Code Script - User Manual

Introduces how to edit code scripts

Introduction to the concept

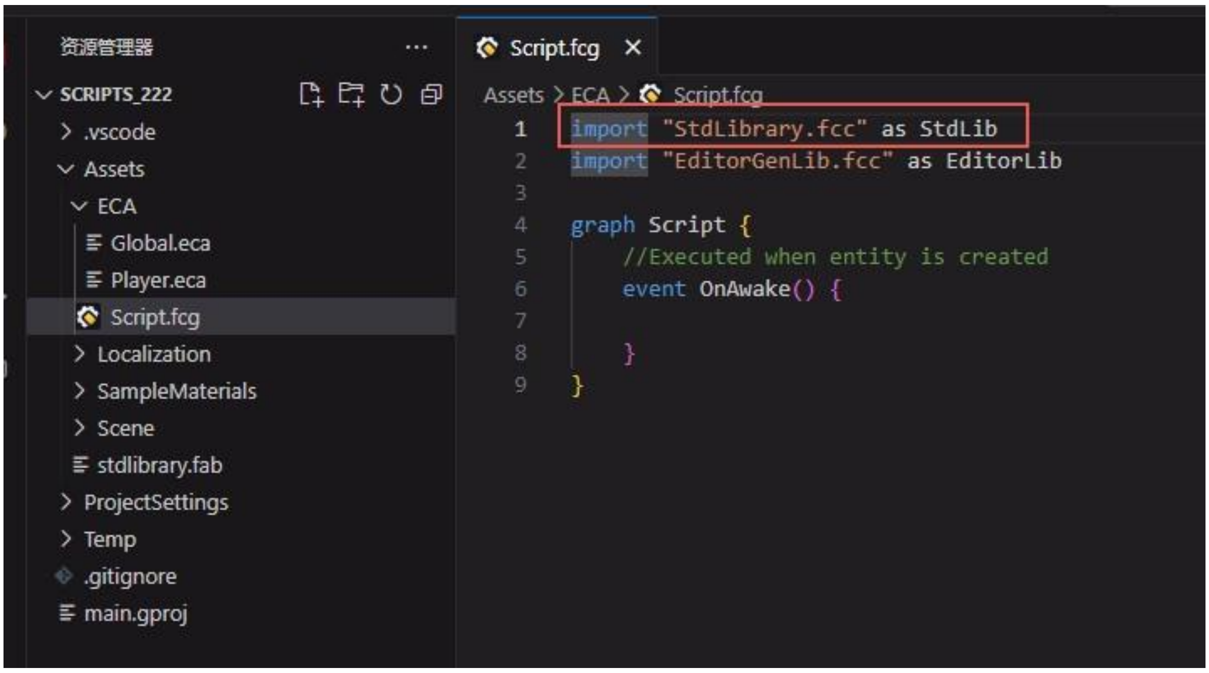
Server scripts and client scripts

Scripts on different running platforms can use different events and interfaces. The scope of the element can be confirmed by querying the library. Server scripts and client scripts cannot call each other directly, they can only notify the other end of the call by means of events.

Scripts with .fcg extension

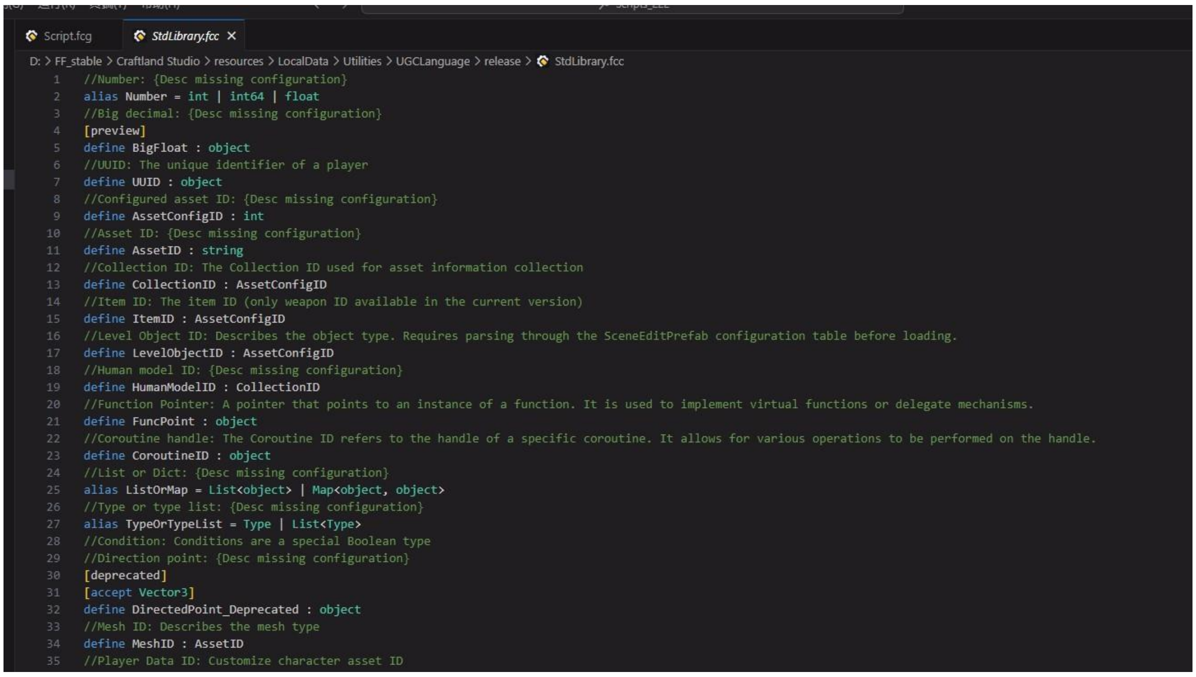
storehouse

The library has a .fcc suffix.
After creating a code script, the standard library is automatically added:



```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib

graph Script {
    //Executed when entity is created
    event OnAwake() {

    }
}
```

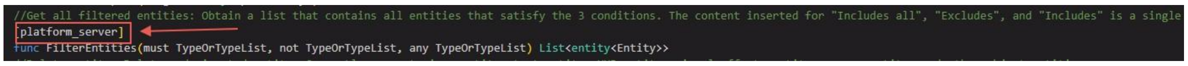
Standard library files:



Script.fcg StdLibrary.fcc

D: > FF_stable > Craftland Studio > resources > LocalData > Utilities > UGCLanguage > release > StdLibrary.fcc

```
//Number: {Desc missing configuration}
alias Number = int | int64 | float
//Big decimal: {Desc missing configuration}
[preview]
define BigFloat : object
//UUID: The unique identifier of a player
define UUID : object
//Configured asset ID: {Desc missing configuration}
define AssetConfigID : int
//Asset ID: {Desc missing configuration}
define AssetID : string
//Collection ID: The Collection ID used for asset information collection
define CollectionID : AssetConfigID
//Item ID: The item ID (only weapon ID available in the current version)
define ItemID : AssetConfigID
//Level Object ID: Describes the object type. Requires parsing through the SceneEditPrefab configuration table before loading.
define LevelObjectID : AssetConfigID
//Human model ID: {Desc missing configuration}
define HumanModelID : CollectionID
//Function Pointer: A pointer that points to an instance of a function. It is used to implement virtual functions or delegate mechanisms.
define FuncPoint : object
//Coroutine handle: The Coroutine ID refers to the handle of a specific coroutine. It allows for various operations to be performed on the handle.
define CoroutineID : object
//List or Dict: {Desc missing configuration}
alias ListOrMap = List<object> | Map<object, object>
//Type or type list: {Desc missing configuration}
alias TypeOrTypeList = Type | List<Type>
//Condition: Conditions are a special Boolean type
//Direction point: {Desc missing configuration}
[deprecated]
[accept Vector3]
define DirectedPoint_Deprecated : object
//Mesh ID: Describes the mesh type
define MeshID : AssetID
//Player Data ID: Customize character asset ID
```



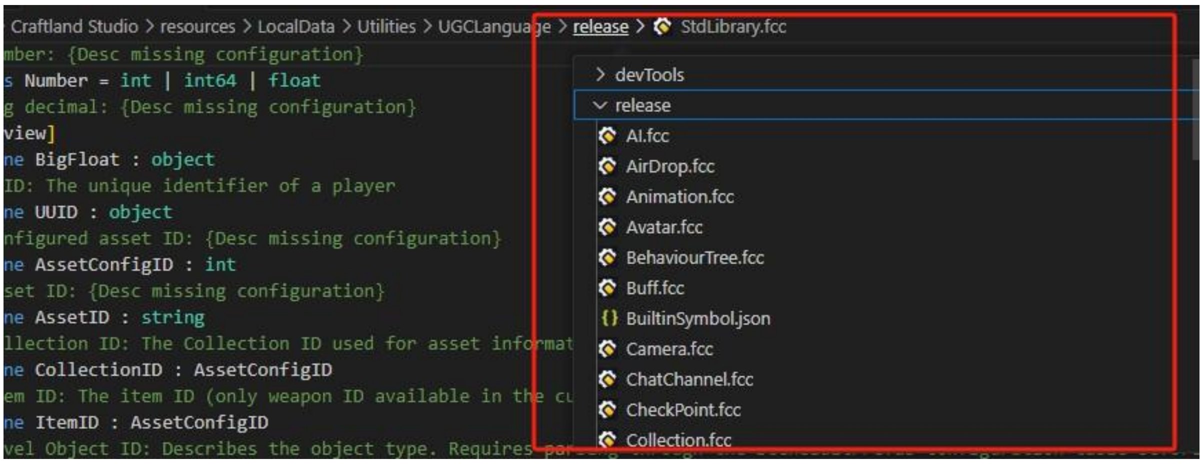
```
//Get all filtered entities: Obtain a list that contains all entities that satisfy the 3 conditions. The content inserted for "Includes all", "Excludes", and "Includes" is a single
[platform_server]
func FilterEntities(must TypeOrTypeList, not TypeOrTypeList, any TypeOrTypeList) List<entity<Entity>>
```

The [platform_server] tag indicates that this element applies to servers and only server scripts may use this element. The [platform_client] tag indicates that the platform for this element is client. Only client scripts may use this element.

Elements without tags can be used by both the server and the client.

[deprecated] indicates that the element has been deprecated, please do not use these elements.

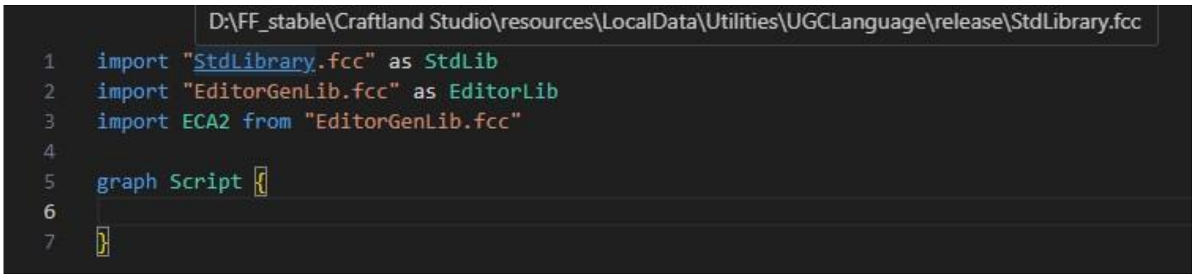
There are official library files in the editor path: \Craftland Studio\resources\LocalData\Utilities\UGCLanguage\release, you can refer to them as needed.



Craftland Studio > resources > LocalData > Utilities > UGCLanguage > release > StdLibrary.fcc

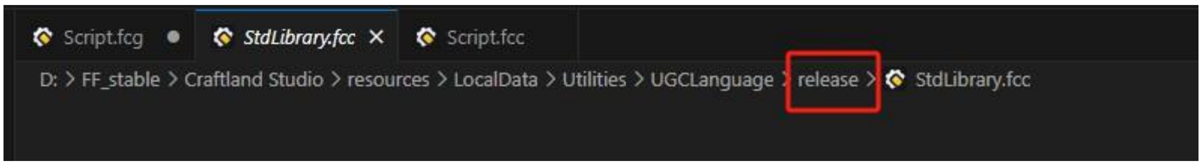
```
mber: {Desc missing configuration}
s Number = int | int64 | float
g decimal: {Desc missing configuration}
view]
ne BigFloat : object
ID: The unique identifier of a player
ne UUID : object
nfigured asset ID: {Desc missing configuration}
ne AssetConfigID : int
set ID: {Desc missing configuration}
ne AssetID : string
llection ID: The Collection ID used for asset informat
ne CollectionID : AssetConfigID
em ID: The item ID (only weapon ID available in the cu
ne ItemID : AssetConfigID
vel Object ID: Describes the object type. Requires
```

Open any code script that references a library, and hold down the Ctrl key and click on the library to jump directly to it.



D:\FF_stable\Craftland Studio\resources\LocalData\Utilities\UGCLanguage\release\StdLibrary.fcc

```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib
import ECA2 from "EditorGenLib.fcc"

graph Script {

}
```

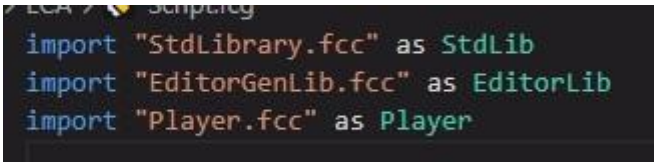

Script.fcg
StdLibrary.fcc
Script.fcc
D: > FF_stable > Craftland Studio > resources > LocalData > Utilities > UGCLanguage > release > StdLibrary.fcc

> release is the path where the official library files are stored.

