Macros

For anyone learning Lua to get the most out of WoW, it’s a safe bet you already have a few macros in your action bars. They can be incredibly useful when you want to perform complex tasks like casting different spells on different targets based on conditions, with just one button. Such conditions can also be quite complex; for example, you can detect whether your current target is hostile or friendly, alive or dead.

If you currently do have any macros in your action bars, did you create them yourself or did you copy and paste them from somewhere? If you created them yourself, did you know exactly what you were doing? Many players think that macros just consist of a simple list of simple slash commands, but they are extremely powerful. Certain slash commands can contain powerful conditions and have a variety of options that can be used.

A macro is always executed line by line, which means you have to insert a new line after every slash command you want to use. For example, a macro that sends two messages to the say chat looks like this:

/say Hello...

/say ...World!

You can add every slash command that is available in the game to a macro, from gameprovided slash commands like /say <text>\_:, through small Lua scripts via /script<Lua code>:, to complex Lua functions in slash commands created by addons. The only restriction is that the number of characters per macro is limited to 255. But we will find ways to circumvent this restriction here.

This chapter is about creating macros, so we will explore how macros work and how we can create extremely powerful macros. We will start with an overview of the available commands for macros.

Available Command

You probably know the basic slash commands, like，/cast <spell>:,/use <item>(which are internally actually the same command), or /castsequence <spell1>, <spell2>, .... But there are many more commands; for example, did you know about the command /petautocasttoggle <pet spell>?

Secure and Normal Slash Commands

Most of the important slash commands we are going to use for macros are described as secure slash commands because they often simply execute a single protected function. All of these slash commands are stored in a table that is accessible only from the file FrameXML\ChatFrame.lua; the name of the local variable holding the table is SecureCmdList. Recall that normal slash commands like the ones created by an addon are stored in the globally accessible table SlashCmdList.

These normal commands are quite boring from our point of view, as everything they can do can also be done by a Lua script. You also already know all the important normal slash commands that are provided by the game, as you use them every day. For example, /raid <text> sends a message to the raid, and that’s something you type in your chat frame all the time. And there is the command /random for loot distribution. Other commonly used normal slash commands are emotes; for example, /facepalm <some guy who just wiped the arid>can be incredibly useful sometime.

However, this chapter does not cover these provided normal commands. You already know the most important ones, as you are using them every day. All other less commonly used normal slash commands deal with managing the guild or the raid, and these functions are often also available in the GUI. Also, these slash commands are often very simple; they often accept just one argument like a text or a target. We want to write powerful macros you can use in boss fights or in the arena.

e Blizzard has a list of all available normal slash commands on its web page if you need additional information about a game-provided normal slash command: <http://www.worldofwarcraft.com/info/basics/slashcommands.html>.

Let’s now focus on secure commands, which are more complicated than they seem at first glance. Adding complexity are the conditions and options you can add to a slash command. Conditions are checked before the command is executed; the slash command is ignored if the conditions are not met. You will see more about conditions later.

We will first explore which slash commands are available and the options they provide. There is one option that is somewhat special as it is available for all slash commands and is internally actually a condition—the target option.

Let’s use a slash command you all know as an example: /cast <spell>. This seems to be a simple command; it casts a spell on your current target. But you can also use /cast [target=focus] <spell> to cast a spell on your focus target. The value of this option is focus in the example, but it can be any valid unit ID or the name of a nearby player or NPC.

target is an option that is available for all secure commands. The option obviously doesn’t make sense for all commands but it is syntactically valid in every secure command; it just doesn’t have an effect in every command.

Only secure commands support the target option; normal slash commands do not. But many normal slash commands, like emotes, take a target for the action as an argument.

This option leads to our first useful macro, casting a spell on our target’s target. This can, for example, be used by healers to target a mob like a boss and automatically heal its current target. Just insert a healing spell of your class to use this macro.

Another useful macro for hunters or warlocks is targeting a spell on the target of the pet. This can be done with the following slash command.

