User Manual

Open Source IRC Client

Author Daniel Hetrick
License GPL 3 (explained)
Source Code https://github.com/nutjob-laboratories/merk
Last modified Monday, October 20, 2025

Table of Contents

Summary

IRC (**Internet Relay Chat**) is a text-based chat system for instant messaging. IRC is designed for group communication in discussion forums, called *channels*, but also allows one-on-one communication via private messages… .

Internet Relay Chat is implemented as an application layer protocol to facilitate communication in the form of text. The chat process works on a client–server networking model. Users connect, using a client—which may be a web app, a standalone desktop program, or embedded into part of a larger program—to an IRC server, which may be part of a larger IRC network. Examples of ways used to connect include the programs Mibbit, KiwiIRC, mIRC and the paid service IRCCloud.

From the Wikipedia entry on IRC, at https://en.wikipedia.org/wiki/IRC

MERK is a free and open source Internet Relay Chat client for Windows and Linux. It uses a "multiple document interface", in which the application works as a parent window that contains other windows for servers, channels, and private chats. The popular Windows shareware client mIRC is an example of another IRC client that uses a multiple document interface.

MERK is written in the Python programming language, using the PyQt library for the graphical interface and the Twisted library for networking. MERK also comes bundled with three other open source libraries:

- **qt5reactor**, for getting PyQt and Twisted to work together
- **pyspellchecker**, which provides the spellchecking mechanism
- **emoji**, providing support for emoji shortcodes

MERK has a scripting engine allowing most functionality to be automated. The core concept of the scripting engine is the **context**: a context is a window, either a channel, private chat, or server window, that the commands executed are intended to interact with. Scripts can be executed on connection, or executed from text input. MERK comes with a script editor with features to make writing scripts easy and fun, with no prior programming experience required.

As IRC is a text-based protocol, MERK features a rich text display, which can be easily configured. MERK supports the display of mIRC colors[1], which can optionally be stripped from messages.

1 https://en.wikichip.org/wiki/irc/colors

Running MERK

MERK comes in two different versions: the Python version, which uses the Python interpreter to run MERK, and a PyInstaller version of MERK, which runs on Windows, and doesn't require the Python interpreter. Both versions behave exactly the same way, and have only minor differences.

PyInstaller Version

Download the Windows version of MERK, and unzip the archive to anywhere you'd like. The archive contains a folder named **`lib`**, the **`merk.exe`** executable, and **`README.html`**. Just double click **`merk.exe`**, to run MERK. That's it!

There's also an installer for MERK. Download the Windows installer version and unzip the archive to wherever you'd like. Double click on **`setup.exe`**, which will guide you through installing MERK on your computer. MERK can be installed wherever you'd like, and can either be installed for a single user, or for all users on your computer.

Python Version

MERK requires several libraries to be installed in order to run: **`Python 3.9+`**, **`PyQt`**, **`Twisted`**, and if you'd like to connect to servers via SSL/TLS, **`PyOpenSSL`** and **`service_identity`**. If you're running MERK on Windows, you may also need **`pywin32`**. All of these libraries can be installed easily with PIP, the Python package installer. To install the base requirements, open a terminal, and enter:

```
pip install PyQt
pip install Twisted
pip install PyOpenSSL
pip install service_identity
```

If you're using MERK on Windows, also enter:

```
pip install pywin32
```

Once all the requirements are installed, unzip the downloaded archive of MERK, use the terminal to navigate to the directory you unzipped MERK to, and type:

```
python merk.py
```

Updating MERK

As MERK stores all its configuration files separate from the executable/installation, updating MERK to the latest version is easy.

Zip File Version

You can do this one of two ways: either delete **`merk.exe`** and the **`lib`** folder, wherever you extracted them do, and unzip the new version of MERK in the same folder; or unzip the new version of MERK in the same directory and overwrite all files.

Installer Version

This version of MERK is even easier to update. Just download the installer of the newer version of MERK, unzip **`setup.exe`**, and double click on it. You don't have to uninstall the older version, the new version will overwrite the old one.

Command-Line Arguments and Options

The command-line interface of MERK works identically on all platforms.

```
usage: python merk.py [--ssl] [-p PASSWORD] [-c CHANNEL[:KEY]] [-C SERVER:PORT[:PASSWORD]]
                      [-S SERVER:PORT[:PASSWORD]] [-n NICKNAME] [-u USERNAME] [-a NICKNAME]
                      [-r REALNAME] [-h] [-d] [-x] [-t] [-R] [-o] [-f] [-s FILE]
                      [--config-name NAME] [--config-directory DIRECTORY] [--config-local]
                      [--scripts-directory DIRECTORY] [--user-file FILE]
                      [--config-file FILE] [--reset] [--reset-user]
                      [--reset-all] [-Q NAME] [-D] [-L]
                      [SERVER] [PORT]

Connection:
  SERVER                Server to connect to
  PORT                  Server port to connect to (6667)
  --ssl, --tls          Use SSL/TLS to connect to IRC
  -p, --password PASSWORD
                        Use server password to connect
  -c, --channel CHANNEL[:KEY]
                        Join channel on connection
  -C, --connect SERVER:PORT[:PASSWORD]
                        Connect to server via TCP/IP
  -S, --connectssl SERVER:PORT[:PASSWORD]
                        Connect to server via SSL/TLS

User Information:
  -n, --nickname NICKNAME
                        Use this nickname to connect
  -u, --username USERNAME
                        Use this username to connect
  -a, --alternate NICKNAME
                        Use this alternate nickname to connect
  -r, --realname REALNAME
                        Use this realname to connect

Options:
  -h, --help            Show help and usage information
  -d, --donotsave       Do not save new user settings
  -x, --donotexecute    Do not execute connection script
  -t, --reconnect       Reconnect to servers on disconnection
  -R, --run             Don't ask for connection information on start
  -o, --on-top          Application window always on top
  -f, --full-screen     Application window displays full screen
  -s, --script FILE     Use a file as a connection script

Files and Directories:
  --config-name NAME    Name of the configuration file directory (default: .merk)
  --config-directory DIRECTORY
                        Location to store configuration files
  --config-local        Store configuration files in install directory
  --scripts-directory DIRECTORY
                        Location to look for script files
  --user-file FILE      File to use for user data
  --config-file FILE    File to use for configuration data
  --reset               Reset configuration file to default values
  --reset-user          Reset user file to default values
  --reset-all           Reset all configuration files to default values

Appearance:
  -Q, --qtstyle NAME    Set Qt widget style (default: Windows)
  -D, --dark            Run in dark mode
  -L, --light           Run in light mode
```

Using Command-Line Options

MERK's command-line options allow users to do many things on startup. All of these uses are completely optional, and never have to be used. Most command-line options feature a long version (for example **`--donotexecute`**) and a shorter version (**`-x`**, which does the same thing).

If user settings are in place (that is, the default nickname, username, etc), command-line options can be used to connect to one or more IRC servers automatically on startup. For example, to automatically connect to the DALnet IRC network, you can use:

```
merk.exe us.dal.net 6687
```

This will automatically connect to DALnet, executing any connection script previously set up with MERK. To prevent the connection script from executing, try:

```
merk.exe --donotexecute us.dal.net 6667
```

Multiple servers can be connected to, as well, though the method is a little different. Use the **`-C`** option to connect to normal IRC servers, and the **`-S`** option to connect to IRC servers via SSL/TLS. In the next example, we're going to connect to the Libera network via SSL/TLS, and DALnet:

```
merk.exe -S irc.libera.chat:6697 -C us.dal.net:6667
```

If you want MERK to skip asking for a server to connect to on startup, use the **`--run`** option:

```
merk.exe --run
```

MERK can even be configured to run on a USB thumb drive! For this example, assume that MERK has been extracted into the root directory of a USB thumb drive. In the same directory as **`merk.exe`**, create a text file, and type this into it:

```
merk.exe --config-local
```

Save this file as **`merk.bat`**. Now, to run MERK off of the thumb drive, double click on **`merk.bat`** (which is a Windows batch file[2]). This will run MERK normally, but store all of the configuration files in a folder named **`.merk`** in the drive where you're running MERK from. MERK is now completely portable!

2 https://en.wikipedia.org/wiki/Batch_file

Resetting MERK to Default Settings

If your installation of MERK becomes unusable or for any other reason, you can reset MERK back to default settings with the following command-line option:

```
python merk.py --reset
```

If you are running MERK with the PyInstaller executable, use:

```
merk.exe --reset
```

To reset all user settings, use the **`--reset-user`** command-line option. This will remove all user settings, including your nickname, alternate username, username, realname, connection history, and any connection scripts:

```
python merk.py --reset-user
```

To reset *all* settings, and return MERK configuration files to their default state with all default settings, use **`--reset-all`**. This will reset both your user file, **`user.json`**, and the settings file, **`settings.json`**, to all default values.

```
merk.exe --reset-all
```

Directories and Configuration Files

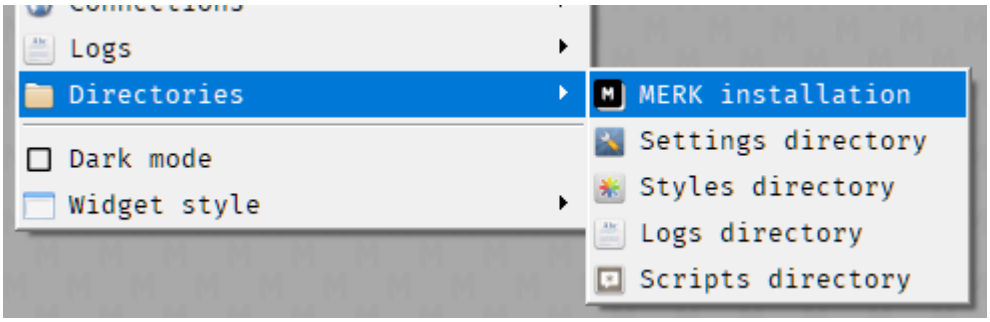
MERK stores all its settings in a directory it creates in the user's home directory, named **`.merk`**. Inside this directory, MERK creates:

- **`logs`**. This directory is where MERK stores channel and private chat logs.
- **`styles`**. This directory is where MERK stores text style files, and the palette used for dark mode.
- **`scripts`**. This directory is where MERK stores, and first looks for, scripts. This is the default directory chosen when running a script via the server window toolbar, input menu, or right click menus, or when saving a script in the editor.
- **`settings.json`**. This file is where MERK stores and loads application settings.
- **`user.json`**. This file is where MERK stores user information, such as the chosen nickname, username, and the like, as well as the application's connection history and any connection scripts.

When using the **`/script`** command, if a full filename is not provided, MERK will look for the script in several locations, in order:

1. The **`scripts`** directory.
2. The settings directory (by default, **`.merk`** in the user's home directory).
3. The application's installation directory.

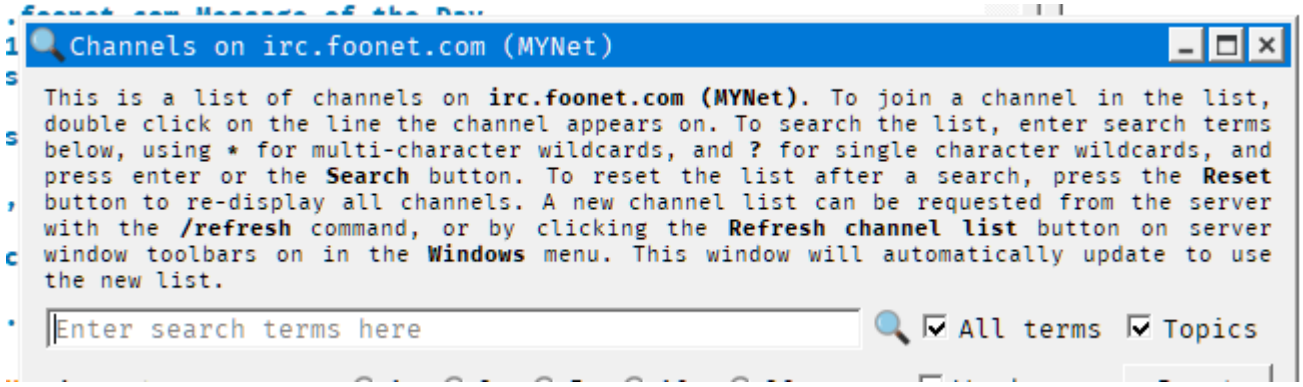
First, MERK will attempt to find the script using the provided filename, and if the script is still not found, it will append the default file extension (which is **`.merk`**) to the filename and search again. This same pattern is used with the **`/edit`** and **`/insert`** commands.



These folders can be opened in your default file manager from the client by clicking on the appropriate entry in the "Directories" sub-menu, near the bottom of the "Settings" menu.

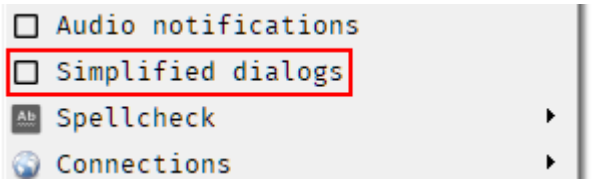
New User Help

Most dialogs feature text explaining how the dialog or the settings in it work.

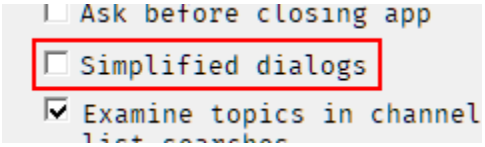


The explanation text from the channel list dialog.

While new users of MERK may find these helpful, experienced users may not want to see them. To hide the help text on dialogs, turn on "Simplified dialogs" in the settings menu or the settings dialog:



The "Simplified dialogs" option in the "Settings" menu. Click this entry to simplified dialogs on and hide the help text.

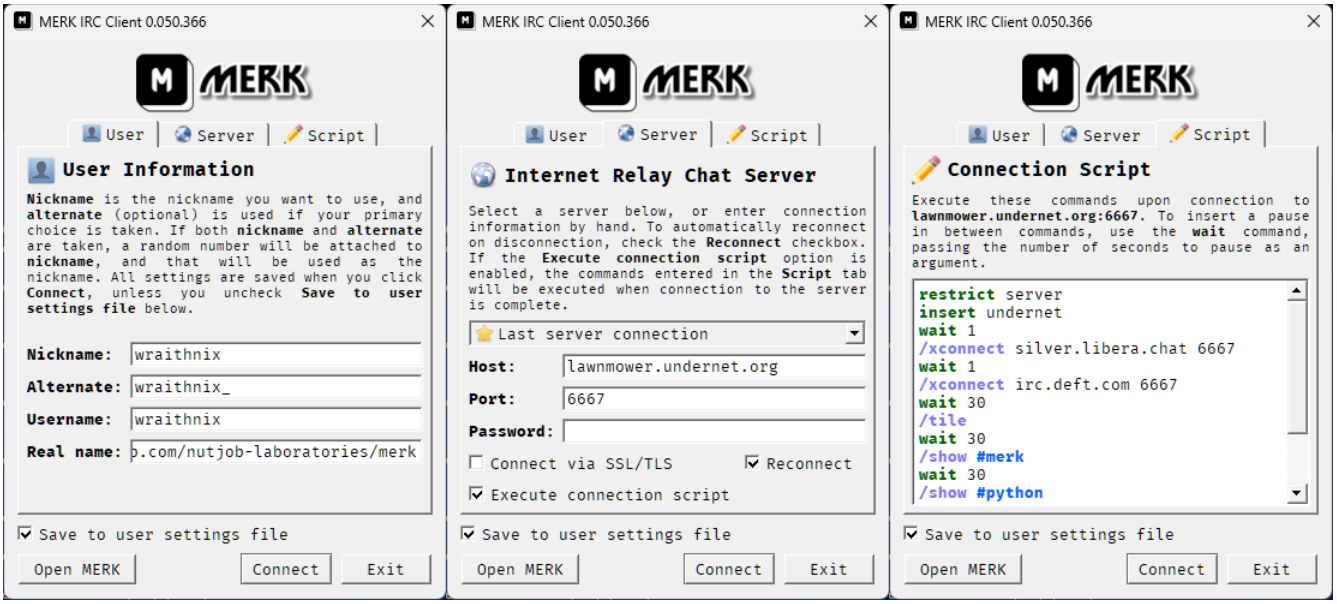


The "Simplified dialogs" option on the first page of the "Settings" dialog.

"Simplified dialogs" is turned off by default, showing the help text on dialogs every time a dialog is opened.

Connection Dialog

When you first run MERK, a connection dialog is displayed, allowing you to connect to an IRC server. The dialog has three tabs: **User Information**, **Internet Relay Chat Server**, and **Connection Script**.