Using a library requires a reference to it in the header of the file with the syntax:

import "library file" as alias

For official library files and new custom library files created within the editor, there is no need to add a path.



```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib
import "Player.fcc" as Player
```

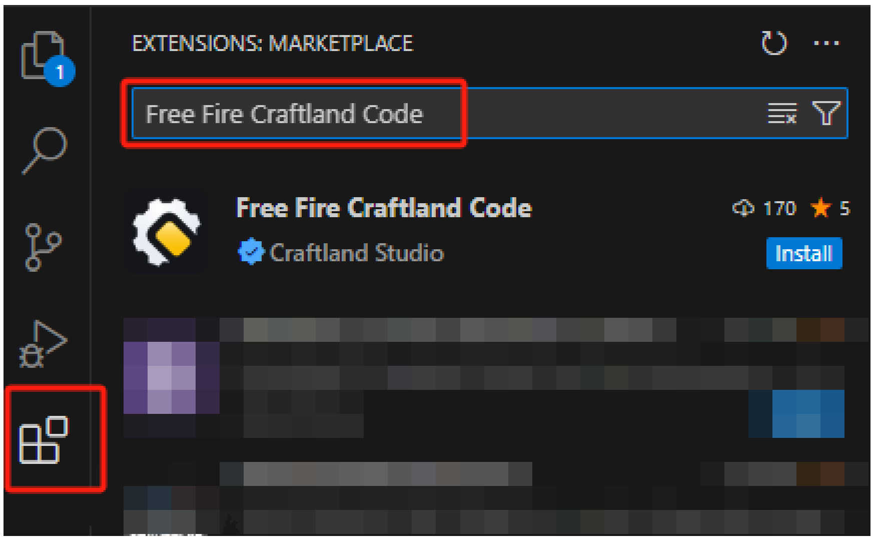
> EditorGenlib.fcc is the repository of registered assets, mainly used to reference library support for meta-scripts Player Custom Libraries, please see the Custom Libraries related section below.

Environmental preparation

VS Code plugin

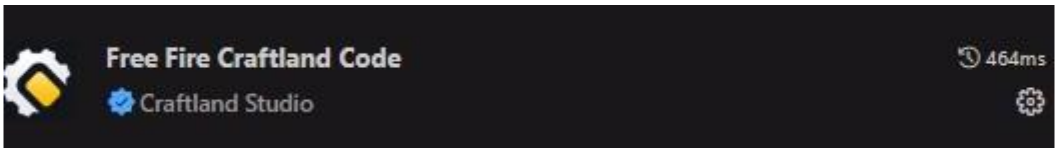
Installation of plug-ins

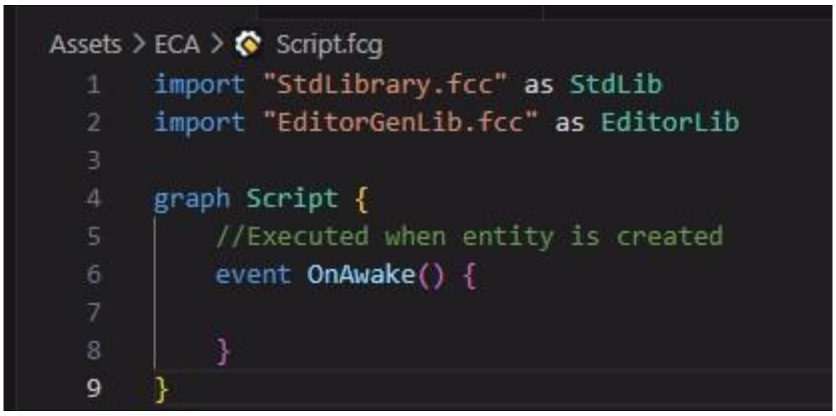
The plugin has been released to VS Code's plugin marketplace, search for Free Fire Craftland Code or some of the keywords at VS Code Plugins:



Click Install.

Once the installation is complete, you can find the plugin `FFUGCLanguage` plugin, open the code footer with VS Code! This file, see the corresponding syntax highlighting is in effect, it means that the plug-in installation is successful.





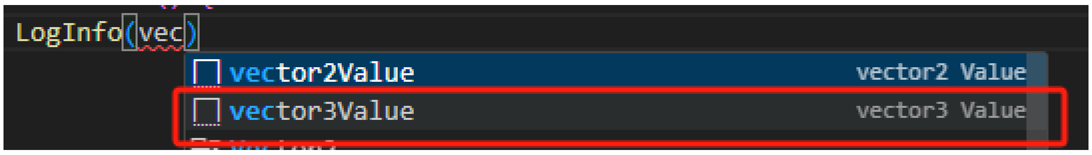
Assets > ECA > Script.fcg

```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib

graph Script {
    //Executed when entity is created
    event OnAwake() {

    }
}
```

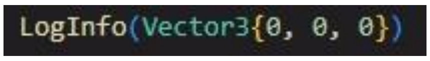
Important Features

Plugin details can be found in the description of the plugin in VS Code. Here are some important plugin features.

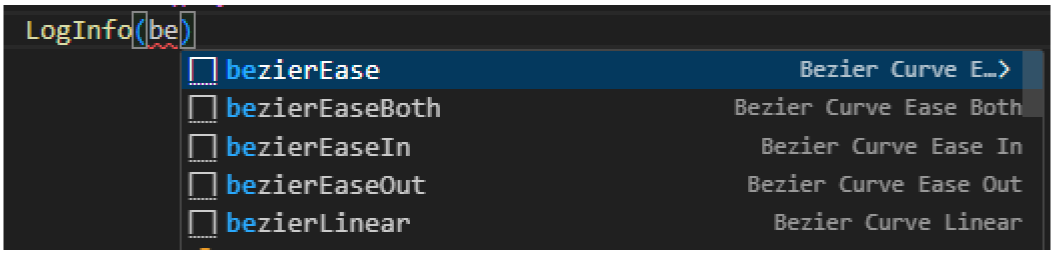
1. **Code Snippet Completion**: When inputting code snippets in fixed formats such as Bezier curves, vectors, etc., but more complex, you only need to input keywords to trigger code snippet completion.



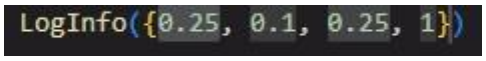
```
LogInfo(vec)
  vector2Value                                  vector2 Value
  vector3Value                                  vector3 Value
```



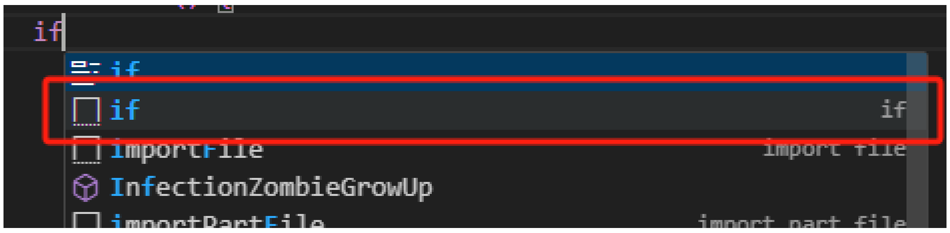
```
LogInfo(Vector3{0, 0, 0})
```



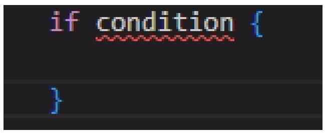
```
LogInfo(be)
  bezierEase                                  Bezier Curve E…>
  bezierEaseBoth                          Bezier Curve Ease Both
  bezierEaseIn                              Bezier Curve Ease In
  bezierEaseOut                            Bezier Curve Ease Out
  bezierLinear                              Bezier Curve Linear
```



```
LogInfo({0.25, 0.1, 0.25, 1})
```

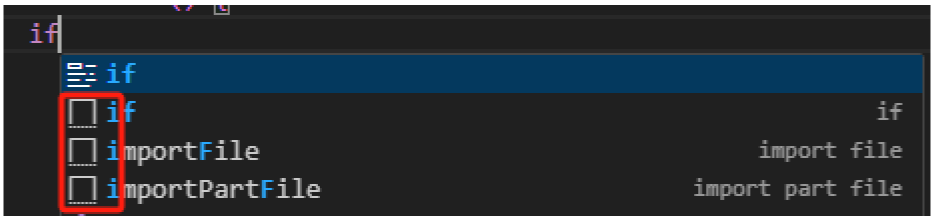


```
if
  if
  if                                                          if
  importFile                                         import file
  InfectionZombieGrowUp
  importPartFile                                import part file
```



```
if condition {

}
```

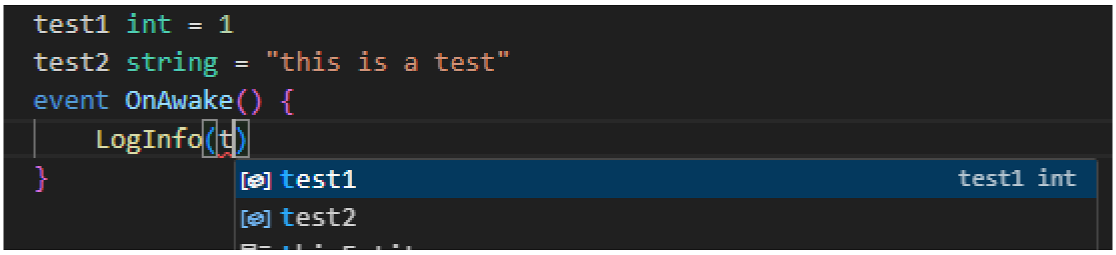
Auto-completion with this flag is code snippet completion in the plugin.



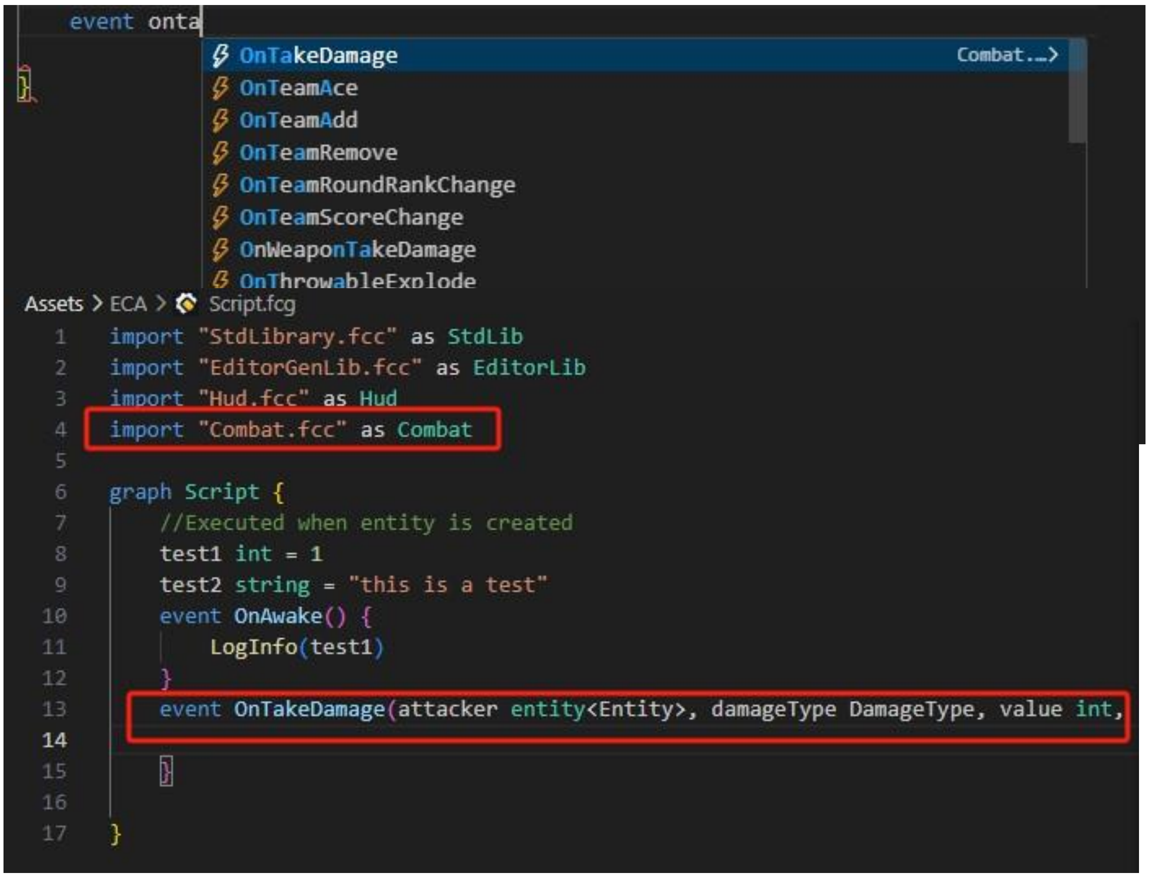
```
if
  if
  if                                                          if
  importFile                                         import file
  importPartFile                                import part file
```

> The first if in the figure is the VS comes with the keyword complement, the second to the fourth for the plug-in defined in the three different code snippets of the complementary

2. **Intelligent hints and completions**: auto-completion of variables that have appeared in the context, auto-completion of event and API names, prompting for formal parameters and return value information when calling a function, and automatically referencing libraries when using events or APIs that don't reference libraries.



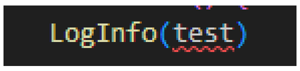
```
test1 int = 1
test2 string = "this is a test"
event OnAwake() {
    LogInfo(t)
}
```



```
event onta
```

```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib
import "Hud.fcc" as Hud
import "Combat.fcc" as Combat

graph Script {
    //Executed when entity is created
    test1 int = 1
    test2 string = "this is a test"
    event OnAwake() {
        LogInfo(test1)
    }
    event OnTakeDamage(attacker entity<Entity>, damageType DamageType, value int,

    }

}
```

3. **Diagnostics**: Code with syntax and

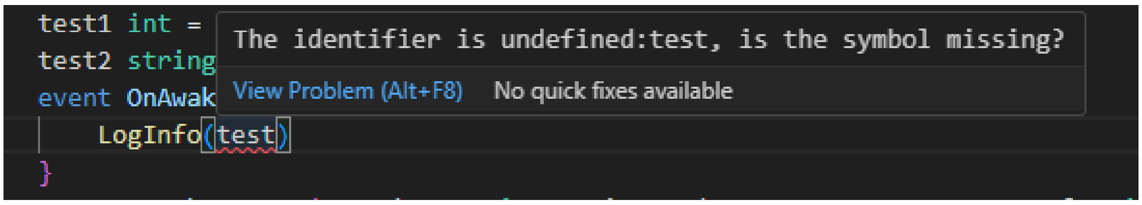


```
LogInfo(test)
```

semantic errors are marked red Hover

over the error to display an error

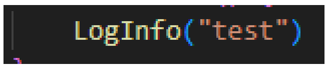
message:



```
test1 int =
test2 string
event OnAwak
    LogInfo(test)
}
```

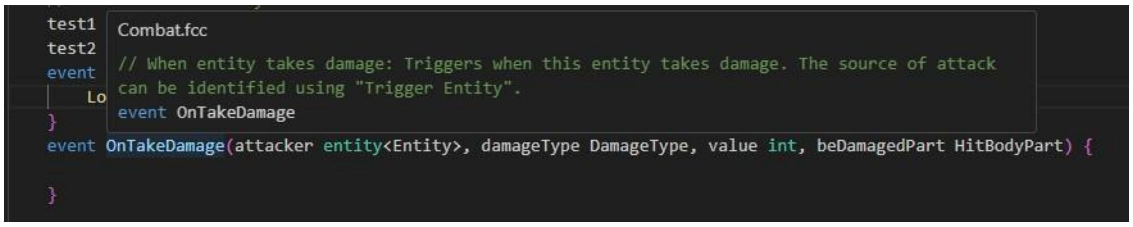
The identifier is undefined:test, is the symbol missing?