List of Secure Commands

Table 12-1 lists all available secure commands that deal with casting spells or using and equipping items. Some more advanced slash commands, like /castsequence, are explained in detail later.

All secure slash commands are written in Lua. You can find them in the file FrameXML\ChatFrame.lua(search for SecureCmdList there). Reading the code behind a slash command can help you understand it.

Your character starts attacking <target> or your current target if it is omitted. It is also possible to use the target option instead.

Randomly selects one of the spells or items from the comma-separated list and casts or uses it.

Casts <spell1> when it is executed for the first time, <spell2> when it’s clicked for the second time, and so on. Its arguments may also be items. This command is even more powerful with its additional reset option, which is explained in detail later in this chapter.

Equips the item <itemName> from your inventory. Alias: /eq

Equips <item>: in the slot with the ID<slot>. Slot IDs are explained later

These few commands that mainly deal with spells and items are the most important ones for macros. Note that you can usually cast just one spell per macro because of the global cooldown. However, there are a few spells that do not activate the global cooldown, like the Shaman and Druid spell Nature’s Swiftness. The following macro casts an instant Healing Wave if Nature’s Swiftness is ready. It casts a normal Healing Wave if you have cooldown on Nature’s Swiftness.

/cast Nature’s Swiftness

/cast Healing Wave

You can also use this macro with a Druid; just replace Healing Wave with a Druid healing spell like Healing Touch. Examples of other spells that also do not activate the global cooldown are Inner Focus (Priest) and Presence of Mind (Mage). You can build similar macros with these two spells.

But there are more slash commands that do more than cast spells. Table 12-2 lists all the secure commands that deal with your target and focus target; these commands can also come in handy.

Targets <name>. This also targets nearby players or NPCs starting with <name>； you can use /targetexact <name> to avoid this behavior. It is also possible to use the target option instead of the argument. Alias: /tar

Targets the nearest unit with the exact name <name>. You can use the target option instead of the argument.

Assists <target>, it is also possible to use the target option instead of the argument. Alias: /a <target>

Table 12-3 lists slash commands that are especially important for hunters and warlocks, as they deal with commanding your pet.

Causes your pet to start attacking <target>. It is also possible to use the target option instead.

Enables auto casting of your pet’s spell <spell>

Table 12-4 lists all slash commands that deal with raid roles, like main tank or main assist assignments. You might wonder why the following functions are protected, as they don’t seem to deal with functionality that could automate the game. The reason is that if they were not protected, you could write an addon that always sets the player with the lowest health in your raid as main tank. A unit frame addon could then use a secure group header that shows all of your main tanks. This would always display the player with the lowest health, and the purpose of the secure/tainted code system is to prevent such addons.

Sets target as the new main tank. It is also possible to use the target option instead. Alias: /mt

These were the four main categories of secure slash commands. But there are a few more miscellaneous commands available. Table 12-5 lists all remaining secure commands.

Requests a duel with <target>. It is not possible to use the target option here.

Executes the OnClick handler of the secure frame stored in the global variable <Secure frame> with <mouseButton>. This is an extremely powerful command as it allows us to click on arbitrary secure button templates; you will see how we can use this later.

Two commands in the tables in this section were not fully explained there: /castsequence and /equipslot. Let’s take a closer look at each.

There is an additional option for the cast sequence command: the reset option, which defines criteria that determine when to reset to the first spell in the cast sequence. This option is not required; by default, a cast sequence restarts when it reaches the last spell in the list. You might expect that the reset option, like the target option, is added in square brackets to the beginning of the command’s arguments, but that is not the case. The reset option must be placed between the optional conditions (that is, the target option) and the list of spells. The following example illustrates this.

/castsequence [target=focus] reset=30 <spell1>,<spell2>,...

This starts a 30-second timer every time the macro is executed and resets to the first spell when the timer expires. This means that it resets if you don’t use the cast sequence for 30 seconds. But reset accepts more values than just numbers; it can also be a string that defines a trigger that resets the action. The following triggers are available.

target resets the sequence when you change your target.

combat triggers when you leave combat.

