mIRC Scripting Primer

Version 1.00

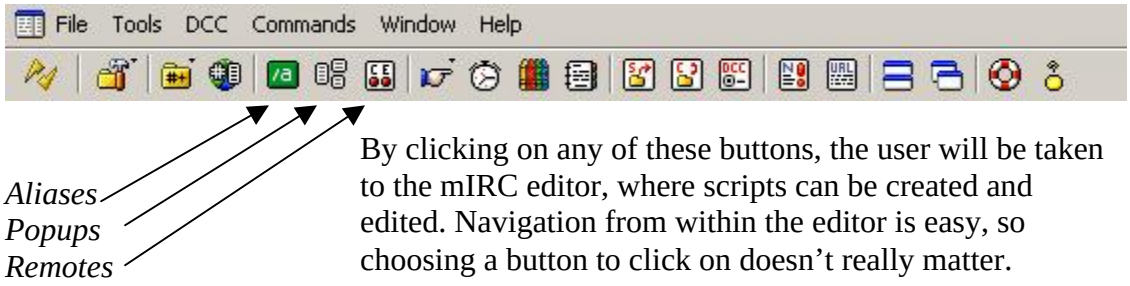
Introduction

This paper is a basic mIRC scripting primer- packed with step-by-step tutorials. This white paper assumes a basic comprehension of IRC commands (perhaps six months of IRCing should be adequate). This paper will run through various concepts and programming aspects, but by no means can this paper replace the standard help files which list all commands and their syntaxes. In order to fully utilize this paper, you will need at least one copy of the mIRC application and an Internet connection. A small segment towards the end deals with modifying the mIRC executable- trying is not recommended, however a HEX editor would be a requirement if one wishes to attempt understanding mIRC at a more complex level.

If you are familiar with IRC commands, you will soon notice that scripting is extremely simple. All you really have to do is understand some basic control structures, some formatting and a few more commands. You will also learn that /help is your best friend.

Fine Tuning mIRC

There are three distinctly different types of 'scripts'. Commonly a 'script' is a file that contains three types of scripts: Aliases, Remotes and Popups. Aliases are custom commands that can be used like functions. Remotes are instructions to react to particular conditions- like a particular phrase or ctcp request. Popups define the interface.



By clicking on any of these buttons, the user will be taken to the mIRC editor, where scripts can be created and edited. Navigation from within the editor is easy, so choosing a button to click on doesn't really matter.

The mIRC editor cannot be maximized unfortunately, but it can be resized, and the fonts it uses to display scripts can be changed. These modifications are highly recommended, as they will simplify the scripting process by making the editor far less intimidating.

Before getting into scripting, we recommend changing several of mIRC's built in options. Explore the file-options dialog box for the following items, and change them if desired:

IRC:	Check 'Show Addresses' Uncheck 'Use Short Joins/parts'
Sound:	(Requests) Uncheck 'Accept Sound Requests'
Sound:	(Requests) Uncheck 'Listen for !nick file get requests'.
General:	Set the Window Buffer to a high number. (300,000+)
General:	Remove the line separator altogether.
General:	*Escape key minimizes windows (it's helpful!)*

Aliases

Now that you are familiar with mIRC's interface, lets start scripting. Open up the aliases editor and input the following:

/cool /me thinks scripting is cool.

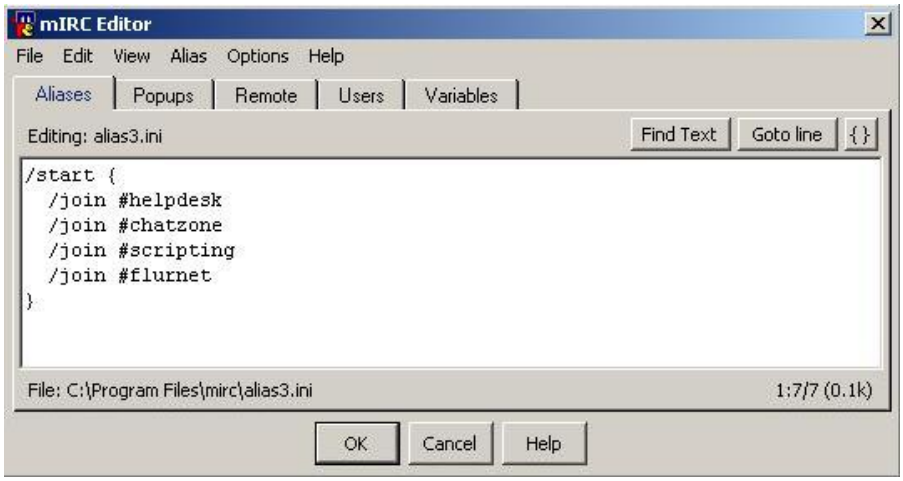
Click OK to save (or if you want to be complicated, select save from the file menu and hit cancel, hehe.) Join any channel and type /cool

** flur thinks scripting is cool.*

When mIRC receives the 'command' /cool, it performs /me thinks scripting is cool. Simply speaking, mIRC scripting is creating 'macros' out of commonly used IRC commands. However, you will notice when you get into scripting that there are far more IRC commands then you had ever anticipated. You will also come to realize that /help is a neophyte's best friend. It's also my best friend.