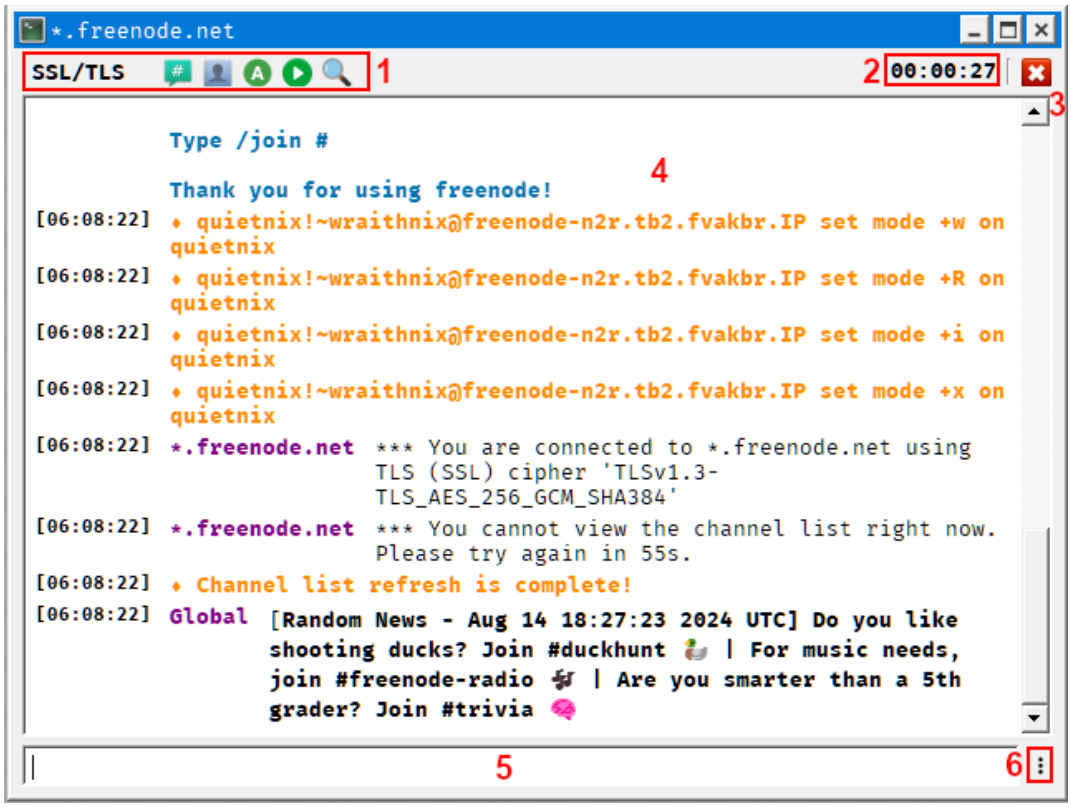


- **User Information** is where the user enters their various user information for connection. **Nickname** is the nickname you'd like to use, and **Alternate** (optional) will be used if that nickname is already taken; if both are taken, a random number is generated and added to **Nickname**.
- **Internet Relay Chat Server** is where the user enters the IRC server to connect to. By default, the last server MERK connected to is pre-entered. **Host** is the IP address or hostname of the desired IRC server, and **Port** is the server's port. **Password** is the server's password, if one is required; leave blank if one is not required. Click **Connect via SSL/TLS** to connect via SSL/TLS, and click **Reconnect** if MERK should automatically try to reconnect on disconnection. Click **Execute connection script** to execute the connection script on the next tab as soon as MERK connects to the server.
- **Connection Script** has a small script editor for editing the script MERK will execute upon connection to the server. The syntax highlighting settings can be edited with the "Settings" dialog. If **Execute connection script** is unchecked, this script will not be executed.
- **Save to user settings files**, if checked, will save any information entered into the dialog to **`user.json`**, and will be loaded automatically the next time MERK is started up.

Click **Connect** to connect to the IRC server using the entered information, or **Exit** to close MERK. To start MERK without connecting to a server, click **Open MERK**.

Server Windows

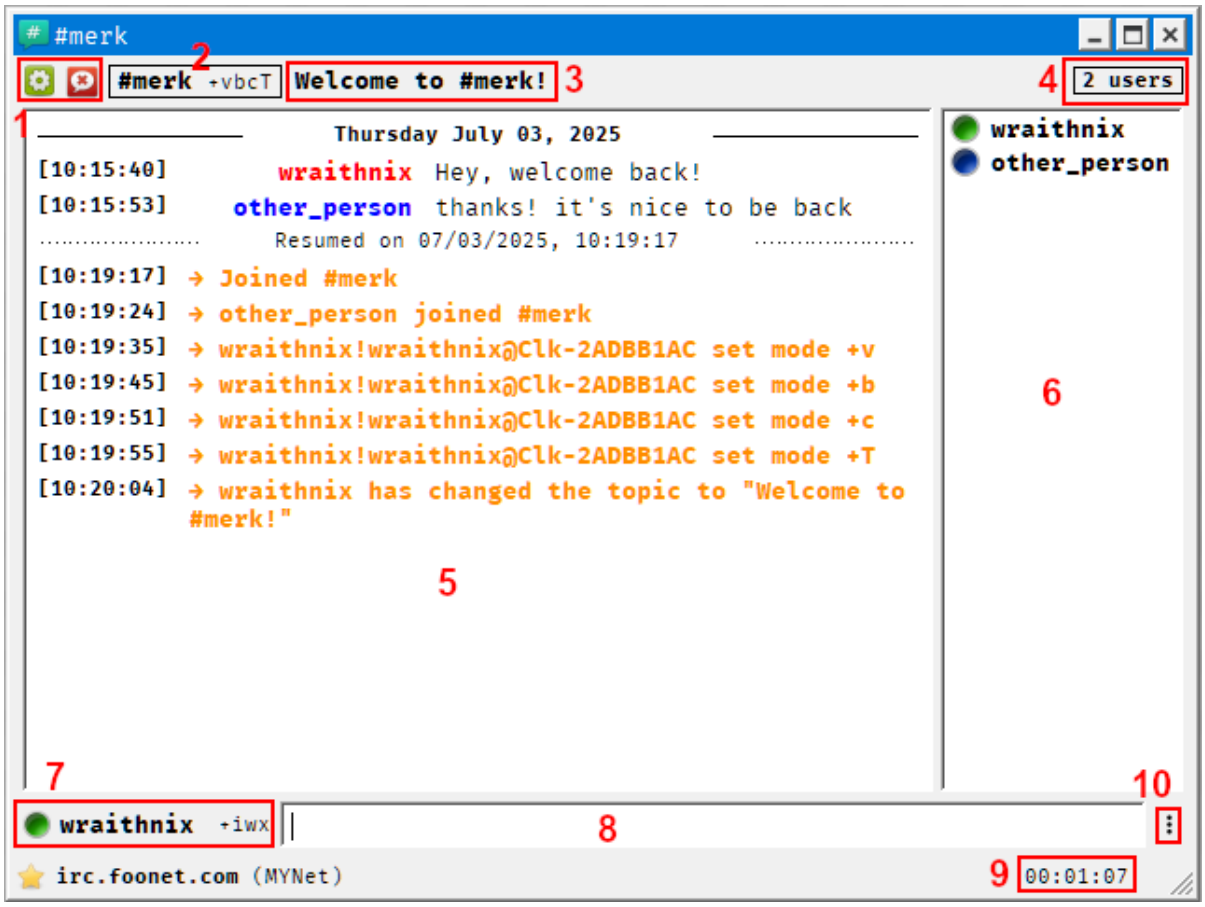
Server windows are the first windows you will see in MERK; they appear as soon as connection to an IRC server begins. Server windows behave differently from channels or private chat windows: closing a server window does not disconnect from the server, it only hides the window. To view a hidden server window, click on its entry in the "Windows" menu or the system tray menu. You cannot chat to other users from a server window without using commands like **`/msg`** or **`/notice`**.



1. **Toolbar**. Buttons that perform basic actions; some on the IRC server, such as joining a channel, changing your nickname, and setting your away status, and others on the client, like selecting a script to run, refreshing the channel list from the server, and opening the channel list dialog. Clicking the button labeled "TCP/IP" (for normal connections) or "SSL/TLS" (for encrypted connections) will show a menu with information about the server.
2. **Connection uptime**. This displays how long MERK has been connected to the server.
3. **Disconnect**. Pressing this button issues a **`QUIT`** command and quickly disconnects from the IRC server.
4. **Display**. Displays any messages from the server, as well as notices, outgoing private messages, and the like.
5. **Text input widget**. Type commands in here, and press "enter" to execute them.
6. **Input menu**. Clicking on this brings up a menu that allows you to do various tasks, like changing the spellchecker's language. This button is present on channel or private message windows, too.

Channel Windows

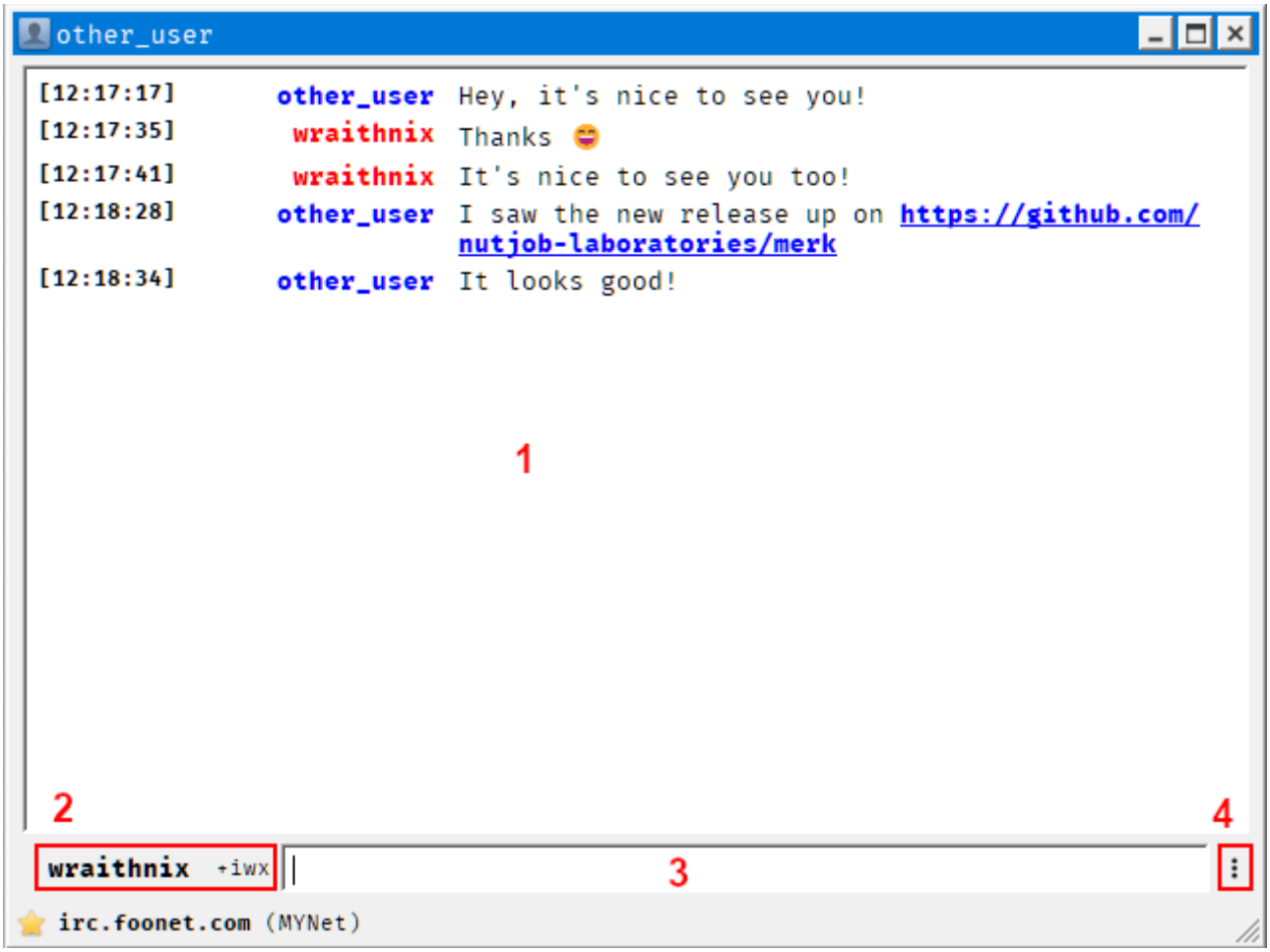
Closing a channel window leaves the channel.



1. **Mode Editor** and **Banlist**. The mode editor button displays a menu that allows the user to set or remove popular channel modes, if their status allows it; if they are not a privileged enough user, the button is hidden. The banlist displays a list of users that have been banned from the channel; if the banlist is empty, the button is hidden.
2. **Name and mode display**. Here, the channel name and any channel modes are displayed.
3. **Topic**. The channels topic is displayed here. Click on the topic to edit it, and press enter to send any changes to the server.
4. **User count**. How many users are currently in the channel
5. **Chat display**. Channel chat, as well as system messages, are displayed here.
6. **User list**. A list of users in the channel is displayed here. Privileged users have special icons next to their name (green for channel operators, blue for voiced users, etc.), and normal users do not. Nicknames are displayed in bold if the users are present, and in normal weight if they are away. Double click a user's name to open a private chat window.
7. **Nickname**. This displays the currently used nickname, and any user modes set.
8. **Text input widget**. Type your chat or commands here, and press "enter" to send them to the server or client.
9. **Uptime**. This displays how long the client has been connected to the channel.
10. **Input menu**. Clicking on this brings up a menu that allows you to do various tasks, like changing the spellchecker's language.

Private Chat Windows

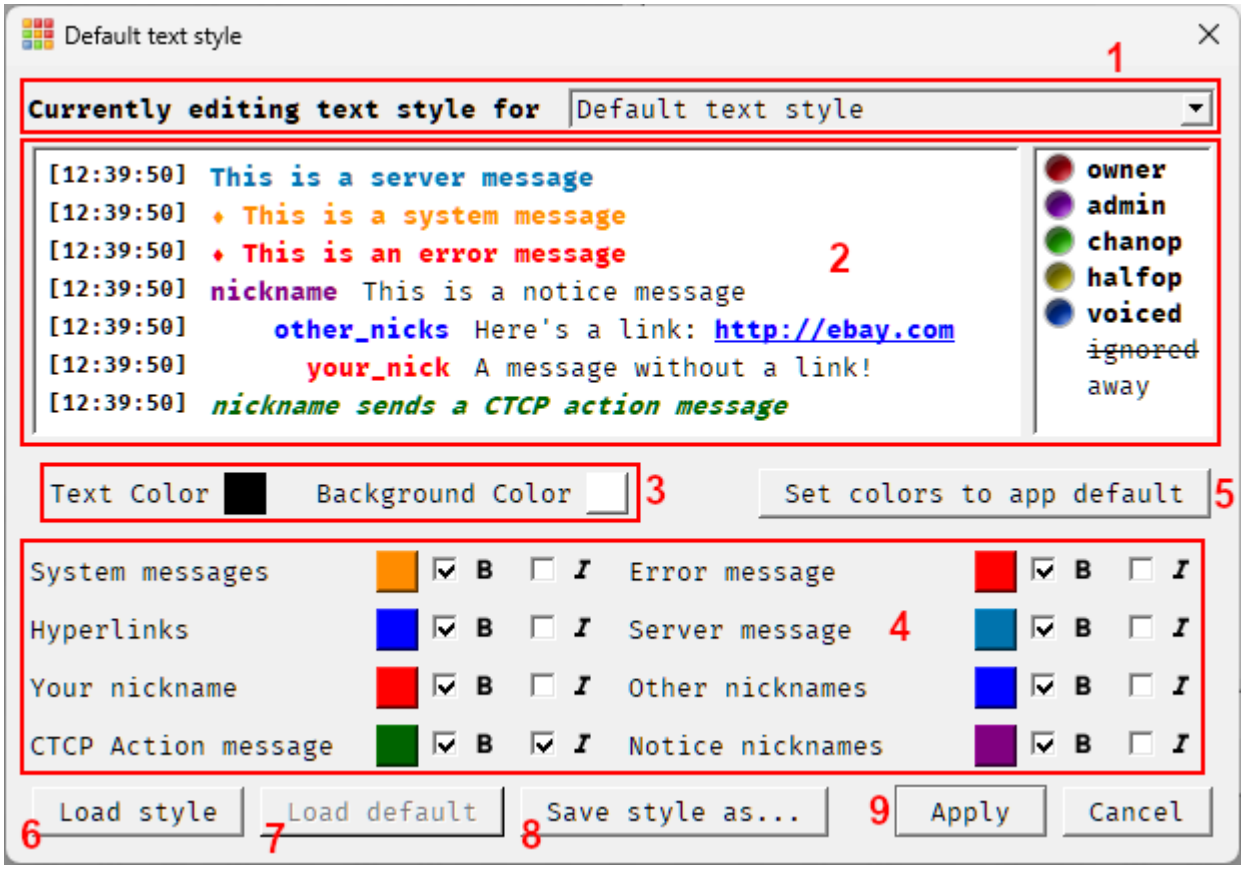
Closing a private chat window does not leave the chat, or block the sender; it only closes the window.