The keyboard modifiers ctrl, alt, and shift reset the sequence when you hold the key while executing the macro.

The code behind the slash command handler simply uses string.find on this option and resets the sequence if it finds a matching trigger, so you can just concatenate multiple triggers to reset it when one of them occurs. For example, reset=ctrltarget30, resets it when you Ctrl-click the macro, when you change your target, or when you haven’t used the macro for 30 seconds.

The secure command /equipslot <slot> <item> takes an item slot ID the item will be equipped in. The commands /use <item> and /cast <item> also accept such an ID instead of the name of an item.

There are two types of inventory IDs: strings (such as HeadSlot for your head) and numbers (such as 1 for your head). Macros can only use the numeric identifiers, while addons can (and should) also use the string identifiers in API functions like this.

id, texture = GetInventorySlotInfo(slotName)

The numbers are important for us here because we cannot use the strings in macro commands. We can now write simple, yet powerful macros that make use of items. Imagine you have two of these fancy “activate to do x extra damage for y seconds” trinkets equipped. They usually share a short cooldown, so you can’t activate both of them at the same time. It can be useful to have a macro that activates the trinket that is currently not on cooldown.

This is easy with the /use（or /cast） command; we can just try to activate both trinkets in the same macro.

If one of them is on cooldown it will be skipped with the usual “this item is not ready yet” message. Later you will see how to suppress this error message.

The macros we’ve tried thus far are quite simple, and you probably already knew how to use most of the commands. So let’s create a few more advanced macros with conditions.

Conditions are added in square brackets to a macro, just like the target option (which is syntactically actually a condition). A good example of a simple condition is exists, which evaluates to true if the command’s target exists or to false if the target doesn’t exist. The following slash command casts <spell> on your target if you have a target and does nothing otherwise.

You can also use the target option to modify the command’s target. The exists condition then also refers to the unit that is referenced by the target option. The following command casts <spell> on your focus target if you have one.

Multiple conditions are always separated with commas in a single command. We could now add other conditions to this macro, but let’s get an overview of the existing conditions first.

Use the red question mark as icon for your macros, as it automatically chooses an appropriate icon based on the conditions.

Table 12-6 lists all available conditions with a short description. Some more complicated conditions, like equipped, are explained in detail later in this section.

true if the macro was clicked with mouse button x, works just like the button suffix for attributes of secure templates. x can also be any string that can be passed as a button to the /click command.

true if you are currently channeling <spell>. The spell is optional, just using channeling without a spell evaluates to true if you are currently channeling any spell

true if you have an item of the type <itemType> equipped. The argument is slightly more complicated and explained in detail later. Alias: <itemType>

true if you are in Outlands or Northrend and it is possible to mount. Caution: this condition evaluates to true in zones like Wintergrasp or Dalaran (a bug that renders this condition almost completely useless)

Valid group types are party and raid, evaluates to true if you are in such a group

true if your currently active pet is <name>. The argument is optional, it is also possible to just use pet to check for any pet.

true if you are currently in the stance or shapeshift form with the ID x. Stance IDs are explained in detail later. The ID is optional, you can use just stance to check if any stance or shapeshift form is active. Alias: form:x

These conditions can be used with all secure commands except the duel commands, /duel and /forfeit

One of the more complicated conditions from this list was equipped, which takes the type of an item instead of its name. Every item has a type and a subtype, which can be used, for example, to categorize an item within an auction house. The identifier that can be used in this macro condition is the localized string of the item type as it is shown in the auction house. For example, you can use the condition amqella equipped:Cloth to check whether you are currently wearing a cloth item. It doesn’t make much sense to use the armor type here; the interesting option is using a weapon type. For example, rogues can use the following macro to combine Backstab (which requires a dagger) and Sinister Strike in a single macro depending on the currently equipped weapon.