Let's try a slightly more complicated alias now. We are going to try and write an alias that will join channels we commonly use- let's name it /start so that we can just type /start as soon as we connect to an IRC server:

```
/start {
  /join #helpdesk
  /join #chatzone
  /join #scripting
  /join #flurnet
}
```



This alias performs more than one command, hence we open {}'s. This is part of mIRC's scripting control structure which we will discuss in more detail a little further. However, it's important to note that when within these brackets, the / (control character) is no longer required- but mIRC doesn't mind if you include them.

Getting accustomed to using {}'s is critical- they become extremely necessary with more complicated scripts, however, the same alias used in the last example could be written like this:

/start /join #helpdesk | /join #chatzone | /join #scripting | /join #Flurnet

When within a script, mIRC interprets a pipe character "|" as a carriage return- it performs the commands from left to right, joining #helpdesk first and #Flurnet last. However, using pipe characters instead of brackets is considered poor programming etiquette- so throughout this paper we will use them as little as possible.

Let's try an alias that utilizes some variables now. We will try and write an alias that will react to an extra variable, such as a nick (ie. /op nick):

/fakevoice { mode $chan +v-v $1 $1 }

Are you afraid? Don't be, we haven't even scratched the surface. Let's break this alias down- /fakevoice is the name of the command. After the name begins the commands, to make things clearer we encapsulated the commands with those brackets, and omitted including the command character before our first command. What are those $ things? They are variables (we will discuss them in more detail later.) Before executing this alias, mIRC will replace all variables with their actual contents. In this case, $chan will be replaced with the channel on which we issued the command and $1 will be replaced with the first word we type after the command. So, lets join a new channel (we will need ops to try this alias out) and type /fakevoice nick. mIRC will replace all the variables, and perform something like this:

/mode #newchannel +v-v nick nick

This is what should happen:

**** flur sets mode: +v-v flur flur*

For those of you that aren't too familiar with IRC commands, the mode command lets you issue more then one mode change per line- we basically gave 'nick' (in my case, I typed /fakevoice flur) a +voice and a –voice in the same line, so flur gets nothing- hence the teasing name "fakevoice".

Control Structures

We saw multi-line aliases encapsulated with {}'s. This is considered part of the control structure, because these brackets define the alias. Scripting is all about creating aliases and controlling them in an intelligent way. Control structure is the intelligence that controls how the script acts. Let's write some aliases as practical examples, but first lets take a look at the 'if' command.

if (v1 operator v2) { commands }
elseif (v1 operator v2) { commands }
else { commands }

If is just like any other mIRC command, except that it's rarely used on the command line (as in, while you are chatting). However, this is probably the most commonly used command while scripting. It evaluates two variables and performs commands accordingly.

v1 and v2 can be strings, integers or variables. There are far more operators then you'd expect that can evaluate the two variables. We'll start off with a simple alias that utilizes the if command:

```
/testnick {
  if ( $me == $1 ) { /say $1 is $me }
  else { /say $1 is not $me }
}
```

Issue *"/testnick blahblah"* on a channel, and your client will say "blahblah is not nick" with nick being what ever your current nickname is set to. Let's analyze this alias more closely. Once *testnick* is run, it compares v1 with v2 using == as the operator. So, if $me (which mIRC will replace with your current nick- described in the variables section of this paper) is equal to $1 it will perform *{ /say $1 is $me },* otherwise it will perform *{ /say $1 is not $me }.*

We could have made the script case sensitive by using === instead of ==. New mIRC binaries can perform complicated comparisons whereas in the past one had to script them oneself. Type /help if-then-else for a full list of operators. Let's utilize the *elseif* command by adding on to this alias:

```
/testnick {
  if ( $me == $1 ) { /say $1 is $me }
  elseif ( $1 == $null ) { /echo you forgot to include a variable, stupid. }
  else { /say $1 is not $me }
}
```

Now the alias will check the first if statement, if it fails it will try *elseif*, if that fails too it will default to else. What our alias does now is if the nick doesn't match it will check to

see if a variable ($1) was defined- maybe the user was confused and tried typing /testnick without specifying a nick- if this is the case, $1 will equal $null (a variable that defines an empty set) and thus the command *{ /echo you forgot to include a variable, stupid. }* will be performed. Naturally, multi-line commands are allowed, provided they are within {}s.

If you notice commands you aren't accustomed to, such as /echo, you should type /help /echo and read up on it. You can have as many *elseif* statements as you like, but you should note that once an if statement is satisfied the script will perform the commands it is instructed to then halt.

