	returns a range of tokens from index n to m (preserving the delimiter)
$	getTokenTo($tokens, n)	returns all tokens from the beginning up to index n (preserving the delimiter)
$	getTotalTokens($tokens)	returns the total number of tokens in this string
$	tokenize("string", ["delimeter"])	converts "string" into a series of tokens split apart by the specified "delimiter". The default delimiter is a space " ". Returns $tokens

Timers

A timer in jIRCii is a way of executing a set of commands when a certain interval of time has passed.

$ addTimer(&closure, n, [r], [$scalar])	adds a timer that executes &closure in an on going manner, it executes every n milliseconds and repeats r times. The $timer scalar return is a reference to this specific timer. If specified, $scalar is passed as an argument to &closure each time the timer is executed.
setTimerResolution(n)	Sets the overall timer resolution to n milliseconds. The timer resolution is how often jIRCii checks to see if there is a timer that needs to be fired. Obviously lower timer resolutions can mean poorer performance on a slower computer.
stopTimer($timer)	stops the specified timer.

Appendix A – Client Configuration Variables

This list is by no means a comprehensive list of all the configuration variables within jIRCii. The best thing to do is view the file jirc.prop after you've played around with the options for awhile. All config options in jIRCii have default values. These default values are not automatically saved to the config file.

You can set these values from within jIRCii using:
/eval setProperty(*"property"*, *value*)

active.*	"true", "false"; determines wether messages are echoed to active The * portion refers to: ctcp, notice, notify, reply, query, whois
auto.*	"true", "false"; auto /window options The * refers to: chat, chatclose, hide, join, part, query
auto.connect	@list of servers that are automatically connected to on client startup
client.encoding	name of charset encoding jIRCii uses for text coming from irc server
dcc.exists	0, 1, 2, 3; what to do when a dcc send occurs and the file exists 0 = "Ask", 1 = "Overwrite", 2 = "Resume", 3 = "Ignore"
dcc.high	integer; the high port in the dcc port range
dcc.localinfo	"Server Method", "Normal Method", or use an ip address how to resolve or what ip address is to be used for dcc local info
dcc.low	integer; the low port in the dcc port range
dcc.onchat	0, 1, 2; what to do when a dcc chat request comes in 0 = "Ask", 1 = "Auto Accept", 2 = "Ignore"
dcc.onsend	0, 1, 2; what to do when a dcc send request comes in 0 = "Ask", 1 = "Auto Accept", 2 = "Ignore"
dcc.saveto	the directory received files are saved to
ident.system	the reply returned by the ident server for specifying your OS
ident.userid	the userid for the ident server
ident.port	integer; the port number for the ident server
ignore.masks	@list; all of the masks in the ignore list
kick.message	the default kick reason
listbox.enabled	"true", "false"; enable / disable the channel nickname listbox

listbox.position	0, 1; set the position of the channel listbox 0 = left, 1 = right
load.default	“true”, “false”; enable / disable loading of the default script
load.menus	“true”, “false”; enable / disable loading of the menus script
log.enabled	“true”, “false”; enable / disable logging of irc chats
log.saveto	the directory log files are saved to
log.strip	“true”, “false”; enable / disable stripping of text attributes from logged text.
Log.timestamp	“true”, “false”; enable / disable time stamping of all logged text.
Message.quit	the default quit message
notabs.border	int; size (px) of border around jIRCii with server tabs disabled
notify.users	@list; all of the users on the notify list
option.showmotd	“true”, “false”; enable /disable the /MOTD output
option.timestamp	“true”, “false”; enable / disable timestamping
perform.enabled	“true”, “false”; enable / disable the perform on connect feature
reconnect.time	int; number of seconds to sleep between reconnect attempts
script.files	@list; all of the current loaded scripts with full pathnames
switchbar.enabled	“true”, “false”; enable / disable the switchbar
switchbar.hilight	“true”, “false”; enable / disable switchbar highlighting on activity
switchbar.fixed	“true”, “false”; enable / disable fixed width buttons in the switchbar
switchbar.position	0, 1; set the position of the switchbar 0 = top, 1 = bottom
switchbar.sort	“true”, “false”; enable / disable auto-sorted switchbar buttons
ui.buffersize	int; maximum size of jIRCii scrollback buffer
ui.font	“*FontName*-PLAIN-*FontSize*”; font used by jIRCii
ui.native	“true”, “false”; enable / disable the native look and feel does not take immediate effect.. jIRCii must be restarted
ui.openfiles	the application to use when opening files on the users specific OS
ui.sbarlines	integer; number of lines for the statusbar
ui.sdi	“true”, “false”; enable / disable the single document interface mode does not take immediate effect. jIRCii must be restarted
ui.showbar	“true”, “false”; show the menubar… be careful about disabling this
ui.showsbar	“true”, “false”; show the statusbar.. setting ui.sbarlines to 0 does the same thing.
ui.showtabs	“true”, “false”; show the server tabs
ui.usetoolbar	“true”, “false”; shows the lame newbie toolbar…
update.ial	“true”, “false”; update internal address list when joining a channel (done by performing a /who #channel when joining)
user.altnick	the users altnick from the setup dialog
user.email	the users email address from the setup dialog
user.nick	the users nickname from the setup dialog
user.rname	the users real name from the setup dialog
version.addons	if set the contents of this variable will be reported with jIRCii's version information rather than a goofy tagline
version.string	setting this variable will alter the internal version string for the client