This casts Backstab and cancels the macro if you are wielding a dagger; otherwise, it casts Sinister Strike. You will see a way to write such a macro with a single cast command in the next section

Item categories are always plural, so using [equipped:Dagger] does not work

Two other important categories are One-Hand and Two-Hand they evaluate to true if you are wielding any one-handed or two-handed weapon.

The other condition that is not fully explained in the table is stance:x. Stances are the different battle stances of warriors, as well as druid shapeshift forms, the shadowform and Spirit of Redemption of priests, rogue stealth, the shaman’s ghost wolf, and the warlock’s demon form. The stance ID 1 refers to the first stance or shapeshift that is visible in your shapeshift/ stance bar, 2 to the second, and so on. For example stance:1 refers to the warrior’s battle stance or the druid’s bear form

The stance ID can refer to different forms/stances depending on the talent specialization (spec) of a druid or priest. For druids, stance:5 refers to the Moonkin form if they have the Balance spec and to Tree of Life if they have the Restoration spec. Priests are slightly more complicated; stance：1 refers to Shadowform if it is available and to Spirit of Redemption if this is available but Shadowform is not. Spirit of Redemption is stance:2. if both talents are specced

Auras of paladins and aspects of hunters are not stances, even though they show up on the shapeshift/stance bar. This means you cannot use the stance condition to check which aura or aspect is currently active.

We’ve now discussed all the conditions that can be used. Let’s combine them to build complex expressions.

The simplest way of checking two conditions doesn’t actually combine two different conditions. All conditions in the form condition:argument accept multiple arguments separated with slashes which will be combined with a logical OR. The following command casts <spell> if you have Shift or Alt pressed

But we can also combine two different conditions in a single command. The simplest combination is the conjunction (combining the conditions with the logical AND).

You’ve already seen an example of this in the last section, as the target option is syntactically a condition.

We can just add another condition separated with a comma to check both of them. The following example casts Resurrection on your target if it exists and is dead

It is also useful to have conditions that check whether your target exists and is not dead. We need the logical NOT to do this

The negation of a condition can also be done easily. Just add the prefix no to a condition to negate it (apply the logical NOT). Thus, nodead means your target is alive. The following macro casts <spell> on your target if it exists and is alive:

The logical NOT is restricted to a single condition; you cannot add brackets in a macro condition. Let’s try to negate the condition from the last example. Imagine you were writing this negated expression as a Lua expression; it would look like this.

In a macro it is not possible to use a condition like [no(exists,nodead)]. We need to transform this expression into an equivalent expression that can be represented in a macro. We can do this in a simple case like this just by analyzing the meaning of the condition. It means that our target either does not exist or is dead. This can be written as (not exists) or dead in Lua syntax. You will see how we can use a logical OR in a macro in just a bit

The transformation between these two forms is not always that obvious, especially if you have more complex expressions. But there is a simple law in boolean algebra that can be used to do this transformation. De Morgan’s Law looks like this

If we apply this to our expression not (exists and not dead) we get (not exists) or (not not dead). Obviously, not not dead means the same as dead, so we get (not exists) or dead.

This law also applies to terms with or instead of and

But we still can’t use the expression in a macro, as we haven’t discussed the logical OR yet; let’s see how this works

To combine two conditions with a logical OR, add another block of expressions in square brackets. The example from the last section then looks like this

This casts <spell> if your target doesn’t exist or is dead. The OR operator is also somewhat restricted. Imagine you have an expression in the form (A or B) and (C or D). You might think it is possible to write the macro condition as [A][B]，[C][D], but that is not possible. You also have to transform your expressions into the disjunctive normal form, breaking the expression down into multiple subexpressions, which can be as simple as normal conditions and are combined with logical OR.

