

Contents

DialogueQuest User Manual	2
DialogueQuest for non-coders	2
Writing Dialogue - DQD	2
Basics	2
Comments	2
Flag Solving	3
BBCode and Text Effects	3
See Also	3
Characters	3
The Say Statement	4
See Also	5
The flag statement	5
The choice statement	7
The Branch Statement	7
Choice	8
Evaluate	9
Flags	10
Flag operators	11
See Also	12
The signal statement	12
The call statement	13
The exit statement	13
DialogueQuest specific BBCode	13
The speed bbcode	13
The pause statement	14
See Also	14

DialogueQuest User Manual

DialogueQuest for non-coders

DialogueQuest features a standalone program called [DialogueQuestTester](#) that allows running dialogues without a Godot environment.

Writing Dialogue - DQD

Basics

DQD stands for DialogueQuest Dialogue and is the dialogue format of DialogueQuest.

The DQD format uses the .dqd file extension.

DQD is a simple text-based format, that goes something like this:

```
statement | param1 | param2 | ...
```

Every line starts with a statement which 'moves forward' in the line like a pipeline.

The most basic and most used statement is the [say](#) statment, which looks like this:

```
1 say | joe | Hello DialogueQuest
   say | You don't even need a character
```

Comments

DQD Support comments.

A line that starts with // is considered a comment, and will not be parsed/executed.

Comments are useful for explaining things like branches, flags, or even leaving a comment for your team on their good work :)

Comments can also be used to temporarily disable parts of the dialogue without deleting them.

An example of comments:

```
// The line bellow is commented and will not run. This
   is a comment too by the way!
2 // say | This is a comment, you will not see this
   dialogue
```

```
say | This is not a comment, you will see it
```

Flag Solving

See [flag](#)

If you have set a flag, you can get it's value with the special syntax `${flag}`

For example:

```
flag | inc | 5 | monkeys
2 say | There are ${monkeys} little monkeys jumping on the
  bed.
```

BBCode and Text Effects

In order to have text effects and formatting such as **bold text**, *italic text*, and much more.

BBCode is a well-known format, and you can find out more about it [on the Godot documentation](#), but here's a basic example:

```
say | italian_man | [i]I am speaking in italic! No not
  italian...
2 say | brave_man | I am brave and [b]bold[/b] in the face
  of danger.
say | small_man | [font_size=8]Please don't make fun of
  my font size, I'm quite insecure about it.
```

See Also

[characters](#)

[say](#)

[choice](#)

[branch](#)

[DialogueQuest BBcodes](#)

Characters

Characters are simply a collection of data, and have the following properties:

An ID - This is how they will be referred to in [DQD](#). The ID is not shown in-game.

A Name - The name that will be displayed in the in-game dialogue.

A Color - The color Associated with the character, used mainly for displaying their name.

A Portrait - An image that will be displayed when the character is speaking.

The Say Statement

The say statement is the most common statement in DialogueQuest.

It's usage is:

```
1 say | [character_id] | [speech]
   say | [character_id] | [speech] | [speech2]
3 say | [character_id] | [speech] | [speech2] |
   say | [speech]
5 say | [speech] | [speech2]
   say || [speech] | [speech2] |
```

The basic use case would be:

```
say | my_character | Hey, I am saying something
```

And:

```
1 say | There is dialogue without character. Perhaps it is
   a ghost...
```

The character_id field can also be provided empty for the same result:

```
1 say | | I am still a ghost...
```

If you want to pause in the middle, you can use multiple speech pipes as so:

```
1 say | DialogueQuest is absolutely | legen|dary!
```

If you end the say statement with an empty pipe, the dialogue will advance without user input:

```
1 say | dude1 | Hey man so I heard about this game called
   DeshanimQuest and |
   say | dude2 | Yeah whatever dude
3 say | dude1 | Hey don't cut me off like that!
```

If using it without a character, you **must** provide an empty character:

```
1 say | This is not going to work... |
```

```
1 say | | This does work though! |
```

See Also

Writing Dialogue

[BBCode and Text Effects](#)

The flag statement

A flag, is simply a value that can exist, or not exist.

The act of creating a flag is called raising it, afterwards we can check if it exists, and what it is set to.