1. **Chat display**. Private chat is displayed here, as well as system messages.
2. **Nickname**. This displays the currently used nickname, and any user modes set.
3. **Text input widget**. Type your chat or commands here, and press "enter" to send them to the server or the client.
4. **Input menu**. Clicking on this brings up a menu that allows you to do various tasks, like changing the spellchecker's language.

Style Editor

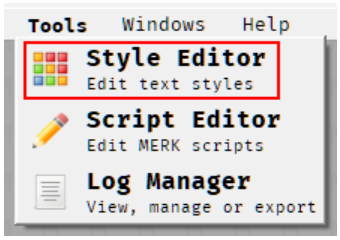
MERK has a text style engine that colors and styles all chat text, and can be edited by users with the style editor.



1. **Style selector**. Select what text style to edit. When launched from the "Tools" menu, this will default to editing the default text style; when launched from context menus or the **`/style`** command, the text style of the window that launched the style editor will be selected. The text style of any window currently in use can be selected.
2. **Display**. This is what the text style will look like in the client. Any changes in color or style will be displayed here instantly. If editing the default text style, or the text style of a channel, an example user list is shown as how it will appear in the client. The example user list is not shown when editing the text style of server windows or private chats.
3. **Background and foreground color**. Set the color of the text and the background color here.
4. **Message styles**. Change the color and style of individual message types here.
5. **Set colors to app default**. Set all colors to the default style that ships with MERK. This is different from the "default" style that is applied to server windows and any windows that do not have a style.
6. **Load style**. Here, you can open any existing MERK style file for editing. Colors and styles will be loaded and displayed.
7. **Load default**. This will load in whatever style the user has set as the default text style. This button is disabled when editing the default text style.
8. **Save style as…**. Save this style to a file. It will not be applied, only saved to a file.
9. **Apply** and **Cancel**. Applying this style automatically saves it. Pressing the "cancel" button closes the dialog, and all changes are discarded.

How Text Styles Are Applied

All chat windows start by using the default style. All text styles currently in use can be edited with the "Style Editor", found in the "Tools" menu

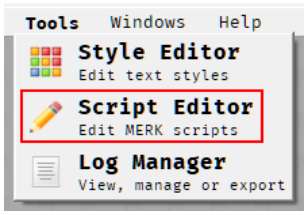


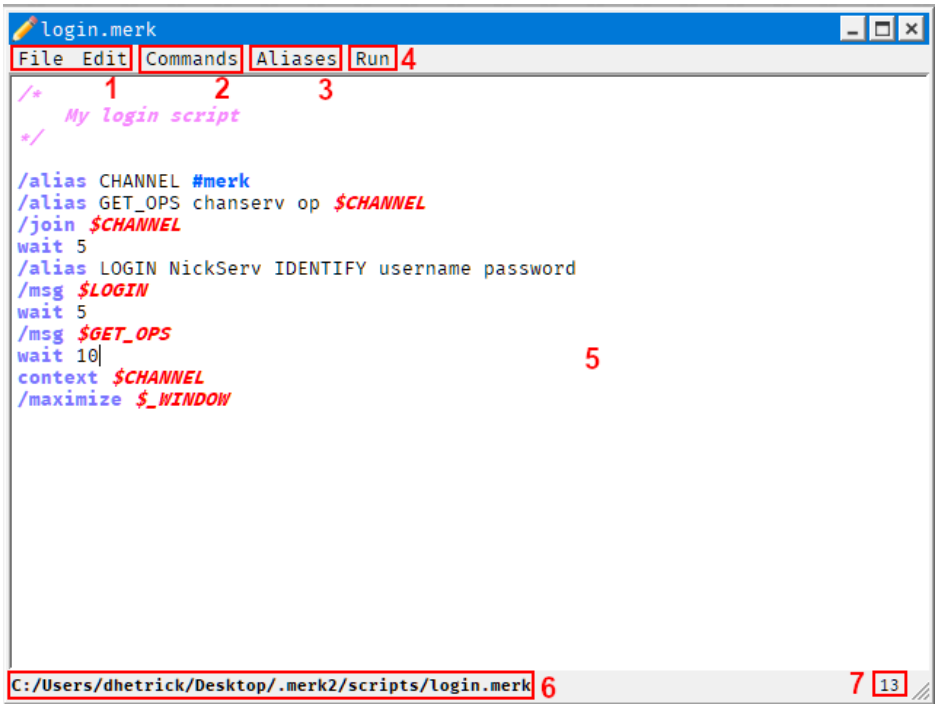
All chat windows can have their own styles which can be edited by selecting the "Style Editor" option from the "Tools" menu, or "Edit [NAME]'s text style" in the input options menu, or the chat display right click menu. Styles for channel and private chat windows are saved with the IRC network of the channel or private chat in mind, so they will load no matter which server the client is connected to. For example, if the user has set a text style for the **#merk** channel on the EFnet network, it will load and be applied to the **#merk** channel window if the user is connected to **irc.underworld.no** on port 6667, **irc.choopa.net** on port 9999, or **irc.prison.net** on port 6667, as all of these servers are on the EFnet IRC network.

Server window text styles are specific to the server being connected to, regardless of what network the server is on. So, if the user has set a style for **irc.prison.net** on port 6667, the server window text style for that connection will be loaded. If the user connects to **irc.choopa.net** on port 9999, the default style will be loaded; even though both servers on the EFnet network, they are still different servers.

Script Editor

To launch the script editor, use the **`/edit`** command, or select **Script Editor** from the "Tools" menu



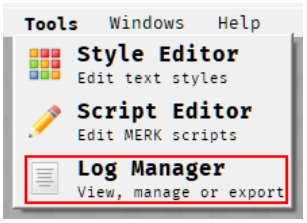


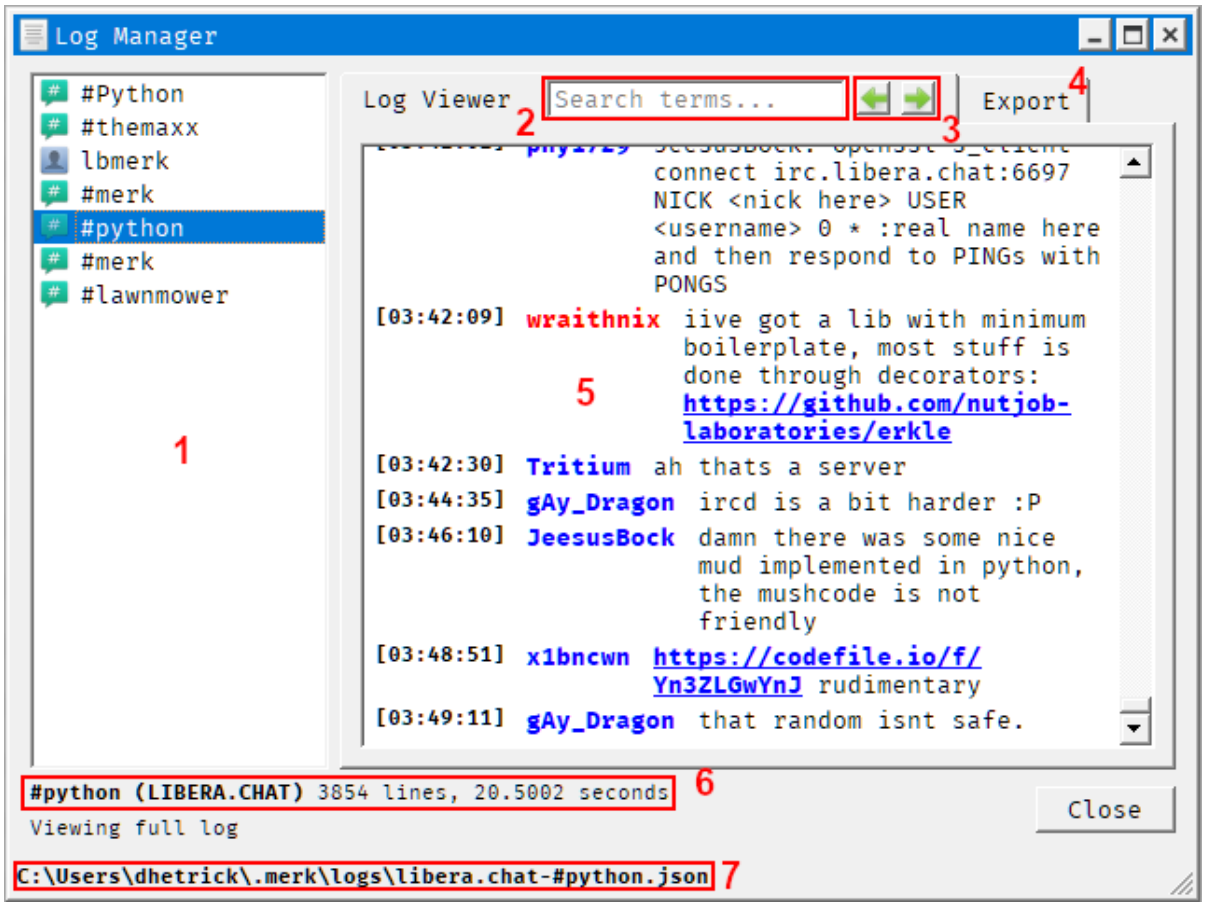
1. **File** and **Edit Menus**. All the normal selections of a text editor, like opening and saving files, cut and paste, find and replace, etc. Connection scripts can also be opened for editing, as well as created.
2. **Commands**. Each entry in this menu allows the user to insert a command into the open script. Click the desired command, fill out the entries in the dialog that pops up, if needed, and the command will be inserted into the script.
3. **Aliases**. Insert built-in aliases into the script.
4. **Run**. Run the currently open script in any context/window available. The user also can run the script in all contexts/windows simultaneously. Scripts are executed with no arguments and no filename.
5. **Script display**. Features syntax highlighting. Colors used for the display can be set in the "Settings" dialog.
6. **Filename**. The currently open script's filename is displayed here.
7. **Line number**. The line number the cursor is currently on.

The colors and styles used for the syntax highlighting can be changed in the "Settings" dialog. Comments, commands, channels, aliases, and script-only commands can be styled individually. These colors and styles will also be used in the "Connection Script" tab of the connection dialog.

Log Manager

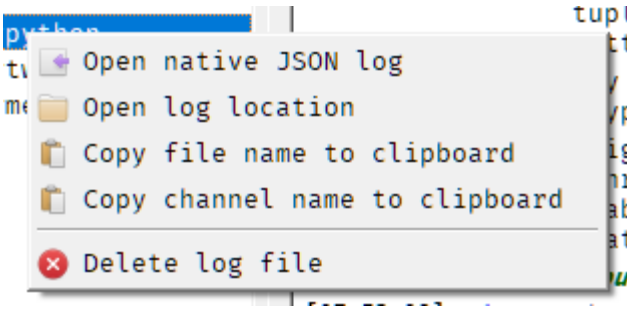
The log manager allows users to view, delete, and export MERK logs. It can be launched from the "Tools" menu





1. **Logs**. A full list of all logs in MERK. Hover the mouse over the log name to see what IRC network the log is from. Click a log name to view information about the log, as well as use export options. Double click a log name to view the contents of a log.
2. **Search**. Search the log for specific words. Typing in the terms and pressing enter will find the first instance of the term in the log; hitting enter again will find the next, and so on.
3. **Next Result** and **Previous Result**. Move forward and backwards in the word search results.
4. **Log Viewer** and **Export Tabs**. Click tabs to switch functions. The **Export** tab has settings and functionality to export MERK logs to JSON or a custom delimited format.
5. **Log display**. Logs are loaded in full for viewing. For longer logs, this may take some time. Logs use the whatever default text style the user has set.
6. **Log information**. Contains the log name, what IRC network the log is from, how many lines of chat the log contains, and how long it took to render the entire log, if the log is being viewed.
7. **Log filename**. The full filename of the current log.

Additional log options can be seen by right-clicking on the log's name:



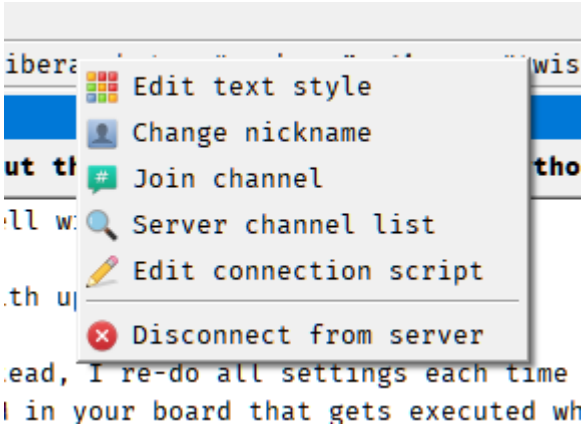
If the "Open native JSON log" option is selected, the JSON file that MERK uses will be opened in the default application that the user's operating system to open JSON files. There is additional information in the native log format that tells MERK how to render the log for viewing; to use MERK logs with other applications, the log should be exported to strip this information out.

Logs that are very large will not be fully loaded for the log display; only the last 5000 lines of chat are loaded, as it can take up to a minute to render a log this large, depending on the speed of the computer running MERK. This can be changed by using the **`/config`** command: the setting **`log_manager_maximum_load_size`** sets the maximum number of lines to load.

The Windowbar

The "windowbar" is a widget that is located by default on the top of the main window that displays a list of open subwindows. Clicking on a window's name switches "focus" to that window, bringing it to the front; double click the window name to bring the window to the front and maximize it. By default, the windowbar only shows channel windows, private chat windows, and script editor windows, but you can use the "Settings" dialog (or the windowbar's right click menu) to show other window types. Think of the windowbar as a sort of task manager, only for windows in MERK.

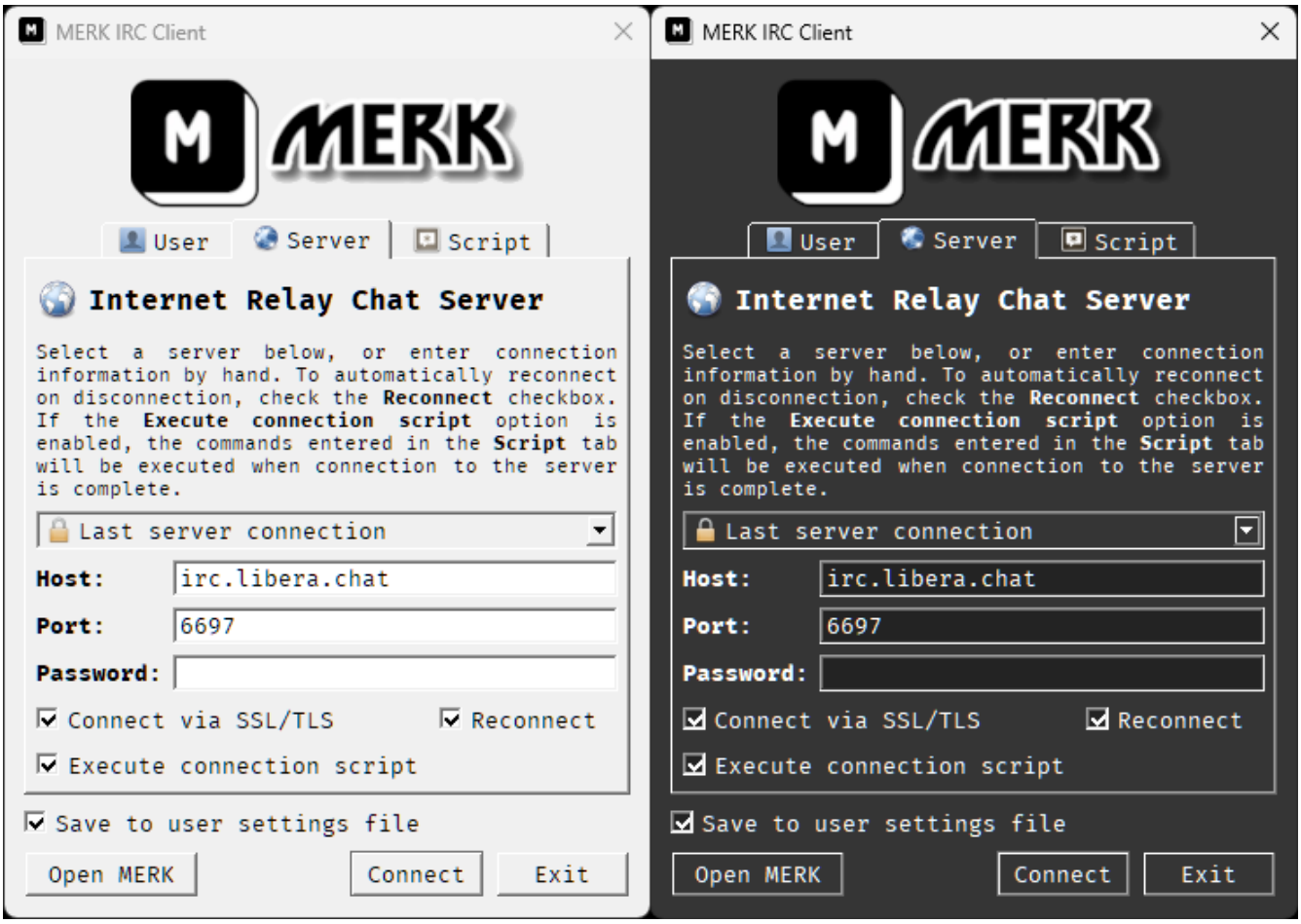
Right click on an entry in the windowbar for more options. Each window type shows different options. Right click on the windowbar itself to change settings for the windowbar. Most things about the windowbar can be changed, including the order windows are shown, what windows are shown, whether icons are shown on entries, and more.



