Guide to using the LSPDFR API and Documentation – Albo1125

Table of Contents

How do LSPDFR API plugins differ from normal, standalone RPH plugins? 2
Getting started: referencing the LSPDFR API in Visual Studio 2
Adding an entry point for our LSPDFR API plugin 3
Creating a callout for LSPDFR – Stolen Vehicle 5
- Preparation – Creating our callout's class in an organised fashion 5
- Inheriting from LSPDFR's Callout class and declaring our variables 6
- Overriding OnBeforeCalloutDisplayed() 7
- Overriding OnCalloutAccepted() 8
- Overriding Process() 9
- Overriding End() 10
- Registering our callout 10
- Other methods that can be overridden 10

Building our assembly and using our plugin 11
Further Development Tips 12
Appendix A: LSPD_First_Response.Mod.API.Functions documentation 13

Before I start, I'd like to thank the developers of LSPDFR for their outstanding work which has made LSPDFR and its API possible.
A basic knowledge of C#, Visual Studio and RPH developing is advised before using this guide. I've created a video series to help you obtain this here: https://www.youtube.com/playlist?list=PLEKypmos74W96pL9HW8BTQsyGtBTIyOst

How do LSPDFR API plugins differ from normal, standalone RPH plugins?

Standalone plugins for RPH are different in the sense that they need to be loaded manually via the RPH console and often have no interaction with other plugins; LSPDFR API plugins are automatically loaded by LSPDFR, are located in the Plugins/LSPDFR folder and can interact with the LSPDFR API functions. Also, there are a few requirements to which LSPDFR API plugins must adhere (we'll get to that later).

Getting started: referencing the LSPDFR API in Visual Studio

Much in the same way as referencing RAGEPluginHookSDK.dll as we did in Part 2 of the video series, we now need provide a reference to the LSPDFR assembly. The difference is that the LSPDFR API is included in the normal .dll file; there's no need to download and reference a new SDK file.
To add a reference to the LSPDFR API, open a new (or your existing) project in Visual Studio. On the **menu bar**, click **Project** and then **Add Reference**. Click **Browse** and navigate to the location of **LSPD First Response.dll.** Select it and click **OK**. Confirm its checkbox is checked and click **OK** again. You have now referenced the LSPDFR API. If you need to update your reference later (e.g. a new version of LSPDFR is released) simply replace the LSPDFR.dll file you referenced – the reference will automatically update.
Note: If you haven't done so yet, you also need to reference the RAGEPluginHook SDK to start developing LSPDFR API Plugins. Check out Part 2 of my video series for more information.

Adding an entry point for our LSPDFR API plugin

Now that we've referenced the LSDPFR API, we need to make sure our own plugin has the expected class structure. To start with, LSPDFR needs a place to kick off our plugin's code from – this is generally called an entry point, much like the EntryPoint class in standalone RPH plugins. The LSPDFR API kicks off our plugin in our plugin's Main class. This Main class has to inherit LSPDFR's abstract Plugin class and needs to override a few of the abstract Plugin class's methods (some extra information on what exactly this means can be found here: http://rbwhitaker.wikidot.com/c-sharp-abstract-classes).

Keeping this in mind, we'll start by adding a new class to our project in Visual Studio. Click **Project** and click **Add Class**. Rename it to **Main.cs** and click Add. You should now be presented with a new window containing a basic code setup with the information we entered.

Next, we're going to tell Visual Studio we're **using** the LSPDFR API and the RAGE SDK; this saves us the effort of having to type the entire path of the LSPDFR API / RAGE SDK every time we use it in our code. To do this, add the following line of code under the last **using** statement (at the top of your code):

```
using LSPD_First_Response.Mod.API;
using Rage;
```

Next, we need to make our Main class **inherit** from LSPDFR's Plugin class. To do this, change your code so the Main class declaration looks like this:

```
public class Main : Plugin
{
```

Now we need to override two of the Plugin class's abstract methods: Initialize() and Finally(). These two methods are called when our plugin is respectively created and cleaned up.

```
public class Main : Plugin
{
    public override void Initialize()
    {

    }
    public override void Finally()
    {

    }
}
```

