Triggers in CocoMUD

Triggers may very well be the most powerful feature of any MUD client. CocoMUD offers an advanced system of triggers, which will be discussed and presented with examples in this documentation.

What is a trigger?

Usually, when you enter a command, the server to which you are connected "replies" with one or more lines. This reply, just like the command you sent, is mere text (sometimes with little additions, like colors).

Triggers can intercept a specific line of text and react to it to do something. That's the basic definition of any trigger: I react to something (one or more lines sent by the server) and I do something in response (playing a sound, changing the text, sending another command, saving some parts of the line, moving the cursor and so on).

Since triggers are so powerful, their setting has been spread on several layers. You don't need to understand, or even know, the most advanced type of trigger to use this feature. This documentation will walk you through the basic setting of triggers, until the setting of most advanced ones.

Creating a trigger

Let's begin with something simple: a trigger that plays a sound when someone (anybody) speaks in a channel.

This will obviously differ on every MUD, so don't hesitate to adapt this example. For this example, we'd say we can use the "chat" command to send a message to everyone logged to the MUD. The command we will type should be like this:

```
chat hello everyone!
```

And if the MUD behaves, it should reply with:

```
[public] You say: hello everyone!
```

Similarly, if someone else speaks on this channel, you might see something like:

```
[public] Aaron says: hi there
```

In short, if we want to intercept the messages on this channel, we need to watch for lines that begin by "[public]" (that is the word "public" between brackets).

> Why should we intercept these messages, anyway?

If you're new to triggers, you might wonder why it's useful. The answer in that specific case could be that, if we receive a message on the public channel, the client could play a sound to alert us. That's on this task we're going to work.

So we're going to see how to create a trigger that:

- Reacts when it receives a line beginning with "[public]".
- Plays a sound when it does.

Like most features on CocoMUD, triggers can be created either through the interface or through SharpScript. We'll see the former first:

Through the interface

To add, edit or remove triggers, select **Game** in the menu bar, then **Triggers**.

You will find yourself in the list of current triggers configured for this world. This list might very well be empty at first. To create a trigger, click on the **Add** button.

A new dialog box opens. The cursor should be in an edit field, where you're asked to type your trigger. That's the reaction part, the place where you will specify what fires your triggers.

In our example, we've determined that our trigger should be executed whenever a line "beginning by [public] is detected. You can type [public] in this field, but don't leave it just yet: if we create a trigger [public], it will only fire when the server sends a line containing **only** [public]. That's not the case for us, we want for our trigger to fire when the line begins with, not only contains [public]. To do that, we'll have to add an asterisk (*) after [public]. If we type [public]* in our trigger, CocoMUD will know that we are looking for lines that begin with [public] followed by something (no matter what).

You might recognize this syntax, it's the same we use to create aliases with variables. And that's not a coincidence, we'll see later why.

For the time being, you can write in this edit field:

```
[public]*
```

Press Tab: you find yourself on a list of possible actions to be performed when this trigger fires. In our example, we want our trigger [public]* to play a sound. So browse this list until you find:

```
Play an audio file
```

Then press Tab again and click on the **Add action** button.

You are now asked to configure this action. What audio file do you want to play when this trigger fires? You have a **Browse** button if you want to select the file in your file system. And once selected, you have a **Test** button, to check that CocoMUD can play this audio file just right (CocoMUD supports .wav and .ogg files). If everything works, just press **OK**. The action will be added. The cursor is moved into a list where you see only one item:

```
#play whatever.wav
```

Of course, the "whatever.wav" will be replaced by your file path and name.

This is the sharp editor: it is used by every feature (aliases or macros for instance) that needs to execute complex actions at some point. The window can be a little intimidating at first, but as you've seen, adding an action isn't that complicated. So here is a brief summary of the trigger dialog with the sharp editor:

- The first thing to set is the trigger itself (the part that will fire the trigger, [public]* in our case).
- Below is a list of actions currently associated with this trigger. It's a list of actions (what to do when this trigger fires)? You can add more than just one action. Note that this list is disabled if it's empty.
- An **Edit** button, to edit the selected action. For instance, in our case, if you've changed your mind and want to play a different audio file, you need to select this line and click on **Edit** to change the audio file.
- A **Remove** button, to remove this line of action.
- Still below is another list of actions. This time, it's a list of actions you could desire to add. That's the list we browsed through when trying to play a sound if this trigger fires. We won't describe all of them, but you'll see other examples in this documentation.
- Then is a button to **Add** the action you have selected. Remember that you can connect a trigger with zero, one, two or more actions, there's no real limit.
- There are other checkboxes and options in this dialog that we will discuss later.