Should be:



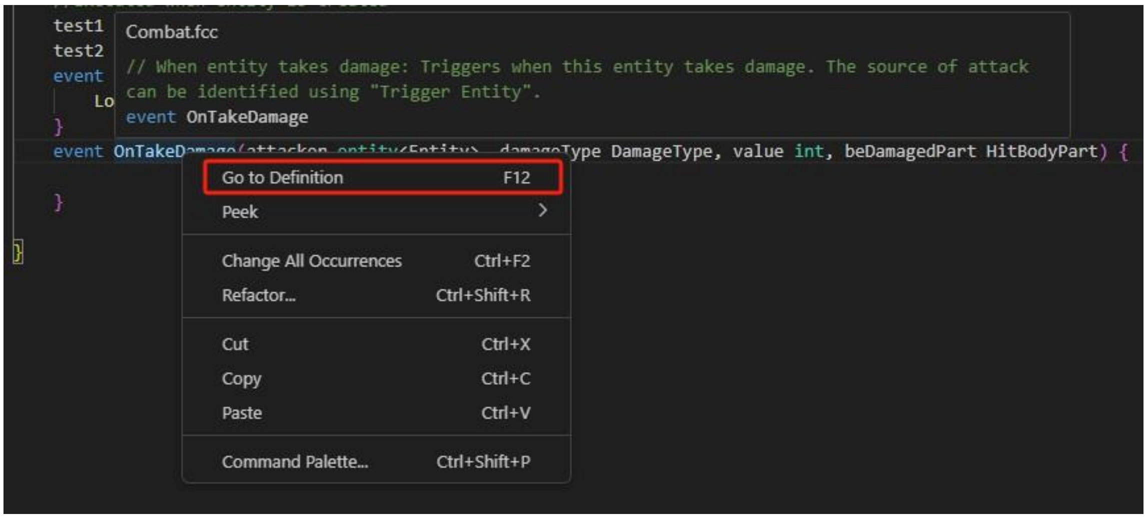
```
LogInfo("test")
```

4. **Hover Tips**: mouse over the code of interest to see the details

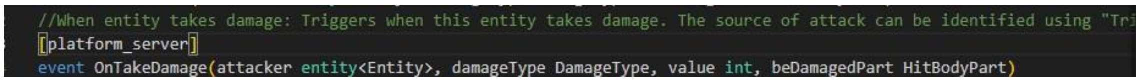


```
test1   Combat.fcc
test2   // When entity takes damage: Triggers when this entity takes damage. The source of attack
event   can be identified using "Trigger Entity".
   Lo   event OnTakeDamage
}
event OnTakeDamage(attacker entity<Entity>, damageType DamageType, value int, beDamagedPart HitBodyPart) {

}
```

5. **Jump to definition**: You can jump to the definition position by F12 or right clicking the corresponding code.



```
test1   Combat.fcc
test2   // When entity takes damage: Triggers when this entity takes damage. The source of attack
event   can be identified using "Trigger Entity".
   Lo   event OnTakeDamage
}
event OnTakeDamage(attacker entity<Entity>, damageType DamageType, value int, beDamagedPart HitBodyPart) {

}
```



```
//When entity takes damage: Triggers when this entity takes damage. The source of attack can be identified using "Tri
[platform_server]
event OnTakeDamage(attacker entity<Entity>, damageType DamageType, value int, beDamagedPart HitBodyPart)
```

Script structure

The following is an outline of the structure of a code script, a code script file that contains three sections:

- script member
  - Variable Definition
  - Function Definition
  - Event Definition

That means that a script's first-class member can only be one of these three categories.

```
graph script name {
    Script Variable Name Type = Initial Value

    func function name (list of formal parameters – optional) {
        //do sth...
    }

    event listener event name () {
        //do sth...
    }
}
```

We introduce the notion of a `code block`, which is the content of code wrapped in a set of curly braces `{...}` wrapped in a set of curly braces {...}.

As an example, it would be wrong to write an expression like call log printing directly in a first-class member of a script, but it could be written in a block of code for a function or event:

```
graph Demo {
    //LogInfo("Hello")  //this is the wrong place to write it

    func Hello() {
        LogInfo("Hello")  //  can be written in a block of code for a function or event
    }

    event OnAwake() {
        LogInfo("Hello")  //  can be written in a block of code for a function or event
    }
}
```

Editorial Related

basic grammar

The basic syntax of the code script can be found in detail in the corresponding section of the link below: Script Additional Instructions - User Manual.md

data type

The data types of the code scripts can be viewed in detail in the corresponding section of the link below: Scripts-Additional-Descriptions-User-Manual.md

operator (computing)

Operators are used to perform mathematical or logical operations at program runtime. The inbuilt operators are:

- arithmetic operator
- Relational operators
- Logical operators
- Assignment operators
- Other operators

The operators of the code script can be viewed in detail in the corresponding section of the link below: Script Additional Instructions - User Manual.md

process control

The default execution order of the lines of code is top-down order, in order to achieve more complex flow control, we need to introduce flow control statements.

The flow control of the code script can be viewed in detail in the corresponding section of the link below: Script Additional Instructions - User Manual.md

variant

Variables need to be declared in the code before they can be used, i.e. created. There are several types of variables:

local variable

The general form of declaring local variables is to use the `var` keyword, whose type can be automatically inferred by the compiler in the presence of a right-hand assignment statement; otherwise, they should be declared voluntarily.

Local variables are scoped from the declaration location to the end of the statement block in which they are located.

```
func Demo() int {
    // Complete statement
    //var value int = 10
    // Omit type declarations
    //var value = 10
    // Omit the assignment
    statement on the right
    hand side var value
    int
    value = 10 //ok
    if value == 10 {
        var localVar = 20
        value = localVar //ok
    }
    value = localVar //not ok
    return value
```

script variable

The scope of a script variable is the current script, and script variables can also be accessed in other scripts through entity instances after they have been defined.

```
graph HelloWorldGraph {
    SayTipWords string = "Hello, " //script variable

    func SayHello(name string)
        { std.PrintString(SayTipWords +
        name)
    }

    event OnAwake() {
        start SayHello("FF_UGC")
    }
```

Component Properties

Component properties are scoped to the current component, and component properties are all publicly available data that can be accessed in any script through an entity instance.

```
graph EntityDataStore {
    func EntityPropModify(input int) int
        { thisEntity.EcoKillMoney = input
        return thisEntity.EcoKillMoney
    }

    func EntityPropModifyV2(input int) int
        { thisEntity<std.Global>.EcoKillMoney = input
        return thisEntity<std.Global>.EcoKillMoney
    }
}
```

Value types and reference types

The basic types bool, int, float, string and Vector2, Vector3, Quaternion are all value types, which cannot be nil and must have a value. Value types make a value copy when they are assigned a value.

List, Map, and entity are all reference types, and their values may be nil. a reference type only changes the address it points to when it is assigned a value, and a change in the contents of its address changes the values of all the variables that point to it.

function (math.)

A function is a block of statements that organises a number of related statements together to perform a task. To use a function, you need:

- Define function
- call function

Defining functions

When defining a function, you are actually declaring the associated elements of the function, the basic structure of which is as follows:

```
func function name (formal parameter argument name formal parameter type, ...) Return value type {

}
```

Here is a simple example:

```
func TryAddExp(player entity<Player>, exp int) bool {
    //...
    return false
}
```

call function

You can call it using the function name, here is a simple example:

```
event OnAwake() {
    //...
    // Call the user function of the current script.
    var isSuc = TryAddExp(curPlayer, 15)
    // Standard library calls
    LogInfo(Format("AddExp %v:%v", List<object>{curPlayer, isSuc}))
}
```

Synchronous and Asynchronous Functions

A synchronous function is one that executes and returns immediately, with no asynchronous processes in between;

An asynchronous function is a function implementation that uses an API that performs asynchronous waiting, such as `WaitForSeconds`. Syntactically, asynchronous functions must be declared with the `async` keyword:

```
async func
    ShowGameTime(){ while(thisEntity<Global>.GameTimeMs
    < 10000){
        LogInfo("GameTime:" + thisEntity<Global>.GameTimeMs)
        WaitForSeconds(1000)
    }
```

Calling Asynchronous Functions

When calling an asynchronous function, we can

choose how it is executed:● start

- ○ Call the asynchronous function without blocking and continue with the next line
- wait
  - ○ Call an asynchronous function, block until the asynchronous function finishes before executing the next line

In the default state, the default is the WAIT method, which will wait for the asynchronous function to execute.

```
event OnAwake()
    { LogInfo("1")
    start TryAddExp(curPlayer,
    15) LogInfo("2")
    wait TryAddExp(curPlayer,
    15) LogInfo("3")
}
```

Contagious asynchronous functions

If you call an asynchronous function from within a custom function, and you choose to call it as a `wait`, then

`async`:
the custom function must also be declared as a

```
async func
    ShowGameTime(){ while(thisEntity<Global>.GameTimeMs
    < 10000){
        LogInfo("GameTime:" + thisEntity<Global>.GameTimeMs)
        //async's API
        WaitForSeconds(1000)
    }
}


async func Start() {
    //wait up an async
    function wait
    ShowGameTime()
    LogInfo("Show End!")
```

output parameter

The return statement can be used to return only one value from a function. However, you can use out to return multiple values from a function, and the output parameter assigns the function's output to itself.

The basic form is as follows:

```
out var form parameter name
```

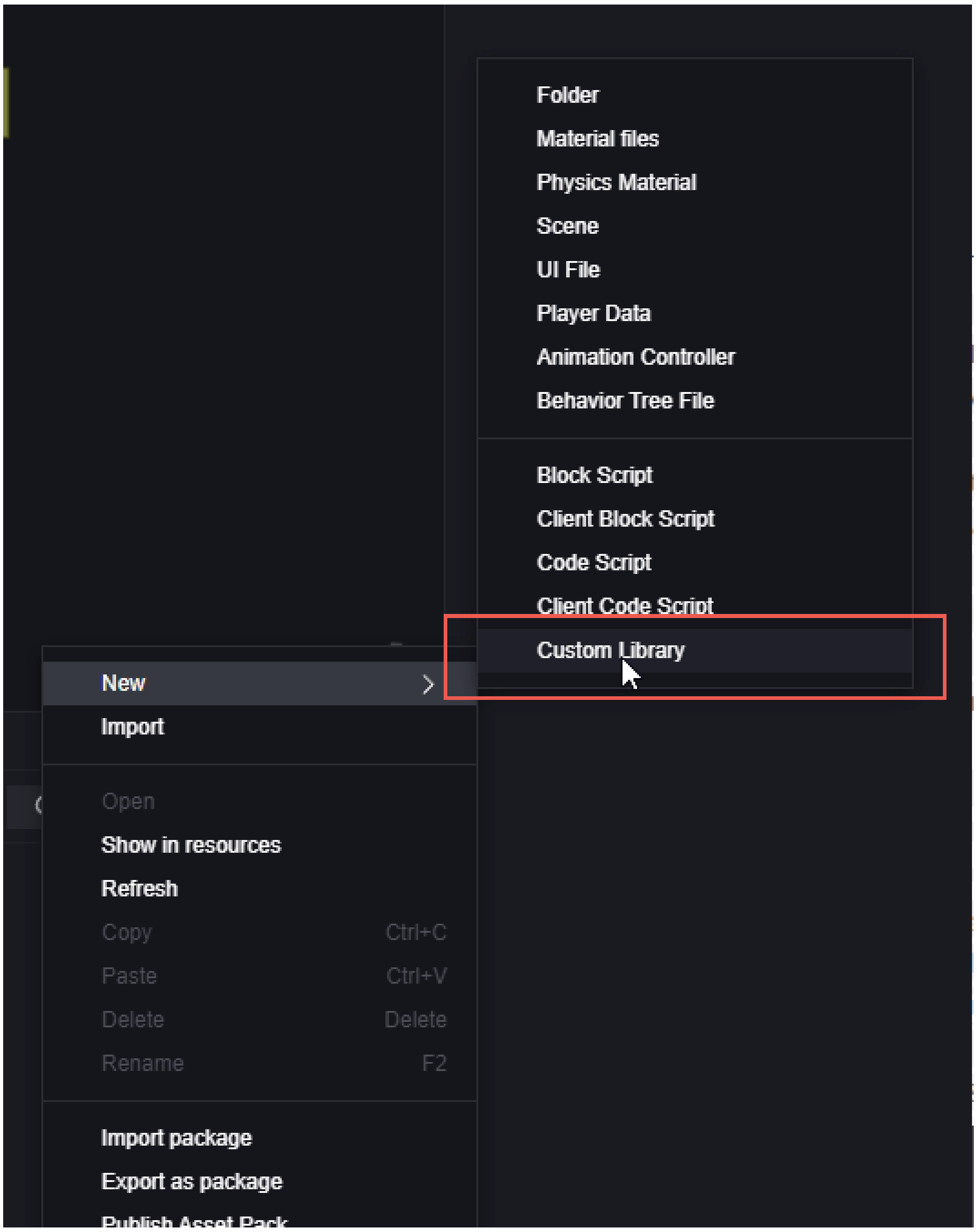
Here is an example:

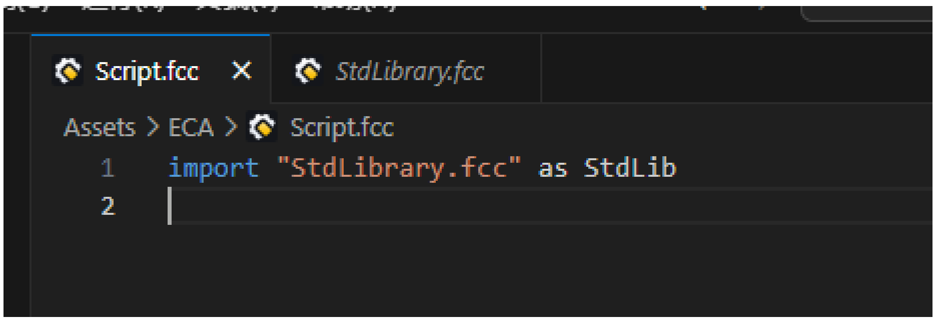
```
import "StdLibrary.fcc" as
std import "List.fcc" as list
import "EconomyModule.fcc" as economy

graph APIOutParam {
    func CreateWallet() int {
        NewWalletForEntity(nil, thisEntity, out var walletEntity)
        IncreaseMoneyForWallet(walletEntity, MoneyType.Basic, 1)
        return Length( walletEntity<Wallet>.Content)
    }
}
```

An output parameter must be explicitly declared to be an output parameter of out both when it is defined and when it is called, which is essentially defining a local variable at the function call, and assigning a value to this local variable defined outside the domain inside the function.