Now, we are going to make these methods actually do something. We're going to use LSPDFR's **Functions.OnOnDutyStateChanged** event that notifies us whenever the player goes on or off duty, and we're going to create our own event handler for this (extra information here: http://rbwhitaker.wikidot.com/c-sharp-events). Additionally, I'm going to show you how to add a few useful log messages that are useful for debugging your plugin later. We're also going to add a new method that we will later expand to register the callouts we create.

```
public class Main : Plugin
    {
        public override void Initialize()
        {
            Functions.OnOnDutyStateChanged += OnOnDutyStateChangedHandler;
            Game.LogTrivial("Plugin LSPDFR_API_Guide " +
System.Reflection.Assembly.GetExecutingAssembly().GetName().Version.ToString() + " has
been initialised.");
            Game.LogTrivial("Go on duty to fully load LSPDFR_API_Guide.");
        }

        public override void Finally()
        {
            Game.LogTrivial("LSPDFR_API_Guide has been cleaned up.");
        }

        private static void OnOnDutyStateChangedHandler(bool OnDuty)
        {
            if (OnDuty)
            {
                RegisterCallouts();
            }
        }

        private static void RegisterCallouts()
        {
            //We will implement this method later.
        }
    }
```

In the **Initialize()** override, we **subscribe** our **event handler** to the Functions.OnOnDutyStateChanged event. We also **log** our plugin's **name** and **version** to the RPH log.
In the **Finally()** override, we **log** that our plugin has been cleaned up.
In our **OnOnDutyStateChangedHandler** event handler, we first check if the player is on duty and if so, we call our RegisterCallouts() method.
In our **RegisterCallouts()** method we currently don't do anything yet. We will add more to this later on.
That's our Main class done; LSPDFR will now successfully load our plugin.

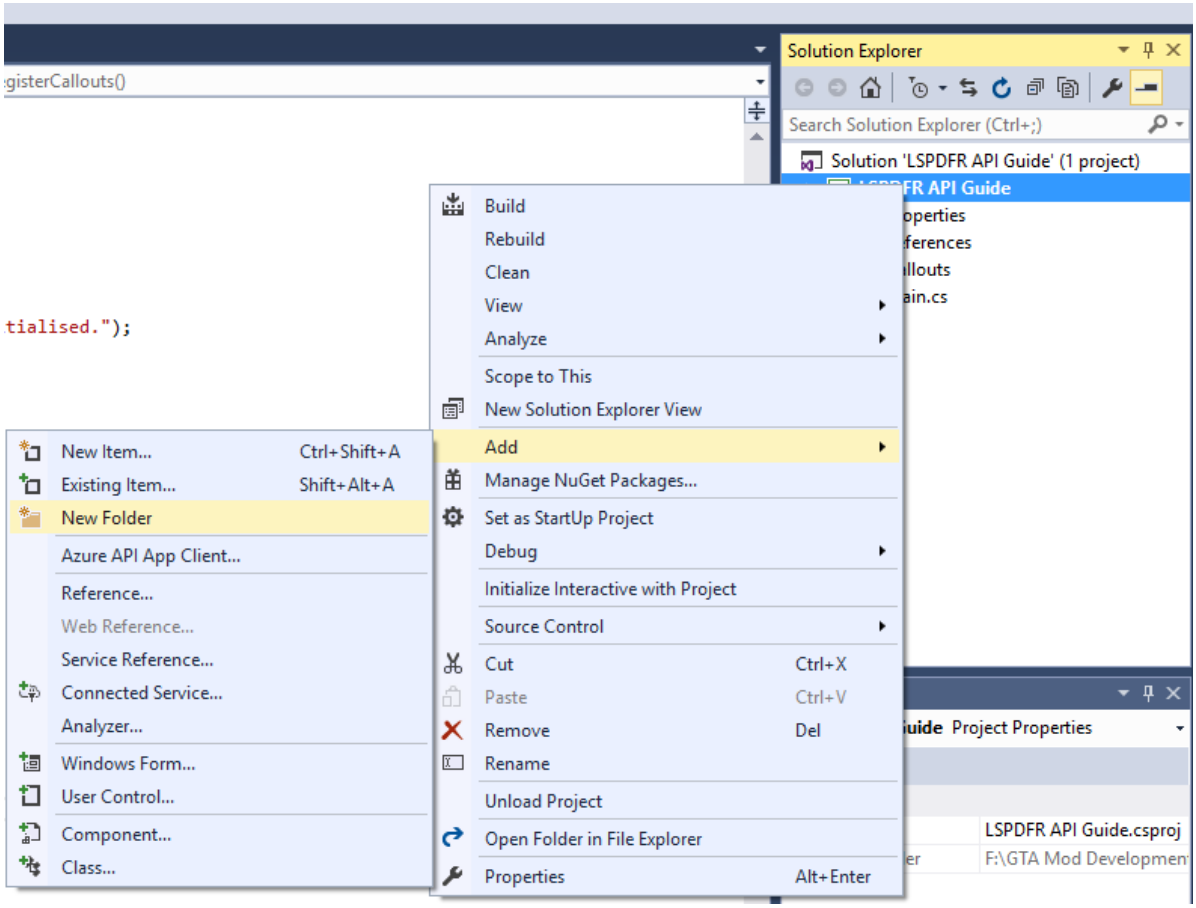
Creating a callout for LSPDFR – Stolen Vehicle