The windowbar right click menu for a server window.

Although immobile by default, the right click menu and the "Settings" dialog can make the windowbar movable; the windowbar can float, or be "docked" at the bottom or the top of MERK's main window.

Regular and Dark Mode



Normal Mode *Dark Mode*

MERK can be operated in "normal" mode, seen above to the left, or in "dark" mode, on the right. To switch to "dark" mode, select it in the "Settings" menu or in the "Settings" dialog. MERK will have to be restarted for it to take effect, and you'll be prompted to restart MERK automatically.

For more advanced users, if you want to edit the palette that "dark" mode uses, all of the colors used by the application are stored in a file in the **`styles`** directory named **`dark.palette`**. The file format is specific to MERK, but you can edit it with a text editor. All colors are stored in the hexadecimal format used by HTML. To re-create the file and reset the "dark" mode values back to the default, delete **`dark.palette`** and restart MERK; the application will regenerate the file with default values.

Commands and Scripting Guide

Command List

IRC Commands

All of these commands are related to IRC, in some way, shape, or form. Most of them are present in most other modern IRC clients.

Command	Description
/away [MESSAGE]	Sets status as "away"
/back	Sets status as "back"
/ctcp REQUEST USER	Sends a CTCP request to a user. Valid requests are TIME, VERSION, USERINFO, SOURCE, or FINGER
/invite NICKNAME CHANNEL	Sends a channel invitation
/finger TEXT...	Sets the CTCP FINGER response. Pass ***** as the argument to clear the FINGER response text.
/join CHANNEL [KEY]	Joins a channel
/kick CHANNEL NICKNAME [MESSAGE]	Kicks a user from a channel
/knock CHANNEL [MESSAGE]	Requests an invitation to a channel
/list [TERMS]	Lists or searches channels on the server; use "*" for multi-character wildcard and "?" for single character
/me MESSAGE...	Sends a CTCP action message to the current chat
/mode TARGET MODE...	Sets a mode on a channel or user
/msg TARGET MESSAGE...	Sends a message
/nick NEW_NICKNAME	Changes your nickname
/notice TARGET MESSAGE...	Sends a notice
/oper USERNAME PASSWORD	Logs into an operator account
/part CHANNEL [MESSAGE]	Leaves a channel
/ping USER [TEXT]	Sends a CTCP ping to a user
/quit [MESSAGE]	Disconnects from the current IRC server
/quitall [MESSAGE]	Disconnects from all IRC servers
/raw TEXT...	Sends unprocessed data to the server
/reclaim NICKNAME	Repeatedly attempts to change nickname to **NICKNAME** until successful; by default, attempts are made every 30 seconds
/refresh	Requests a new list of channels from the server
/time	Requests server time
/topic CHANNEL NEW_TOPIC	Sets a channel topic
/userinfo TEXT...	Sets the CTCP USERINFO response. Pass ***** as the argument to clear the USERINFO response text.
/version [SERVER]	Requests server version

Command	Description
/who NICKNAME [o]	Requests user information from the server
/whois NICKNAME [SERVER]	Requests user information from the server
/whowas NICKNAME [COUNT] [SERVER]	Requests information about previously connected users

All Other Commands

These commands are specific to MERK, and are for using and manipulating the client. Commands with a gray background are for use in scripts *only*; they cannot be used in the text input widget, and will only produce an error. Some of these commands cannot be used in scripts.

Command	Description
/alias [TOKEN] [TEXT...]	Creates an alias that can be referenced by **$TOKEN**. Call with only **TOKEN** as the argument to see **TOKEN**'s value. If **TEXT** is a mathematical statement, it will be evaluated, with the resulting value stored as the alias' value. Call without any arguments to see all aliases and their values.
/bind SEQUENCE COMMAND...	Executes **COMMAND** every time key **SEQUENCE** is pressed. **SEQUENCE** should contain no spaces, and should be a string like "Ctrl+M" or "Alt+Space". **COMMAND** will be executed in whatever window/context has focus when **SEQUENCE** is pressed.
/clear [SERVER] [WINDOW]	Clears a window's chat display. **SERVER** is optional if **WINDOW** belongs to the same context.
/close [SERVER] [WINDOW]	Closes a subwindow. **SERVER** is optional if **WINDOW** belongs to the same context.
/config [SETTING] [VALUE...]	Changes a setting, or searches and displays one or all settings in the configuration file
/config export [FILENAME]	Exports the current configuration file
/config import [FILENAME]	Imports a configuration file into settings
/config restart	Restarts MERK using the same command-line arguments used to start MERK
/connect SERVER [PORT] [PASSWORD]	Connects to an IRC server
/connectssl SERVER [PORT] [PASSWORD]	Connects to an IRC server via SSL
context WINDOW_NAME	Moves execution of the script to **WINDOW_NAME**; can only be called from scripts
/delay SECONDS COMMAND...	Executes **COMMAND** after **SECONDS** seconds
/edit [FILENAME]	Opens a script in the editor; if called without an argument, opens an editor window
end	Immediately ends a script. Can only be called from scripts.
exclude WINDOW...	Prevents a script from executing in **WINDOW**'s context. Multiple **WINDOW**s can be specified. Can only be called from scripts.
/exit [SECONDS]	Exits the client, with an optional pause of **SECONDS** before exit

/find [TERMS]	Finds filenames that can be found by other commands, like **/script** or **/edit**. If called without any arguments, **/find** will list all files visible to commands. Can use ***** for multi-character wildcards and **?** for single character wildcards.
/focus [SERVER] [WINDOW]	Sets focus on a subwindow. **SERVER** is optional if **WINDOW** belongs to the same context.
/fullscreen	Toggles full screen mode
goto LINE_NUMBER	Moves execution of the script to **LINE_NUMBER**. The only script-only command that can be issued from an **if** command. Can only be called from scripts.
halt [MESSAGE...]	Halts a script's execution, and displays an error **MESSAGE** with line number and file name. Can only be called from scripts.
/help [COMMAND]	Displays command usage information
/hide [SERVER] [WINDOW]	Hides a subwindow. **SERVER** is optional if **WINDOW** belongs to the same context.
if VALUE1 OPERATOR VALUE2 COMMAND...	Executes **COMMAND** if **VALUE1** and **VALUE2** are true, depending on **OPERATOR**. Valid **OPERATOR**s are **(is)** (result is true if **VALUE1** and **VALUE2** are equal), **(not)** (result is true if **VALUE1** and **VALUE2** are not equal), **(in)** (result is true if **VALUE1** is contained in **VALUE2**), **(gt)** (result is true if **VALUE1** is a greater number than **VALUE2**), **(lt)** (result is true if **VALUE1** is a lesser number than **VALUE2**), **(ne)** (result is true if **VALUE1** and **VALUE2** are numbers and are not equal), and **(eq)** (result is true if **VALUE1** and **VALUE2** are numbers and are equal). Can only be called from scripts.
/ignore USER	Hides a **USER**'s chat in all chat windows. This can be set to a nickname or hostmask. Capitalization is ignored. Use ***** as multiple character wildcards, and **?** as single character wildcards.
insert FILE [FILE...]	Inserts the contents of **FILE** where it appears in the script. **FILE** should be a MERK script. If a filename contains spaces, put it in quotation marks. Multiple files can be passed as arguments. Can only be called from scripts.
/macro NAME SCRIPT [USAGE] [HELP]	Creates a macro, executable with **/NAME**, that executes **SCRIPT**. **USAGE** and **HELP** are used for the command list displayed with **/help**.
/maximize [SERVER] [WINDOW]	Maximizes a subwindow. **SERVER** is optional if **WINDOW** belongs to the same context.
/minimize [SERVER] [WINDOW]	Minimizes a subwindow. **SERVER** is optional if **WINDOW** belongs to the same context.
/move [SERVER] [WINDOW] X Y	Moves a subwindow to **X** (left and right) and **Y** (up and down) coordinates. **SERVER** is optional if **WINDOW** belongs to the same context.

/msgbox MESSAGE...	Displays a messagebox with a short message
only WINDOW...	Restricts a script to only executing in **WINDOW**'s context. Multiple **WINDOW**s can be specified. Can only be called from scripts.
/play FILENAME	Plays a WAV file
/print [WINDOW] TEXT...	Prints text to a window
/prints [WINDOW] TEXT...	Prints a system message to a window
/private NICKNAME [MESSAGE]	Opens a private chat subwindow for **NICKNAME**
/random ALIAS LOW HIGH	Generates a random number between **LOW** and **HIGH** and stores it in **ALIAS**
/reconnect SERVER [PORT] [PASSWORD]	Connects to an IRC server, reconnecting on disconnection
/reconnectssl SERVER [PORT] [PASSWORD]	Connects to an IRC server via SSL, reconnecting on disconnection
/rem [TEXT...]	Does nothing. Can be used as a target for **goto**
/resize [SERVER] [WINDOW] WIDTH HEIGHT	Resizes a subwindow. **SERVER** is optional if **WINDOW** belongs to the same context.
restrict channel|server|private	Prevents a script from running if it is not being executed in a **channel**, **server**, or **private** chat window. Up to two window types can be set. Can only be called from scripts.
/restore [SERVER] [WINDOW]	Restores a subwindow. **SERVER** is optional if **WINDOW** belongs to the same context.
/s FILENAME [ARGUMENTS]	A shortcut for the **/script** command.
/script FILENAME [ARGUMENTS]	Executes a list of commands in a file. If the script has a file extension of **.merk**, it may be omitted from **FILENAME**.
/shell ALIAS COMMAND...	Executes an external program, and stores the output in an alias
/show [SERVER] [WINDOW]	Shows a subwindow, if hidden; otherwise, shifts focus to that subwindow. **SERVER** is optional if **WINDOW** belongs to the same context.
/style [SERVER] [WINDOW]	Opens a window's text style editor. Cannot be called from a script
/unalias TOKEN	Deletes the alias referenced by **$TOKEN**
/unbind SEQUENCE	Removes a bind for key **SEQUENCE**. To remove all binds, pass ***** as the argument
/unignore USER	Un-hides a **USER**'s chat in all chat windows. This can be set to a nickname or hostmask. Capitalization is ignored. To un-hide all users, use ***** as the argument.
usage NUMBER [MESSAGE...]	Prevents a script from running unless **NUMBER** or more arguments are passed to it, and displays **MESSAGE**. Can only be called from scripts
/user [SETTING] [VALUE...]	Changes a user setting, or searches and displays one or all settings in the user file. Pass ***** as **VALUE** to set a setting as blank

wait SECONDS	Pauses script execution for SECONDS; can only be called from scripts
/window [COMMAND] [X] [Y]	Manipulates the main application window. Valid commands are **move**, **size**, **maximize**, **minimize**, **restore**, **readme**, **settings**, **logs**, **cascade**, **tile**, **next**, and **previous**. Call with no arguments to see window information and a list of subwindows.
/xconnect SERVER [PORT] [PASSWORD]	Connects to an IRC server & executes connection script
/xconnectssl SERVER [PORT] [PASSWORD]	Connects to an IRC server via SSL & executes connection script
/xreconnect SERVER [PORT] [PASSWORD]	Connects to an IRC server & executes connection script, reconnecting on disconnection
/xreconnectssl SERVER [PORT] [PASSWORD]	Connects to an IRC server via SSL & executes connection script, reconnecting on disconnection

Script-Only Commands

These commands can only be called from scripts. Attempts to use them in the text input widget will fail and show an error.

Command	Description
context WINDOW_NAME	Moves execution of the script to **WINDOW_NAME**; can only be called from scripts
end	Immediately ends a script. Can only be called from scripts.
exclude WINDOW...	Prevents a script from executing in **WINDOW**'s context. Multiple **WINDOW**s can be specified. Can only be called from scripts.
if VALUE1 OPERATOR VALUE2 COMMAND...	Executes **COMMAND** if **VALUE1** and **VALUE2** are true, depending on **OPERATOR**. Valid **OPERATOR**s are **(is)** (result is true if **VALUE1** and **VALUE2** are equal), **(not)** (result is true if **VALUE1** and **VALUE2** are not equal), **(in)** (result is true if **VALUE1** is contained in **VALUE2**), **(gt)** (result is true if **VALUE1** is a greater number than **VALUE2**), **(lt)** (result is true if **VALUE1** is a lesser number than **VALUE2**), **(ne)** (result is true if **VALUE1** and **VALUE2** are numbers and are not equal), and **(eq)** (result is true if **VALUE1** and **VALUE2** are numbers and are equal). Can only be called from scripts.
goto LINE_NUMBER	Moves execution of the script to **LINE_NUMBER**. The only script-only command that can be issued from an **if** command. Can only be called from scripts.
halt [MESSAGE...]	Halts a script's execution, and displays an error **MESSAGE** with line number and file name. Can only be called from scripts.
insert FILE [FILE...]	Inserts the contents of **FILE** into the script. **FILE** should be a MERK script. If a filename contains spaces, put it in quotation marks. Multiple files can be passed as arguments. Can only be called from scripts.
only WINDOW...	Restricts a script to only executing in **WINDOW**'s context. Multiple **WINDOW**s can be specified. Can only be called from scripts.
restrict channel|server|private	Prevents a script from running if it is not being executed in a **channel**, **server**, or **private** chat window. Up to two window types can be set. Can only be called from scripts.
usage NUMBER [MESSAGE...]	Prevents a script from running unless **NUMBER** or more arguments are passed to it, displaying **MESSAGE**. Can only be called from scripts.
wait SECONDS	Pauses script execution for SECONDS; can only be called from scripts

Non-Script Commands

These commands cannot be called from scripts. Scripts that use these commands will prevent the script from being executed, and show an error.

Command	Description
/style [SERVER] [WINDOW]	Opens a window's text style editor. Cannot be called from a script

Context-less Commands

These commands can be called without specifying the channel, chat, or window they are for. They will run in the current context. Commands with a gray background are IRC specific commands.

Command	Description
/bind SEQUENCE COMMAND...	Executes **COMMAND** every time key **SEQUENCE** is pressed. **SEQUENCE** should contain no spaces, and should be a string like "Ctrl+M" or "Alt+Space". **COMMAND** will be executed in whatever window/context has focus when **SEQUENCE** is pressed.
/clear	Clears the current window's chat display
/close	Closes the current subwindow
/focus	Sets focus on the current subwindow
/hide	Hides the current subwindow
/invite NICKNAME	Sends a channel invitation to the current channel
/kick NICKNAME [MESSAGE]	Kicks a user from the channel
/maximize	Maximizes the current subwindow
/me MESSAGE...	Sends a CTCP action message to the current chat
/minimize	Minimizes the current subwindow
/mode MODE...	Sets a mode on the current channel
/move X Y	Moves the current subwindow to **X** (left and right) and **Y** (up and down) coordinates.
/part [MESSAGE]	Leaves the channel
/print TEXT...	Prints text to the current window
/prints TEXT...	Prints a system message to the current window
/resize WIDTH HEIGHT	Resizes the current subwindow
/restore	Restores the current subwindow
/show	Shows the current subwindow, if hidden
/topic NEW_TOPIC	Sets the current channel's topic
/unbind SEQUENCE	Removes a bind for key **SEQUENCE**. To remove all binds, pass ***** as the argument

Subwindow Management Commands

These commands can be called without specifying the channel, chat, or window they are for. They will run in the current context. Commands with a gray background are IRC specific commands.