Custom Libraries

Support user to create custom library file, create library file by creating a new library file when you create a new script at the asset.



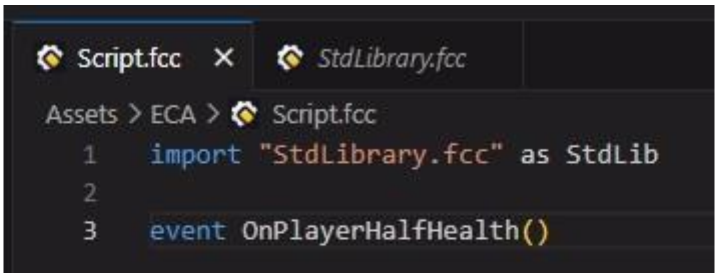


```
import "StdLibrary.fcc" as StdLib
```

Custom events, components, types, enumerations are supported within the custom library.

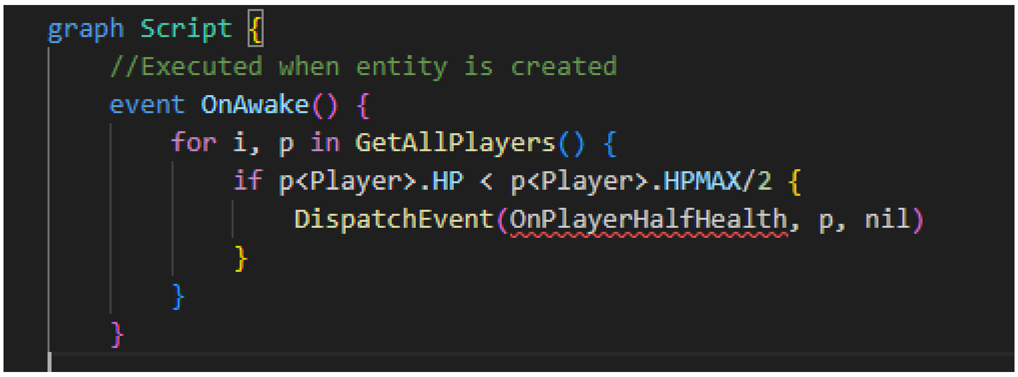
Custom Events

Register the required events in the library:



```
import "StdLibrary.fcc" as StdLib

event OnPlayerHalfHealth()
```

Event logic needs to be written in code scripts, and code scripts with event logic written in them need to be guaranteed to run when used.

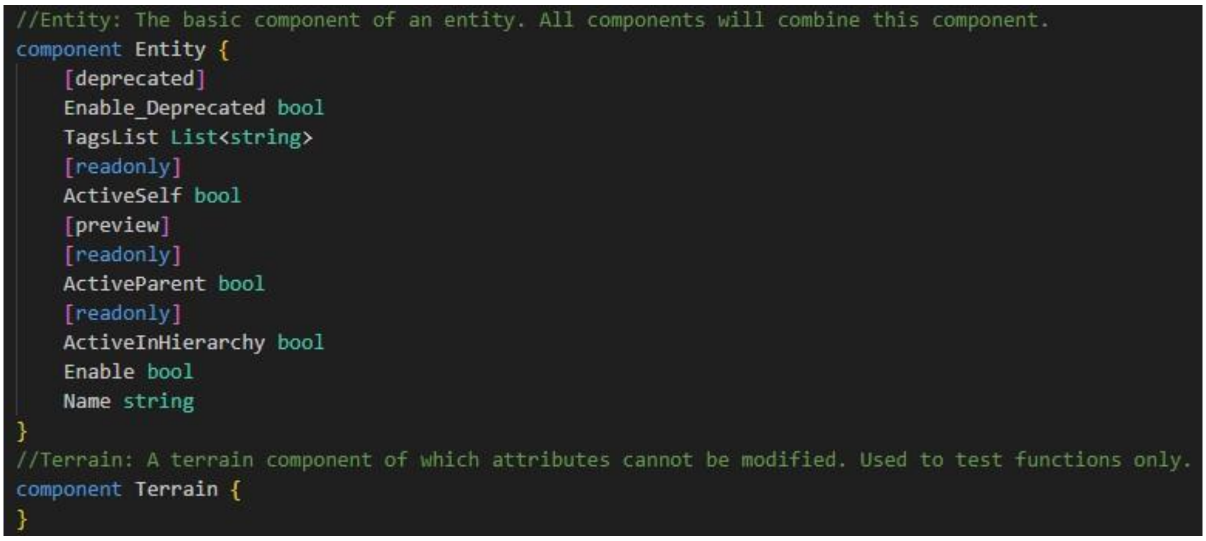


```
graph Script {
    //Executed when entity is created
    event OnAwake() {
        for i, p in GetAllPlayers() {
            if p<Player>.HP < p<Player>.HPMAX/2 {
                DispatchEvent(OnPlayerHalfHealth, p, nil)
            }
        }
    }
```

> DispatchEvent is an interface for custom events. When an event is activated, you can send a custom event signal to the library, and the event in the custom library will receive the signal to trigger the event logic.

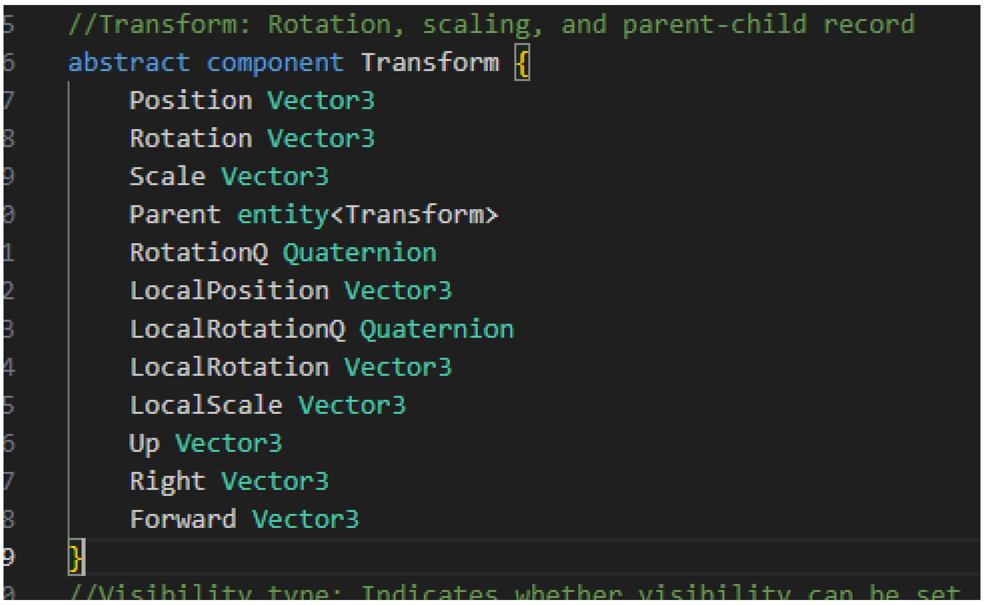
Custom Components

Refer to the standard library for definitions of components:



```
//Entity: The basic component of an entity. All components will combine this component.
component Entity {
    [deprecated]
    Enable_Deprecated bool
    TagsList List<string>
    [readonly]
    ActiveSelf bool
    [preview]
    [readonly]
    ActiveParent bool
    [readonly]
    ActiveInHierarchy bool
    Enable bool
    Name string
}
//Terrain: A terrain component of which attributes cannot be modified. Used to test functions only.
component Terrain {
}
```

Abstract components:

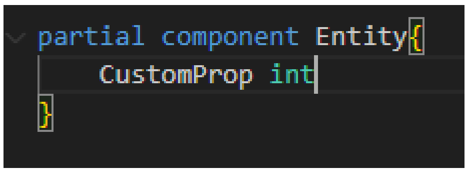


```
//Transform: Rotation, scaling, and parent-child record
abstract component Transform {
    Position Vector3
    Rotation Vector3
    Scale Vector3
    Parent entity<Transform>
    RotationQ Quaternion
    LocalPosition Vector3
    LocalRotationQ Quaternion
    LocalRotation Vector3
    LocalScale Vector3
    Up Vector3
    Right Vector3
    Forward Vector3
}
//Visibility type: Indicates whether visibility can be set
```

When defining a component, you can also define the properties in it.

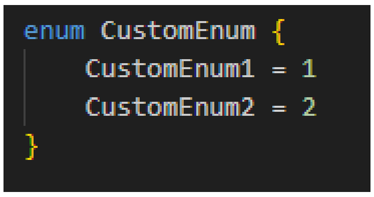
Extending Official Component Properties

Use the partial keyword to extend the properties of an official component:



```
partial component Entity{
    CustomProp int
}
```

Custom Enumerations



```
enum CustomEnum {
    CustomEnum1 = 1
    CustomEnum2 = 2
}
```

Importing Scripts

You can import library files and code scripts.

Library files contain type definitions, component definitions, event definitions, API definitions, script declarations, and so on. Script file contains script property definitions, event handling functions, custom functions and so on.

The import method is as follows:

```
import "file path" as alias
```

The following are examples of importing standard and custom library files:

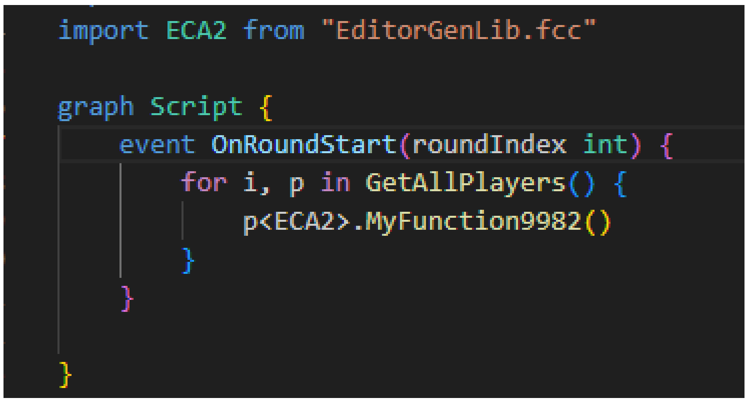
```
import "StdLibrary.fcc" as std
import ". /MyEditorGenLib.fcc" as
```

- When the file path is a filename, it is looked up from the standard library included in the compiler When the file path is a file path, it is
- looked up according to the file path

Importing Meta Scripts

The graph element scripts are stored in the EditorGenlib library as one graph per file. To reference the graph meta-scripts, you need to be careful:

1. When the meta-script is a non-static script, you need to specify the entity on which the script is mounted at the time of use.



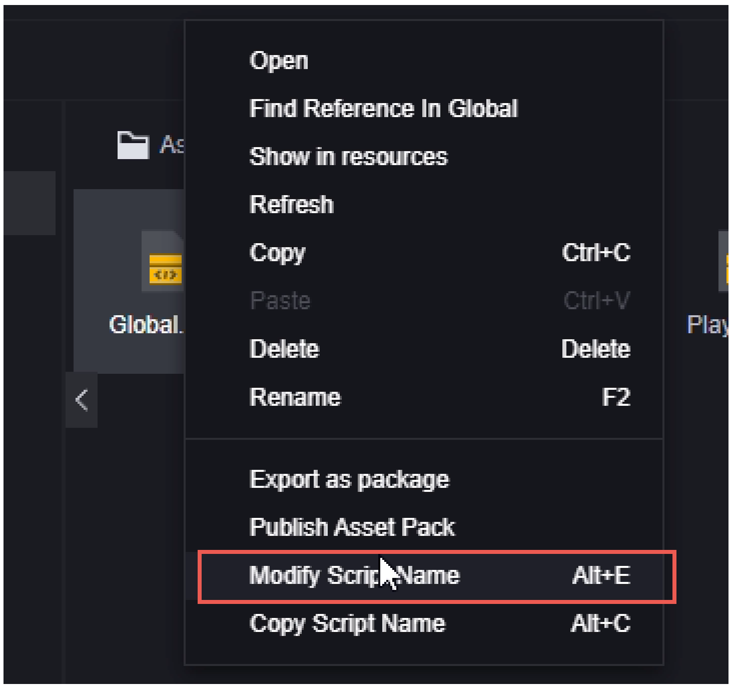
```
import ECA2 from "EditorGenLib.fcc"

graph Script {
    event OnRoundStart(roundIndex int) {
        for i, p in GetAllPlayers() {
            p<ECA2>.MyFunction9982()
        }
    }

}
```

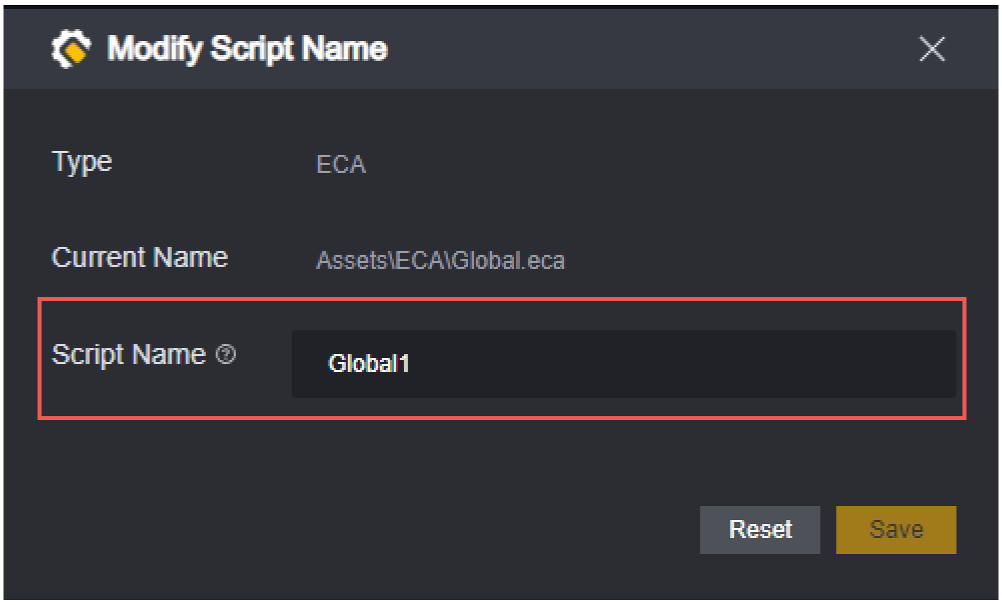
> ECA2 is a tuple script mounted on the player entity with member MyFunction9982.