Just like our Main class, LSPDFR expects any callout classes we create to have a predetermined structure.

Preparation – Creating our callout's class in an organised fashion

To keep our project organized, we're going to keep any callouts we create in the **Callouts** folder. We'll need to create this folder first. To do so, first make sure the **Solution Explorer** is visible on the right of your Visual Studio interface. If not, you can display it by clicking **View** on the menu bar and selecting **Solution Explorer.**

To add a new folder, **Right Click** on your project in the Solution Explorer (often the second line), click **Add** and click **New Folder.** A new folder will be created. Rename it to **Callouts.**



Next, we need to add a new class in our Callouts folder. **Right Click** the callouts folder and select **Add**, then **New Item.** Make sure you're creating a **class** and name it **StolenVehicle.cs.** Again, we will be presented with a basic code setup.

Let's add some of our **using** statements again to let Visual Studio know we won't be typing their entire path every time. Add the following under the last **using** statement at the top of your code:

```
using Rage;
using LSPD_First_Response.Mod.API;
using LSPD_First_Response.Mod.Callouts;
using LSPD_First_Response.Engine.Scripting.Entities;
```

We've talked about the first two using statements. The third contains everything we need to create an LSPDFR callout, while the fourth contains some useful functions we can use to get a spawn point.

Inheriting from LSPDFR's Callout class and declaring our variables

All our LSPDFR callouts need to inherit from LSPDFR's **abstract Callout** class. They also need to have a **CalloutInfo** attribute to provide some basic information about our callout: its name and probability. On top of that, we need to override four methods from the Callout class to make our callout unique and make it do what we want. We also need to declare the variables that we're going to need, like **vehicles**, **peds, Blips, Vector3s** (locations) etc.

```
namespace LSPDFR_API_Guide.Callouts
{
    [CalloutInfo("StolenVehicle", CalloutProbability.High)]
    public class StolenVehicle : Callout
    {
        private Ped Suspect;
        private Vehicle SuspectVehicle;
        private Vector3 SpawnPoint;
        private Blip SuspectBlip;
        private LHandle Pursuit;
        private bool PursuitCreated = false;

        public override bool OnBeforeCalloutDisplayed()
        {
            return base.OnBeforeCalloutDisplayed();
        }

        public override bool OnCalloutAccepted()
        {
            return base.OnCalloutAccepted();
        }

        public override void Process()
        {
            base.Process();
        }

        public override void End()
        {
            base.End();
        }
    }
}
```

The **namespace** will most likely be different for you depending on the names you chose before.
The **CalloutInfo** attribute sets our callout's name to StolenVehicle and sets its probability to High.
We also declare a **Suspect**, a **SuspectVehicle,** a **SuspectBlip,** a **Pursuit,** a **PursuitCreated** Boolean (defaulting to false) and a **SpawnPoint** that we can use throughout our callout.
OnBeforeCalloutDisplayed() is called before our callout gets displayed as a notification. **OnCalloutAccepted()** is called when a user accepts our callout. **Process()** is constantly called while our callout is running and is used for our callout's main logic. **End()** is called when our callout ends: it is important to clean up all of the stuff we do here.