An exclamation mark (!) can be used to negate a value, so instead of the *elseif* line we used earlier, we could have written it like this:

```
  elseif ( !$1 ) { /echo you forgot to include a variable, stupid. }
```

This will evaluate whether or not $1 exists, and if not, it will perform the commands encapsulated by the brackets.

v1 and v2 can also be strings, *if ($me == flur)* will work just fine.

Let's have a look at some other types of loops. The alias that follows utilizes a 'goto' loop, heavily frowned upon by programming communities left and right. The goto command is redundant in the face of if and while loops, but goto's are great fun and extremely simple (and versatile) however, debugging can be tedious. Let's rewrite the alias we wrote above using gotos- notice how much longer it is.

```
/testnick2 {
  if ( $me == $1 ) { goto match }
  elseif ( $1 == $null ) { goto novar }
  else { goto nomatch }
:match
  say $1 is $me
  halt
:novar
  echo you forgot to include a variable, stupid.
  halt
:nomatch
  say $1 is not $me
  halt
}
```

When using goto loops, the halt command is extremely important. When mIRC receives a halt instruction, it simply stops processing the rest of the alias. Halt can be replaced with goto end, but then :end must be defined at the bottom of the alias. To break out of infinite loops, press control-break (this is a common situation in goto loops.)

Lastly, we come to while loops, the most versatile of the loop family. A while loop looks similar to an if loop, in the sense that it compares two values over and over again performing the *{ command }* until a particular variable is met. Let's try a new alias as an example. (This example uses some special variables that will be explained in more detail later.)

```
/count {
  var %i = 1
  while (%i <= 10) {
    echo 2 %i
    inc %i
  }
}
```

Looks cryptic? Don't panic- just read on. As usual, we define the aliases name (count) then open our first set of brackets defining the whole alias (the first set of brackets doesn't close till the end). In the first line, before we begin the while loop, we define an internal variable called %i and give it the value "1". Then we begin the loop:

```
  while (%i <= 10) { }
```

So, while %i is smaller than or equal to 10, the script will perform the commands defined in brackets. The above alias may look more complicated then it actually is because the while loop commands are multi-line, hence we opened another set of brackets within the first set (something you should get VERY accustomed to doing.)

What this alias intends to do is to echo "%i", then increment it by 1, and run again until *(%i <= 10)* is satisfied. If you haven't yet typed /help /inc, do so now. The number 2 after the echo command defines the color that echo should echo output in, this is especially important to define in this case because otherwise echo will think %i is a color and give us errors. Read /help /echo.

Variables

In this chapter we will be looking at variables in depth. It is recommended that you are comfortable enough with mIRC to be able to test your variables one way or the other- whether via a combination of scripts and command line, or purely through scripts. All the examples will be defined in aliases and output will be through echo or say, however naturally, variables can be part of control structures as values or even as a form of internal data storage (some scripts might need to store some information for internal use).

Variables in mIRC scripts can be split into two types, 'predefined variables' and 'user variables'. User variables can also be broken down into two groups too- 'local variables' and 'global variables'.

We have encountered various types of variables in the aliases section of this paper- predefined variables such as $nick and $me. All predefined variables have a $ prefix, and all user variables are prefixed with a %. Searching through the mIRC help file for items starting with a $ will show you a list of predefined variables. We'll discuss some of the more important ones, but be aware that there are plenty more for those that befriend /help.

We've seen $1 representing the first argument of an alias, for example:

/command variable1 variable2 variable3

When executed, this alias returns:

variable1 as $1
variable2 as $2
variable3 as $3
variable2 variable3 as $2-
variable1 variable2 variable3 as $1-

$0 returns the number of space-delimited tokens in the $1- line. So, in this example, $0 will return 3

To make things simpler, we will illustrate this concept with an alias:

```
/vardemo1 {
  echo 1 "$1" = $1
  echo 1 "$2" = $2
  echo 1 "$3" = $3
  echo 1 "$1-" = $1-
  echo 1 "$2-" = $2-
}
```

Try running it with a few variables (/vardemo1 these are variables) The output should look something like this:

"$1" = these
"$2" = are
"$3" = variables
"$1-" = these are variables
"$2-" = are variables

The double $$ means that this command will only be executed if a parameter is specified. If you specify only one parameter when more then one variable utilize double $$'s, the command will not be executed. You can also do $$?1 or $?1 which instructs mIRC to fill this value with parameter one if it exists. If parameter one doesn't exist, ask for it. In the first case the parameter is necessary for the command to be executed, in the second case it isn't.

Similarly, mIRC can be directly instructed to prompt the user for the variable by using *$?="prompt"*. Let's try an example directly off the command line instead of writing an alias. In order to do this, we must use two command characters (//command). This instructs mIRC to parse the line before executing it, thus processing all variables.

//echo $?="What is your name?"