2. The tuple script name cannot be a reserved keyword.

When a meta-script is named as a keyword, the script must be changed by the following method before it can be referenced successfully:

Right-click on the meta-script, select Modify Script Name, and make the change. This action does not affect the displayed filename, but simply modifies the name of the script used for the code.





Type ECA

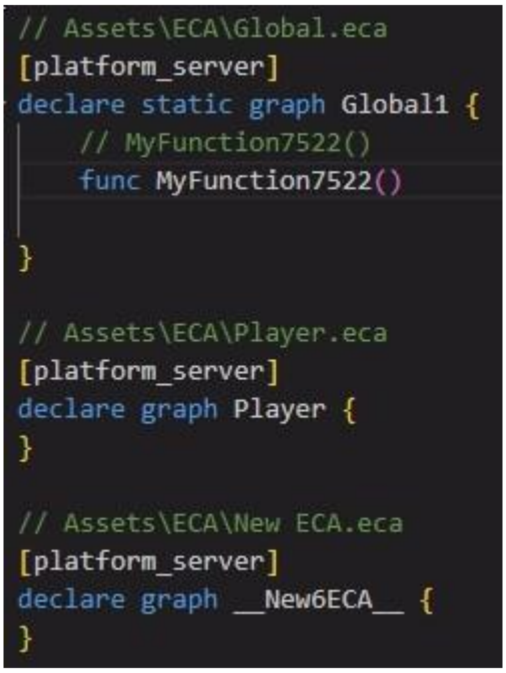
Current Name Assets\ECA\Global.eca

Script Name Global1

The keyword cannot be used as a user-defined identifier and is not limited to custom tuple script names. The following statement can be used for a tuple script that meets the conditions:

```
import script name from "EditorGenLib.fcc"

//or

import Script Name as Script Alias from "EditorGenLib.fcc"
```

The script name and path information for a script can be viewed by accessing the EditorGenLib.fcc library file



```
// Assets\ECA\Global.eca
[platform_server]
declare static graph Global1 {
    // MyFunction7522()
    func MyFunction7522()

}

// Assets\ECA\Player.eca
[platform_server]
declare graph Player {
}

// Assets\ECA\New ECA.eca
[platform_server]
declare graph __New6ECA__ {
}
```

Below is an example of the import:

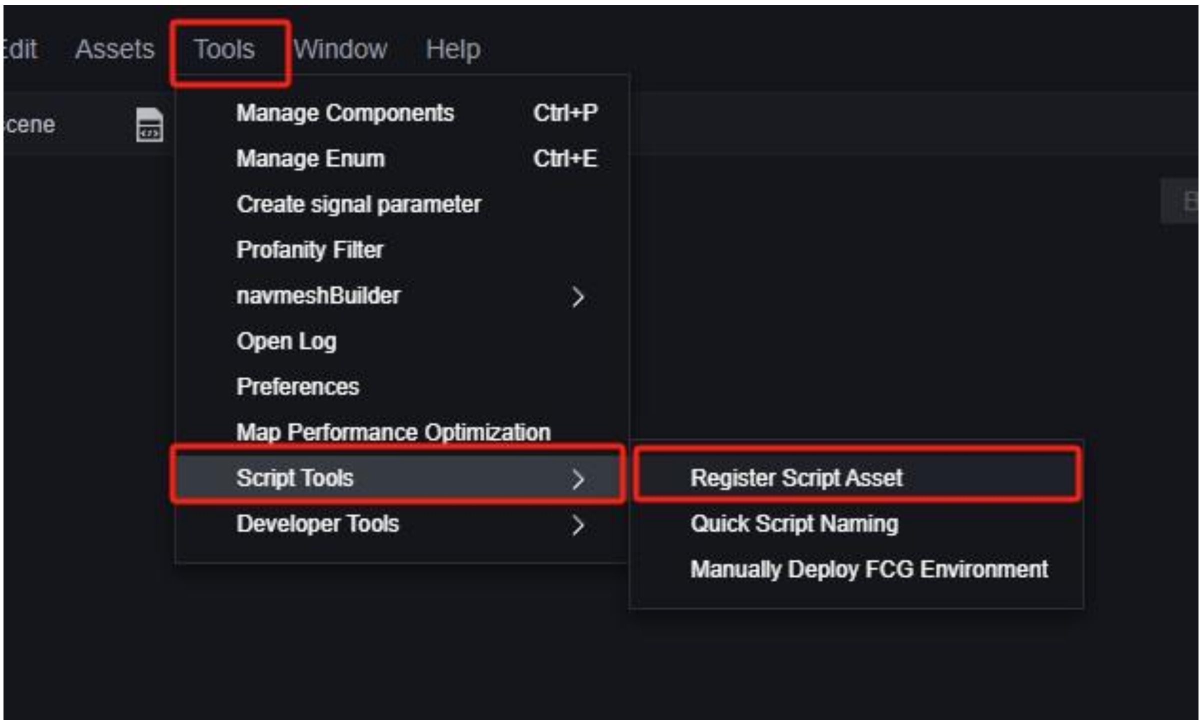
```
import Global1 as GLB from "EditorGenLib.fcc"
```

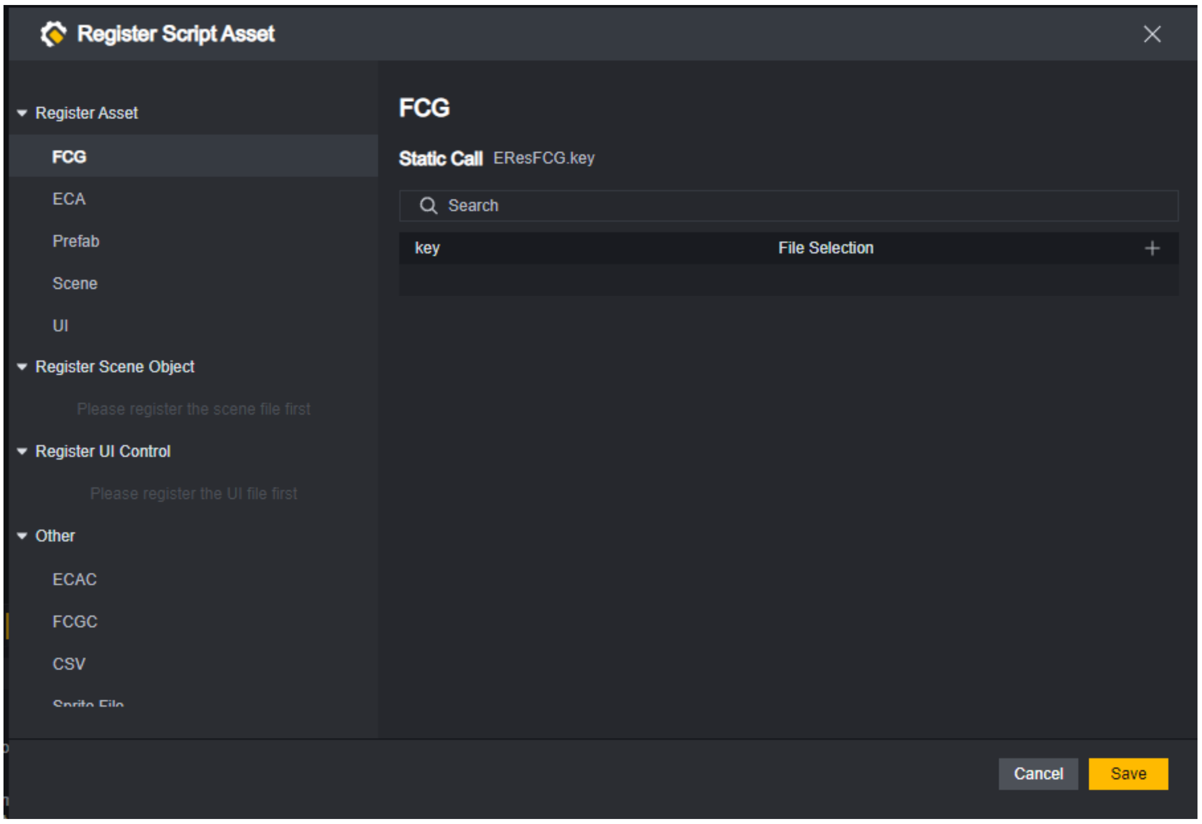
Access to assets

Although scripts can use some constants and literals to specify asset IDs, this approach is less maintainable and prone to problems. Therefore, accessing assets in scripts requires the use of asset registration.

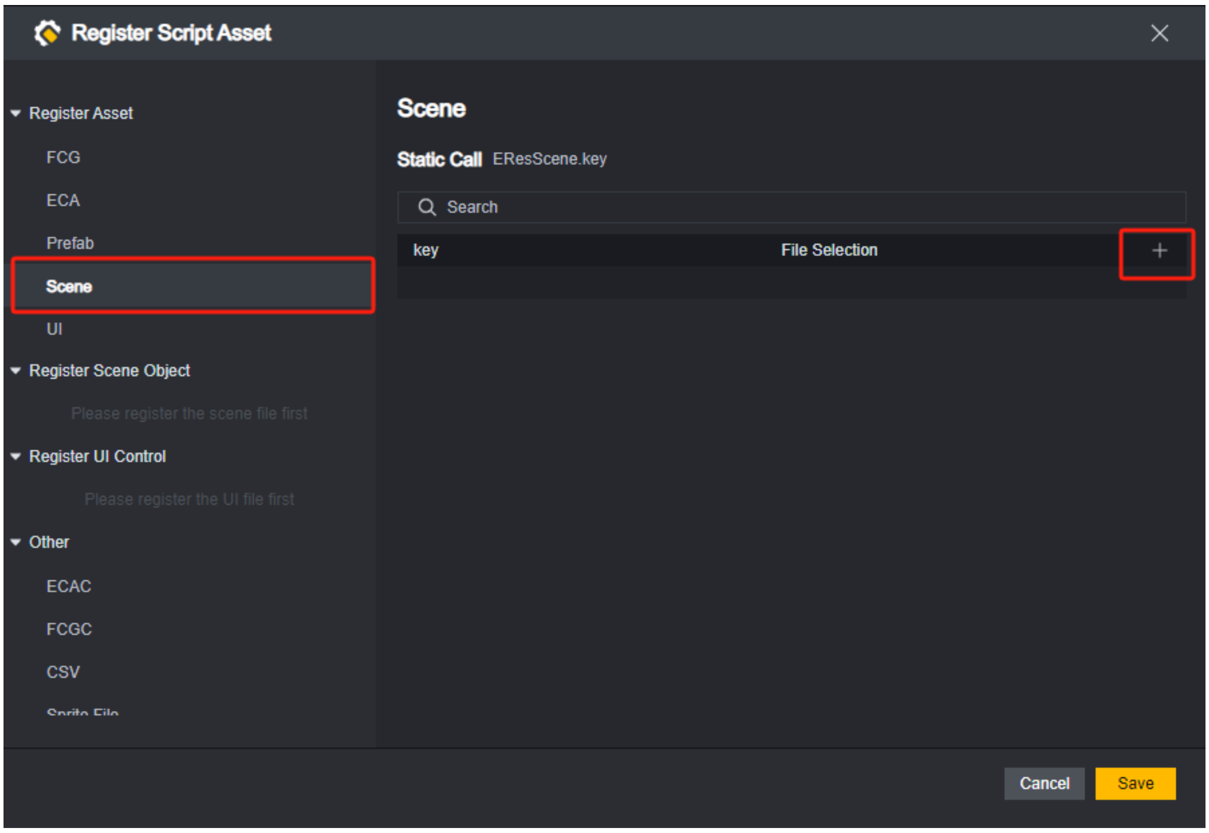
Asset registration

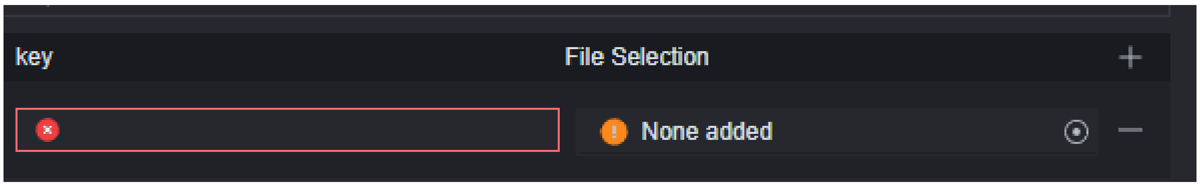
Click Tools - Script Tools - Script Asset Registration to open the Asset Registration menu in the FE Editor:



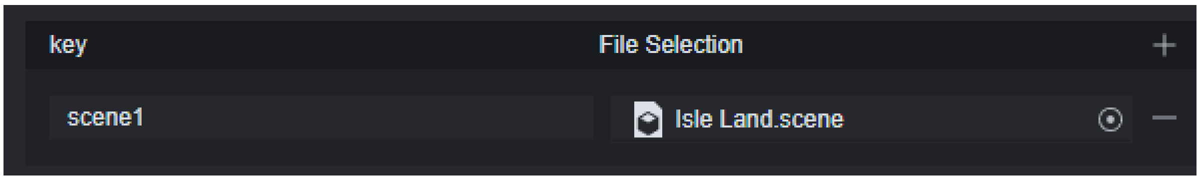


On the left hand side select the asset class you want to register and then select the plus sign to add a registered asset:

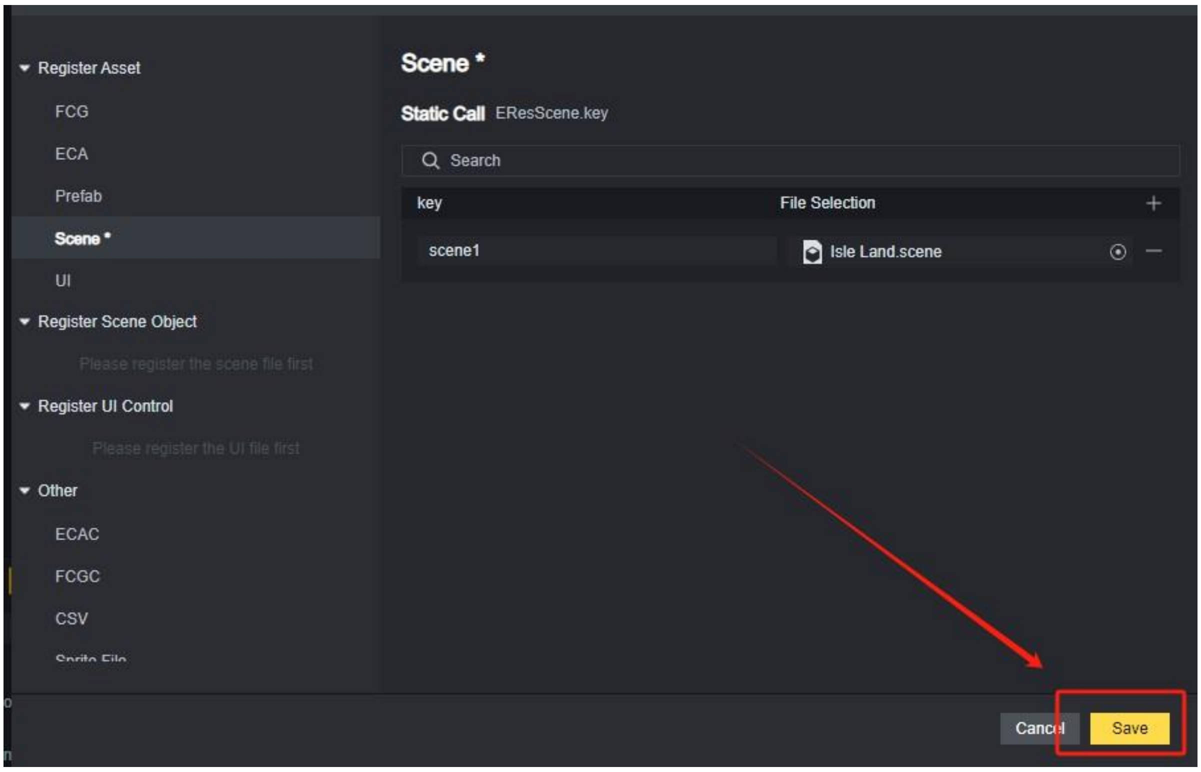




Enter a legal key name and select the corresponding asset.



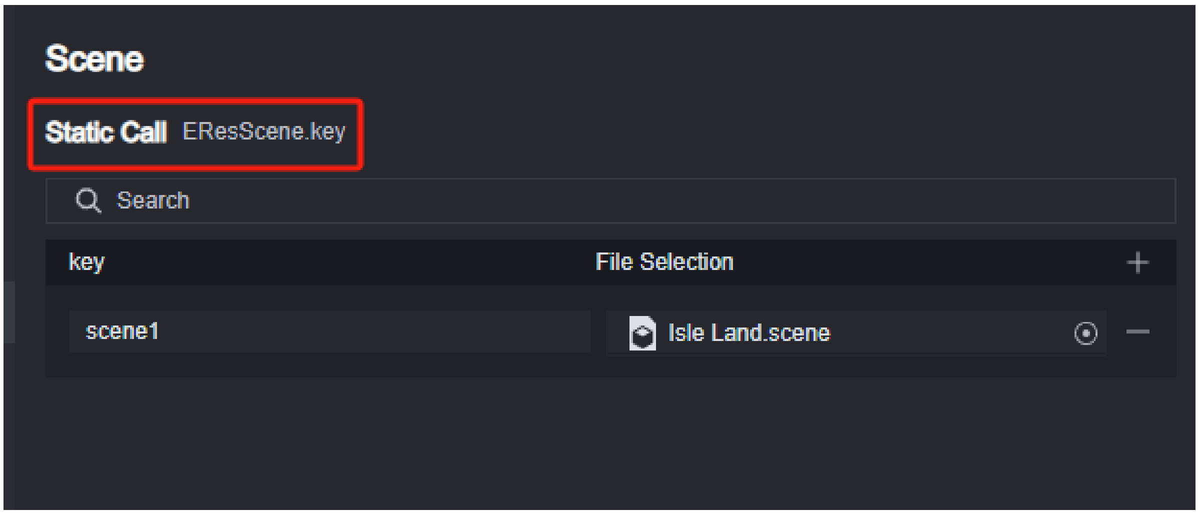
Click the Save button below to complete the asset registration.

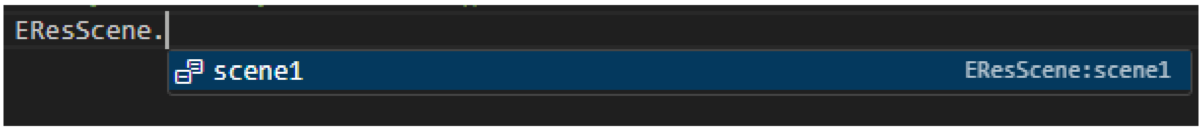


> For scenes and UI, special attention needs to be paid
> There are objects in the scene and widgets in the UI, and you can register both the scene and the UI, as well as the objects and widgets in them, but the registration is sequential: make sure that the scene and the UI file are registered before you register the objects and widgets in them.

static call

Call the corresponding asset in the script using the static call syntax in the FE Editor menu:





```
EResScene.
    scene1                                          EResScene:scene1
```

The statically called assets can be viewed at Enumeration in the EditorGenLib.fcc library

image-20240723182251643

dynamic call

Assets accessed based on enumeration, while convenient, lose flexibility: consider the scenario if the mapping of an item in the game is determined by configuring the csv table. This is when assets need to be accessed dynamically.

In EditorGenLib.fcc, a special Res graph is generated. its member variables are the maps that collect each asset. image-20240723182513253

The value of the map is the corresponding asset ID, so it also generates an enumeration EResKeyXXX based on the asset type, and fills the enumeration with registered assets by auto-completion:

image-20240723182603330

Based on the map access approach, the need for dynamic access to assets can be realised. For example, an asset key read from a csv can be used directly as an index to a map.

```
var CSVKey string
// ...
// Get the key by reading the csv table.
var MySceneID = Res.Scene[CSVKey]
```

compiling

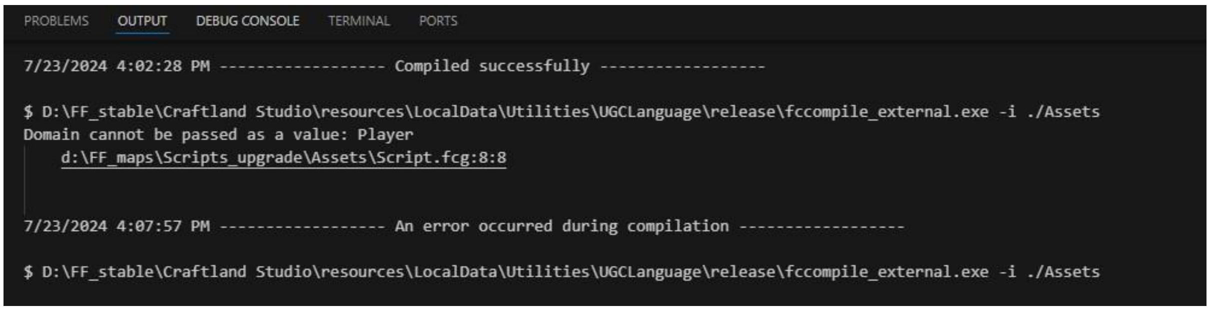
Starting DEBUG or saving the project will compile it automatically.

In VS Code, use the shortcut Alt+B to manually compile the current file.

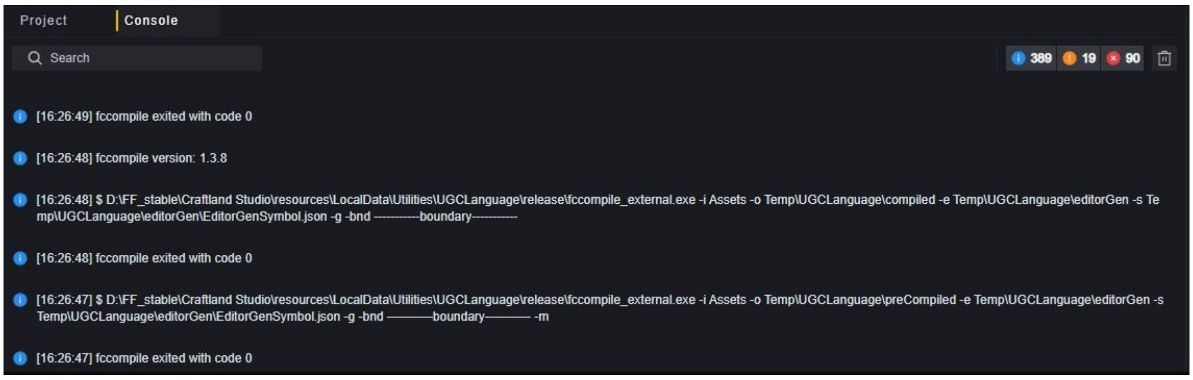
> This is what the Free Fire Craftland Code plugin does

In the FE editor, use the shortcut Alt+B to compile manually.

Compilation errors and messages can be viewed in the console of the corresponding software.



```
7/23/2024 4:02:28 PM ------------------ Compiled successfully ------------------

$ D:\FF_stable\Craftland Studio\resources\LocalData\Utilities\UGCLanguage\release\fccompile_external.exe -i ./Assets
Domain cannot be passed as a value: Player
    d:\FF_maps\Scripts_upgrade\Assets\Script.fcg:8:8


7/23/2024 4:07:57 PM ------------------ An error occurred during compilation ------------------

$ D:\FF_stable\Craftland Studio\resources\LocalData\Utilities\UGCLanguage\release\fccompile_external.exe -i ./Assets
```



```
[16:26:49] fccompile exited with code 0

[16:26:48] fccompile version: 1.3.8

[16:26:48] $ D:\FF_stable\Craftland Studio\resources\LocalData\Utilities\UGCLanguage\release\fccompile_external.exe -i Assets -o Temp\UGCLanguage\compiled -e Temp\UGCLanguage\editorGen -s Te
mp\UGCLanguage\editorGen\EditorGenSymbol.json -g -bnd -----------boundary-----------

[16:26:48] fccompile exited with code 0

[16:26:47] $ D:\FF_stable\Craftland Studio\resources\LocalData\Utilities\UGCLanguage\release\fccompile_external.exe -i Assets -o Temp\UGCLanguage\preCompiled -e Temp\UGCLanguage\editorGen -s
Temp\UGCLanguage\editorGen\EditorGenSymbol.json -g -bnd -----------boundary----------- -m

[16:26:47] fccompile exited with code 0
```

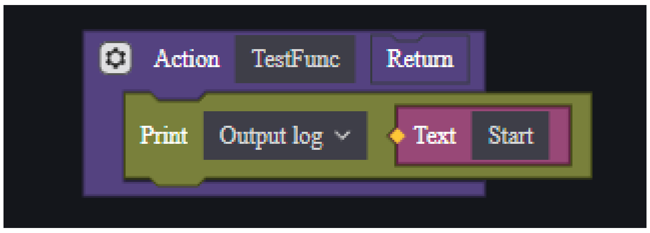
typical example

Demonstrates how to enable code scripts with a simple example:

Customise a function in a meta script to print the log "start". In the custom code script, run this custom function every time the player fires, and after printing "start", deduct 25 current life points from the player's life value if the player's life value is greater than 25 points or more, and deduct 1 point from the player's life value if the player's life value is less than or equal to 25 points.

> This is a pointless requirement setting, used only to demonstrate the use of code scripts

First implement the custom function in the graphical metacode. The function is called TestFunc. the script is called CustomFunc.

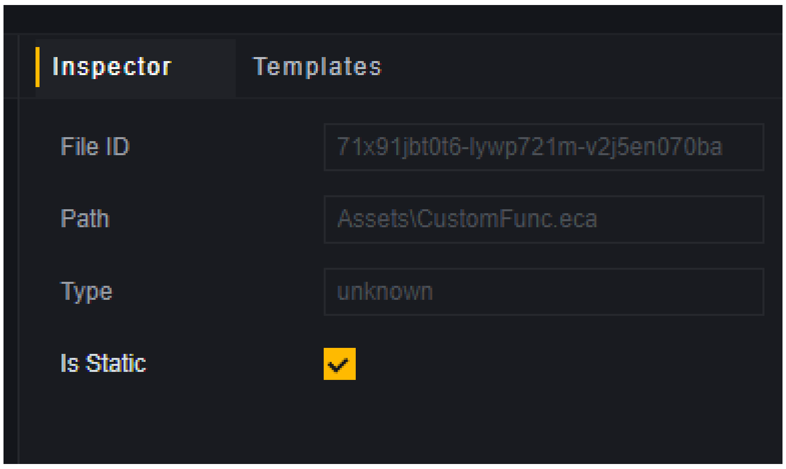


Action TestFunc Return

Print Output log Text Start

> Since the script file is named CustomFunc and is not a reserved keyword, there is no need to make additional changes to the script name.