Overriding OnBeforeCalloutDisplayed()

Let's start with **OnBeforeCalloutDisplayed().** We need to do all of our preparation work here, like setting the callout's spawn point, setting its notification message and playing the Police Scanner audio.

```
public override bool OnBeforeCalloutDisplayed()
{
   SpawnPoint =
World.GetNextPositionOnStreet(Game.LocalPlayer.Character.Position.Around(250f));

   ShowCalloutAreaBlipBeforeAccepting(SpawnPoint, 30f);
   AddMinimumDistanceCheck(20f, SpawnPoint);

   CalloutMessage = "Stolen Vehicle";
   CalloutPosition = SpawnPoint;

   Functions.PlayScannerAudioUsingPosition("WE_HAVE CRIME_GRAND_THEFT_AUTO
IN_OR_ON_POSITION", SpawnPoint);

    return base.OnBeforeCalloutDisplayed();
}
```

To find a SpawnPoint, we use Rage's **World.GetNextPositionOnStreet(Vector3)** method. We must pass it a Vector3 point, so we generate a Vector3 that's about 250m away from the player using **Game.LocalPlayer.Character.Position.Around(250f).** The **Around extension method** comes from the fourth **using** statement we added earlier (extra information on extension methods here: http://rbwhitaker.wikidot.com/c-sharp-extension-methods).

Next, we tell LSPDFR to show the callout area on the minimap using a circle whose radius is 30m. We also add a minimum distance check to see if the player is far enough away (more than 20m).
We set the **CalloutMessage** (displayed in the notification) to Stolen Vehicle and set the **callout's position** to our SpawnPoint we generated earlier. Then, we **play some audio** using LSPDFR's Police Scanner. The string passed corresponds to the file names that can be found in **GTAV/LSPDFR/Police Scanner**.
Finally, we allow LSPDFR's base method to run to finalize creating the callout using **return base.OnBeforeCalloutDisplayed();**

Overriding OnCalloutAccepted()

Now for the **OnCalloutAccepted()** override. Here, we create our Suspect and their vehicle and add a blip to the suspect that's displayed on the minimap. To spice it up a little, we'll also make the suspect drive erratically.

```
public override bool OnCalloutAccepted()
{
   SuspectVehicle = new Vehicle("ZENTORNO", SpawnPoint);
   SuspectVehicle.IsPersistent = true;

   Suspect = SuspectVehicle.CreateRandomDriver();
   Suspect.IsPersistent = true;
   Suspect.BlockPermanentEvents = true;

   SuspectBlip = Suspect.AttachBlip();
   SuspectBlip.IsFriendly = false;

   Suspect.Tasks.CruiseWithVehicle(20f, VehicleDrivingFlags.Emergency);
   return base.OnCalloutAccepted();
}
```

We create a new **SuspectVehicle**, a Zentorno, at our SpawnPoint. We then set the SuspectVehicle to be **persistent**; this means that it won't be cleaned up by GTA V's memory cleanup.
Next, we create our **suspect**, a driver for our vehicle using Rage's **CreateRandomDriver()** method. We set the suspect to be **persistent** too and also set our suspect's **BlockPermanentEvents** property to true. This ensures our suspect only responds to our commands and is unaffected by GTA V's built-in behaviour (fleeing when gunshots are fired etc.).
We also attach a **Blip** to our suspect and assign it to the variable SuspectBlip. We also set it to be unfriendly.
Lastly, we assign our suspect a **CruiseWithVehicle task** to make them drive around with their vehicle at a speed of 20 meters per second. We set the VehicleDrivingFlags to Emergency to make our suspect drive erratically.