If we had issued that command with one command character, mIRC would have literarily echoed *$?="What is your name?"*. However, we used the double //, so mIRC asked us what our name was, then proceeded to /echo the variable. $? Will return nothing if the user clicks cancel- it's often a good idea to halt the alias if this is the case. You should know how to do that using an *if statement*.

$1, $2, $3, etc also represent nicks when used within Nick List popups which we will discuss in more detail in the chapter entitled 'popups'.

$ip	-	Returns your IP address.
$hostname	-	Returns your hostname.
$time	-	Returns your system time.
$timestamp	-	Returns a short form timestamp.
$network	-	Reutrns the network you are on.
$server	-	Returns the server you are on.
$os	-	Returns the OS you are on (95, 98, etc).
$mircdir	-	Returns mIRC's root directory.
$mircexe	-	Returns full path and filename of the mIRC binary.
$titlebar	-	Returns titlebar text (set by /titlebar *).
$version	-	Returns mIRC version.
$fullname	-	Returns full name specified in the Connect dialog.
$chan	-	Returns current channel, $null if none. # does the same thing.

$address	-	Returns address of associated user in the form userid@host.domain.
$fulladdress	-	Returns the full address of the triggering in the form nick!userid@host.domain.
$crc(filename)	-	Returns the CRC of contents of 'filename'.
$exists(file/dir)	-	Returns $true / $false if a file or dir exists.
$file(filename)	-	Returns information about the specified file: Properties: *size, ctime, mtime, atime* $file(mirc.exe).size returns the file size $file(mirc.exe).ctime returns creation time $file(mirc.exe).mtime returns last modification time $file(mirc.exe).atime returns last access time

Listing variables is boring, so let's move on- however a lot of emphasis is put on the exploration of /help for more of these predefined variables.

The $read variable is an extremely useful scripting utility. It's slightly more complicated then the average variable, but on account for what it does- learning how to use it can be rewarding. $read allows a script to read particular lines from a file.

The full syntax of $read at the time of publication is:
$read(filename, [ntsw], [matchtext], [N])

//echo $read(funny.txt)
Reads a random line from the file funny.txt.

//echo $read(funny.txt, 24)
Reads line 24 from the file funny.txt.

//echo $read(info.txt, s, mirc)
Scans the file info.txt for a line beginning with the word mirc.

*//echo $read(help.txt, w, *help*)*
Scans the file help.txt for a line matching the wildcard text *help*.

If you specify the s or w switches, you can also specify the N value to specify the line you wish to start searching from in the file, eg.:

*//echo $read(versions.txt, w, *mirc*, 100)*

If the n switch is specified then the line read in will not be evaluated and will be treated as plain text. If the first line in the file is a number, it must represent the total number of lines in the file. If you specify N = 0, mIRC returns the value of the first line if it's a number. If the t switch is specified then mIRC will treat the first line in the file as plain text, even if it is a number.

Just for reference, most of that blurb on $read was taken directly from mIRC's /help. Reading about /write and /writeini is also a good idea if you intend to work with files.

Here are some manipulative predefined variables:

*$calc((3*2/12)*12)*	-	Performs a mathematical calculation.
$chr(N)	-	Returns the character with ASCII number N. (ie: $chr(65) returns A)
$count(string,sub)	-	Returns the number of times 'substring' occurs in 'string'.
$int(N)	-	Returns integer part of a floating point number with no rounding.
$left(text,N)	-	Returns the N left characters of text.
$right(text,N)	-	Returns the N right characters of text.
$len(text)	-	Returns the length of text.
$longip(address)	-	Converts an IP address to long value & vice-versa.
$lower(text)	-	Returns text in lowercase.
$upper(text)	-	Returns text in uppercase.

As you can probably imagine, variables such as these can be used to perform anything from complex mathematical equations to filtering out a particular part of a file when used in conjunction with $read or $readini.

Finally, the last predefined variable we will be documenting, $rand. This variable is used as a crude randomizer engine, but it must be used with care- specially within security applications, as it isn't as random as one may think.

$rand(v1,v2)
This works in two ways. If you supply it with numbers for v1 and v2, it returns a random number between v1 and v2. If you supply it with letters, it returns a random letter between letters v1 and v2.

$rand(a,z) returns a letter in the range a,b,c,...,z
$rand(A,Z) returns a letter in the range A,B,C,...,Z
$rand(0,N) returns a number in the range 0,1,2,...,N

A point to mention before concluding our segment on predefined variables is that mIRC uses some variables internally to manipulate the parsing of scripts. A short explanation and a very self-explanatory alias will be used to convey the function of these variables.

$& indicates a long line.
$+ indicates that there should be no spaces.

Let’s take a look at two example aliases:

```
/longline {
  echo This is an example of a long $&
  line that has been split into multiple lines $&
}

/nospaces { echo hi $+ how $+ are $+ you $+ ? }
```

Let’s talk about user-defined variables now.