You can now press on the **OK** button, since we did what we wanted: our [public]* trigger is connected to the action to play a sound.

You will find yourself in the list of triggers again, with your newly created trigger. You can press **OK** again to close this dialog and save the trigger to your configuration. If you don't press **OK**, the trigger will not be saved.

To see if it works, let's try to fire this trigger! You could send to the server something like:

```
chat does it work?
```

If everything goes well (assuming, one more time, that the server answers the way we have planned), it should send you something like:

```
[public] You say: does it work?
```

And you should hear a sound, the sound you have selected. There! It wasn't that complex, was it?

Through SharpScript

As usual, the interface allows to manipulate potentially complex settings, but they all are converted to SharpScript in the end. Setting the trigger we have worked on above in SharpScript is quite simple. You can enter the following command in your MUD client, or write it in your "config.set" file:

```
#trigger [public]* {#play path/to/sound.wav}
```

It's a SharpScript instruction. From left to right:

- #trigger is the name of the action. Here, #trigger just creates a new trigger.
- [public]* is what should fire our trigger. Just like above, we have specified [public]* , which means [public] followed by anything.
- Then we specify the action or actions to be executed when this trigger fires. Here, we have #play path/to/sound.wav. #play is another SharpScript action that just plays the sound you give it in argument, if it can. Notice that we have surrounded our call between braces, because there is a space between the name of the function (#play) and the argument of the function (path/to/sound.wav).

Adding a trigger using the SharpScript syntax may be quicker, but it will not forgive easily if you make an error of syntax. That's one of the reasons why using the interface might be safer.

Editing a trigger

In the dialog box of triggers (menu bar, **Game** -> **Triggers**), you can edit a trigger if needed. You will need to do so if you want to change the part that fires the trigger, or the actions associated with this trigger (for instance, the audio file to be played, in our example).

Removing a trigger

Removing the trigger can also be done through the interface. Just press the **Remove** button and confirm that you do want to remove this trigger. Do not forget to click on **OK** to close the list of triggers. If you don't, the trigger will not be removed.

Using variables in triggers

It is time to look more closely at our previous trigger, in particular at the asterisk sign (*). It is not a coincidence that you may recognize this syntax from the documentation on aliases, for it is exactly the same principle: * means anything.

So here is a list of possible syntax:

Syntax	Meaning
Welcome*	Any line beginning by Welcome .
*dude	Any line that ends with dude .

spoil	Any line containing spoil, at the beginning, the end or the middle. It will also fire if the line contains spoiler for instance.
* spoil *	Any line that contains the word spoil surrounded by spaces. The word spoiler would not fire the trigger in this context.
You earned * credits in *.	Lines like You earned 80 credits in combat. or You earned 10 credits in management. will fire this trigger. Also note that the line You earned some unknown credits in something. will also fire.

In short, an asterisk sign (*) means anything, including a number, a letter, a word, a space, or whatever else... including nothing.

A little word of caution: the syntax of your trigger is really important, and you should check when the trigger would be fired.

```
*public*
```

This trigger would fire when there is the word public everywhere in the line. It means this trigger would fire when you receive the following line:

```
[ooc] Modo says: please talk on the public channel, not here, if you're stuck with a quest.
Yassen tells you: where should I write, if I want my message to be publicly readable?
To send your book to publication, press S.
```

Remember that a trigger can be easily restricted... but can easily be fired. That all depends on you.

Back to variables. The asterisk does two things:

- It determines when the trigger should fire (specifying "anything").
- It writes in variables.

Let's use the same example of our trigger [public]*. What happens when, say, you receive a line like:

```
[public] Edgar says: I don't get it at all, could someone help me?
```

First, the trigger [public]* fires, then the part after [public] is sent to a variable. A variable can store information, and that's just what it does here. Variables are numbered starting with $1, $2, $3 and so on. So in our example, when we receive the above message, the variable $1 is created containing the text:

```
 Edgar says: I don't get it at all, could someone help me?
```

What can we do with it? About anything. Every parameter in our actions could use variables. We will see a concrete example below. For the time being you could display it:

```
#say $1
```

Which should display:

```
 Edgar says: I don't get it at all, could someone help me?
```

Why does it begin with a space?