Command	Description
/close [SERVER] [WINDOW]	Closes a subwindow
/focus [SERVER] [WINDOW]	Sets focus on a subwindow
/hide [SERVER] [WINDOW]	Hides a subwindow
/maximize [SERVER] [WINDOW]	Maximizes a window
/minimize [SERVER] [WINDOW]	Minimizes a window
/move [SERVER] [WINDOW] X Y	Moves a subwindow to **X** (left and right) and **Y** (up and down) coordinates.
/resize [SERVER] [WINDOW] WIDTH HEIGHT	Resizes a subwindow
/restore [SERVER] [WINDOW]	Restores a window
/show [SERVER] [WINDOW]	Shows a subwindow, if hidden
/window [COMMAND] [X] [Y]	Manipulates the main application window. Valid commands are **move**, **size**, **maximize**, **minimize**, **restore**, **readme**, **settings**, **logs**, **cascade**, **tile**, **next**, and **previous**. Call with no arguments to see window information and a list of subwindows.

Additional Command Help

- **`wait`** – ***Can only*** *be called from* ***scripts***
- **`context`** – ***Can only*** *be called from* ***scripts***
- **`end`** – ***Can only*** *be called from* ***scripts***
- **`usage`** – ***Can only*** *be called from* ***scripts***
- **`restrict`** – ***Can only*** *be called from* ***scripts***
- **`insert`** – ***Can only*** *be called from* ***scripts***
- **`only`** – ***Can only*** *be called from* ***scripts***
- **`exclude`** – ***Can only*** *be called from* ***scripts***
- **`if`** – ***Can only*** *be called from* ***scripts***
- **`goto`** – ***Can only*** *be called from* ***scripts***
- **`halt`** – ***Can only*** *be called from* ***scripts***
- **`/style`** – ***Cannot*** *be called from* ***scripts***

Most commands can be issued in both the text input widget and scripts. There are eleven commands, however, that can *only* be issued in scripts: **`wait`**, **`context`**, **`usage`**, **`restrict`**, **`insert`**, **`only`**, **`exclude`**, **`if`**, **`goto`**, **`halt`**, and **`end`**. These ten commands *cannot* be used in the text input widget. There are one command that cannot be called by a script, and can only be used in the text input widget: **`/style`** will display an error and prevent execution if called by a script.

Most commands require a context to be executed in (see ***Context***). Commands that can be issued without explicitly specifying a context are **`/clear`**, **`/invite`**, **`/kick`**, **`/me`**, **`/mode`**, **`/part`**, **`/topic`**, **`/maximize`**, **`/minimize`**, **`/hide`**, **`/show`**, **`/close`**, **`/print`**, **`/prints`**, **`/resize`**, **`/move`**, **`/focus`**, and **`/restore`**; they will be executed in whatever the current context is, and may not function correctly if the current context does not support that command (for example, calling **`/invite`** from a server window). The only exception is **`/me`**: if called from the text input widget of a channel or private chat window, it will send a CTCP action to the current window, using *all* arguments as the text to send in the message. If **`/me`** is called from a server window, the first argument specifies the channel or private chat to send the CTCP message to. For example, to send a CTCP message containing "is using MERK" to the **`#merk`** channel, you could call **`/me #merk is using MERK`** from a server window. When calling **`/me`** from a script, *the command will always send to the current chat if running in a channel or private chat window, and must specify the context if running in a server window*. If a context is specified as the first argument to **`/me`**, and the command is executed in a channel or private chat window, the specified context will be sent as part of the CTCP action message.

The **`goto`** command cannot be used to "jump" to a line that contains any script-only command other than **`end`**. If the "jumped to" line contains *any* script-only command besides **`end`**, the script will display an error and end execution. Use the **`/rem`** command for a target line that "does nothing".

Most commands that require a numerical argument require an integer; that is, a whole or natural number[3]. Several commands can take floating-point numbers[4] as arguments, allowing for fractional numbers: **`if`**, **`wait`**, **`/delay`**, and **`/exit`**. As most of these commands' numerical arguments relate to the amount of time to "pause" a command or script's execution, floating-point numbers allow these commands to "pause" for time periods of less than a second. For example, to "pause" a script for 500 milliseconds, or one half of a second:

```
wait 0.5
```

To convert milliseconds into seconds, divide the number of milliseconds by 1000 to determine the number of seconds. 1 millisecond is 0.001 seconds, 2ms is 0.002 seconds, and so on.

3 ...Natural numbers are the numbers 0, 1, 2, 3, and so on. Wikipedia
4 Wikipedia article on floating-point arithmetic

Using the `goto` Command

The **`goto`** command is extremely powerful, and allows a script to "jump" from one line to another line; that is, it can change the sequence of execution of a script. It is not recommended to use **`goto`** in scripts that contain comments, as comments are stripped out of a script before execution, and may change the line count of a script. Using the **`insert`** command can also alter the line count of a script, so **`goto`** should not be used in scripts that use **`insert`**. Instead, if your script has **`insert`** or comments, execute scripts with **`goto`** using the **`/script`** command.

`goto` cannot be used to "jump" to a line that contains a script-only command other than **`end`**. If **`goto`** is used in this way, an error will be raised, and script execution will halt. The **`/rem`** command can be used as a target, as the command is not a script-only command, and does nothing.

Using **`goto`** is easy: pass the line number you wish to "jump" to as the only argument. Execution of the script will immediately move to the desired line. For example:

```
1 /print This is the beginning of the script!
2 goto 4
3 /print This line will never be executed.
4 /print This line will ALWAYS be executed!
5 end
```

This script:

1. Prints "This is the beginning of the script!" to the current window
2. "Jumps" immediately to line 4
3. Prints "This line will ALWAYS be executed!" to the current window, which is the result of the command on line 4.
4. End the script.

With the **`goto`** command in place, line 3 will never be executed, as the **`goto`** command skips right over it.

`goto` is the only script-only command that can be called from an **`if`** command; all other script-only commands are forbidden, and will display an error.

WARNING! There are no protections in place preventing a script from entering an infinite loop[5]. **`goto`** can lock up or crash MERK.

Please use **`goto`** carefully and sparingly.

5 ...An infinite loop (or endless loop) is a sequence of instructions that, as written, will continue endlessly, unless an external intervention occurs, such as turning off power via a switch or pulling a plug. Wikipedia

Using the `if` Command

The **`if`** command allows MERK scripts to have a small amount of flow control[6]. It compares two values, and if the values' comparison is true, executes a command. The entity that sets how the comparison works is called the **operator**. MERK has seven operators:

Operator	Description
`(is)`	True if the first value is equal to the second value. Values are treated as strings, and are case in-sensitive.
`(not)`	True if the first value is *not* equal to the second value. Values are treated as strings, and are case in-sensitive.
`(in)`	True if the first value is contained in the second value; for example, **`o (in) pop`** evaluates to true because "pop" has "o" in it. Values are treated as strings, and are case in-sensitive.
`(lt)`	True if the both values are numbers, and the first value is less than the second value. If either value is not a number, an error is displayed and script execution stops.
`(gt)`	True if both values are numbers, and the first value if greater than the second value. If either value is not a number, an error is displayed and script execution stops.
`(eq)`	True if both values are numbers, and the first value is equal to the second value. If either value is not a number, an error is displayed and script execution stops.
`(ne)`	True if both values are numbers, and the first value is not equal to the second value. If either value is not a number, an error is displayed and script execution stops.

The first value is passed as the first argument, followed by the **`OPERATOR`**, followed by the second value. All other arguments should contain the command to execute if the comparison is true. **`if`** can execute almost any command available, with one major exception: **`if`** *cannot execute any script-only command other than* **`goto`**. Any attempt to use **`if`** to execute a script-only command besides **`goto`** will result in a "Line contains no command" error, and script execution will be halted. Remember, **`goto`** cannot be used to "jump" to a line with any script-only command other than **`end`**.

As an example, here's a script that tests if the user has a specific nickname (**`merk`**), and uses the **`/reclaim`** command if someone use is using the nickname.

```
1 if merk (is) $_NICKNAME goto 4
2 /print "merk" nickname in use, trying to reclaim it…
3 /reclaim merk
4 end
```

This script:

1. Tests if the user's current nickname is **`merk`**, and if it is, calls **`goto`** to move execution to line 4, which ends the script
2. Prints a message to the user telling them that the script is going to use the /reclaim command
3. Issues the **`/reclaim`** command to reclaim the **`merk`** nickname
4. Ends the script

6 ...Control flow (or flow of control) describes how execution progresses from one command to the next. Wikipedia

Values are tokenized like script arguments; values can have whitespace in them as long as they are contained in quotes. Values can also be mathematical statements, which are evaluated before comparing with the **`if`** statement's operator. For example, let's assume that you have a number stored in an alias named **`mynum`**, and you want to determine if that number is even or odd:

```
1 if "$mynum % 2" (eq) 0 goto 4
2 /print $mynum is odd
3 end
4 /print $mynum is even
5 end
```

Using the `/rem` Command

The **`/rem`** command does nothing. It can be used to add "comments" to a script, or as a target for the **`goto`** command. Since the **`goto`** command cannot be used to "jump" to a line that contains a script-only command, place a **`/rem`** command in the line before the desired script-only command, and "jump" to the line with the **`/rem`** command.

For example, the following code will display an error on line 3 and exit:

```
1 goto 3
2 /print This will never display
3 goto 4
4 /print This will display
```

As line 3, the target the **`goto`** command "jumps" to in line 1, contains a script-only command (another **`goto`**), an error will be displayed and the script will exit. To make this script work in exactly the same way (other than updating the target of the second **`goto`** command), we can use the **`/rem`** command:

```
1 goto 3
2 /print This will never display
3 /rem This line does nothing
4 goto 5
5 /print This will display
```

Using the `/print` and `/prints` Commands

The **`/print`** command can be used to print text to the current or another window; use the name of the window context as the first argument to print to another window. If this window cannot be found, the text will print to whatever the current window context is. The window specified by the first command must be "connected" to the current context (that is, they share an IRC server connection).

The **`/prints`** command works exactly the same way, only it prints a "system" message, like the messages emitted by most commands.

Both **`/print`** and **`/prints`** can print HTML, and are not written to the log.

```
/* This will print to the current window */
/print Hello world!

/* This will print to the #merk channel window.
   If the client is not in #merk, it will print to the current window. */
/print #merk Hello world!

/* This will print some HTML as a system message */
/prints <i>This is in italics!<i><br><u>And this is underlined on a new line!</u>
```

Using the `restrict`, `only`, and `exclude` Commands

The **`restrict`** script-only command restricts a script's execution to a specific context. The first argument sets what type of context the script will function in: **`server`** restricts the script's context to server windows or connection scripts, **`channel`** restricts the script's context to channel windows, and **`private`** restricts the script's context to private chat windows. A restricted script will *not* execute in another context, and will show an error. Up to two context types can be passed, so **`restrict private channel`** would prevent a script from being executed in server windows. So, to restrict a script's execution to chat windows only, you could use:

```
restrict private channel
/print This will only print to chat windows
```

The similar **`only`** script-only command can be used to restrict a script's execution to specific contexts; for example, specific channels, server windows, or private chats. Pass the name of the window as an argument to the command; an unlimited number of arguments can be passed to the command. For example, to restrict a script's execution to channels named **`#merk`** or **`#merkirc`**, you could use:

```
only #merk #merkirc
/print This will only print to windows named #merk or #merkirc
```

The **`exclude`** script-only command works just like the **`only`** command, only it prevents a script from executing in specific contexts. Pass the name of the window as an argument to the command; an unlimited number of arguments can be passed to the command. To prevent a script's execution in any channel named #merk or #merkirc, you could use:

```
exclude #merk #merkirc
/print This will not print to windows named #merk or #merkirc
```

Using the `insert` Command

The **`insert`** script-only command reads in the contents of any file passed as an argument to it, and "inserts" it into the script where it is called. Any built-in aliases (see ***Built-In Aliases***) in the inserted script will reference the script being executed, not the script being **`insert`**ed, including any arguments passed to the calling script. For example, assume you have a script named **`stuff.merk`**, and it contains:

```
/print Hello from $_SCRIPT!
```

In another script, we use the **`insert`** command to insert this file into the script **`test.merk`**:

```
/print This is my main script!
insert stuff.merk
/print And now my script is complete!
```

Once processed, the script that will be executed will look like:

```
/print This is my main script!
/print Hello from test.merk!
/print And now my script is complete!
```

The **`insert`** command can be used to insert multiple files into a script; pass each file's name as a separate argument to **`insert`**, or issue **`insert`** multiple times. Arguments passed to the **`insert`** command are tokenized like script arguments, so filenames with spaces in them can be passed to **`insert`**, as long as they are contained in quotation marks.

`insert`ed files may contain **`insert`** as well, up to a maximum "depth" of 10. That is, a file can **`insert`** a file that calls **`insert`**, which can **`insert`** call a file that calls **`insert`**, which can **`insert`** a file that calls **`insert`**, which can **`insert`** a file that calls **`insert`**, and so on, up to a maximum of 10 "layers" of files that call **`insert`**. Changing this behavior is only possible by using the **`/config`** command on the **`maximum_insert_file_depth`** setting, or by editing **`settings.json`** directly with a text editor.

Using the **`insert`** command can alter the line count of a script, and thus the **`goto`** command should not be used in an **`insert`**ed script, or in a script that uses **`insert`**.

Using `/show`, `/hide`, `/close`, and `context` Commands

`/show` and **`context`** switch contexts programmatically, but in slightly different ways. **`/show`** will show a window hidden with the **`/hide`** command, and also move focus to that window; that means that any commands issued without context will now be issued in that window's context. **`/hide`** simply "hides" a window without closing that window. A hidden window is no longer visible, but will still appear in the windowbar; it is *not* closed, and if the window's context is a channel, the client will still be present in that channel. **`/show`** will *not* move "focus" to a subwindow, and *cannot* be used to switch context; to switch to a windows context, use the **`/focus`** command. **`/show`** and **`/hide`** can be issued from the text input widget.

`context` can only be issued in scripts, and *completely* moves the script's context to the new window.

Both **`/hide`** and **`/show`** can take up to two arguments. Pass the **name of the window** to hide or show as the **first argument** if the window shares a context (that is, the same server connection) as the window issuing the command. If the window to be shown or hidden is in another context (a different server connection), pass the **name of the server window** as the **first argument**, followed by the **name of the window** to be shown or hidden as the **second argument**. The IRC server's hostname is normally used for the **name** of the server window, but you can use the address used to connect to the server, or **address:port** to switch to the server window's context. To "select" the server window with one of these commands, use **`*`** as the name of the window along with the name of the server.

For the following example, assume that we have two active connections: we are in the channels **`#merk`** and **`#python`** on **`silver.libera.chat`**, and the channels **`#qt`** and **`#merk`** on **`lawnmower.undernet.org`**. Our test script is executed in **`#python`**'s context, on **`silver.libera.chat`**:

```
/* Here, we hide #merk's window in the current context */
/hide #merk

/* Now, we switch context to lawnmower.undernet.org's window */
context lawnmower.undernet.org

/* This shows to #merk, only it's on the UnderNet server,
   not the Libera server */
/show #merk

/* In order to show #merk's window on Libera, we use the server
   argument of /show */
/show silver.libera.chat #merk
```

`/close` works exactly the same as **`/show`** and **`/hide`**, only instead of showing or hiding a window, it closes a window. If the window is for a channel, the channel will be left.

You can use the **`/windows`** command to see a list of currently available subwindows.

Using the `/resize`, `/move`, and `/window` Commands

The **`/resize`** and **`/move`** commands can be used to resize and move MERK subwindows, respectively.

Much like **`/hide`** and **`/show`**, **`/resize`**'s and **`/move`**'s first two arguments set what window the command is going to work on. Pass the name of the window as the first argument if the window shares the same context as the window the command is being executed in. If the window belongs to the context of another server, pass the name of the server followed by the window name as the first two arguments.