There is also a useful law for transforming such expressions—the distributive law

Let’s try to apply this law to the expression （A or B）and（C or D） to transform it into a form that is suitable for a macro

This form of the expression has the same meaning as the original form and can be used as a macro condition that looks like this

That was a highly abstract process. Fortunately, it is rarely needed, as you rarely have such complex expressions in a macro. But it’s still good to know how you can transform logical expressions. You can also apply these laws to expressions in Lua to shorten or clarify them. However, let’s get to something more practical now: useful macros that incorporate a logical or in their expression

The first block always evaluates to true as it doesn’t contain a condition, so the second block is never executed and its target option has no effect. But we can also add conditions to the block to set the target option depending on these conditions. We can use this technique to extend one of our first macros in this chapter, the one that casts a healing spell on your target’s target

You can target a mob and heal the player that is currently tanking it with this macro, but it is useless when we have a friendly player targeted. The following macro checks whether our current target is friendly and casts a heal on it if so; otherwise it casts the heal on the target’s target

Note that the first target option is actually not required, as target is set to target by default

We can extend the macro even further by first checking for the unit ID mouseover. This unit ID works with secure unit templates that allow you to heal players by hovering your raid frames with your mouse. The next check can then be the target followed by the target’s target. It may happen that help fails on all of these unit IDs, but we can still cast the spell on ourselves in this case. The following is the ultimate macro for lazy healers; it performs all of the checks mentioned earlier and selects a target that can be healed:

Note that target=target is required in the second block, as the first block changed the target setting to mouseover. Just using [help] as the second would be pointless, as it would check whether the mouseover target is friendly, which is not the case. We know that because if it were friendly, the second block would never be reached

It is not necessary to add checks to see if the targets are alive here, because the condition help does a little more than just check whether the target is friendly; it checks whether the target is a valid target for a beneficial spell, and dead targets are not valid targets for such spells (except for resurrection spells which cannot be used together with a help check)

We can now build complex expressions with the conditions, but the form of the whole command is still quite simple: an if <expression> then do something end construct. Having an elseif or else block would be a nice addition

The only control structure that is available in a macro is an if-else-elseif block. There is no such thing as a loop in a macro

An else or elseif block is started with a semicolon in a macro. This is then followed by a condition and the macro arguments to use. I previously presented a macro that casts Backstab if you have a dagger equipped and Sinister Strike otherwise

This macro uses the stopmacro command, and as I said, we are going to try another version of it that consists of just one line. We just need to add a semicolon followed by the contents of the else-block, which is here simply using Sinister Strike as the argument cast

An elseif-block uses an additional condition after the semicolon, like the following macro, which casts a damage spell if you have a hostile target and a healing spell if your target is friendly

Another interesting use of this control structure is using the btn or mod condition to execute different actions based on the mouse button or keyboard modifier used. The following macro can be used by rogues to apply a poison to both weapons by clicking it first with the left and then with the right mouse button

The first command uses the poison, which then expects you to choose a target. The second command then uses the pending spell on your main hand if you left-click the macro and on the off hand if you right-click the macro

Another really powerful macro is one for warriors that tries to interrupt a spell. Warriors can interrupt spells in Battle and Defensive Stance with Shield Bash (which requires a shield). Pummel can be used in Berserker Stance; this spell doesn’t need a specific weapon type. The following macro casts Pummel if you are in Berserker Stance; otherwise it checks if you have a shield equipped and then casts Shield Bash. If this check also fails, meaning that you are in Battle or Defense Stance and do not have a shield equipped, it casts Berserker Stance so you can just execute the macro a second time.

Note that it is not possible to cast Berserker Stance and Pummel with just one click, as Berserker Stance triggers your global cooldown.

One thing you might have noticed in this and most of the earlier macros is that they automatically show the correct spell if you chose the red question mark as macro icon. But we can also control which spell and tooltip are displayed

A macro always uses the icon and cooldown of the spell it casts when you execute it under the current circumstances. But this is not always sufficient, and it is possible to control this behavior by using the commands #show and #showtooltip at the beginning of the macro. These two commands also require the question mark icon for your macro

Both commands are quite simple. They take a single argument, which is then shown in your action bar for this macro. The following macro shows up as Attack in your action bar and behaves just as if you had the normal Attack and not a macro in the action bar. But it casts Flash Heal when you click on it

The command ­#showtooltip follows exactly the same syntax; the only difference is that you also get a tooltip for the spell when you place your mouse over the macro. It also sets the icon. The ­#show command just sets the icon, and the tooltip shows the name of the macro.