If you wonder about it, just put the trigger with the line side-by-side, that might help:

- Trigger: [public]*
- Line: [public] Edgar says: I don't get it at all, could someone help me?

Can you see it? Our line puts "public" between brackets, then a space, then the name of the one speaking... while our trigger just says "public" between brackets and anything after that. Which includes our space.

The solution? To slightly change our trigger:

```
[public] *
```

This time, we put a space between the right bracket and the asterisk sign. So now if we receive the line:

```
[public] Edgar says: I don't get it at all, could someone help me?
```

And we display $1, we'll see:

```
Edgar says: I don't get it at all, could someone help me?
```

Spaces may be the main source of confusion in your triggers. The best advice is to look closely at the lines you receive from the server, and use only the * sign when you don't know what will be put there.

CocoMUD channels in triggers

CocoMUD has a more interesting feature of channels. A channel is basically a list of events. It will store these events and you can display this list whenever you want. It might be useful, for instance, when exploring the game and not paying particular attention to some messages (on the public channel, or the ooc channel, for instance). Afterward you might want to read them, so you display the list of messages.

You will find more informatioon in the documentation about channels.

Mute triggers

In some cases, when you receive a line, you don't want for it to appear.

Does it really happen?

In some contexts. For instance, some MUDs send regular ambiance messages to the client. That can be great to set the ambiance, but that's usually not very user-friendly for people using screen readers.

```
A cloud of sparks from your campfire soars toward the darkening sky.
```

Nice and, well, yeah, when you're near a campfire, it cracks and smokes and sends sparks, but with a screen reader, that's not really

useful. So we could just remove this line.

To do so, create a trigger. Through the interface:

- Go to the menu bar, **Game** -> **Trigger**.
- Add a trigger.
- Paste the line: A cloud of sparks from your campfire soars toward the darkening sky.
- No need to select an action, we won't do anything (except if you want to add a crackling sound when that happens).
- Tab until you find the "mute trigger" checkbox. Check it.
- Press **OK** several times to validate and save.

If the client receives this line, it will just ignore it and not display it.

You could have done the same thing in SharpScript:

```
#trigger {A cloud of sparks from your campfire soars toward the darkening sky.} {} +mute
```

Notice that the second parameter is empty (just {}, meaning we don't perform any action). The third parameter is a flag, beginning by + or - and followed by the name of the flag. Here, +mute means we activate the mute flag to set our trigger as a mute trigger.

As pointed out, you can have a mute trigger that performs actions, like playing a sound, displaying something, sending a command, putting the line (or part of the line) in a channel, and so on.

Mark triggers

Mark triggers can be useful for accessibility. They will put the cursor directly on the line that fires the trigger. It is useful in some contexts: for instance, when you're exploring, you want the cursor to be put on the list of exits, rather than at the bottom of the window where you will have to press the up arrow key several times.

It's the same principle to create this trigger:

- Open the menu bar, **Game** -> **Triggers**.
- Click on **Add** to add a trigger.
- Put the part that should fire the trigger. In our example, perhaps something like: Obvious exits: * .
- Tab to check the "mark trigger" checkbox.

The next time you will receive the line beginning with "Obvious exits", the cursor will be moved directly on it.

The same thing in SharpScript would be:

```
#trigger {Obvious exits: *} {} +mark
```

Triggers with substitution

In some cases, when you receive a trigger, you want to modify the line that fired the trigger. One of the common case is to shorten the message. Some MUDs have very long messages for some channels, like this:

```
Somebody publicly speaks on the 'ooc' channel in a worried voice: is it safe?
```

While this is great, it would be nice to shorten it and perhaps write it differently. Like:

```
[ooc] Somebody: is it safe? (with a worried voice)
```

The way to do that is to create a trigger, and add a line of substitution. The line of substitution will replace the line that fired the

trigger in the client.

- Open the menu bar, **Game** -> **Triggers**.
- Click on **Add** to add a new trigger.
- Write the part to fire this line. Here we might have something like:

```
* publicly speaks on the '*' channel *: *
```

- Notice that we have $1 containing the name of the speaker, $2 containing the name of the channel, $3 containing the tone of the voice and $4 containing the message.
- Tab until you find the empty text field "Message to substitute to the trigger line, if any".
- In it put:

```
[$2] $1: $4 ($3)
```

Does it look understandable? Take the time to read the trigger, the line that should fire it and the substitution.