Set this meta-script file as a static script for subsequent references. If not set as a static script, the script needs to be mounted on an entity that is active within the game and used when referenced.



Inspector Templates

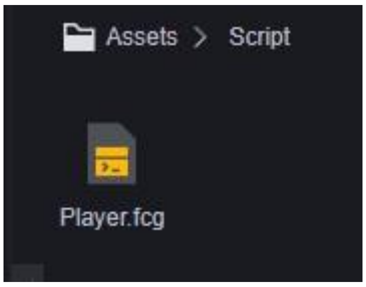
File ID 71x91jbt0t6-lywp721m-v2j5en070ba

Path Assets\CustomFunc.eca

Type unknown

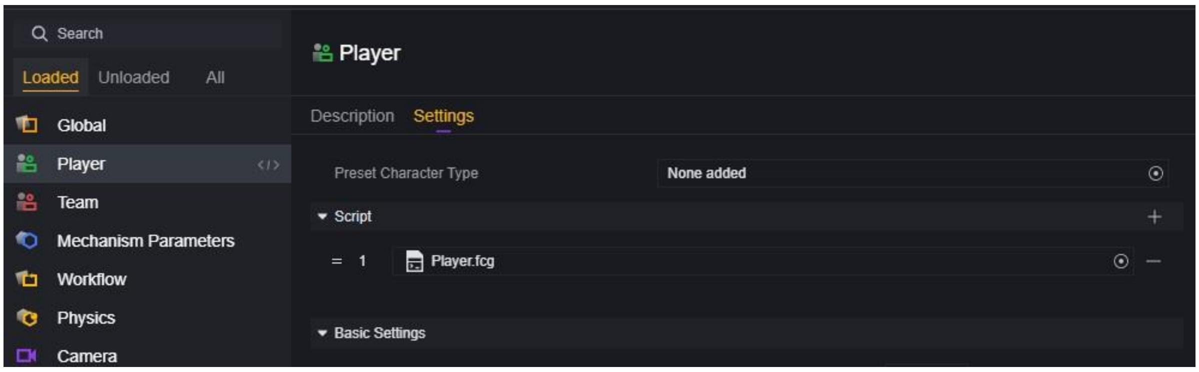
Is Static ✓

A new code script was created and configured as a server script because reducing the life value is a message that needs to be communicated to the server. Also because each time the player fires is an event that requires listening to the player's behaviour, the script was chosen to be mounted on the player entity.



Assets > Script

Player.fcg



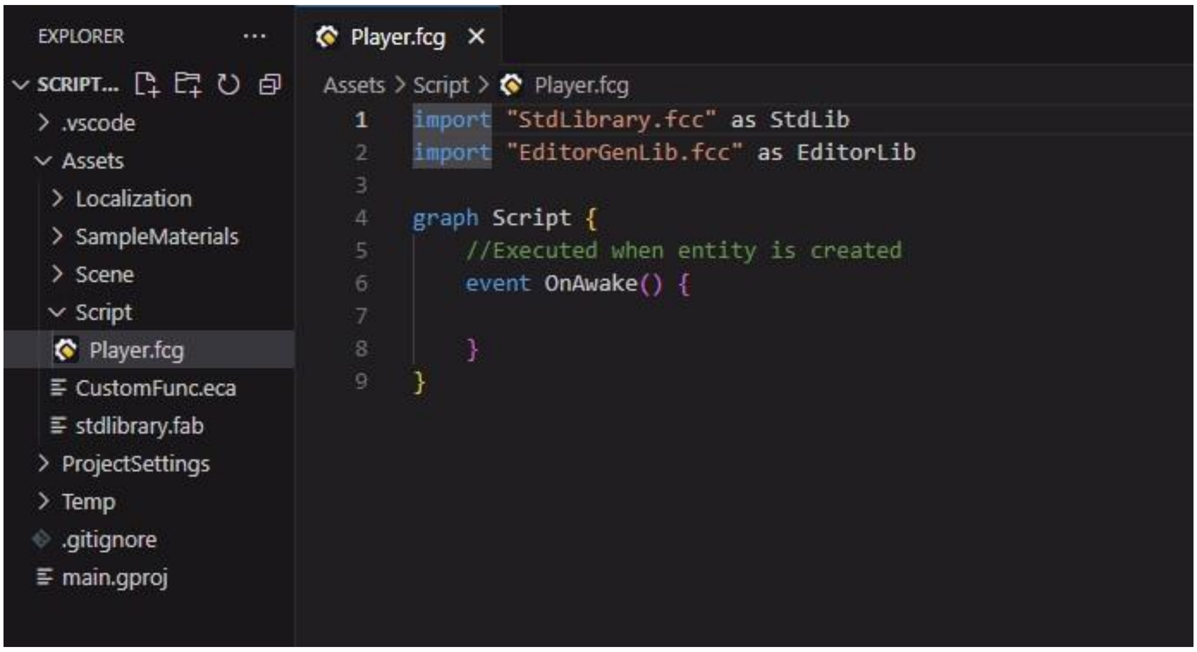
Preset Character Type None added

Script

1 Player.fcg

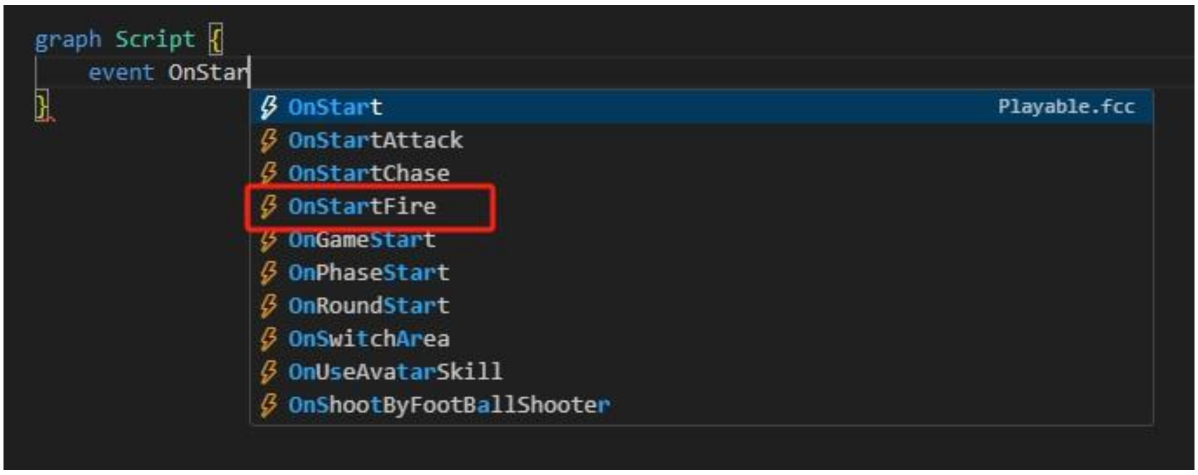
Basic Settings

Open Player.fcg for editing:



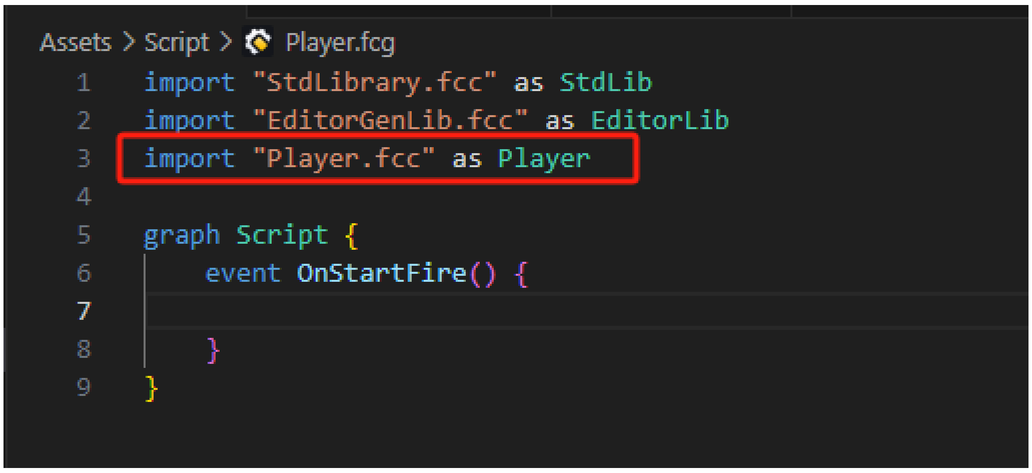
```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib

graph Script {
    //Executed when entity is created
    event OnAwake() {

    }
}
```

First confirm that the event trigger is when the player fires, and try to select the corresponding event via VS Code auto-completion:



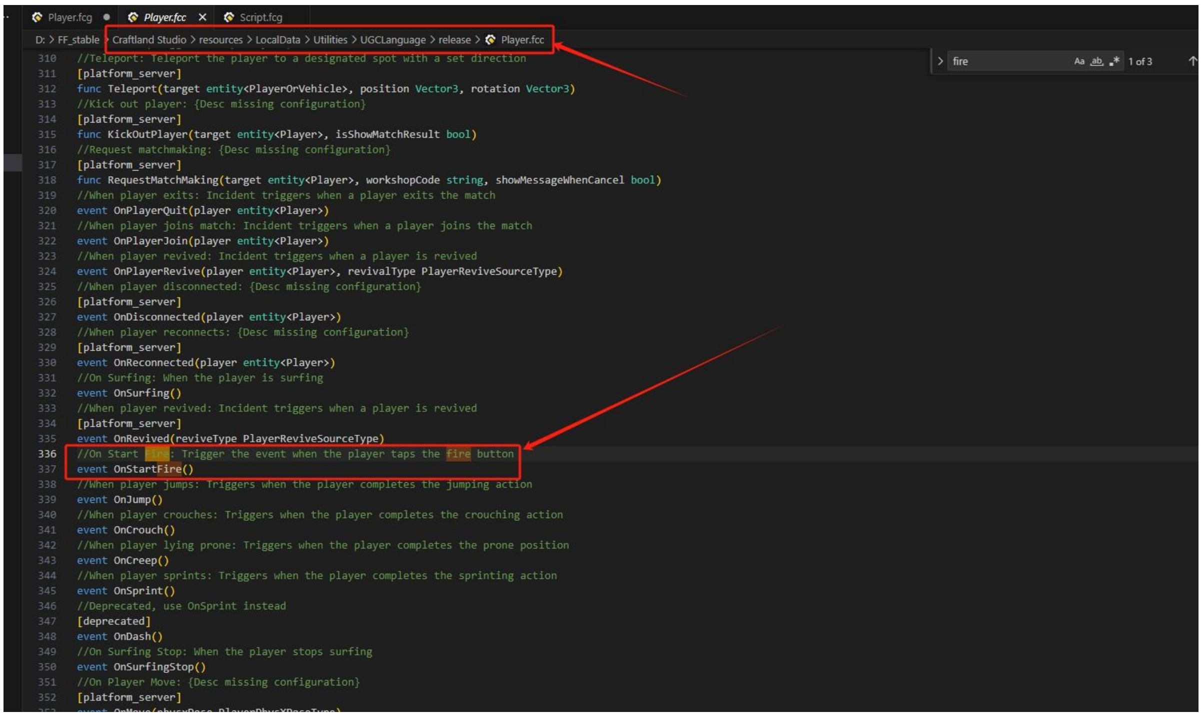
```
graph Script {
    event OnStar
}
```

Auto-completion of the event will reveal an automatic reference to the Player library where the event is located:



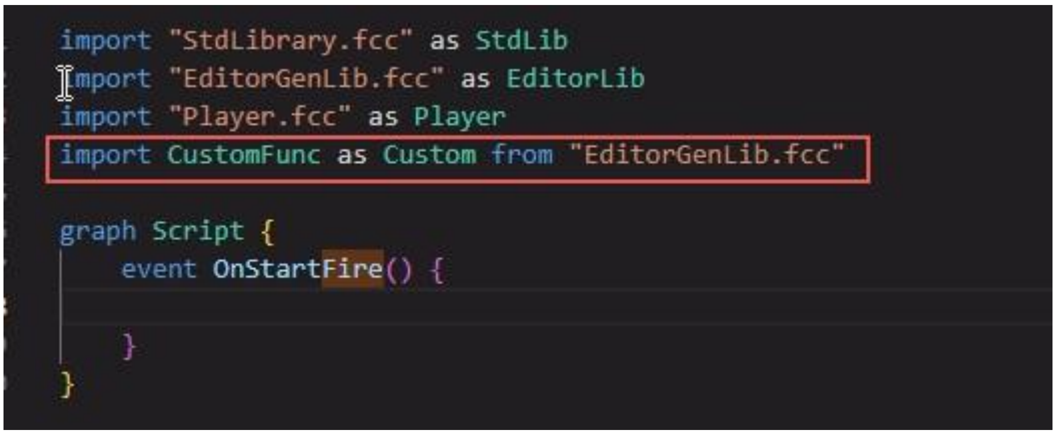
```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib
import "Player.fcc" as Player

graph Script {
    event OnStartFire() {

    }
}
```

If you don't get the event you want by auto-completion or you are not sure about the event, you can directly open the library of the corresponding module and search for it.