Overriding Process()

Next up is our **Process()** override. This is the main logic of our callout. You must **never** call GameFiber.Sleep() or GameFiber.Wait() in this method without creating a new GameFiber beforehand.

We are going to make our callout wait for the player to get close to the suspect. Then, we're going to create a pursuit. When the pursuit finishes, we will end our callout.

```
public override void Process()
{

   base.Process();
   if (!PursuitCreated && Game.LocalPlayer.Character.DistanceTo(Suspect.Position) < 30f)
   {
       Pursuit = Functions.CreatePursuit();
       Functions.AddPedToPursuit(Pursuit, Suspect);
       Functions.SetPursuitIsActiveForPlayer(Pursuit, true);
       PursuitCreated = true;
   }

   if (PursuitCreated && !Functions.IsPursuitStillRunning(Pursuit))
   {
       End();
   }
}
```

As you can see, **base.Process()** is always called first to handle the basic callout handling. Then, we use an **if statement** to check whether the player is close to the suspect (closer than 30m). In this case, we create a pursuit, add our suspect to the pursuit and set it to be active. However, we need to ensure this only happens once, which is where our PursuitCreated Boolean comes in. Once we create a pursuit, we set it to true, ensuring the **if statement** will never be passed again: the if statement can only be passed if PursuitCreated is equal to false (!PursuitCreated).

Once our pursuit has been created, our second if statement comes into play as PursuitCreated is now equal to true. If the pursuit has been created, we check whether the pursuit is still running using **Functions.IsPursuitStillRunning(LHandle).** If this equals false, we know the pursuit has ended and we call our End() method to clean our callout up.

Overriding End()

In our **End()** override, we want to make sure all of the things we created are cleaned up. In this case, the following will suffice:

```
public override void End()
{
   base.End();
   if (Suspect.Exists()) { Suspect.Dismiss(); }
   if (SuspectVehicle.Exists()) { SuspectVehicle.Dismiss(); }
   if (SuspectBlip.Exists()) { SuspectBlip.Delete(); }
}
```

Again, we call the base.End() method first to let LSPDFR clean up the basics of our callout internally. Afterwards, we check if our Suspect and our SuspectVehicle **exist** and if so, we **Dismiss()** them. Dismissing them hands them back to GTA V's behaviour handling and cleanup, meaning they will be cleaned up like any other ped or vehicle in the GTA world. Basically, we're undoing what we did earlier with BlockPermanentEvents and setting them to be persistent. I prefer Dismissing over deleting for increased realism.
However, we want our blip to simply disappear from the minimap once the callout ends. That's why we call its Delete() method immediately once our callout ends.
Note: It's often important to check if something **exists** before trying to do something with it. If something doesn't exist and you try to call one of its methods, RPH will crash.

Registering our callout

Remember the empty RegisterCallouts() function we created in the Main class? Now that we've created our callout, we can register it there. Go back to your Main class and add code to register your callout like so:

```
private static void RegisterCallouts()
{
    Functions.RegisterCallout(typeof(Callouts.StolenVehicle));
}
```

Other methods that can be overridden

These methods can also be overridden; however they aren't required to be for this callout.

OnCalloutDisplayed() called when callout notification is displayed.

OnCalloutNotAccepted() called when callout is not accepted.

Building our assembly and using our plugin

To build our plugin, we need to make sure our target .NET framework is set to 4.6 or higher. In the menu bar, click **Project** and **[YourPluginName] properties**. From the **Target Framework** dropdown menu, select **.NET Framework 4.6.**
You can also set basic information for your assembly, such as the version, author and description. Click the **Assembly Information** button and fill in the fields you find necessary. I recommend filling in at least the Description and both the **Assembly and File version** fields. Make sure you increase your version number with every update you release for debugging and clarity purposes.

When you're done, it's time to build. In the menu bar, click **Build** and click **Build Solution.** Any potential errors will appear in the Output window below – if there are no errors, your build will succeed and your assembly will be located in the Bin/Debug folder in the location where you saved your project, along with all its references.
To use your assembly, copy it to the Plugins/LSPDFR folder. It'll be loaded by LSPDFR automatically via the Main class we created.