The same trigger in SharpScript would be:

```
#trigger {* publicly speaks on the '*' channel *: *} {} {[$2] $1: $4 ($3)}
```

Important note: we have three arguments to the #trigger action here. The first one is still the name of the trigger, the second one is still the action to be performed (empty in our case). The third one is our substitution. If the field is empty, there's no substitution. That's just what happened for all of our previous tests.

Triggers fired by a regular expression

This section is more advanced, beware.

The asterisk symbol is great, but it doesn't offer much flexibility. Hopefully, we can connect a trigger with a regular expression . Regular expressions won't be described in this documentation, that's clearly off topic, but there are plenty of resources out there to learn it.

As far as CocoMUD is concerned, there's not much to know: if the trigger begins with a ^ symbol, it's a regular expression and CocoMUD will treat it as is.

```
^You receive \d+ XP.$
```

This will trigger when you receive the line "You receive ... XP.", with "..." being one ore more numbers. This won't fire the trigger if you receive the following message: "You receive a lot of XP."

An important thing to note when using regular expressions in triggers, however, is that, if you want to capture parts of the line, you have to use groups:

```
^You receive (\d+) XP.$
```

Then the number of XP will be put in $1. You can also use named groups and call them with $myname afterward.

Advanced triggers in Python

The SharpScript engine is really great... but it doesn't allow everything. And its aim is to remain light and not allow everything. It's not a programming language. But Python is. And CocoMUD is developed in Python. The SharpScript engine is designed to send code to Python when the user asks it to do so. That means you can have triggers that execute much more complex actions. At the

same time, the Python code can access all functions defined by the SharpScript engine, which keeps the code simple and potentially very powerful.

We'll take an example as usual, but keep in mind the possibilities are really endless.

The game sends a line when earning XP, but it also displays the total number of XP needed to level up.

For instance:

```
You receive 37 XP and need 500 to level up.
```

Let's say we want to extract these two numbers and display a percentage. xp / total * 100. That's not going to work with pure SharpScript.

For the time being, it's not possible to use the interface to manipulate Python code. So we'll need to do it in the "config.set" file directly.

```
#trigger {You receive * XP and need * to level up.} {+
    # $1 contains the number of XP
    # $2 contains the total number to level up
    xp = args["1"]
    total = args["2"]

    # We're going to try and convert these numbers
    try:
        xp = int(xp)
        total = int(total)
    except ValueError:
        # The numbers can't be converted, but do nothing
        pass
    else:
        percentage = xp * 100.0 / total
        say("You receive {}/{} XP ({}%).".format(xp, total, int(percentage)))
}
```

Wow! That was some trigger! Let's quickly review it:

- The line that should fire the trigger shouldn't be much of a surprise by now.
- The second parameter begins by a {+ (a left brace followed by a plus sign). This tells CocoMUD that what follows is Python code.
- Note that all the code is indented. This is not just for readability this time, this is a Python requirement. You can use a single space or a tabulation to indent, that's your choice, I usually use 4 spaces because it's easier to read, but that's only a convention.
- Some comments. Don't underestimate their positive impact.
- We extract the two numbers (XP and total). To access them, we use args which is a dictionary containing all variables. Remember that the first variable is $1, we need to access it through args["1"].
- We need to convert these numbers. Why? There are still strings at that moment, it's as if they had been sent by the user. And CocoMUD doesn't do anything about it, so you need to do it manually. That's why we convert these numbers in a try/except/else statement. Notice that if the conversion fails, we do nothing.
- Next we create the percentage. It's Python 2, so we need to explicitly state that it should take into account floating points.
- And finally we use the say() function. That's just like using the #say function in SharpScript, it's the same thing (not the same syntax because we're in Python). All SharpScript functions can be accessed like that, so you could use play() or send() or feed() and more.

I'm not really happy about my previous trigger... I'm not against some lines of code, especially since I find them much more readable than when you try to do everything in script, but... we could definitely work on something safer with the help of regular expressions.

```
#trigger {^You receive (\d+) XP and need (\d+) to level up.$} {+
    # $1 contains the number of XP
    # $2 contains the total number to level up
    xp = int(args["1"])
```

```
    total = int(args["2"])
    percentage = xp * 100.0 / total
    say("You receive {}/{} XP ({}%).".format(xp, total, int(percentage)))
}
```

Okay, using regular expressions in your trigger is a bit more technical, but the gain for our code isn't to be dismissed.

If you want more help about using Python code in SharpScript, just refer to the section describing SharpScript.