Remote Objects: Documentation & Setup Guide

(Initial prototype version)

Links:

- Unity package download
    - Unity Development Project GitHub (Pong example)
    - Unity Project Example GitHub (Helicopter Game)
- Python package download
    - Python package GitHub

Remote Objects allow for a game made in Unity to send commands to external microcomputers (such as Raspberry Pi) over Wi-Fi to create unique immersive experiences involving lights, sound, and more. The intended use-case is as a simple option to enhance interactive games and experiences with remotely-connected devices (for example to add realtime reactive lighting at a game expo display), as well as for creating new games that make novel use of physical space in gameplay.

The system uses a modular design allowing developers to create their own modules for their specific use-case. The customisation element is currently functional but admittedly a bit clunky, especially on the remote device code side.

This version includes two modules:

- ***RemoteAudioSource*** - Play sounds through remote devices.
- ***Remote Arduino*** - Control Arduino output pins remotely.

Disclaimer

This version is a proof-of-concept prototype. There are many areas that could be improved or have a layer of clunkiness.

The current system is designed to work between the Unity game engine and devices running my Python package. Continued development may bring support for various game engines, and the device codebase may eventually outgrow its current Python implementation.

My testing has been using a pair of Raspberry Pi 3 Model B's, running various versions of the standard Raspbian image.

Slash commands

At its core, the system is based around simple human-readable "slash commands", which can include many arguments separated by slashes. The general format is /[module]/[function]/[arguments].

For example,

- /audio/play/snare - Plays a sound clip named "snare".
- * /servo/turn/30/fast - Turns a servo motor 30 degrees at a preset "fast" speed.
- * /led/strobe/19/100/10 - Strobe an LED on pin 19 once every 100ms for 10 seconds.

** These are hypothetical commands that do not currently have an implementation.*

Commands can be sent over UDP to running remote devices. Whilst the current prototype is implemented exclusively with Unity, any program that sends a valid slash command to a remote device's IP on the correct port should work. As such, it should be relatively straightforward to implement the system in other environments.

The default port for remote devices is 32019, although this can be tweaked in the ini.

Contents

1. **Host-side (Unity) Documentation**
2. **Remote Device (Python) Documentation**
3. **Example setup guide** ← *Start here if you're new.*

Host-side (Unity) Documentation

While the Unity package should work with most relatively recent versions of Unity, it has only been tested on 2021.3.17f1.

Remote Objects

One of the main ideas in my design for this system is that *RemoteObjects* act as extensions to the existing *GameObject* paradigm present in Unity. Each *RemoteObject* is associated with a corresponding *RemoteDevice*. *RemoteObjects* can have *RemoteComponents*, and *RemoteComponents* are somewhat based on the structure of standard *MonoBehaviour* components.

Fallback mode

Fallback mode allows RemoteComponents to act as regular components when a device is not linked to a remote object. For example, a RemoteAudioSource will act as a regular AudioSource and play sounds directly through the host PC's audio. This is useful for quick prototyping and works well as a backup in case a device is not working.

At this stage, fallback mode is very barebones and isn't really something you should use for more complex solutions beyond a simple backup or testing tool. In future, I'm looking to integrate a system using UnityEvents to enable a more easily customisable system.

The Remote Object component includes a boolean for "Auto Fallback Mode". This will automatically enable fallback mode on any RemoteObjects that are not linked to a device. This is applied at the end of the identification flow.

Reference

Properties

RemoteDevice remote

(Cannot be set in inspector) The RemoteDevice representing the device this RemoteObject is currently linked to. May be null.

string debugIPAddress

If set on Awake, this RemoteObject will automatically configure its remote to connect to a given IP address. Avoid using this where possible, instead set devices using the Identification UI.

string remoteName

The name of this RemoteObject. This will be shown in the Identification UI, and potentially other UI in the future.

Sprite remoteIcon

The icon to show for this RemoteObject in the Identification UI, and potentially other UI in the future.

bool autoFallbackMode

If set to true, this object will automatically enter Fallback Mode (see above) when it is not linked to a device. For example, if ticked, RemoteAudioSources will instead play through the computer's audio when no linked device is selected.