`/resize` takes two additional arguments: the new width of the subwindow, and the new height of the subwindow. So, to set a subwindow's size to a width of 800 pixels, and a height of 600 pixels, you can use:

```
/resize #merk 800 600
```

`/move` works similarly, and also takes two additional arguments: the new **`X`** and **`Y`** values of the subwindow. The **`X`** value sets where the top left corner of the subwindow will be located from left to right, and the **`Y`** value sets where the top left corner of the subwindow will be located from top to bottom. An error will be shown if invalid values are used (like negative numbers, or if it would move the window to where it would no longer be visible).

This example uses the **`/move`** command to move the subwindow for **`#merk`** on **`tungsten.libera.chat`** to a new location 950 pixels to the right, and 500 pixels from the top:

```
/move tungsten.libera.chat #merk 950 500
```

To find out exactly how big the area that contains the subwindows is, as well as what subwindows are available, call the **`/window`** command with no arguments.

The **`/window`** command is used to manipulate the main application window, as well the subwindows "contained" in it. To move the main application window, pass **`move`** as the first argument to **`/window`**, followed by the **`X`** and **`Y`** values. To resize the window, pass **`size`** as the first argument to **`/window`**, followed by the **`width`** and **`height`**:

```
/* Moves the main application window to the coordinates 200,200 */
/window move 200 200

/* Resizes the main window to 1024x768 */
/window size 1024 768
```

To maximize, minimize, or restore the main application window, pass **`maximize`**, **`minimize`**, or **`restore`** to **`/window`** as the only argument, respectively.

/window can also be used to open dialogs and other types of subwindows. To open up the README, pass **readme** as the only argument to **/window**. To open up the settings dialog, pass **settings** as the only argument to **/window**. To open the log manager, pass **logs** as the only argument to **/window**. To open the log manager with only logs from certain targets, pass the name of the IRC network to show only logs from that network as an argument to **/window logs**, or pass a search term to show only logs with that term in the name of the channel or private chat to **/window logs**:

```
/* Opens up the README */
/window readme

/* Opens up the settings dialog */
/window settings

/* Opens up the log manager */
/window logs

/* Opens up the log manager, only showing logs from the EFnet network */
/window logs EFnet

/* Opens up the log manager, only showing logs channels named #merk */
/window logs #merk
```

/window can also arrange subwindows in common patterns. Call with **cascade** as the only argument to cascade all subwindows, and call with **tile** as the only argument to arrange all subwindows in a tiled pattern.

/window can also be used to "move" focus to another subwindow. Call with **next** as the only argument to move focus to the "next" subwindow, and call with **previous** as the only argument to move to the "previous" subwindow.

```
/* Cascades all subwindows */
/window cascade

/* Tiles all subwindows */
/window tile

/* Moves focus to the "next" subwindow */
/window next

/* Moves focus to the "previous" subwindow */
/window previous
```

The order subwindows are ordered in can be set in the "Settings" dialog, or by changing the **subwindow_order** setting with the **/config** command. Valid values are **creation** (the default; the order subwindows were created in), **stacking** (the "visual" order the subwindows are in), and **activation** (the order subwindows have been activated in).

Using the `/ignore` Command

The /**ignore** command hides chat from a given nickname or user. However, messages from an /**ignore**d user are still received and logged, they are just not displayed. That user's chat is hidden from *all* chat displays, no matter what server they are on. You can pass a nickname (which will hide all chat from any user with that nickname) or a hostmask (which will hide chat from only users with that hostmask) to the /**ignore** command. The /**ignore** list is saved to the configuration file, and will be applied universally until the user is /**unignore**d. The **/ignore** list *cannot* be edited by the **/config** command; the only way to unignore a user is either through the right click userlist menu, the settings menu, or with the **/unignore** command.

The **/ignore** command can also be used with wildcards. Use ***** to substitute for any number of characters, and **?** to substitute for a single character. For example, to ignore any user that has "annoying.com" anywhere in their hostmask or nickname, you could use:

```
/ignore *annoying.com*
```

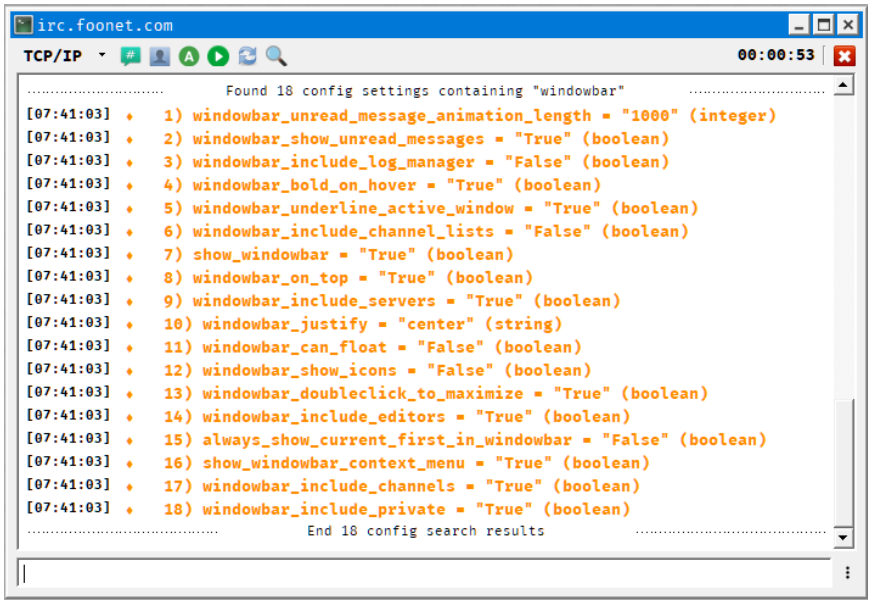
Users **/ignore**d in this way *must* be **/unignore**d with the **/unignore** command. Right clicking on an ignored user in the userlist will not give an option to unignore the user. To show the users messages again, either call **/unignore** with the specific entry used to ignore them as an argument (so, **/unignore *annoying.com*** in the example above), or clear the ignore list by clicking on "Clear ignore list" in the settings menu, or calling **/unignore *** to clear the ignore list.

Call **/ignore** with no arguments to see the ignored user list in full. Attempting to add an entry to the ignore list that already exists will result in an error.

Using the `/config` and `/user` Commands

The **`/config`** command allows users to edit the main MERK configuration file, **`settings.json`**, from within the client. ***Warning!*** It is possible to break or otherwise "mess up" MERK's configuration with this command. If this occurs, see ***Resetting MERK to Default Settings*** to undo the damage.

If called with no arguments, **`/config`** will list all the settings that can be edited with the command. To search settings, pass search terms to the command as an argument. For example, to search all settings that have the word "windowbar" in them, you could execute **`/config windowbar`**; this will print a list of all the settings that match the search term:



*A search with **`/config`** for settings containing "windowbar"*

Each listing contains the name of the setting, what the setting's value currently is, and what type of variable the setting is. To change a setting, pass the name of the setting as the first argument to **`/config`**, followed by the new setting value. The new value will be checked to make sure it's valid, and if so, stored as the new setting's value. For example, to change the **`show_windowbar`** setting to "False", you could execute:

```
/config show_windowbar false
```

If a setting's value is a string, all arguments after the setting will be assumed to be part of the new value. For example, if using **`/config`** to change the default "away" message to "I'm busy right now!", you could execute:

```
/config default_away_message I'm busy right now!
```

MERK should be restarted after changing settings with `/config`. Although many settings are applied instantly, some settings will only be applied after restarting.

/user works exactly the same way as **/config**, only it edits the user configuration file. For most settings, this just changes default values. However, for **finger** and **userinfo**, this setting changes the values sent when MERK receives a CTCP FINGER or USERINFO request, respectively.

To set a value to a blank string, pass ***** as the value to set.

Here's some example uses of **/user**:

```
/* This sets the default nickname */
/user nickname new_nickname

/* This sets the default username */
/user username new_nickname

/* This sets the CTCP FINGER response */
/user finger Hey! Stop pointing at me!

/* This clears the CTCP USERINFO response */
/user userinfo *
```

/config can also be used to import and export configuration files. To export the current configuration file, pass **export** as the first argument to **/config**, followed by the filename to export the configuration to; if no filename is passed, the user will be prompted for a filename. To import a configuration file, pass **import** as the first argument to **/config**, followed by the filename of the configuration file to import; if no filename is passed, the user will be prompted for a file to import. After importing a configuration file, MERK should be restarted as soon as possible.

/config can also be used to restart MERK. Use **restart** as the only argument to **/config** to restart MERK, using the same command-line initially used to start MERK.

Using the `/connect` and the Other Connection Commands

These commands are used to connect MERK to IRC servers:

- **`/connect`** connects to an IRC server, and does *not* execute any existing connection scripts
- **`/connectssl`** connects to an IRC server via SSL/TLS, and does *not* execute any existing connection scripts
- **`/xconnect`** connects to an IRC server, and executes any existing connection scripts
- **`/xconnectssl`** connects to an IRC server via SSL/TLS, and executes any existing connection scripts

Pass the hostname or IP address of the server as the first argument to these commands, and the port number to connect to as the second argument. These commands can be issued from the text input widget or from scripts. Please see ***Connecting to Servers*** for an example connection script that uses these commands.

These commands will not automatically reconnect on disconnection. To ensure that MERK reconnects to the server on disconnection, add **`re`** before connect in all of these commands:

- **`/reconnect`** connects to an IRC server, and does *not* execute any existing connection scripts, automatically reconnecting on disconnection
- **`/reconnectssl`** connects to an IRC server via SSL/TLS, and does *not* execute any existing connection scripts, automatically reconnecting on disconnection
- **`/xreconnect`** connects to an IRC server, and executes any existing connection scripts, automatically reconnecting on disconnection
- **`/xreconnectssl`** connects to an IRC server via SSL/TLS, and executes any existing connection scripts, automatically reconnecting on disconnection

Using the `/quit` and `/quitall` Commands

These commands are used to disconnect MERK from an IRC server. The **`/quit`** command disconnects from the IRC server associated with the context the command was issued in. It can be issued without an argument, or with a "quit" message as all arguments to the command; this will be used instead of the default "quit" message.

The **`/quitall`** works exactly the same way as **`/quit`**, only it will disconnect from *all* servers that MERK is currently connected to.

```
/* This will disconnect from the IRC server associated
   with the context the command is issued in. It will
   use "See you later!" as the "quit" message. */
/quit See you later!

/* This will disconnect from *all* connected servers with
   the same message used in the last example. */
/quitall See you later!
```

Using the `/shell` Command

The **`/shell`** command allows a script, or command, to execute an external process and store any output in an alias. The first argument to the command is the alias to store the output in, and all other arguments are executed as an external process.

For example, let's use **`/shell`** to write a script that calls the **`fortune`** Linux/UNIX program to display a fortune-cookie-like "fortune". We're going to call **`fortune`** with the **`-s`** command-line flag to generate a short fortune, store it in an alias named **`FORTUNE`**, and then send it as a message to the current chat:

```
restrict channel private
/shell FORTUNE fortune -s
/msg $_WINDOW Here's a fortune: $FORTUNE
```

This script:

1. Restricts the script's execution to channel and private chat windows, as it sends a message to the current window, which won't work in server window contexts.
2. Executes **`/shell`**, which calls **`fortune -s`** and stores the output in **`FORTUNE`**.
3. Sends a message to the current chat window containing the fortune.

Using the `/reclaim` Command

The **`/reclaim`** command can be used to "reclaim" a nickname currently in use by another user. When used, MERK will periodically try to change the nickname to the desired nickname until the desired nickname is "claimed" by the client. By default, MERK will attempt to change the nickname every 30 seconds; this can be changed by using **`/config`** to change the **`reclaim_nickname_frequency`** setting to the desired number of seconds.

Using the `/delay` Command

The **`/delay`** command is used to delay a command's execution for a set amount of time. Pass the number of seconds to delay the execution of the command as the first argument, and all other arguments will be used as the command to execute. For example, to wait 5 minutes (which is 300 seconds) before changing your nick to **`merk_user`**, you could use:

```
/delay 300 /nick merk_user
```

The command to be executed with **`/delay`** cannot contain any script-only commands, *even in scripts*. The command will fail with a "no command found" error.

Using the `/macro` Command

The **`/macro`** command allows users to create their own "commands"; it creates a named command, just like MERK's other commands, that executes a script, passing any arguments to the script. Arguments to the **`/macro`** command are tokenized like arguments passed to scripts, so if you have an argument that contains whitespace in it, contain it in quotes.

Macro names, much like aliases, cannot contain punctuation (with the exception of the underscore, _). They also cannot have the same name as existing commands.

As an example, we're going to create a macro named **`/hello`**. First, we'll write the script the macro will execute. Our macro will take one argument, a name, and send a message containing a greeting to that name to the current context. Any and all script commands can be used, including script-only commands:

```
restrict channel private
usage 1 /hello NAME
/msg $_WINDOW Hello, $_1!
```

Save this script to a file named **`hello.merk`** in MERK's "scripts" folder. Now that our script exists, we can create the macro for our command:

```
/macro hello hello.merk
```

Now, we can use our macro! To send a greeting to a user named "bob", just enter:

```
/hello bob
```

Thanks to the **`usage`** script-only command, if we call our macro with less arguments than we need, the scripts usage text will be displayed. And thanks to the **`restrict`** script-only command, if this macro is called from a server window, the macro will not execute and an error will be displayed.

Arguments to macros are tokenized (and treated) just like arguments to scripts. In the example above, the command **`/hello bob`** is treated exactly like the command **`/script hello.merk bob`** was called. If you wanted to send a greeting to a user that contains spaces in the "name" argument, use quotes to contain the argument; for example, to send a greeting to "Bob the Builder", you could use the **`/hello`** macro like:

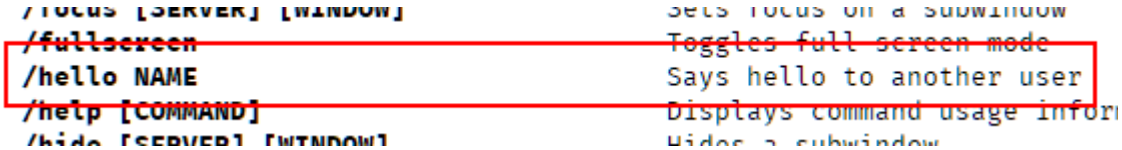
```
/hello "Bob the Builder"
```

Macros are not saved by MERK, so they must be recreated every time MERK is started. An easy way to make sure that the macros you use are always available is to create them in your connection scripts.

The **/macro** command can take two optional arguments; these arguments set the text that is displayed in the command list displayed by the **/help** command. The first optional argument sets the usage text (displayed in the left hand column of the **/help** display), and the second optional argument sets the command description (displayed in the right hand column of the **/help** display). To modify the previous example to use the usage and help arguments, we could create the macro with:

```
/macro hello hello.merk "/hello NAME" "Says hello to another user"
```

This adds our usage and help information to the **/help** display:



It is not possible to pass the help text to the **/macro** command without passing the usage text first; while both arguments are optional, and usage text may be passed *without* help text, if help text is desired, usage text *must* be passed as an argument first. If both usage and help text arguments are omitted, the macro is still added to the **/help** display; usage is set to the name of the macro, and the help text is set to "Executes script ", with the name of the script the macro executes.

Macros *cannot* have the same name as existing commands. Macro names must also start with a letter, and not a number or other symbol. Multiple macros *cannot* have the same name; a new macro with the same name as an existing macro will "overwrite" the older macro.

Macros are treated like commands in a lot of respects. If autocomplete for commands is turned on, autocomplete will work for macros. Macros are added to the list of commands that displays when the **/help** is used, and macros will be highlighted just like commands in the text input widget. If scripting is turned off in settings, the **/macro** command will be disabled.

Using the `/bind` and `/unbind` Commands

The **`/bind`** command allows users to create their own hotkeys for MERK. The first argument to **`/bind`** should be the key sequence that triggers the hotkey; it should contain no spaces. Valid key signifiers include "Ctrl" (for the control key), "Shift" (for the shift key), "Alt" (for the alt key), and "Meta" (for the meta key). For example, the sequence "Ctrl+M" will trigger if the control key and the letter "M" are pressed at the same time. Capitalization does not matter (so "Ctrl+M" is the same bind as "ctrl+m" or "Ctrl+m").

All other arguments after the first are the command to execute when the key sequence is pressed. The command will be executed in the context of whatever window is active when the sequence is pressed. So, to print the words "Hello world" to the current window with the control key and the letter "H" are pressed, you could use:

```
/bind Ctrl+H /print Hello world
```

Only a single command can be assigned to a bind; if the desired action is complex or contains more than one command, use the **`/script`** command. Also note that if the command contains any built-in aliases, the aliases will reference *the context of the window that the bind was set in at the time the bind was set*, and not the context of the active window when the hotkey is pressed. Commands that use built-in aliases should be in a script file that can be executed with the **`/script`** command; then, the build-in aliases in the script will reference the context of the active window triggering the bind.