It is also possible to use the conditions known from secure commands here, and semicolons are used for simple if-then-else constructs. The following macro creates a button that shows Backstab if you have a dagger and Sinister Strike otherwise:

But the macro that used the same conditions earlier already had this functionality (except for the tooltip, which just showed the macro’s name). The icon of the macro was already correct, and you rarely need a tooltip for an action button. So you might wonder what the real use for these commands is. The answer is that you cannot always use simple slash commands with conditions in your macro. For example, you need this functionality when the actual work is done by Lua code, as /script commands are not parsed to determine the icon to display. You will see an example of this when we create a mount macro at the end of this chapter.

You will also need to use #showtooltip if you have a stopmacro command in your macro and you want to show a different icon if the command’s condition is met, as the game does not check the stop conditions when determining the icon/tooltip to display. However, you will rarely need to display a different icon based on a stopmacro condition, as you can usually avoid using stopmacro altogether by using control structures

Another possible use for ­#showtooltip is when your macro calls other macros or executes the click actions of secure buttons. For example, let’s assume you have two macros in your first action two action bar slots, ActionButton1 and ActionButton2. You now want to create a macro that executes macro 1 if you have a hostile target and macro 2 otherwise

The button would show no icon at all if we didn’t use ­#showtooltip here, as WoW does not parse the buttons referenced by /click when determining the icon to display. But why would we want to write such a macro that simply calls another macro? Because both of the other macros can also be up to 255 characters long, which means we have just successfully circumvented the 255-character limit of a macro. But there are other ways to do this without having to split up your macro into many small parts. Let’s see how we can use this combination with /click and secure action button templates to create really long macros

Circumventing the 255-Character Limi

You rarely need more than 255 characters in a macro, but one case where you do is when you want to do the “kill all rare mobs” achievement. It can be really useful to have a macro that tries to target all rare spawns and displays a message when it manages to target one. You can then fly around in the spawn areas of the rare mobs and click the macro all the time

We need to create a secure action button with Lua code and set its type to macro and its attribute macrotext to something like the following

This would obviously not fit in a normal macro, so let’s build a secure action button for it. The best place to create such a button to use in a macro is in a separate addon. I have a small addon called RandomCrap, which just consists of an XML and a Lua file that hold some functions and templates I use regularly. But you can also enter the following code in an addon like TinyPad if you just want to test it

The simplest way to create such a macro is to write a string with the 23 /targetexact commands. Note that delimiting the string with square brackets allows you to use new lines in the string, so you can just write your long macro literally in your code. But it is smarter here to create a table with all the names of all the bosses and automatically generate the macro text from this string. It is shorter and more flexible, for example; you can later easily change the target command from /targetexact to /target if you want to allow partial matches:

The code first generates the /targetexact part of the macro text and then creates a secure action button and sets its type to macro. The attribute macrotext is then set to the string generated earlier, followed by the stop condition and the code that displays a message.

This technique works with every macro that requires more than 255 characters, but it also has a small disadvantage: you have to use the ­#show (or ­#showtooltip) command to get an icon and tooltip in your action bar. We can now build a macro that clicks on our invisible secure action button with the long macro attribute. The macro is quite simple as the /click command expects a global variable containing a button

One last topic remains, and it actually has little to do with traditional macros—using Lua in a macro. Doing this is quite simple: just use the slash command /script <lua code>. But what Lua could be of use in a macro? Remember that it is not possible to cast a spell or use an item from Lua, as these functions are protected and can therefore only be called from secure code. Code from a macro can never be secure