It's usage is:

```
flag | raise | [flag]
2 flag | set | [value] | [flag]
flag | inc | [flag]
4 flag | inc | [amount] | [flag]
flag | dec | [flag]
6 flag | dec | [amount] | [flag]
flag | delete | [flag]
```

A basic example would be:

```
1 flag | raise | is_using_dialogue_quest
3 // This will happen
branch | flag | is_using_dialogue_quest
5     say | We are using DialogueQuest.
branch | end
7
// This will not happen
9 branch | no_flag | is_using_dialogue_quest
    say | We are NOT using DialogueQuest.
11 branch | end
```

You can also use flag | inc and flag | dec to use integer (whole number) flags:

```

    flag | inc | money
2
    // Will say `I have 1 money`
4 say | I have ${money} money

6 flag | inc | 6 | money

8 // Will say `I have 7 money now`
  say | I have ${money} money now
10
    flag | dec | money
12
    // Will say `I have 6 money now`
14 say | I have ${money} money now

```

You can use `flag | set` to set a flag as an arbitrary value like so:

```

    flag | set | Mage | player_class
2
    // Will say `Oh sick! I am a Mage`
4 say | Oh sick! I am a ${player_class}

6 flag | set | 20 | number_of_enemies

8 // Will say `We have 20 enemies here, that's a lot!`
  say | We have ${number_of_enemies} enemies here, that's
    a lot!

```

And finally, you can delete a flag as well:

```

1 flag | raise | road_is_safe

3 // Will say `The player proceeds forward` (in Italics)
  branch | flag | road_is_safe
5     say | [i]The player proceeds forward
  branch | end
7 branch | no_flag | road_is_safe
    say | [i]The player stays back
9 branch | end

11 flag | delete | road_is_safe

13 // Will say `The player stays back`
  branch | flag | road_is_safe
15     say | [i]The player proceeds forward
  branch | end
17 branch | no_flag | road_is_safe

```

```

    say | [i]The player stays back
19 branch | end

```

The choice statement

The choice statement will bring up a menu with items that the user has to choose from.

It is inherently dependant on the [branch](#) statement

It's usage is:

```

1 choice | [choice1] | [choice2]...

```

For example:

```

1 say | Which one do you like better? Apples or Oranges?
  choice | Apples | Oranges | You can't compare
3
  branch | choice | Apples
5      say | Doctors hate you
  branch | end
7 branch | choice | Oranges
      say | Juicy!
9 branch | end
  branch | choice | You can't compare
11      say | You're just so smart, aren't you?
  branch | end

```

The Branch Statement

The branch statement allows dialogue to happen in different ways depending on a variety of factors.

When a branch statement is encountered, the dialogue can go in one way or another, like a fork in the road or *branches* of a tree.

It is recommended to first understand [flag](#), [choice](#), and [flag solving](#) as they are essential for understanding branching.

It's usage is:

```

branch | choice | [choice1] | [choice2]...
2 branch | evaluate | [expression]
  branch | end
4
branch | flags | [flag1] | [flag2]...

```

```

6 branch | flag | [flag]
  branch | flag | [flag1] | [flag2]...
8 branch | no_flag | [flag]
  branch | no_flag | [flag1] | [flag2]...
10 branch | flag > | [flag] | [value]
  branch | flag < | [flag] | [value]
12 branch | flag = | [flag] | [value]
  branch | flag >= | [flag] | [value]
14 branch | flag <= | [flag] | [value]

```

A simple example of a branch would be:

```

say | Let's see about this branching thing
2
flag | raise | loves_dialogue_quest
4
branch | flag | loves_dialogue_quest
  // We will see this
  say | I love DialogueQuest!
8 branch | end

10 branch | no_flag | loves_dialogue_quest
  // We will not see this
12 say | I HATE DialogueQuest!
branch | end

```

A branch checks a **condition**, and if it finds that condition to be **true**, it runs the contents until it reaches the next branch | end statement.

Choice

When using choices, we must use the branch | choice statement, like so

```

choice | a | b
2
branch | choice | a
  say | We picked A
4 branch | end
6 branch | choice | b
  say | We picked B
8 branch | end

```

We do not have to provide a branch for every choice.

We can also check for multiple choices, like so:


```

choice | a | b | c | d
2
branch | choice | a | b
4     say | We picked either A or B!
branch | end

```

Evaluate

evaluate is the most complex branch statement, and will use [GDScript](#) to solve the branch.