For an example, let's create a bind that sends a greeting to the current window. It will use a built-in alias to send a message to the current channel or private chat. First, we'll create our script:

```
restrict channel private
/msg $_WINDOW Hello!
```

Save this script to a file named **`hello.merk`** in your "scripts" directory. Now that we have our script, we'll create our bind:

```
/bind Ctrl+H /script hello
```

Now, no matter what channel or private chat MERK is in, every time the control key is pressed at the same time as the "H" key is pressed, the message "Hello!" will be sent to the current chat. If a server window is active when the key sequence is pressed, an error will be shown.

Key sequence binds are not saved, so they must be recreated every time MERK is ran. The best way to make sure that your binds are created is to place your **`/bind`**s in connection scripts.

Be aware that this command can "overwrite" default hotkeys, like "Ctrl+C" for copy and "Ctrl+V" for paste, and MERK will not prevent this.

To remove an existing bind, use the **`/unbind`** command. Pass the key sequence of the bind to remove as the first argument; just like with the **`/bind`** command, capitalization doesn't matter, and the sequence should not contain spaces. For example, to remove the binding set in the example above:

```
/unbind ctrl+h
```

This removes our "Hello!" hotkey. Please note that the capitalization does *not* have to be the same as the **`/bind`** command that created the hotkey.

To remove all created hotkey binds, pass **`*`** as the only argument to **`/unbind`**.

Scripting MERK

There are two types of scripts in MERK: connection scripts, and all other scripts. Each script runs in its own thread, and no commands are blocking[7]; each command is executed immediately after being issued, without waiting for the command's process to end or resolve.

Connection Scripts

Connection scripts are the scripts entered into the connection dialog, and are intended to be executed as soon as the client connects to the server. Unlike other scripts, they are stored in the user configuration file, and, outside of connection, can only be executed with the script editor. Connection scripts have the context of the associated server window created when connection begins (see ***Context*** and ***Writing Connection Scripts***).

All Other Scripts

All other scripts are, well, *scripts*: a list of commands, one per line, issued in order. Scripts have a context, which is the window that they are called from or executed in. They can be executed in several ways:

- From the "**Run**" button on a server window's toolbar. The script will be executed in the server window's context.
- From the "**Run**" entry in a window's input menu. The text in the text input widget will be replaced with a call to the **`/script`** command to execute the selected file.
- From the "**Run**" entry in a window's chat display right-click menu. The text in the text input widget will be replaced with a call to the **`/script`** command to execute the selected file.
- By issuing the **`/script`** command. The script will be executed in the window that the command was called from's context.
- From the "**Run**" menu in a script editor window. The user can select which context to run the script in, or optionally select to run the script on *all* windows simultaneously (with each window running that script in the window's context).

When using the **`/script`** command, scripts are searched for as outlined in ***Directories and Configuration Files***; if the script can be found in this directory search, the path to that script can be omitted. For example, if a script named **`example.merk`** is in the **`/scripts`** directory, calling **`/script example.merk`** will execute that script. If the file extension to the script is **`.merk`**, than the file extension can be omitted; **`/script example`** will also execute the script in the previous example.

Scripts can have comments. Comments must begin with **`/*`** and end with **`*/`**, and can span multiple lines. Commands issued within comment blocks will be ignored, as will any text inside the comment block. For single line comments, the **`/rem`** command can be used; the command does nothing, and any arguments to it are ignored.

7 ... a process that is blocked is waiting for some event, such as a resource becoming available or the completion of an I/O operation. Wikipedia

Errors

For the most part, MERK handles script errors in two ways. Errors in script-only commands will prevent execution, while errors in other commands will halt the script execution when the error is encountered. If any of the following script-only command errors or conditions are detected, ***the script will not execute***:

- Calling **`wait`** with a non-number argument
- Calling **`usage`** with the wrong number of arguments
- Calling **`usage`** with a non-number as the first argument
- Calling **`restrict`** with the wrong number of arguments
- Calling **`restrict`** with an argument that is not server, channel, or private
- Calling **`only`** with no arguments
- Calling **`end`** with any arguments
- Calling **`insert`** no arguments
- Calling **`insert`** with a file that cannot be found, doesn't exist, or can't be read. Errors of this type will *not* display the filename the faulty **`insert`** call was in.
- Calling **`context`** with a context that doesn't exist
- Executing a script in a context that is not allowed by **`restrict`**
- Executing a script in a context that is not allowed by **`only`**
- Executing a script in a context that is not allowed by **`exclude`**
- Calling a command that cannot be called in scripts

Every other command error will display an error message (if script error messages are turned on in settings), and halt execution of the script. Error messages will be displayed for:

- Lines that do not contain a command
- Lines that start with `/` and are not followed by a valid command
- Calls to commands with an incorrect number of arguments
- Calls to commands with invalid arguments
- Scripts being executed in the "wrong" context
- Errors in command execution
- Using **`goto`** to "jump" to a line that contains a script-only command

If an error is encountered, an error message with be displayed. The error message will contain:

- **The line number the error occurred on**. On files that contain the **`insert`** command, the line number may not be accurate, as the script will include the **`insert`**ed file. Comments are stripped from scripts before execution, which may also result in inaccurate line numbers in error reports.
- **The name of the file the error is located in**. If the error is in a connection script, the filename displayed will be **`SERVER:PORT`** for the server's connection script. If the filename is not known or does not exist, the filename will be set to **`script`**. Errors in calling the **`insert`** command will *not* contain the filename the error occurred in, only what the erroneous call was.
- **A description of the error**.

Context

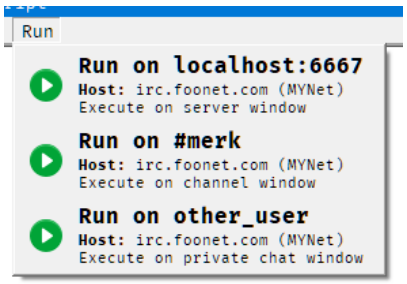
A script or command's *context* is a reference to the window the script or command is being executed in. Context is, for the most part, only necessary for scripts; the context for any commands issued by a window's text input widget is the window the command is being issued in.

Some commands can ignore an argument if they are for the current context; for example, when issuing the **`/part`** command, you can ignore the **`CHANNEL`** argument if the command is intended to be executed in the current window's context:

```
/* This leaves the current channel */
/part

/* This invites a user to the current channel */
/invite my_friend

/* This kicks a user from the current channel */
/kick my_enemy

/* This gives a user operator status in the current channel */
/mode +o my_friend

/* This sets the topic in the current channel */
/topic Welcome to my channel!
```

Context-less commands should *not* be issued by scripts, as it can get confusing if you run the script in the wrong context. However, if the script is being ran in any window's context, context-less commands are available to the script.

When running a script from a window, either through the server window's toolbar, or the input menu, the script is always ran in that window's context. The script editor "Run" menu allows you to choose which context to run the script in. If MERK is connected to more than one server, and in more than one channel or private chat on each server, the "Run" menu will give options to run the script in all of each context; for example, you can run a script in all connected channels.



*An example "Run" menu from the script editor. The client is connected to a server on **`localhost:6667`**, and is in the channel **`#merk`** while having a private chat with **`other_user`**. Each selection will run the script in the specified context.*

A window's context is "connected" to any other window contexts that share the same network stream. Commands issued from the text input widget in server or chat windows can only effect other windows that share the same server connection. This is called a "shared context".

For example, let's assume that MERK is connected to two servers, **`irc.example.com`** and **`irc.other.net`**. On **`irc.example.com`**, MERK is connected to two channels, **`#merk`** and **`#python`**. On **`irc.other.net`**, MERK is connected to the channel **`#qt`**. In this example, **`#merk`** and **`#python`** have a shared context, while **`#qt`** doesn't have a shared context with the other two window. Commands issued in **`#qt`** will not be able to have an effect on **`#merk`** or **`#python`**, and vice versa.

Scripts can use the **`context`** command to "move" the script to another context. The **`context`** command will search for all windows, no matter what context. The order **`context`** looks for windows is:

1. Windows that have a shared context with the context the script was executed in.
2. Windows from all contexts.
3. Server windows.

`context` will move to the *first* window it finds with the name passed to it. If you are in multiple channels with the same name, this may be problematic.

If a script is intended to only be ran in a specific type of context, the **`restrict`** command can be used, in one of three ways:

- **`restrict server`** – The script will only run in server windows or connection scripts
- **`restrict channel`** – The script will only run in channel windows
- **`restrict private`** – The script will only run in private chat windows

A script that has been "restricted" will *only* run in the context specified, and will not execute and show an error if it is ran in another context. Up to two contexts can be passed to **`restrict`**. For example, to make sure that a script is ran *only* in chat windows, use **`restrict channel private`**.

Outside of scripts, it *is* possible to "move" to another window's context via commands. The **`/show`** command can move to the context of any other window.

Aliases

Aliases are tokens that can be created to insert specific strings into your input in the client (if the "Interpolate aliases into input" setting is turned on, which is the default) or into your scripts. They function kind of like variables[8] do in programming languages. As an example, let's create an alias named 'GREETING', and set it to the value 'Hello world!':

```
/alias GREETING Hello world!
```

Now, if you want to insert the string "Hello world!" into a command or any output, you can use the alias interpolation symbol, which is **$** by default, followed by the alias's name, to insert your alias into the command or output:

```
/msg #mychannel $GREETING
```

This sends a message to **#mychannel** that says "Hello world!" to everyone in the channel!

Alias names *must* start with a letter, and not a number or other symbol. This is to prevent overwriting built-in aliases created for each window's context (see ***Built-In Aliases***). Alias names also cannot contain punctuation, except for underscores.

To create an alias, use the **/alias** command. To see a list of all aliases set for the current window, issue the **/alias** command with no arguments. To delete an alias, issue the **/unalias** command.

Aliases can also be used as macros, and can contain an entire command. For example, let's say that you like to issue a greeting to everyone that enters a channel, but typing **/msg #mychannel Hello, and welcome!** is a pain to type every time someone joins, you could create this alias:

```
/alias GREETING /msg $_WINDOW Hello, and welcome!
```

Now, whenever someone joins your channel, just type **$GREETING** into the text input widget to send your message! The above example uses a built-in alias, which is explained in ***Built-In Aliases***.

All aliases are *global* in scope; that is, they are available to all and every script executed on the client after they are created. They can also be changed by any script or command. Aliases created by connection scripts will be available and visible to any scripts executed after the connection script (including other connection scripts). Any script can also delete an alias with the **/unalias** command.

Built-in aliases cannot be deleted with the **/unalias** command. The client will display an error that says the alias doesn't exist if attempted.

8 https://en.wikipedia.org/wiki/Variable_(computer_science)

Built-In Aliases

Each window has a number of aliases for use that are built-in to the window's context, and do not require the user to create them. Built-in alias names start with an underscore (_) and are all uppercase. Some built-in aliases (see ***Script Arguments***) are only created in certain circumstances; these have a gray background.

Alias	Value
_0, _1,...	Any arguments that have been passed to the currently running script; the first argument will be set to **$_1**, the second argument will be set to **$_2**, and so on. **$_0** will contain all arguments, separated by spaces; if no arguments have been passed, **$_0** will be set to **none**.
_ARGS	The number of arguments passed to a script. If no arguments have been passed to the script, this will be set to **0**.
_CLIENT	The name of the IRC client, MERK.
_CONNECTION	If the connection type of the server the window is connected to; if connected via SSL/TLS, this will be set to **SSL/TLS**, otherwise it will be set to **TCP/IP**.
_COUNT	The number of users in the current channel. For server and private chat windows, this will be set to **0**.
_CUPTIME	The number of seconds that MERK has been running.
_HOST	The reported hostname of the server the window is connected to; if that is not known, then this will be set to the server's address, a colon, and the server's port.
_DATE	The current date, in "MM/DD/YYYY" format.
_DAY	The current day of the week.
_EDATE	The current date, in "DD/MM/YYYY" format.
_EPOCH	The current time in UNIX epoch format.
_FILE	The full filename of the currently running script. If called from an **/insert**ed file, this will contain the name of the script being executed, not the **/insert**ed file. If the current script does not have a filename, this will be set to **script**.
_HCHANNELS	The number of hidden channels on the server the window is connected to. If this is not known, this will be set to **0**.
_MODE	Any modes set on the user associated with the window. If no modes are set, this will be set to **none.**
_MONTH	The name of the current month.
_NETWORK	The network the server the window is connected to is on; if this is not known, this will be set to **unknown**.
_NICKNAME	The user's current nickname.
_ORDINAL	The current day of the month.
_PORT	The port on the server the window is connected to.
_PRESENT	If the window the alias is being used in is a channel window, this will contain a list of users in that channel, separated by commas. If there are no users in the channel, or if this is used in a non-channel context, this will be set to **none**.
_REALNAME	The user's realname, as set in user settings.
_RELEASE	The URL for the current latest release of MERK, as known at the release of the running version.

Alias	Value
_RVERSION	The current release version of MERK, as known at the release of the running version.
_SCHANNELS	The number of visible channels on the server the window is connected to. If this is not known, this will be set to **0**.
_SCOUNT	The number of visible users on the server the window is connected to. If this is not known, this will be set to **0**.
_SCRIPT	The name of the file, without the full path, of the currently running script. Only present in scripts that have been executed with the **/script** command. If called from an **/insert**ed file, this will contain the name of the script being executed, not the **/insert**ed file. If the current script does not have a filename, this will be set to **script**.
_SERVER	The server the window is connected to; this will be the address used to connect to the server, not the server's reported hostname.
_SOFTWARE	The server software the server the window is connected to is using. If this is not known, this will be set to **unknown**.
_SOURCE	The URL to MERK's source code.
_STAMP	The current time, following the format setting for timestamps.
_STATUS	If the window is associated with a channel, this will contain the window's channel status (**operator**, **voiced**, etc.); otherwise, this will be set to **normal**.
_SUPTIME	How long, in seconds, the client has been connected to the server associated with the current window.
_TIME	The current time, in 24-hour format.
_TOPIC	If the window the alias is being used in is a channel window, this will contain the channel's topic, if there is one. If the channel does not have a topic, or if the window is not a channel, this will be set to **No topic**.
_UPTIME	How long the window the script is being ran in has been connected or has been in use, in seconds.
_VERSION	The current version of MERK in use.
_USERNAME	The user's username, as set in user settings.
_WINDOW	The name of the window the script is being used in.
_WTYPE	The type of window the alias is being used in; either **server** for server windows, **channel** for channel windows, or **private** for private chat windows.
_YEAR	The current year.

Built-in aliases can be very useful in scripts, where the script may not "know" what context it is running in:

```
/* This sends a message to the current channel */
/msg $_WINDOW Hello, everybody! My name is $_NICKNAME

/* This sets the current channel's topic */
/topic $_WINDOW We've been around for $_UPTIME seconds!
```

They can also be used to display information about the client:

```
/* This sets the default away message */
/config default_away_message $_CLIENT $_VERSION – I'm away right now
```

Script Arguments

Arguments can be passed to a script with the **/script** command; just pass them as arguments to the command following the script file name. The **/script** command is the *only* way to pass arguments to a script.

Script arguments are tokenized[9] differently than arguments for commands. In arguments for most commands, arguments are considered to consist of either a single word, with no spaces, or a number of words separated by spaces. A channel name, for example, or a nickname may be an argument; the **/msg** command looks for a channel or nickname as the first argument, with all other arguments being the message to be sent. Arguments to scripts can contain spaces, and the number of arguments is important. To use an argument with spaces in it, contain the argument with quotation marks.

```
/* This calls a script with a single argument */
/script myscript.merk "Hello, world!"

/* Here are multiple arguments, with spaces in each */
/script test.merk "First argument!" "Second argument!" "And third!"
```

To access these arguments, a built-in alias is created for each one, and another is created for all arguments. Each built-in alias is named with the number of the argument: **$_1** for the first argument, **$_2** for the second, **$_3** for the third, and so on. The built-in alias **$_0** contains all arguments passed to the script, joined by single spaces; if no arguments have been passed to the script, **$_0** will contain the string **none**.

To make sure your script is called with the right number or arguments, use the **usage** command. As the first argument to **usage**, pass the number of arguments your script requires. All arguments after this first will be displayed as the error message if your script it called with an improper number of arguments.

As an example, let's write a script that sends a greeting to someone in the current chat. Our script will require a single argument, a name. When executed with the right number of arguments, it will send the greeting to chat, and if executed with too few arguments, will tell the user how to use the script. Open the script editor, and paste the following code into it, saving the file as **greet.merk**.