The mIRC scripting area has its own tab for user-defined variables. However, the variables you see there (and can edit, set or unset simply by modifying the text) are only global user defined variables. Local variables cannot be edited directly, so we’ll leave them till last. Let’s conduct a few examples with global variables- we will do it all from command line, you should be feeling comfortable about doing this by now- while being aware of how these principals can be used within the context of a script.

```
//set %easy This is easy!
//echo %easy
```

The first command creates a variable called %easy and references it to “This is easy!”. The /set command can also be used to increment and decrement variables, read /help /set for more details. The second line obviously echoes the contents of the global variable %easy. A point to notice however, is that we did not unset this variable- this means that it must still exist- which is precisely correct. Search for the variable we just set in under the ‘variables’ section of the mIRC editor; you should find it at the bottom of the list:

```
%easy This is easy!
```

You can edit the name of the variable or its contents directly, but doing so may impact a script dramatically. Global variables are good places to save information about the script, constantly changing statistical information (such as a particular status) or just a temporary place to store a string. Variables can be used as tokens (particularly important in flood-protection related applications, as discussed in the chapter entitled ‘remotes’.)

Adhering to an intelligent naming convention with global variables is a great idea because mIRC allows the unset command to use wildcards. %common.unique is a good habit to get into when utilizing various global variables that are utilized by the same function- unsetting becomes as simple as %common.*. Are you reading /help /unset yet?

Local Variables

Local variables are very similar to global variables- in the sense that they are dynamically set and unset, and they can contain the same types of information. The only difference is that a local variable is temporary: it can only be accessed within the function that defined

it. For example if we use it in a particular alias, another alias would not be able to call the variable. Similarly, local variables are unset as soon as the function terminates (normally or abnormally).

Local variables also cannot be edited directly, as they often expire seemingly instantaneously. Editing memory is beyond the scope of this paper, so let's write up an alias to try and make some sense of this instead. Actually, let's look at an alias we used earlier:

```
/count {
  var %i = 1
  while (%i <= 10) {
    echo 2 %i
    inc %i
  }
}
```

Before consulting help, you should have realized that /var is the command to create and define local variables. We named the single variable in the previous alias '%i', this is an extremely typical name for a temporary variable- and this is exactly the reason local variables exist. If several instances of a particular alias were running at the same time using global variables, the script would almost certainly malfunction as several threads are trying to read and write variable data to the same location. Internal variables address this issue directly. Also, lazy programmers like myself can enjoy the fact that local variables need not be unset- actually, they cannot be unset until they are automatically purged at the end of the scripts 'run'.

/var –s will print the result when a value is set (verbose output).

Internal variables do not necessarily need to be initialized with information, so "var %local" is a legal argument. Also, var allows multiple variables to be initialized within one statement, the condition is that they are separated by commas:

```
/var %x = hello, %y, %z = $me
```

There is no documentation regarding how mIRC reacts to a variable call from within a script when both global and local variables exist by the same name. However, our personal experience up until publication was that a script will always check for a local variable before checking for a global one. This seems to always be the case, but we have no guarantees that this is how things will continue to be.

Remotes

In order to use remotes you should be comfortable with aliases, if statements, and all sorts of variables, as remote utilize them all. Remotes are responsible for causing a script to react to certain conditions- for example- a remote could be used to perform a command when anyone says a particular word, or when the client joins a certain channel- or maybe even any channel.

Remotes have a strange syntax that takes a little getting used to- but they are ultimately intelligent and versatile. Remotes have their own simple control structure, and are scripted the same way aliases are – within the mIRC editor, under the remotes tab. There are two distinct types of remotes- "Group Remotes" and "Global Remotes". Global remotes are always active, and group remotes can be turned on and off.

There are two types of global and group remotes: there are those that react to conditions met by the local user, and remotes that react to conditions by other irc users. Testing these two types of remotes varies, as in order to try out a remote you may need to load a clone or have a friend on IRC match the condition. Don't worry- this will be more apparent momentary, when we start trying some.

Remotes always start with "On *:" where * can be a wildcard or a userlevel (either a name or a number.) We will discuss userlevels in more detail later. Let's try a simple remote.

```
on *:TEXT:hello:#:{
  /msg # hello $nick
}
```

After the on *: instruction, we define the condition. In this case we specified "TEXT', thus mIRC will wait for a particular string to be said by anyone apart from the local user (you). In order to test this remote, you will need a friend or a clone to say "hello" in any channel you are on. After TEXT: we specified the string to look for- wildcards are allowed, and mIRC is sensitive to that- as we are about to find out. Let's see what happens when a user says hello.

<remote> hello
<local> hello remote

It worked! The 'hello' text condition was met, and thus mIRC performed the instructions encapsulated within brackets. However, if 'remote' had said 'hello local' the script wouldn't have worked. This is because of mIRC's remote wildcard sensitivity. If we had used "on *:TEXT:hello*:#:{}" then the script will react to any string that begins with hello. If we had used "*hello*", then the script would have reacted to any string with the word hello in it.