Further Development Tips

There are quite a lot of examples and code samples available on the LSPDFR GitHub – feel free to use them and to submit your own at https://github.com/LMSDev/LSPDFR-API

If you have any questions regarding LSPDFR API development, you can post a topic here: http://www.lcpdfr.com/forums/forum/894-api-development/

Try keeping your code organised and keep your classes in appropriate folders as we've done above. Comments help a lot if you're doing complicated things – you'll likely not remember what you did a few days after you did it.

Make full use of IntelliSense as it's a great time saver and really helps you get started as it'll do all the hard searching for you.

If you need to sleep or wait, do so in a separate GameFiber if you haven't already created one. Sleeping a core LSPDFR GameFiber (e.g. the Process method) usually results in fatal errors.

Once your project grows, Visual Studio has some neat features to make it easier to read your code. Make use of the collapse feature for methods you've finished and aren't editing for optimising your code reading.

If you'd like to create a menu, you can do so using RAGENativeUI. More information: http://www.lcpdfr.com/forums/topic/54970-dev-tool-rel-ragenativeui/ .

And above all: Be unique and have fun!

Appendix A: LSPD_First_Response.Mod.API.Functions documentation

Function	Description
void AddCopToPursuit(LHandle Pursuit, Ped Cop)	Pass an LHandle Pursuit instance and a Cop ped. Adds the Cop to the pursuit as backup.
void AddPedToPursuit(LHandle Pursuit, Ped Suspect)	Pass an LHandle Pursuit instance and a Suspect ped. Adds the Suspect to the pursuit as a Suspect to be chased.
LHandle CreatePursuit()	Returns an LHandle Pursuit instance. Used to create a pursuit to be further manipulated with other functions.
void ForceEndCurrentPullover()	If the player is performing a traffic stop, instantly ends it.
void ForceEndPursuit(LHandle Pursuit)	Pass a pursuit that is currently running. Immediately aborts the pursuit.
LHandle GetActivePursuit()	Returns the LHandle Pursuit instance that's currently running. If no pursuit is running, returns null.
System.Reflection.Assembly[] GetAllUserPlugins()	Returns an array of all the LSPDFR plugins that are currently running. Used to safely access other plugins' APIs.
LHandle GetCurrentPullover()	Returns a Pullover LHandle. If the player is not performing a traffic stop, returns null.
Ped GetPedArrestingOfficer(Ped suspect)	Pass an arrested Suspect ped. Returns the Cop ped that arrested the suspect (can also return the player).
Persona GetPersonaForPed(Ped ped)	Pass a ped. Returns its persona. Persona is part of *LSPD_First_Response.Engine.Scripting.Entities.*
Ped GetPulloverSuspect(LHandle pullover)	Pass an active Pullover LHandle. Returns the Driver ped of the suspect vehicle. Most common use: GetPulloverSuspect(GetCurrentPullover());
Ped[] GetPursuitPeds(LHandle pursuit)	Pass an active Pursuit LHandle. Returns a Ped[] array containing all the suspect peds.
String GetVehicleOwnerName (Vehicle vehicle)	Pass an existing vehicle. Returns a string containing the owner name of the vehicle.

Function	Description
WorldZone GetZoneAtPosition(Vector3 pos)	Pass a Vector3 position. Returns a WorldZone variable containing information about the WorldZone at that position. WorldZone is part of *LSPD_First_Response.Engine.Scripting.Entities.*
Bool IsCalloutRunning()	Returns a Boolean indicating whether any callout is currently running.
Bool IsPedArrested(Ped ped)	Pass a ped. Returns a Boolean indicating whether the ped is arrested or not.
Bool IsPedGettingArrested(Ped ped)	Pass a ped. Returns a Boolean indicating whether the ped is currently being arrested or not.
Bool IsPedInPrison(Ped ped)	Pass a ped. Returns a Boolean indicating whether the ped is currently in prison or not.
Bool IsPedStoppedByPlayer(Ped ped)	Pass a ped. Returns whether the ped is currently stopped by the player (i.e. affected by long hold E)
Bool IsPlayerPerformingPullover()	Returns a boolean indicating whether the player is currently performing a traffic stop or not.
Bool IsPoliceComputerActive()	Returns a boolean indicating whether the player is currently using the Police Computer. Recommended to be used before checking for key input yourself to avoid conflicts.
Bool IsPursuitStillRunning(LHandle Pursuit)	Pass a Pursuit LHandle. Returns a Boolean indicating whether the Pursuit LHandle is active or not.