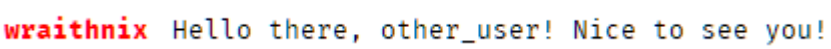
```
/*
     This script requires a single argument.
     If none or more than one argument is passed,
     display script usage information.
*/
usage 1 Usage: /script $_SCRIPT NAME

/msg $_WINDOW Hello there, $_1! Nice to see you!
```

9 https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization

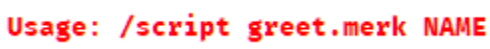
Let's execute our script! In the text input widget, type the following and hit enter:

```
/script greet other_user
```

Our greeting is sent to the current chat:



If we executed our script with no arguments, an error message is displayed:



Connection scripts are *never* called with any arguments. Scripts executed by any way other than the **`/script`** command are *never* called with any arguments.

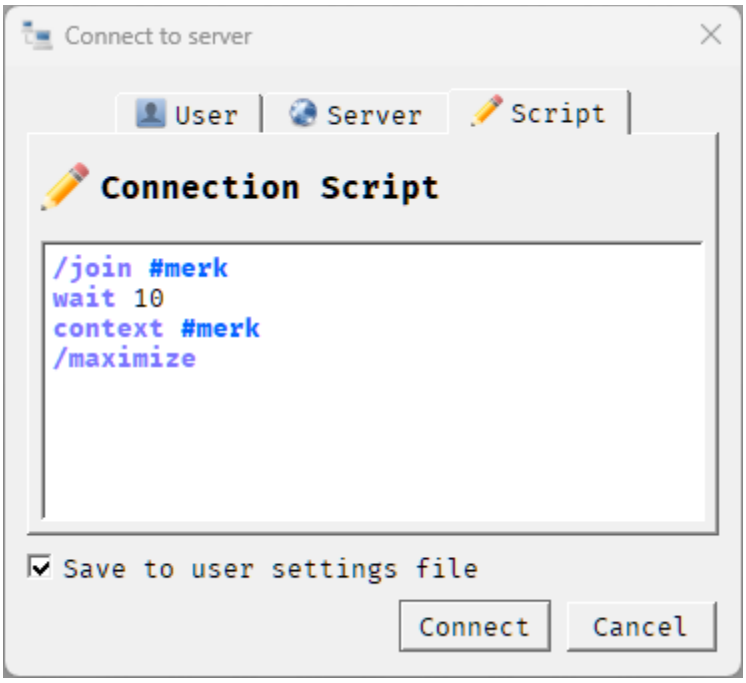
Several built-in aliases are created for scripts. **`$_FILE`** contains the full filename of the script (including path) being called, **`$_SCRIPT`** contains the filename of the script without the path, and **`$_ARGS`** contains the number of arguments the script was called with. The example above has a use of the **`$_SCRIPT`** built-in alias. If a script is executed from the editor (and thus doesn't have a filename), both **`$_FILE`** and **`$_SCRIPT`** will be set to **`script`**.

Scripts executed with the "Run" menu in the editor are *never* called with any arguments, and will only have the **`$_SCRIPT`** or **`$_FILE`** built-in aliases if the script has been saved to or loaded from a filename. Scripts executed with the "Run script" button on server window toolbars will have the **`$_SCRIPT`** and **`$_FILE`** built-in aliases, but as they are *never* called with arguments, will have the **`$_ARGS`** built-in alias set to **`none`**.

Argument built-in aliases can be used with **`insert`**ed scripts, but they will always reference the script that is being executed, not the **`insert`**ed script. For example, if an **`insert`**ed script includes the built-in alias **`$_1`**, that alias will interpolate to the first argument that was passed to the script that is being executed, *not* the **`insert`**ed file.

Writing Connection Scripts

Connection scripts are the scripts that can be entered in the connection dialog, and are executed as soon as the client completes connecting to a server. A connection script's context is the server window created when connecting. In fact, calling **`restrict server`** to restrict the connection script's execution to a server window will pass successfully, while using **`restrict`** with any other context type will cause the connection script to not execute.



To issue commands that will have an effect on another window, use the **`context`** command to move the script to that window's context.

Before **`context`**ing to another context, be aware that that window (and the context) may "not exist" yet. The channel window may not be rendered yet, the private chat that you intended to start has not started yet, etc. The **`wait`** command will help you in these situations, so you can make sure that all the contexts for your script have been created before you issue commands.

For example, let's say that when you connect to your favorite server, automatically join your favorite channel, **`#merk`**, say hello, and maximize the channel window. Your connection script might look like:

```
/alias FAVORITE #merk
/join $FAVORITE
/msg $FAVORITE Hello, everybody!
wait 10
context $FAVORITE
/print $_WINDOW Maximizing $FAVORITE!
/maximize $_WINDOW
```

How long to **`wait`** after connection will take some trial and error, due to many factors: the speed of your Internet connection, the speed of your computer, how busy the server is, how big of a log the client is loading for display, among other things. When in doubt, a longer **`wait`** is preferable to a shorter one, to make sure that your script executes properly. When first writing a connection script, try **`wait 30`** to pause the script for 30 seconds, and tweak from there.

Example Scripts

Wave

This is a simple script that sends an emoji to the current chat, and adds a shortcut to executing the script. When executed in a channel or private chat window, it will send the "wave" emoji to the current chat. It also creates an alias, allowing the user to type **`$wave`** to send the wave emoji.

```
/*
   Wave Script
   By Dan Hetrick
*/

restrict channel private
/msg $_WINDOW :wave:
/alias wave /script $_FILE
```

What this script does specifically:

1. Restrict the scripts execution to channel and private chat windows.
2. Sends the "wave" emoji to the current chat
3. Creates an alias named "wave" that will re-execute the script

Greeting

This script sends a greeting to the as a private message to a user. It takes a single argument, the username of the person the greeting is being sent to.

```
/*
   Greeting Script
   By Dan Hetrick
*/

usage 1 Usage: /script $_SCRIPT nickname
/msg $_1 Hello! Nice to see you!
```

What this script does specifically:

1. Makes sure the script is called with a single argument
2. Sends a greeting as a private message to the user set in the single argument

To make this a little easier, we'll create a macro for our greeting script. Assuming that our greeting script was saved to a file named **`greeing.merk`** in MERK's "scripts" directory:

```
/macro greet greeting "/greet USER" "Sends a greeting to the current chat"
```

Now, we can call our script with a new custom "command", **`/greet`**. So, if we wanted to send a greeting to a user named "Bob":

```
/greet Bob
```

This would send a message to the user "Bob" that consists of "Hello! Nice to see you!".

Example Connection Script

This script should be set as a connection script. Upon connection to the server, it will log the user into **NICKSERV**, join a channel the user owns, tell **CHANSERV** to give them operator status in the channel, set the channel topic, maximize the channel's window, and send a greeting to the channel

```
/*
   Example Server Connection Script
*/

restrict server
/alias USERNAME my_username
/alias PASSWORD my_password
/alias CHANNEL #my_channel
/alias TOPIC Welcome to my channel!
/alias GREETING $_NICKNAME is here, everybody!

/msg nickserv IDENTIFY $USERNAME $PASSWORD
wait 5
/join $CHANNEL
/msg chanserv OP $CHANNEL
wait 10
context $CHANNEL
/topic $TOPIC
/maximize
wait 1
/msg $_WINDOW $GREETING
```

What this script does specifically:

1. Restrict the script's execution to server windows
2. Sets an alias for the user's **NICKSERV** username
3. Sets an alias for the user's **NICKSERV** password
4. Sets an alias for the user's channel
5. Sets an alias for the channel's topic
6. Sets an alias for the greeting to send once everything else is done
7. Logs into **NICKSERV** with the set username and password
8. Waits 5 seconds
9. Join the set channel
10. Tells **CHANSERV** to give the user operator status in the set channel
11. Waits 10 seconds
12. Switches contexts to the channel window
13. Sets the current channel's topic
14. Maximizes the current channel window
15. Waits 1 second
16. Sends the greeting to the current channel

Inserting Files

If there's data or aliases that you want to use in more than one script, the **/insert** command makes that easy. In the last example, a script was used to login to **NICKSERV**. In this example, we're going to store our login information in one script, and use it in another.

login.merk

```
/*
   NICKSERV Login
*/

/alias USERNAME my_username
/alias PASSWORD my_password
/alias LOGIN_TO_NICKSERV /msg NICKSERV IDENTIFY $USERNAME $PASSWORD
```

Now, to login to **NICKSERV** from another script, use the **/insert** command to insert this file into the script, and issue the full command:

```
insert login.merk
$LOGIN_TO_NICKSERV
```

Once all aliases have been interpolated into the script, this will end up being the script that is executed:

```
/msg NICKSERV IDENTIFY my_username my_password
```

Showing the Local Temperature

This script will use the **/shell** command to fetch the currently temperature in a specific location. You may have to edit the call to get the temperature in your location. I live in Detroit, Michigan, in the United States of America, so that's the location I'm going to use. We're going to use the **curl** executable to get the temperature from https://wttr.in/.

```
restrict channel private
/shell TEMP curl.exe -s "https://wttr.in/Detroit?format=%t"
/msg $_WINDOW It's currently $TEMP here in Detroit
```

This script:

1. Restrict the script's execution to channel or private chat windows only, as it sends a message to the current window.
2. Use the **/shell** command to call **curl.exe** to fetch the current temperature, and store it in the **TEMP** alias.
3. Send a message to the current window containing the **TEMP** alias.

Connecting to Servers

Scripts can call other scripts, allowing scripts to be "chained"; that is, to execute one after the other. This script is an example of a connection script that connects to multiple servers, and executes multiple scripts.

First, let's create our initial connection script. We're going to use it upon connection to UnderNet. It will login to our X account before connecting to two other servers.

```
restrict server
/msg X@channels.undernet.org login username password
/join #merk
/xconnect palladium.libera.chat 6667
/xconnectssl irc.underworld.no 6697
```

This script:

1. Restricts the script's execution to server window contexts.
2. Sends a private message to UnderNet's user service bot, logging into an account.
3. Joins the **`#merk`** channel.
4. Connects to **`palladium.libera.chat`** on port 6667, executing any existing connection script when it connects.
5. Connects to **`irc.underworld.no`** via SSL/TLS, on port 6697, executing any existing connection script when it connects.

Scripts can also execute other scripts with the **`/script`** command.

Dice Rolling Script

This script uses the **`/random`** command to simulate a dice roll and send the result to the current channel. Pass the number of sides of the dice to roll as the first argument to the script; so, to roll a 20 sided die, pass **`20`** as the argument to the script:

```
restrict channel private
usage 1 Usage: /script $_SCRIPT NUMBER_OF_SIDES
/random roll 1 $_1
/msg $_WINDOW Rolling a $_1 sided die: $roll
```

This script:

1. Restricts the script's execution to chat window contexts.
2. Sets the usage text for the script, and makes sure that the script is called with one argument.
3. Generates a random number from 1 to the number passed as an argument to the script. If the argument is not a number, the script will stop executing and display an error. The generated random number is stored in the alias **`$roll`**.
4. Sends a message to the current chat showing the random number that was generated.

In the game Dungeons & Dragons, rolling a 20 on a 20-sided die is a "critical", which usually means the roll succeeds or deals extra damage. Here's a version of the script rolls a 20-sided die, and notifies if a critical is rolled:

```
1 restrict channel private
2 /print Rolling...
3 /random roll 1 20
4 if $roll (is) 20 goto 7
5 /msg $_WINDOW Rolled a $roll
6 end
7 /msg $_WINDOW Critical! Rolled a $roll
8 end
```

This script:

1. Restricts the script's execution to chat window contexts.
2. Generates a random number from 1 to the number passed as an argument to the script. If the argument is not a number, the script will stop executing and display an error. The generated random number is stored in the alias **`$roll`**.
3. Displays a message to the user
4. Checks to see if the roll was a 20, and if it was, "jumps" to line 7
5. Displays the roll to the current chat
6. Ends the script
7. This is the line our **`if`** "jumps" to; it displays to the current chat that the roll was a critical
8. Ends the script

Now that our dice rolling script is written, we can create a macro for our script, or bind the script to a hotkey. The following examples assume that dice rolling script is saved to a file name **`roll.merk`**, saved in MERK's "scripts" directory.

First, we'll create a macro. Our macro will be named "roll20", and will take no arguments:

```
/macro roll20 roll "/roll" "Rolls a 20 sided die, and shows the results"
```

Now, we've effectively created a new command, **`/roll20`**, that executes the dice rolling script every time it is called.

We can also create a hotkey for our script with the **`/bind`** command. Every time the user presses the control key and the "R" button at the same time, our dice rolling script will get executed in the current active window's context:

```
/bind Ctrl+R /script roll
```

Custom Day-of-the-Week Away Message

This script sets MERK's default away message to a custom message depending on what day of the week it is.

```
if $_DAY (is) Monday goto 9
if $_DAY (is) Tuesday goto 11
if $_DAY (is) Wednesday goto 13
if $_DAY (is) Thursday goto 15
if $_DAY (is) Friday goto 17
if $_DAY (is) Saturday goto 18
if $_DAY (is) Sunday goto 21
halt $_DAY is not a recognized day!
/config default_away_message I hate Mondays
end
/config default_away_message At least it's not Monday
end
/config default_away_message It's Wednesday, my dudes
end
/config default_away_message It's almost Friday...
end
/config default_away_message It's Friday!
end
/config default_away_message I love Saturday!
end
/config default_away_message It's almost Monday...
end
```

This script uses **`if`** and the built-in alias **`$_DAY`** to check what day of the week it is, and sets the default away message, using **`goto`** and **`/config`**, with a special message for each individual day. If, for some reason, the name of the day of the week stored in **`$_DAY`** is not recognized, the script uses the **`halt`** command to show an error message to the user.

"Trout" Command

Many IRC clients have had the ability to create a "trout" command or macro: a command that takes a single nickname as an argument, and sends a CTCP action message to the current chat that says "slaps NICKNAME with a trout". Here's how you can do that in MERK!

First, we're going to write a script that takes a single argument and uses it to send a CTCP action message to the current chat. We'll save this script to a file named **`trout.merk`** in your scripts directory:

```
restrict channel private
usage 1 Usage: /trout NICKNAME
/me slaps $_1 with a trout
```

Now that we have our script, we'll use it to create a macro. We can put this command into the text input widget, a connection script, or another script:

```
/macro trout trout.merk "/trout NICKNAME" "Slaps someone with a trout"
```

Our "trout" command is complete!

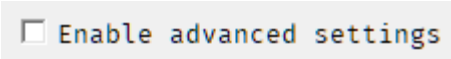
Advanced Settings

The last section in the "Settings" dialog shows a number of settings that can be used to fundamentally change how MERK works. ***WARNING:*** Changing these settings may break your installation of MERK, break any existing scripts, or fill up your hard drive. If this occurs, please see ***Resetting MERK to Default Settings***.

Changing these settings is not recommended, but may be desired.

Options

To change any of these settings, advanced settings must be enabled by clicking this checkbox:



Unchecking this checkbox will reset all the advanced settings to the value stored in the configuration file, and no settings will be saved. This *must* be enabled when clicking the "Apply" or "Apply & Restart" buttons for any changes to be saved. When changing any any advanced setting, restarting MERK is recommended.

- **Connection heartbeat.** This sets how often MERK "pings" the IRC server to keep the network connection active. The default is once every 120 seconds.
- **Max message length**. IRC has a hard limit of 512 characters for chat messages, and this must include the user's nickname and the appropriate chat command. 400 characters is the limit that MERK uses to determine if a chat message needs to be split into multiple messages.
- **Flood protection for long messages**. Chat messages that have been split up for exceeding the maximum message length will be sent once per second, to avoid triggering a server's flood protection. Turned on by default.
- **Show server pings in server windows**. MERK sends "pings" to the IRC server to keep the network connection active. Checking this box will display whenever the client receives a "ping" reply from the server.
- **Save all system messages to log**. Checking this option will save nearly all messages displayed if logging is turned on. This will drastically increase the size of saved logs.
- **Write all network input and output to STDOUT**. This will show all IRC network traffic in STDOUT, which is normally printed to the console MERK is being ran from. This will not display anything if MERK is being executed with the PyInstaller executable (without using additional software to view STDOUT).
- **Write all network input and output to a file in the user's settings directory**. This will write all network input and output to a file or files in the **`settings`** directory. Each IRC connection will have its input and output written to a file named **SERVER-**

`PORT.txt`. So, a connection to **`irc.libera.chat`** on port **`6697`** would have its input and output written to **`irc.libera.chat-6697.txt`**. **WARNING!** This will write a *lot* of data to your hard drive.