

Readme for Macro/Keybind Mod API

API Version 10

Using this API you can add your own features to the Macros Mod scripting engine, before you begin, you should be comfortable with using MCP to decompile Minecraft and make mods and also with building a .jar file containing compiled classes.

Using the API with your development environment

The API package contains both the compiled and obfuscated binaries and also the unobfuscated sources designed for use with the Minecraft Coder Pack (MCP). You can choose to put the binaries into the minecraft jar before decompilation or just use the MCP sources. Use the binaries if you are working on obfuscated code and just want to include a library on your build path.

The sources consist of two Eclipse projects, MacrosAPI and CommonLib. MacrosAPI contains the API classes and CommonLib contains a small number of common upstream dependencies which are not a part of the API. You will need to include both projects in your workspace if you are using the sources with Eclipse.

Requirements for modules

Modules must be provided in .zip or .jar format and must be named **module_** followed by the name of your module, eg. module_mymodule.jar

Actions and providers in your module must implement the correct API version for the version of macros you wish to use the module with.

Check the console output when loading the game to check your module was loaded or ignored.

About the API

There are 3 kinds of classes you can include in your module which provide different functionality:

Script Actions

Script Actions are the "commands" which exist in the scripting language. They can execute in different modes depending on the type of command.

Normal	actions should do their processing in the <code>execute()</code> function and should return <code>false</code> to all of the type enumeration functions (except <code>isClocked()</code> , see below).
Latent	actions should do their processing in the <code>canExecuteNow()</code> function and use <code>execute()</code> as the completion function. They should return <code>false</code> to all of the type enumeration functions.
Stack push	actions can manipulate the stack, they should return <code>true</code> to <code>isStackPushOperator()</code> and implement the <code>executeStackPush()</code> function. They can optionally implement <code>executeStackPop()</code> if they need to provide special handling for certain pop operations. Stack push operators must implement <code>canBePoppedBy()</code> and return <code>true</code> for at least one type of stack pop operator.
Stack pop	actions pair with stack push actions, they should only return <code>true</code> in <code>isStackPopOperator()</code> and the rest of the pop logic is handled by the <code>canBePoppedBy()</code> and <code>executeStackPop()</code> in the corresponding push.
Conditional	actions push the stack, they should return <code>true</code> to <code>isConditionalOperator()</code> and perform their processing in <code>executeConditional()</code> .
Conditional pop	actions are like pop actions but for conditionals, they should likewise return <code>true</code> for <code>isStackPopOperator()</code> but must also implement <code>matchesConditionalOperator()</code> in a similar way to the <code>canBePoppedBy()</code> function on regular stack pushes.

Script Actions are singletons and may be called asynchronously by different simultaneously-running macros. They are thread-safe but should not make any assumptions about the identity of the calling thread.

Because script actions themselves are not generally stateful, they can provide stateful behaviour by storing state data in the command instance which is invoking them. To achieve this the `IMacroAction` instance is provided to all calls against the action and the `instance.SetState()` and `instance.GetState()` can be used to store and retrieve an instance state object which you can use to make stateful behaviour possible.

Script actions also receive tick updates from the script engine regardless of whether they are currently executing, you can use this tick callback to allow your script action to perform processing outside of its execution, ticks callbacks can

be "clocked" (called only once per tick) or "unclocked" (called every frame) and the type of callback is determined by the `isClocked()` function. I recommend returning `true` if you don't plan to use tick events, to reduce the overhead in the script engine.

`isClocked()` also determines the latent behaviour of the script action, since clocked events will experience a 1 tick wait between calls to `canExecuteNow()`, whereas unclocked events will receive a callback every frame.

For regular script actions (latent or otherwise), execution in a script is straightforward. When the action is queued for execution, the script engine first calls the `canExecuteNow()` function. This function should return `true` if the executive should call the `execute()` function, or `false` to pause script execution. You can do as much processing in the `canExecuteNow()` function as you like and return `false` for each tick until your process is "finished", at which point `execute()` will be called.

`execute()` is called once `canExecuteNow()` returns `true`, and should return either `null` or an `IReturnValue` implementation. Return values are not implemented in the current executive, but actions can return a return value appropriate to their function if you choose and it will be supported in the future. `getLocalMessage()` and `getRemoteMessage()` are supported, and can be used to send log messages to the user, or chat messages to the server respectively.

For examples of how the `canExecuteNow/execute` functions interact, consider three existing script actions:

LOG	doesn't need to provide any latent functionality so always returns <code>true</code> in <code>canExecuteNow()</code> . It provides its functionality in <code>execute()</code> and returns a return value with a <code>localMessage</code> .
WAIT	doesn't do anything in <code>execute()</code> , but initialises a timer in its <code>canExecuteNow()</code> function and stores it in the instance state. It returns <code>false</code> until the timer has expired at which point it returns <code>true</code> . It does nothing in <code>execute()</code> except return <code>null</code> .
FOV	uses the <code>canExecuteNow()</code> function to interpolate the player's FOV value by a tiny amount each tick. Once the interpolation is complete it returns <code>true</code> . In the <code>execute()</code> function it sets the FOV to the final value in case the interpolator suffered a rounding error during its processing.

In general, script actions should perform their processing as quickly as possible. Latent functions should process over several ticks by using `canExecuteNow()` as a latent tick function.

Script actions **must** return their action name in the `toString()` function!

Variable Providers

Variable Providers provide dynamic variables into the script engine when scripts are run by the user. Variable providers should be as light weight because they are updated frequently.

Registered variable providers receive a call to `updateVariables()` each frame and after every script action is executed. The provider should calculate/obtain the variable values and store them in an internal cache to avoid recalculating where necessary.

Variable providers should return one of four types in response to a call to `getVariable()`

<code>null</code>	A provider should return <code>null</code> if the variable can not be provided by this provider.
<code>Integer</code>	Boxing is required to allow the engine to recognise the variable type. For numeric variables the provider should return an <code>Integer</code>
<code>Boolean</code>	Boxing is required to allow the engine to recognise the variable type. For boolean variables the provider should return a <code>Boolean</code> .
<code>String</code>	String variables should return a <code>String</code> object.

In general, variable providers should always return the same variables, if variables are calculated dynamically then you should ensure that variables are initialised/returned with a default value when a real value is not available.

Scripted Iterators

Scripted iterators are a special type of variable provider that support providing an array of values inside a `FOREACH` loop. In general, scripted iterators should build their collection in their constructor, and read from the cached values during iteration.

Implementing API objects

The three types of API objects are available in the `IScriptAction`, `IVariableProvider` and `IScriptedIterator` interfaces. To create your custom objects you must implement the relevant interfaces. To complete the implementation of the API, you must annotate your class with an `APIVersion` annotation with value set to the Macros API version you wish to support (currently 10).

`onInit()` is only called for your custom class if the `APIVersion` annotation matches the active API version number. You should register your action or provider with the `ScriptCore` in this function.

For example, in a custom script action:

```
@Override
public void onInit()
{
    // Register this action with the script core.
    ScriptCore.registerScriptAction(this);
}

@Override
public String toString()
{
    return "customaction";
}
```

Script actions MUST override `toString()` and return their action name. This is the name which is then used in scripts to invoke the action.