You saw a simple chat message example for Lua in a macro in the last section. We called ChatFrame1:Addmessage(msg) to display a notification in our chat frame. Another similar use is sending a chat message, but this does not necessarily require Lua code. Simple chat messages can be sent with the usual slash commands like /asy and /raid. It is even possible to include the target in a chat message by using %t in the message; it will be replaced with the name of your current target. This is not a function specific to macros, and you can always use %t in your chat messages to refer to your target in World of Warcraft. The following macro is useful for druids; it casts Rebirth and sends the name of the resurrected player to the raid chat

The macro doesn’t use any Lua at all. But another useful functionality in such a macro is sending a whisper to the player who will be resurrected so he can prepare himself to get back in the fight. You might think it was possible to use /w %t Battle res incoming\_ here. But %t only works in chat messages, not as the target of a whisper. This means we need Lua code to send a whisper to our current target

An obvious disadvantage of using Lua in macros is that you quickly hit the 255-character limit. The solution to this is outsourcing the actual Lua code in a small addon, which then provides a simple slash command or Lua function that is called from a macro. The following code creates the slash command/wtarget <msg>, which sends <msg> to your current target

Besides being shorter and simpler to use, this doesn’t generate an error message when you have no target.

Another good function would be one to schedule messages to be sent after a specific time. This allows you to create countdown macros, which can be very useful, so let’s see how to do it

The simplest way to implement this is just using SimpleTimingLib from your macro. But it’s easier to create a slash command that takes a time in seconds and another slash command that will be executed after this time. The following code creates a slash command that executes another slash command after a given time. It requires the library SimpleTimingLib to be installed, either as a stand-alone version or embedded in the addon

The function runCmd executes the actual command by using a trick—it modifies the text contained in the default chat edit box and calls the function ChatEdit\_SendText(editBox), which is defined in the file FrameXML\ChatFrame.lua. It would also be possible to parse the command here and find the slash command handler or emote that is associated with it, but that would be quite an effort (requiring about 30 lines of code). You can read the function ChatEdit\_ParseText(editBox, send) in the file frameXML\ChatFrame.lua if you are curious to see how slash commands are handled in the default UI

It is also possible to omit the leading slash in the command to be scheduled, as the /in handler adds a slash if the command doesn’t start with a slash. The syntax /in 3 raid Pull now!\_ would also work

It is not possible to execute any secure command with the /in command, because the execution path is tainted by the addon that provides the slash command

Another timer slash command that comes with Deadly Boss Mods is the so-called pizza timer, which can be started with the slash command /dbm timer xx:yy <text>. This shows a DBT timer with the name <text>. You can use this in a macro to track the cooldown of an ability or spell locally. But there is also another, more powerful slash command:/dbm broadcast timer xx:yy <text>, which broadcasts a DBM timer to your party or raid group. This command requires raid assistant or leader status.

Raid leaders can use a macro like the following to announce short breaks with a timer to the raid group

Why would anyone want to suppress an error message? The answer is that there are macros which always generate error messages, even if they are executed successfully. An example of this is the trinket macro we used earlier

This macro tries to activate both trinkets, and this always generates an error message. The second /use command fails because of the global cooldown if the first succeeds. A simple way to get rid of the small red “Item is not ready yet” message is to clear the frame that displays it. This frame is stored in the global variable UIErrorsFrame: its type is MessageFrame. Such a message frame basically works like a chat frame (which is a ScrollingMessageFrame), but without the ability to scroll. You can add messages to such a frame with its method frame:AddMessage(msg) and, most important for us here, clear it with the methodframe:Clear(). A full reference for this frame type (and all other frame types) is available in Appendix A

The following macro clears the UIErrorsFrame after trying to execute the two /use commands

It is of course also possible to place this code in an addon. But the Lua code here is relatively short, so it is not really worth the effort.

A few commonly used macros are unlike other macros, as they are basically just Lua scripts you execute once. These “macros” simply set a so-called cvar (console variable, a variable that stores game settings like your video options), and this can also be done by editing the file World of Warcraft\WTF\config.wtf. Such an option can be set by calling SetCVar(“option”, “value”) and retrieved with GetCVar(“option”).

