- FAQ
- Email
- Discord
- Issue Tracker
- Twitter
- Demo

All code is kept under the `QFSW.QC` namespace

Contents

1. Quick Start

In order to get started as quickly as possible, simply navigate to `Assets/Plugins/QFSW/Quantum Console/Source/Prefabs` and add the `Quantum Console` prefab to your scene. You will then need to add an `EventSystem` to your scene if you do not already have one. After this, you are ready to try out Quantum Console! Various parts of the prefab and theme can be modified and all have tooltips to help you do so

1.2 Text Mesh Pro Support

If you would like to use QC with TMP, please first ensure you have the TMP package/asset installed to your project. Next, navigate to the inspector of a QC instance and click `Enable TMP Support`. This will add the `QC_TMP` symbol and modify the asmdef, putting QC into TMP mode. In this mode, QC will not work with normal Unity UI.

Next, click the `Upgrade to TMP` button. This will upgrade the instance to use TMP components and will rebuild the UI accordingly. *It is advised to backup your instance/scene at this point, in case the upgrade process goes wrong*.

After this, QC will use TMP and everything should work fine. Ensure that the `QC_TMP` symbol is not removed.

2. Adding Commands to the Quantum Console

To integrate Quantum Console into your project you will likely want to add your own commands to the console. To do this, you will need to use the `[Command]` attribute. By adding this to your function, property or field you can turn it into a command that will be loaded by the Quantum Console Processor.

2.1 `[Command]`

The `[Command]` attribute has the following members that can customise and alter the generated command:

- `alias` - the name of the command in the Quantum Console; defaults to the name of the member it is used on
- `description` - description of the command that will be used in command manual generation; defaults to empty
- `supportedPlatforms` - a bitwise enum specifying which platforms the command will be loaded on; defaults to `Platform.All`
- `monoTarget` - see **2.3 Non-static usage** for more; defaults to `MonoTargetType.Single`

The attribute itself can be created with any one of the following constructors:

```
[Command]
[Command(alias)]
[Command(alias, description)]
[Command(alias, description, supportedPlatforms)]
[Command(alias, description, supportedPlatforms, monoTarget)]
[Command(alias, supportedPlatforms)]
[Command(alias, supportedPlatforms, monoTarget)]
[Command(alias, description, monoTarget)]
[Command(alias, description, monoTarget, supportedPlatforms)]
[Command(alias, monoTarget)]
[Command(alias, monoTarget, supportedPlatforms)]
```

2.1.2 Multiple `[Command]`s

Quantum Console allows the usage of multiple `[Command]`s on a single member to create aliases for a single command. To prevent it being a chore of repeating the description and other aspects several times, there are additional attributes that you can add to the member.

- `[CommandDescription]` - this description will be superseded by any defined in the `[Command]`
- `[CommandPlatform]` - this enum will supersede that defined by the `[Command]`

2.2 Static Usage

Using static functions, properties and fields as commands is very easy. Here are 4 examples

```
[Command]
private static void cmd(int a) { }
```

In *scenario 1*, a command called `cmd` will be generated that has no return type and takes an argument of type int

```
[Command("int-prop")]
protected static int someInt { get; private set; }
```

In *scenario 2*, two commands called `int-prop` will be generated:

- The first will take 0 arguments and will return an int; this uses the property.get
- The second will take 1 argument of type int and will not return; this uses the property.set

```
[Command("bool-field")]
public static bool someBool;
```

In *scenario 3*, two commands will be generated called `bool-field`. Fields are treated as auto-properties, so this will behave like *scenario 2*

```
[Command("del")]
public static Func<bool> someDelegate;
```

In *scenario 4*, a command will be generated called `del`. Delegate are treated as a normal method, so this will behave like *scenario 1*

Delegate fields *must* be strongly typed, this means the following is not allowed

```
[Command("del")]
public static Delegate someDelegate;
```

2.3 Non-static Usage

Non-static usage has more caveats than static usage, but is still very simple. Non-static commands **require** that the declaring class is a **MonoBehaviour** type, This is because the object needs to be found in the scene to invoke the command on. Non-static MonoBehaviour commands introduce another concept called the `MonoTargetType`; this defines how to select the target for invocation