It can be used like the following:

```

branch | evaluate | true
2     say | This will always happen.
branch | end
4
branch | evaluate | false
6     say | This will never happen.
branch | end
8
branch | evaluate | 5 == 10
10    say | This won't happen because 5 is not 10 :)
branch | end
12
branch | evaluate | 10 > 5
14    say | This will happen.
branch | end
16
branch | evaluate | 5 != 10
18    say | This will happen.
branch | end
20
branch | evaluate | 5 >= 5
22    say | This will happen.
branch | end
24
branch | evaluate | this == that
26    say | This won't happen.
branch | end
28
branch | evaluate | that == that
30    say | This will happen.
branch | end

```

evaluate can also be used with [flag solving](#)

```

1 branch | evaluate | ${main_character} == joe
   say | joe | Yo, uh-huh
3 branch | end

5 branch | evaluate | ${number_of_corners} == 3
   say | This is my hat
7 branch | end

9 // You can also use the 'or', 'and', '&&', '||'
   statements to check multiple conditions.
branch | evaluate | ${number_of_corners} > 3 or
   ${number_of_corners} < 3
11   say | This is not my hat
branch | end

```

Flags

The `branch | flag` statement is quite versatile, and can be used in a few ways:

This is the simplest flag check:

```

branch | flag | some_basic_flag
2   say | This really is basic
branch | end

```

We can also check multiple flags at the same time:

```

1 branch | flag | red_flag | green_flag
   say | I'll choose either anyway
3 branch | end

5 flag | raise | green_flag
branch | flag | red_flag | green_flag
7   // We will see this
   say | Great!
9 branch | end

11 flag | delete | green_flag
   flag | raise | red_flag
13 // Only red_flag is raised at this point
branch | flag | red_flag | green_flag
15   // We will see this
   say | Still great...?
17 branch | end

19 branch | flag | orange_flag

```

```

21 // We will not see this
    say | What does an orange flag mean?
branch | end

```

In the above example, the branch will be entered if *any* of the flags are raised.

There is also the alternative branch | flags (note the s for plural), which will only be entered if *all* flags are raised

```

flag | raise | table
2 flag | raise | tea

4 branch | flags | table | tea
    // We will see this
6     say | Tea is great.
branch | end

8
branch | flags | table | plate
10     // We will not see this
    say | Cookies are great.
12 branch | end

```

Finally there is the branch | no_flag statement, which will only be entered if *none* of the flags are raised

```

flag | raise | pickles
2 flag | raise | lettuce

4 branch | no_flag | knife
    // We will see this
6     say | How will I cut my pickles?
branch | end

8
branch | no_flag | pickles | tomatoes | lettuce
10     // We will not see this because
    say | I love my burgers dry as the desert.
12 branch | end

```

Flag operators

The branch | flag also has versions for using comparison operators, such as > (greater than), < (lesser than), = (equals), >= (greater than or equals), and <= (lesser than or equals).

```

flag | set | 10 | stairs
2

```

```

branch | flag >= | stairs | 11
4     // We will not see this
    say | I'm gonna take the elevator.
6 branch | end

8 branch | flag < | stairs | 2
    // We will not see this
10    say | Even a baby can climb these
branch | end
12
branch | flag = | stairs | 10
14    // We will see this
    say | The perfect amount of stairs
16 branch | end

```

See Also

[flag](#)

[choice](#)

[flag solving](#)

[GDScript Control Flow](#)

[What are Expressions?](#)

[GDScript Expression class](#)

The signal statement

The signal statement does not quite do anything for the user.

It's functionality is sending a “message” of sorts for the Godot developer to implement into concrete functionality.

It's usage is:

```
signal | [param1] | [param2]...
```

For example:

```
1 signal | play_song | Nightcall - Kavinsky
```

The developer can for example check for the signal value “play_song”, and play the song accordingly.

Also see:

[Developer manual entry for signals](#)

The call statement

This is advanced functionality, and requires coding knowledge to use

The call statement allows you to run GDScript code directly from a DQD.

It's usage is:

```
1 call | [GDScript code]
```

Using call, you can run any GDScript code.

By default, this will run code as an Expression object.

DialogueQuest has a setting that runs the code in a GDScript instance, which is more powerful, however it is experimental.

The exit statement

The exit statement will end the dialogue early.

It's usage is:

```
exit |
```

DialogueQuest specific BBCode

If you haven't already, check out the [BBCode and Text Effects](#) section.

DialogueQuest implements a few custom BBcodes:

The speed bbcode

The speed bbcode sets the dialogue speed (letters per second) within the bounds of the BBCode.

For example:

```
say | This is a regular say statment
2 say | [speed=1]This is a suuuper slooow say statment
say | [speed=500]This is a really fast say statment
4 say | [speed=10]Now I'm slow[/speed][speed=100] And now
I'm fast
```

The pause statement

The pause statement makes the dialogue pause for a specified time (in seconds) before automatically continuing.

For example:

```
say | I have hi[pause=0.5]-hiccups and a bit of a  
s[pause=0.1]-s[pause=0.1]-s[pause=0.1]-tutter
```

See Also

[DQD](#)

[BBCode and Text Effects](#)