Player.fcg
Player.fcc
Script.fcg
D: > FF_stable > Craftland Studio > resources > LocalData > Utilities > UGCLanguage > release > Player.fcc
fire
1 of 3

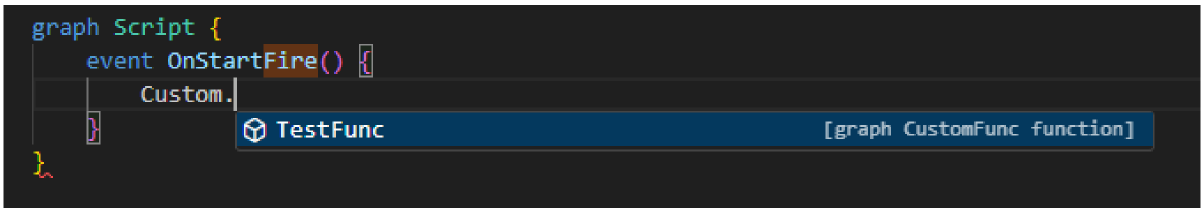
```
//Teleport: Teleport the player to a designated spot with a set direction
[platform_server]
func Teleport(target entity<PlayerOrVehicle>, position Vector3, rotation Vector3)
//Kick out player: {Desc missing configuration}
[platform_server]
func KickOutPlayer(target entity<Player>, isShowMatchResult bool)
//Request matchmaking: {Desc missing configuration}
[platform_server]
func RequestMatchMaking(target entity<Player>, workshopCode string, showMessageWhenCancel bool)
//When player exits: Incident triggers when a player exits the match
event OnPlayerQuit(player entity<Player>)
//When player joins match: Incident triggers when a player joins the match
event OnPlayerJoin(player entity<Player>)
//When player revived: Incident triggers when a player is revived
event OnPlayerRevive(player entity<Player>, revivalType PlayerReviveSourceType)
//When player disconnected: {Desc missing configuration}
[platform_server]
event OnDisconnected(player entity<Player>)
//When player reconnects: {Desc missing configuration}
[platform_server]
event OnReconnected(player entity<Player>)
//On Surfing: When the player is surfing
event OnSurfing()
//When player revived: Incident triggers when a player is revived
[platform_server]
event OnRevived(reviveType PlayerReviveSourceType)
//On Start Fire: Trigger the event when the player taps the fire button
event OnStartFire()
//When player jumps: Triggers when the player completes the jumping action
event OnJump()
//When player crouches: Triggers when the player completes the crouching action
event OnCrouch()
//When player lying prone: Triggers when the player completes the prone position
event OnCreep()
//When player sprints: Triggers when the player completes the sprinting action
event OnSprint()
//Deprecated, use OnSprint instead
[deprecated]
event OnDash()
//On Surfing Stop: When the player stops surfing
event OnSurfingStop()
//On Player Move: {Desc missing configuration}
[platform_server]
```

After the player fires, the custom function: TestFunc() needs to be run first, which requires a reference to CustomFunc.eca.



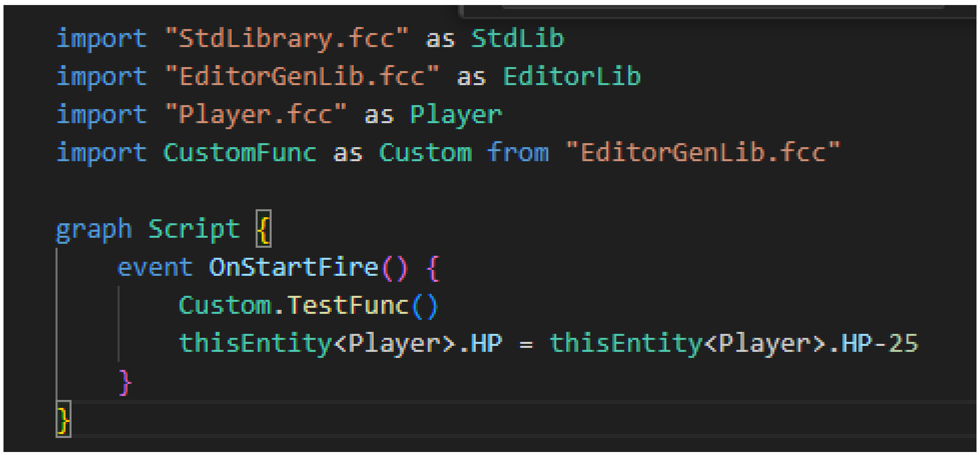
```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib
import "Player.fcc" as Player
import CustomFunc as Custom from "EditorGenLib.fcc"

graph Script {
    event OnStartFire() {

    }
}
```

TestFunc(), as a member of CustomFunc, can be quickly filled in with the "Alias. " for quick filling:



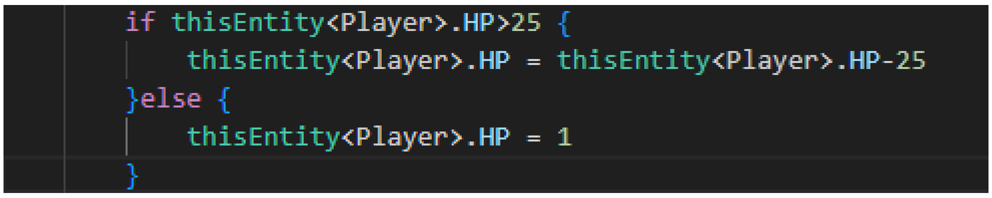
```
graph Script {
    event OnStartFire() {
        Custom.
    }       TestFunc                                [graph CustomFunc function]
}
```

Take -25 to the current player's lifesteal attribute.



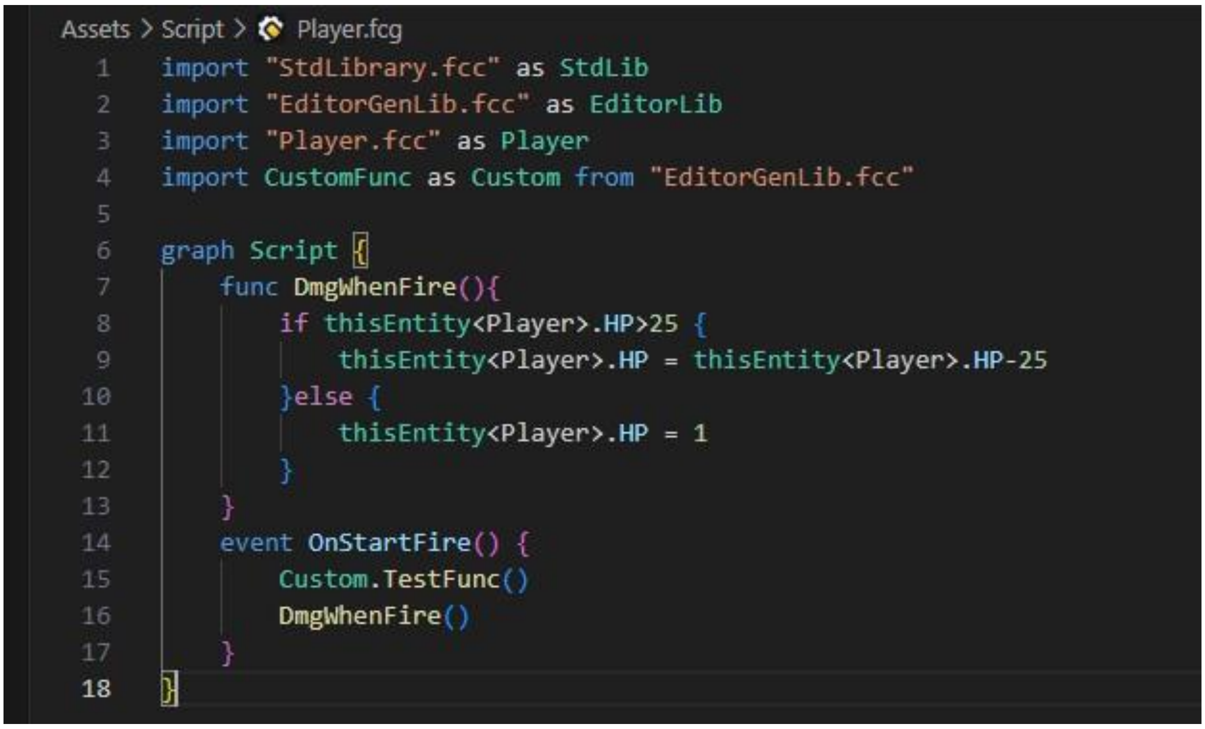
```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib
import "Player.fcc" as Player
import CustomFunc as Custom from "EditorGenLib.fcc"

graph Script {
    event OnStartFire() {
        Custom.TestFunc()
        thisEntity<Player>.HP = thisEntity<Player>.HP-25
    }
}
```

A judgement needs to be made on the player's current life value, and it is only set to 1 if it is insufficient:



```
if thisEntity<Player>.HP>25 {
    thisEntity<Player>.HP = thisEntity<Player>.HP-25
}else {
    thisEntity<Player>.HP = 1
}
```

Because the script structure restricts the existence of only one level of structure: function, event or variable definitions. So it is necessary to wrap a layer of functions around conditional judgements.

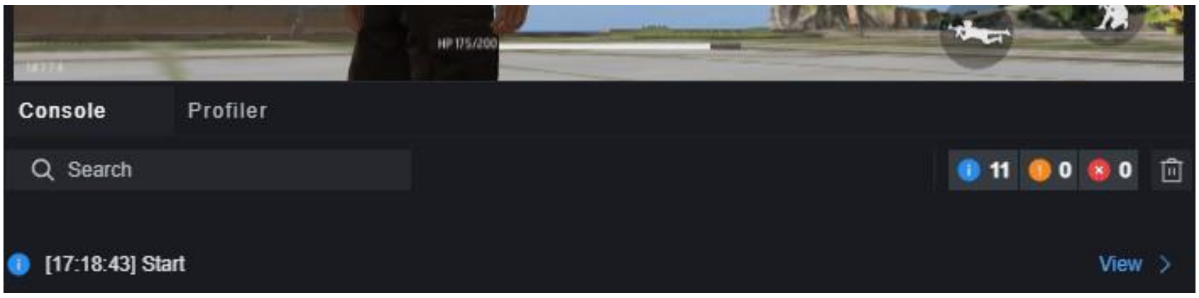


```
Assets > Script > Player.fcg
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib
import "Player.fcc" as Player
import CustomFunc as Custom from "EditorGenLib.fcc"

graph Script {
    func DmgWhenFire(){
        if thisEntity<Player>.HP>25 {
            thisEntity<Player>.HP = thisEntity<Player>.HP-25
        }else {
            thisEntity<Player>.HP = 1
        }
    }
    event OnStartFire() {
        Custom.TestFunc()
        DmgWhenFire()
    }
}
```

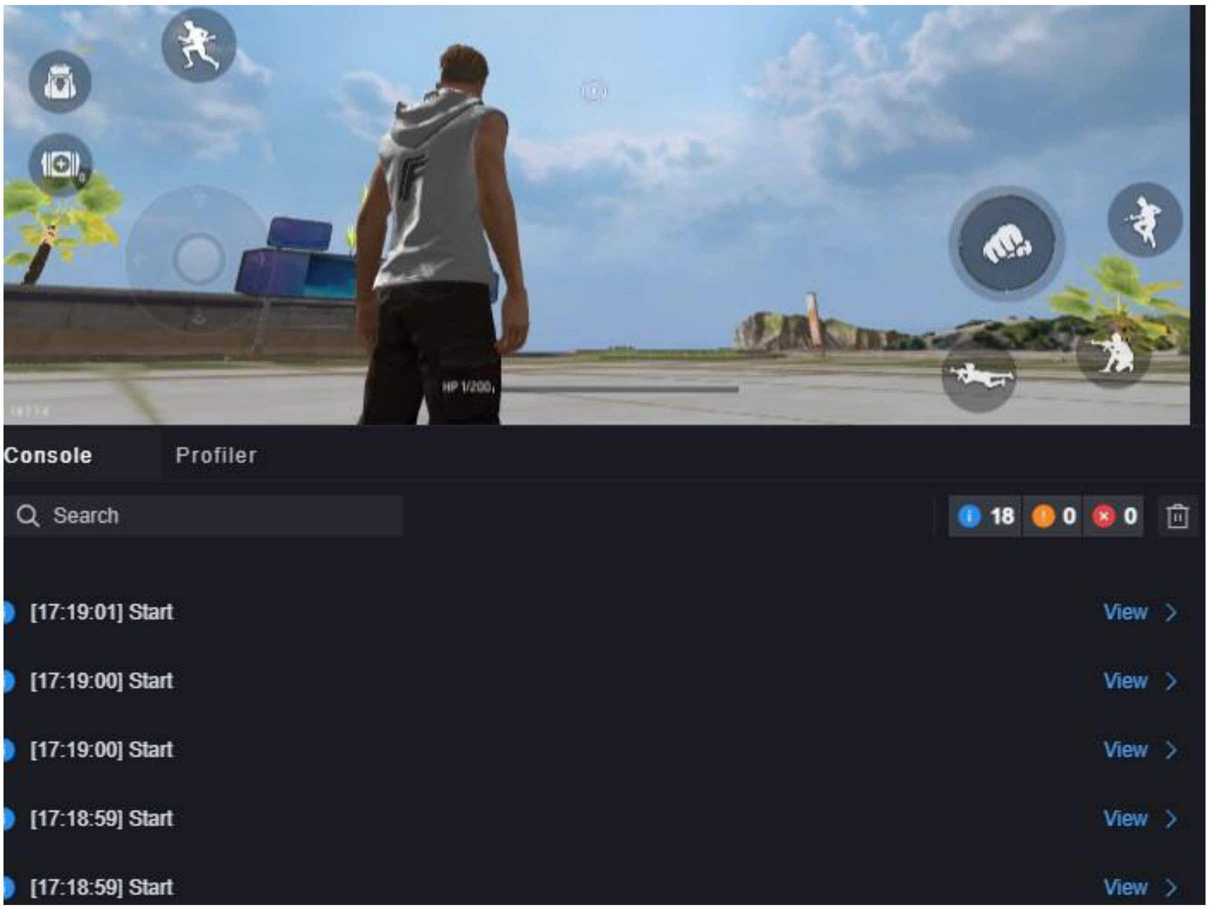
The logic for HP minus 25 can also be handled quickly using the following code:

```
thisEntity<Player>.HP -= 25
```

Run the test:



```
[17:18:43] Start
```



```
[17:19:01] Start    View >
[17:19:00] Start    View >
[17:19:00] Start    View >
[17:18:59] Start    View >
[17:18:59] Start    View >
```

Performance was in line with expectations.

Here is the code for the code script:

```
import "StdLibrary.fcc" as StdLib
import "EditorGenLib.fcc" as EditorLib
import "Player.fcc" as Player
import CustomFunc as Custom from "EditorGenLib.fcc"

graph Script {
    func DmgWhenFire(){
        if thisEntity<Player>.HP>25 {
            thisEntity<Player>.HP = thisEntity<Player>.HP-25
            //thisEntity<Player>.HP -= 25
        }else {
            thisEntity<Player>.HP = 1
        }
    }
    event   OnStartFire()
        {  Custom.TestFunc
        ()  DmgWhenFire()
    }
}
```