Functions	Description
Void PlayScannerAudio(string audio)	Pass a string. String must comply with a file name in GTAV/LSPDFR/Police Scanner and any folder therein. Examples: Passing “WE_HAVE” will play any audio file with file name “WE_HAVE” and any number behind it, so it may play “WE_HAVE_01” or “WE_HAVE_02” at random. Passing “CITIZENS_REPORT_01” will always play “CITIZENS_REPORT_01” specifically. Multiple files can be passed separated by spaces, e.g. “CITIZENS_REPORT_01 CRIME_GRAND_THEFT_AUTO_01”. Audio will only play if no other scanner audio is currently playing.
Void PlayScannerAudioUsingPosition(string audio, Vector3 position)	Pass an audio string and a Vector3 position. Audio string is used in the same way as PlayScannerAudio(string audio). To play audio with information about the Vector3 position, add “IN_OR_ON_POSITION” somewhere in the audio string. Audio will only play if no other scanner audio is currently playing.
Void RegisterCallout(System.Type callout)	Pass your callout’s type. Registers the callout so it can occur via LSPDFR’s callout system. Example: RegisterCallout(typeof(MyCallout));
Vehicle RequestBackup(Vector3 pos, EBackupResponseType responsetype, EBackupUnitType unittype)	Pass a Vector3 position, an EBackupResponseType variable and an EBackupUnitType variable. Spawns the requested backup unit and returns the vehicle of the backup unit. *EBackupResponseType and EBackupUnitType are part of LSPD_First_Response.*

Function	Description
Void SetCopAsBusy(Ped cop, bool busy)	Pass a Cop ped and a bool indicating whether to set the cop to busy or not. If busy, LSPDFR shouldn't affect the cop's behaviour.
Void SetPedAsArrested(Ped suspect)	Pass a suspect. Sets them as arrested. Does not make them handcuffed etc. Likely to be used to make cops ignore the ped.
Void SetPedAsCop(Ped ped)	Pass a ped. Sets the ped to be a cop recognised by LSPDFR. Doesn't change the model.
Void SetPedCantBeArrestedByPlayer(Ped ped, bool toggle)	Pass a ped and a Boolean. If true, the player can't arrest or stop the ped.
Void SetPersonaForPed(Ped ped, Persona persona)	Pass a ped and a Persona instance. Assigns the ped the Persona instance in the police computer. Persona is part of *LSPD_First_Response.Engine.Scripting.Entities.*
Void SetPursuitCopsCanJoin(LHandle Pursuit, bool toggle)	Pass a pursuit LHandle and a Boolean. If false, cops won't join pursuits unless the AddCopToPursuit method is used. Used to prevent ambient cops from automatically joining pursuits.
Void SetDisablePursuitAI(LHandle Pursuit, bool toggle)	Pass a Pursuit LHandle and a Boolean. If true, completely disables the pursuit AI for both suspects and backup units. Current use is not recommended.
Void SetPursuitIsActiveForPlayer (LHandle Pursuit, bool toggle)	Pass a Pursuit LHandle and a Boolean. If true, sets the pursuit to be active for the player (flashing blip etc.). If false, sets the pursuit to be inactive for the player – the player will join when they get close to a suspect.
Void SetVehicleOwnerName(Vehicle vehicle, string name)	Pass a Vehicle and a string. Sets the Vehicle's owner to the string in the police computer.
Void StartCallout(string name)	Pass a string. Starts the callout that has a name equal to the string as set in the CalloutInfo attribute.

Function	Description
Void StopCurrentCallout()	If a callout is currently running, forces it to end and calls its End() method.