- `Single` - Targets the first instance found of the MonoBehaviour
- `All` - Targets all instances found of the MonoBehaviour
- `Registry` - Targets all instances registered in the QuantumConsoleProcessor registry. See **2.4 Quantum Registry** for more

Target location and selection is performed automatically

2.4 Quantum Registry

The Quantum Console Processor supports a feature called the registry. By using the registry in conjunction with `MonoTargetType.Registry`, you can have full manual control over which objects are used for invocation. To add an object to the registry, either use `QFSW.QC.QuantumRegistry.RegisterObject<T>(T target)`, or the runtime command `register-object<T>` Multiple objects can be registered per type, and will not be removed from the registry unless they are destroyed or manually removed. Objects can be removed from the registry with `QFSW.QC.QuantumRegistry.DeregisterObject<T>(T target)` and `deregister-object<T>`

2.5 Restrictions

to prevent command invocation ambiguity, commands must be unique by name or arg count. This means two commands having the same alias is legal, but must have different arg counts If they have the same arg count and the same name, you must override the alias to make them unique.

Additionally, the parameters must all be of a supported type. The full list of supported parameter types is as follows:

- `int` `float` `decimal` `double` `string` `bool`
- `byte` `sbyte` `uint` `short` `ushort` `long`
- `Vector2` `Vector3` `Vector4` `Quaternion` `Color` `Type`
- `ulong` `char` `GameObject`
- all types deriving from `Component` or `MonoBehaviour`
- `Array` `List<T>` `Stack<T>` `Queue<T>` granted that the element type is also supported
- all `enum` types

If either of these restrictions are broken, the Quantum Console Processor will discard the command and throw a warning

2.6 Extra Attributes

Quantum Console comes with extra, non essential attributes designed to make your life using QC easier.

2.6.1 `[CommandPrefix]`

By using the `[CommandPrefix]` attribute on a class, you can easily add an attribute to all of the commands declared inside of it.

```
[CommandPrefix("test.")]
public static class MyClass
{
    [Command("foo1")]
    private static void Foo1() { }

    [Command("foo2")]
    private static void Foo2() { }
}
```

This example code would produce 2 commands, `test.foo1` and `test.foo2`. This can be very useful for grouping together commands.

If no name is supplied, then just like `[Command]`, it can use the caller name.

```
[CommandPrefix]
public static class MyClass
{
    [Command]
    private static void Foo1() { }

    [Command]
    private static void Foo2() { }
}
```

This will yield the commands `MyClassFoo1` and `MyClassFoo2`.

The attribute also works with nested classes, and with multiple attributes on the same class.

```
[CommandPrefix("test1.")]
[CommandPrefix("test2.")]
public static class MyClass
{
    [CommandPrefix("a.")]
    private static class MyClass1
    {
        [Command("foo")]
        private static void Foo() { }
    }

    [CommandPrefix("b.")]
    private static class MyClass2
    {
        [Command("foo")]
        private static void Foo() { }
    }
}
```

This will yield the commands `test1.test2.a.foo` and `test1.test2.b.foo`

3. Interfacing with the Quantum Console Processor

It is recommended that you use the included prefab to do this. This prefab includes a `QuantumConsole` script that provides processor I/O, auto complete, formatting and much more. You are free however to use your own prefab with the `QuantumConsole` script or to make your own script entirely, as all the core functionality resides in the static processor.

3.1 Getting Started

1. To get started, use the `help` command. This will give you a brief intro into using the console
2. To see more help about any specific command, use `man commandName` to see its user manual
3. To see all the commands loaded by the processor, use `all-commands`

3.2 The `man` Command

The man command will generate a user manual for the specified command. By itself, it will be able to display all the available signatures of the command, and the type of each parameter used. To add a description to the command, please include it in the `[Command]` or `[CommandDescription]`. To give a parameter a description, please use `[CommandParameterDescription]`

3.3 Argument Parsing

By default, arguments are separated by whitespace. This means that `spawn player 10` would be interpreted as command called `spawn`, that takes 2 arguments, `player` and `10`. This works fine for most use cases, however issues can happen when strings are involved. If you want to pass a single argument with a space in it (for example, a string) you must enclose it with `" "`. Therefore, `spawn "player 10"` would instead be parsed as a command called `spawn` with 1 argument, `player 10`. If you want to have a string that contains the `"` character in it, then you must escape the `"` character by writing `\"` instead; this will prevent the parser from using it for argument splitting and will instead recognise it as a plain `"`.