Let's look at a few more remotes before we differentiate between global and group remotes. As aforementioned, remotes always begin with 'On', so searching /help for 'on' will show a list of supported remotes. There are many, going through them is mandatory- we will not discuss them all in this paper- and syntaxes may vary from one remote to the next.

on <level>:TEXT:<matchtext>:<><?><#[,#]>:<commands>*

This is the syntax for the TEXT remote- instead of using # which depicts that the remote should work on all channels, we could have used "*?-*" meaning the remote should only work if the conditions are matched within a private message. Alternatively, we could have specified a channel or several channels by name (separated by commas) or even *- implying that the remote should check everything for the string.

The NOTICE and ACTION remotes follow the same syntax as TEXT… this implies that on TEXT will not check actions and notices- this may seem like a drag, but accuracy can be critical, and writing if statements to ensure the accuracy of a condition can be far more tedious.

Some remotes will incur their own variables, let's look at the on op/deop syntax:

on <level>:OP:<#[,#]>:<commands>

$opnick refers to the nickname of the person being opped/deopped,
$vnick the person being voiced/devoiced,

Let's use this information within a remote:

*on *:OP:#:/echo $nick on $chan deopped $opnick $+* .

We added a $+ . purely for entertainment purposes. This will add a period at the end of the line, you should know how to use $+ by now. To test this remote, you will need a friend or clone to op someone on a channel which you are on. Your script should echo something like this when an op is performed:

Remote on #channel deopped local.

Similarly, we can use on BAN to check for conditions within bans. Utilizing the on BAN remote can be made more efficient using the $banmask variable, which depicts ban mask. Can you guess what this script that follows does?

*on *:BAN:#: if ($banmask iswm $address($me,5)) mode # -bo $banmask $iif($nick != $me,$nick)*

Another cryptic looking line- but its nothing five minutes with help cant cure. We will now briefly list some more ‘remote’ global variables.

On part	Triggers when a user parts a channel.
On join	Triggers when a user joins a channel.
On mode	triggers when a user changes a channel mode.
On ping	Triggers when a server sends you a PING, not to be confused with CTCP PING (discussed later.)
On topic	Triggers when the topic within a channel is changed.
On voice	Triggers when a user sets +/-v.
On wallops	Triggers when you receive a wallops message.

There are other remotes that react to local variables, such as on connect- which performs commands as soon as your client connects to an irc server. On start works when the client is started, on notify when someone on your notify list joins or leaves irc. Remotes that utilize local variables work in exactly the same way, but may have their own variables. Remotes can also include sockets, but this is a topic which is beyond the scope of this paper (type /help sockets for more information).

Similar to remotes are ctcp (client-to-client protocol) events. They are also written in the remotes section of the mIRC editor, but they react a little differently. Instead of reacting to specific events, ctcp events react to user-defined ctcp requests. You should already be familiar with some CTCP events, such as PING, FINGER, or VERSION.

In order to PING someone on IRC, you must issue the command /ctcp nick PING. However, mIRC lets you get away simply with /ping nick. In the past, this used to be defined as an alias which the user could edit- however now this alias is built into the binary (we will do some binary hacking towards the end of this paper). Now, the CTCP syntax:

ctcp <level>:<matchtext>:<|#|?>:<commands>*

Let’s try a simple CTCP:

*ctcp *:SCRIPT:*:/msg $nick I’m learning how to script in mIRC!*

In order to test this CTCP, have a friend or clone issue a /ctcp nick script, replacing nick with your nick- they should automatically receive “I’m learning how to script in mIRC!” as private message.

CTCP's are often used as 'triggers'- let's try and write a 'pager' script.

```
Ctcp *:PAGE:*:{
  echo -a $nick $+ paged you at $time $date with message: $2- $+.
  msg $nick $me has been paged.
  splay alarm3.wav
}
```

This remote will play the sound file called 'alarm3.wav' from mIRC's sounds folder when someone sends a CTCP PAGE. The script will echo the nick of the person that paged you along with the date and time it was received. Furthermore, this script checks for $2-, which it defines as a page message. Perform the following:

/ctcp nick page hey wake up

Your client should play a sound and return something like this:

nick paged you at 18:12:42 20/08/2002 with message: hey wake up

Furthermore, our script is instructed to msg $nick informing them that the page has been sent (msg $nick $me has been paged.)

FYI: Ctcp's, just like text-based remotes can have contain a variable instead of a string.

Now we will look at how to change a global remote into a group based one which we can turn on and off at will. We will continue using the pager script, but instead of using a CTCP we'll use an ON TEXT remote, this way instead of sending a CTCP- people can page the script by typing &page <message> on a channel.

```
#pager on
on *:TEXT:&page *:*:{
  echo -a $nick $+ paged you at $time $date with message: $2- $+.
  msg $nick $me has been paged.
  splay alarm3.wav
}
#pager end
```

#pager on and #pager end make the remote group based. This remote is now named 'pager'. If we begin the remote with #pager on then the remote is active- if we used #pager off, then this remote wont work until it is enabled.