One popular example of such a macro is the following one, which allows you to zoom out further by setting the option camerDistanceMaxFactor to 2.5. The distance slider in the interface options menu ends at 2.0

Note that the value of the option is saved in your config.wtf; this means you only have to execute this “macro” once, and you will probably not store this simple Lua script in a macro at all

There are many other options available that can be set to higher values than allowed by the options menu. The video options that control the “Ground Clutter” density and distance are a good example of this. The following macro sets these two options to values beyond the sliders in the video effects menu:

This increases the amount of grass and other vegetation and the range on which it is visible. However, this can decrease your FPS drastically if you don’t have a high-end system; remember that the options in the configuration menu are limited to lower values for a reason

Circumventing the [flyable] bug

Another really useful macro is one that uses your flying mount if you are in a zone where flying mounts are allowed or your normal mount otherwise. This would seem to be a really simple macro if the condition flayble worked properly. You could then simply use the following macro to do this:

But flyable does not work properly. This condition evaluates to true if you are in an area in Outlands or Northrend where mounting is allowed. This worked just fine before Wrath of the Lich King, as flying mounts were allowed everywhere in Outlands that normal mounts were allowed. This is no longer true with Wrath of the Lich King, as you cannot use your flying mount in Dalaran (except for the subzone Krasus’ Landing) or Wintergrasp, but the condition still evaluates to true there. This means the macro fails in these zones

But using a mount is not a protected action, even though it can be done with the secure slash command /cast. The function CallCompanion(type, id), which was introduced in patch 3.0 (the same patch that broke flyable), can be used to summon a non-combat pet or a mount. Type can either be “CRITTER” to summon a non-combat pet or “MOUNT” to use a normal mount. The ID of a mount can be acquired from the default pet and mount menu; the first mount (or pet) that is displayed there has the ID 1, the second the ID 2, and so on

The following code creates a slash command /mount, which uses a random flying or normal mount from a list of available mounts based on the zone you are currently in.

Note that it is necessary here to check whether we are already mounted, and dismount if so because CallCompanion(type, id), unlike the /cast command, does not dismount you. The slash command then uses a good condition to check whether flying in the zone is really allowed and uses a random mount from the table flyingMounts if this is the case. Otherwise it uses a random mount from the table groundMounts. You have to fill these tables with the IDs of your favorite mounts if you want to use a macro based on this script

A possible macro that makes use of an addon providing this slash command might look like this

­#showtooltip is just to get an icon and tooltip for the macro, and it doesn’t really matter that this icon is wrong in Dalaran and Wintergrasp

This chapter demonstrated a different way of interacting with the World of Warcraft API: through slash commands, which seem to be quite simple at first glance. But these secure slash commands can be combined with complex conditions that make them very powerful. We looked at all available secure slash commands, which can execute actions that can normally only be done from secure code (that is, from the default UI) or by using secure templates

We discussed how we can combine multiple conditions to build complex expressions by using logical operators. We then saw how we can use these expressions in simple if-then-else-end blocks in slash command

The next main topic in this chapter was using Lua in macros, and we tried out a lot of useful Lua scripts you can use in your macros. You also saw how to combine macros with secure templates to circumvent the 255-character limit, a workaround that can be important if you want to use long macros

Secure slash commands and secure templates are the only ways of executing certain actions like casting spells. You saw in this chapter how to combine these two techniques to create extremely powerful macros that help you in your raids or PvP matches.

Make sure to check the WoWWiki page <http://ww.wowiki.com/Useful_macros> if you are looking for macros. This page has a quite large compilation of macros for every purpose; you will probably find whatever you are looking for there. You probably already know this list of macros and have copied a few macros from there. But you now know how these macros work and you can customize them to fit your needs. It’s always better to understand a chunk of code and be able to customize it than just to copy and paste it.