3.3.2 Arrays and Collections

Arrays, lists, and other supported collections can be inputted into the console by enclosing the elements within `[]`

```
sort<int> [3, 2, 1]
```

The following example would use the generic command `sort<T>` to sort an array of integers, thus would return `[1, 2, 3]`

Nested arrays may be used in a similar fashion, so `[[Quantum, Console], [Array, Test]]` would be a valid input for a command expecting `string[][]`, `List<List<string>>`, `List<string[]>` etc.

3.4 Events and Callbacks

The `QuantumConsole` component provides various event callbacks that can be subscribed to, allowing you to add your own behaviour into the console without modifying it.

Name	Signature	Description
`OnStateChange`	`Action`	Callback executed when the QC state changes.
`OnInvoke`	`Action<string>`	Callback executed when the QC invokes a command.
`OnClear`	`Action`	Callback executed when the QC is cleared.
`OnLog`	`Action<string>`	Callback executed when text has been logged to the QC.
`OnActivate`	`Action`	Callback executed when the QC is activated.
`OnDeactivate`	`Action`	Callback executed when the QC is deactivated.

4. Macros

Macros, like C macros, allow you to create *shorthands* which will then be expanded before the command is parsed. Macros are defined using `#define` in the Quantum Console, or `QuantumMacros.DefineMacro`

```
#define pi 3.14
#define reset-player "teleport player (0, 0, 0)"
```

Macros can then be used within the console input my prepending them with the `#` as such,

```
rotate object (0, 0, #pi)
#reset-player
```

The usage of the `#` operator causes the macro to be expanded before parsing the input. Macros may not contain hashtags or whitespace in their name.

Note: macros will not be expanded when using #define, this is so that defining nested macros is possible

4.2 Nested Macros

Macro expansions may contain other macros enabling you to create nested macros This makes complex and dynamic macros easier to set up.

```
#define num 5
#define array [#num, #num, 20]
```

With these macros defined, `#array` fully expand to `[5, 5, 20]`. If `#num` was redefined to `10`, then `#array` would expand to `[10, 10, 20]`

5. Async Commands

In addition to normal synchronous commands, QC also supports `async` commands based off of C#'s *async/await* system.

5.1 Async Command Support

In order to write an `async` command, simply write an `async` function as you normally would, then add the `[Command]` attribute.

The following example shows a command that waits `time` ms before returning the value `time`

```
[Command]
private static async Task<int> Delay(int time)
{
    await Task.Delay(time);
    return time;
}
```

If you executed this command with the Quantum Console, then you would see the value `time` appear in the console trace once the async command has finished.

5.2 Blocking vs Non-blocking mode

By default, QC runs in a non-blocking mode for async commands. This means that you can still run other (sync or async) commands whilst an async command is still in progress; a counter will appear next to the input field showing how many async jobs are currently in progress.

If you would like to switch it to a blocking mode, in which the console cannot be used until the current job is done, then enable **Block on Execute** under the **Async Settings** section of the inspector. Additionally, the job counter can be disabled with **Show Current Jobs**.

6. Included Commands

Quantum Console comes with many commands included out of the box for the user's convenience. Some of these commands are *core commands*, whereas others are *extra commands*.

6.1 Core Commands

Core commands are implemented as part of the `QuantumConsoleProcessor` and can be found in `QuantumConsoleCommands.cs` if you wish to edit or remove any of them.