Remote groups can be enabled and disabled using the /enable and /disable commands respectively. When enabling or disabling remote groups, the # is not required. An alias, popup or other remote can trigger a remote group to be enabled or disabled- the utility is endless.

Popups

Let's start wrapping things up by learning about popups. Popups make up the graphical user interface of your mIRC client. When you right click on a channel window, or on a nickname in a channel list you get a list of commands defined in popups. The mIRC client comes with many predefined popups that can be changed.

Before trying to write our own (easy!!!) let's try and improve some of mIRC's pre-defined popups. Open up the mIRC editor, under popups, select nick list. The popup pull-down menu will emulate what the popup will look like- selecting a command from there will scroll you to the commands instructions. Click on 'Control -> Op':

.Op:/mode # +ooo $$1 $2 $3

The variables $$1, $$2, and $$3 represent highlighted nicks on the nick list. Thus, highlighting three nicks and clicking on the Op popup while opped on a channel will set +o for all three nicks. What happens when you highlight four? Only the first three will be opped- we know that most IRC servers allow up to six modes per line, so let's improve our mIRC client:

.Op:/mode # +oooooo $$1 $2 $3 $4 $5 $6

Now we can op six people in one line. This is much faster then issuing a command twice. Naturally, we could have made it op 12 people by using multiple lines:

```
.Op:{
  mode # +oooooo $$1 $2 $3 $4 $5 $6
  mode # +oooooo $7 $8 $9 $10 $11 $12
}
```

Alternatively, op could have just called an alias in which we would have defined the mode instructions.

With a little practise, it will be clear that a simpler way to have modified the original popup to op six people would have been to use $3- and issue six +oooooo's- but this could cause problems on some servers, as your script will request six mode changes for less then six people (assuming only five were selected.)

The syntax for writing popups is similar to the syntax for writing aliases:

(1)Name:{commands}

The strange thing about popups is their prefix- which isn't actually that strange. The number of periods before the name of the popup determines the level at which it is embedded. Take a look at the beginning of the 'Control' segment of the predefined nick list popup:

-
Control
.Ignore:/ignore $$1 1
.Unignore:/ignore -r $$1 1

The first line, -, instructs mIRC to draw a horizontal division rule on the popup. The next line has no command, thus it is considered a heading. Headings that contain no subcommands are grayed out in modern mIRC binaries- whereas in older ones they were ignored altogether.

Control is the heading, and ignore and unignore amongst others are subcommands of Control. If we wanted to have another heading within Control, it would look like this:

Control
.Ignore:/ignore $$1 1
.Unignore:/ignore -r $$1 1
.Another Heading
..Commands:/coolAlias

Experiment a little with popups structure, it shouldn't take more then thirty seconds to grasp this 'embedding' concept.

At the time of writing there were five areas that popups exist: Status, Channel, Chat, Nick List, and Menu Bar. As you can imagine, you cannot use a variable like $1 and expect mIRC to fill it in with a nick within the menu bar or status window.

A general rule of thumb is to instruct popups to call aliases as often as possible- that way all functions can modified from one central location instead of having to modify popups left right and center. Be creative- a popup can do anything from changing the text in the titlebar to enabling a remote.

Keep your interface as simple as possible- testing frequently. Forgetting a dot could cause the popup to appear deformed, and cause popups to be hidden from the user.

In conclusion, let's tie up all these scripting concepts by putting them all into one flat file which we can load up at will, or paste directly into the mIRC editor as a new script. In order to illustrate this, we'll continue our pager script by adding a few more features and compiling it all into one file that can be loaded directly:

```
;
; Flur's modular pager script with flood protection.
;
menu channel {
  Pager
  .On:{
    enable #pager
    echo Pager is now ON
  }
  .Off:{
    disable #pager
    echo Pager is now OFF
  }
}

#pager on
on *:TEXT:&page *:*:{
  if ( %fprot = on ) { halt }
  PAGE
  set %fprot on
 timer5 1 5 unset %fprot
}
#pager end

alias PAGE {
  echo -a $nick $+ paged you at $time $date with message: $2- $+.
  msg $nick $me has been paged.
  splay alarm3.wav
}
```

Finally a brief description of the various new concepts introduced in this example. The alias name {} defines an alias- so instead of typing out all your aliases in the aliases section of mIRC's editor, you can defined them directly in a script file which can be loaded, or as under remotes. Aliases and popups can be defined with the alias or menu prefix. After the prefix, the name of the alias or the popup class must be defined followed by the contents within a set of {}s.

Another new concept introduced in this script is comments. A semicolon (;) implies the text that follows is to be ignored by mIRC. Use them to document your code- a good habit to get used to.

The timer command is also worthy of mention:

/timer[N/name] [-ceomh] [time] <repetitions> <interval> <command>

The timer command is used to perform commands after a delay or continuously based on the commands syntax. In the example above, a global variable called *%fprot* is set with the contents "on" after a page has been successfully received. This variable does not get unset until the timer entitled timer5 expires (*timer5 1 5 unset %fprot*). It takes the timer 5 seconds before it performs its payload- to unset the global variable *%fprot*. This timer will run only once. What is the point? Well- lets look back at the remote, right before the alias 'PAGE' is called:

if (%fprot = on) { halt }

If the global variable is set to on, the script will halt without continuing- thus the page will be ignored. This means that your script will accept a maximum of one page ever five seconds. Being security conscious is critical, as IRC tends to attract lots of malicious activity. The purpose of this particular flood protection is to keep people from being able to instruct several bots to send many pages expecting your script to reply to them all at the same time, this would often cause your script to get disconnected by the IRC server for flooding (trying to send too much text at once.) Based on this concept, it becomes apparent that replacing "halt" with *"/msg $nick Sorry, try again later"* is extremely redundant.

Before concluding with some final advanced notes, let's discuss how to load a script. A a new script can be created via the file menu in the mIRC editor, and write an entire script on one remotes page then saving the script. This file can now be distributed to your friends- and can be loaded by using the command:

/load -rs filename.ext

The load command supports many switches, so reading help (it cannot be stressed enough) is a good idea.

Binary Hacking

This chapter assumes the reader is comfortable with moderately advanced memory manipulation. This is considered a basic form of reverse engineering, not scripting, readers are advised to skip this segment if they are not interested in modifying the mIRC executable. Reverse engineering is considered illegal in many areas, and messing with a binary file can have extremely disastrous side effects- do not perform anything contained in this chapter without being comfortable with using a HEX editor to examine memory, furthermore, don't ever forget to create a backup first.

We mentioned earlier that mIRC has built in scripts that the user is not meant to play with. Such scripts include the VERSION and PING CTCP. By examining the mIRC executable, we can find these scripts and modify them to suit our purposes. The creator of the mIRC IRC client even expected people to do this- he left a comment for those that try. Let's look for the VERSION CTCP within the binary (discussed in far more detail in a paper also written by flur entitled 'mIRC Version Spoofing').

```
0000 4143 5449 4F4E 2000 0056 4552 5349
4F4E 0000 4564 6974 696E 6720 6F75 7420
7468 6520 7665 7273 696F 6E20 7265 706C
792C 2068 7568 3F20 3A29 0000 4E4F 5449
4345 2025 7320 3A01 5645 5253 494F 4E20
6D49 5243 2025 7320 4B68 616C 6564 204D
6172 6461 6D2D 4265 7901 0A00 0056 4552
5349 4F4E ... ... ...
```

..ACTION ..VERSION..Editing out the version reply, huh? :)..NOTICE %s :.VERSION mIRC %s Khaled Mardam-Bey....VERSION

From within the mIRC binary we can instruct mIRC not to respond to a CTCP version reply, or a CTCP PING reply and instead embed them into scripts that can be turned on and off. There are many ways to do this including inserting a modified script into the binary, or a more robust solution, to rename the variables within the binary to variables that will never be called (consider renaming the internal CTCP PING to something like FOOO- who will ever perform a */ctcp nick FOOO* without knowing the ctcp exists?) Once a CTCP is renamed, we can re-write it within mIRC's script editor thus causing our client to reply the way we want it to as opposed to the way it was meant to react.

If there is enough interest, I may be convinced to write a separate paper focusing entirely on mIRC binary hacking, as this is all that we will discuss on this relatively complicated topic.

Conclusion

By now readers should be able to write scripts to perform any desired function. Furthermore, you should be very familiar with the comprehensive help included with mIRC. Congratulations. We will discuss a little bit of scripting history and close with a funny comment involving cookies hidden within mIRC (which can be edited by binary hacking hehehe…)

If you have managed to successfully write your own script file, and tried loading it you would have noticed that you get a very clear warning asking you to verify that the script should be loaded. This is due to a serious flaw in the past with older versions of mIRC that would automatically load any script named script.ini within $mIRCdir.

Malicious scripts that spread by convincing people to run a particular command, or run a particular file to create a script.ini file that does the same thing- these went around like wildfire. The situation has supposedly been remedied, but patches were never created – therefore, finding old clients means finding vulnerable clients. This gave the script.ini file an evil connotation, but in reality, script.ini is just a name that means the same as filename.ext. Don't be afraid to run script.ini files that you wrote yourself (duh!)

In final conclusion, I'd like to take a quote from /help cookies!

The Cookies
A squeaky sound, a multitude of silly comments, a faithful pet, one bouncing dot, and a smiley face. Where oh where can they be? :)