Name	Description
`help`	Shows a basic help guide for QuantumConsole
`man`	Generates a user manual for any given command, including built in ones. To use the man command, simply put the desired command name in front of it. For example, `man my-command` will generate the manual for `my-command`
`commands`	Generates a list of all commands currently loaded by the Quantum Console Processor
`register-object<T>`	Adds the object to the registry to be used by commands with `MonoTargetType` = `Registry`
`deregister-object<T>`	Removes the object to the registry to be used by commands with `MonoTargetType` = `Registry`
`display-registry`	Displays the contents of the specified registry
`use-namespace`	Adds a namespace to the table so that it can be used to type resolution
`remove-namespace`	Removes a namespace from the table
`all-namespaces`	Displays all of the namespaces currently in use by the namespace table
`reset-namespaces`	Resets the namespace table to its initial state
`#define`	Adds a macro to the macro table which can then be used in the Quantum Console. If the macro `name` is added, then all instances of `#name` will be expanded into the full macro expansion. This allows you to define shortcuts for various things such as long type names or commonly used command strings. Macros may not contain hashtags or whitespace in their name. **Note**: macros will not be expanded when using `#define`, this is so that defining nested macros is possible.
`remove-macro`	Removes the specified macro from the macro table
`clear-macros`	Clears the macro table
`all-macros`	Displays all of the macros currently stored in the macro table
`dump-macros`	Creates a file dump of macro table which can the be loaded to repopulate the table using `load-macros`
`load-macros`	Loads macros from an external file into the macro table

6.2 Extra Commands

Extra commands are implemented as in separate classes and files that are found in the `Extras` folder; these commands are added for convenience and are not required.

UtilCommands.cs

Name	Description
`get-object-info`	Finds the first object in the scene with the name `objectName` and displays its transform and component data
`destroy`	Destroys a `GameObject`
`instantiate`	Instantiates a `GameObject`
`teleport`	Teleports a `GameObject`
`teleport-relative`	Teleports a `GameObject` by a relative offset to its current position
`set-active`	Activates/deactivates a GameObject
`send-message`	Calls the method named `methodName` on every `MonoBehaviour` in the target `GameObject`
`add-component<T>`	Adds a component of type `T` to the specified `GameObject`
`rotate`	Rotates a `GameObject`

ScreenCommands.cs

Name	Description
`fullscreen`	Fullscreen state of the application
`screen-dpi`	DPI of the current device's screen
`screen-orientation`	The orientation of the screen
`current-resolution`	Current resolution of the application or window
`supported-resolutions`	All resolutions supported by this device in fullscreen mode
`set-resolution`	Sets the resolution of the current application, optionally setting the fullscreen state too

SceneCommands.cs

Name	Description
`load-scene`	Loads a scene by name into the game
`load-scene-index`	Loads a scene by index into the game
`unload-scene`	Unloads a scene by name
`unload-scene-index`	Unloads a scene by index
`all-scenes`	Gets the name and index of every scene included in the build
`loaded-scenes`	Gets the name and index of every scene currently loaded
`active-scene`	Gets the name of the active primary scene
`set-active-scene`	Sets the active scene to the scene with name `sceneName`

GraphicsCommands.cs

Name	Description
`max-fps`	The maximum FPS imposed on the application. Set to -1 for unlimited
`vsync`	Enables or disables vsync for the application

TypeCommands.cs

Name	Description
`enum-info`	Gets all of the numeric values and value names for the specified `enum` type

TimeCommands.cs

Name	Description
`time-scale`	Gets all of the numeric values and value names for the specified `enum` type

TypeCommands.cs

Name	Description
`enum-info`	The scale at which time is passing by

exec/DynamicCodeCommands.cs

Name	Description
`exec`	Compiles the given code to C# which will then be executed. Use with caution as no safety checks will be performed. Not supported in AOT (IL2CPP) builds. By default, boiler plate code will be inserted around the code you provide. This means various namespaces will be included, and the main class and main function entry point will provided. In this case, the code you provide should be code that would exist within the body of the main function, and thus cannot contain things such as class definition. If you disable boiler plate insertion, you can write whatever code you want, however you must provide a static entry point called Main in a static class called Program
`exec-extern`	Loads the code at the specified file and compiles it to C# which will then be executed. Use with caution as no safety checks will be performed. Not supported in AOT (IL2CPP) builds. By default, boiler plate code will NOT be inserted around the code you provide. Please see 'exec' for more information about boilerplate insertion