Methods

SendCommand(string module, string func, string[] args)

Sends a command to the linked device, with the given module, function and arguments.

This will automatically convert the given arguments to a slash command, e.g.

/[module]/[func]/[args[0]]/[args[1]]/[args[2]]

SendRawCommand(string command)

Sends the *command* to the RemoteDevice with no alteration.

Note: This should be avoided where possible - if you are writing custom code to manually combine strings into a slash command, look into using SendCommand instead.

UpdateFallbackMode()

If autoFallbackMode is enabled, this will first enable or disable Fallback Mode depending on if the linked device (remote) is unset or set, respectively.

Regardless, it runs through each component attached to this RemoteObject and activates or deactivates Fallback Mode accordingly.

Primarily used internally. This may be of use for projects where autoFallbackMode is left disabled and fallback mode is controlled externally. If this is the case, you could avoid individually activating/deactivating each RemoteComponent manually and instead run this method.

ResetRemote

Clears the attached RemoteDevice to null.

RemoteDevice class

This class represents a connected device (known good IP) on the network. You shouldn't need to interact with this class much, however there is utility to accessing *RemoteObject.device* to fetch the device's details or for implementing more advanced systems.

RemoteState state

Signifies the state of this *RemoteDevice*. Uses enum *RemoteState*. The state can be:

- ***Unassigned*** - This IP address has been found on the network and identified as a *RemoteDevice*, however it has not been assigned to a *RemoteObject* yet.
- ***SkippedAssignment*** - The user chose to skip assigning this *RemoteDevice*.
- ***Assigned*** - This *RemoteDevice* has been assigned to a *RemoteObject*.

Socket socket

The UDP Socket (From *System.Net.Sockets*) for this *RemoteDevice*. This should not be used unless you're looking for an easy solution for adding additional low-level networking code.

string ip

The IP Address of this device.

SendNetMessage(string message)

Encodes and sends the unaltered *message* string to the connected socket.

Do not use any other of this class' methods, as they're designed for internal systems.

RemoteManager

For RemoteObjects to function, you need a RemoteManager in the scene. You can get the prefab for RemoteManager from *RemoteObject > Prefabs > RemoteManager*.

There is a *RemoteManager* class, but this section will also discuss the *RemoteIdentificationHandler* class also included on the prefab.

Poking

“Poking” is a slightly bodgy system that regularly sends a ping message to every device to ensure that the connection is kept awake.

In my testing using my phone hotspot, I found a significant delay came up when a command hadn’t been sent for a short while (around 20 seconds, although it’s inconsistent). I assume this is a power-saving feature, but it leads to very inconsistent latency.

Ideally, you should keep this off or find some way to configure your router to mitigate this issue. However, I found this quick fix made a significant improvement in latency on my phone hotspot.

RemoteManager fields

bool doPoking

This allows you to enable or disable poking. This should work at runtime.

float pokeInterval

The time in seconds to wait between each poke. Around 0.2 seems to work well for me. Don’t set this too low, or you will start to get performance issues.

bool debugKeysEnabled

This enables or disables debug keys bound to certain functions in the system. e.g. F9 to start identification UI.

If you’re writing a script and want to make use of this, there’s a public static property variant RemoteManager.DebugKeysEnabled. e.g.

```
if (RemoteManager.DebugKeysEnabled && Input.GetKeyDown(KeyCode.F7)) {...}
```

RemoteIdentificationHandler

The RemoteIdentificationHandler handles the UI flow for configuring which RemoteObjects are linked to which RemoteDevices. You can start the setup flow with the public RemoteIdentificationHandler.Begin() method. With debug keys enabled, you can also press *F9*.

There are two main phases to the identification UI.

1. **Scanning** - Every IP address on the local network is pinged to find all devices. Each found IP is then sent a /ping/ message through the Remote Object System to find if it is a remote device. The UI shows found IPs with a question mark, and then shows a tick for confirmed remote devices.

   The script uses your local IP address to find the range to scan for IPs in - and will only scan from 1-255 on the final IP number. If your IP is 192.168.4.100, the scanned IP range is from 192.167.4.1 to 192.168.4.255 (excluding your own IP).

   There is no way to be absolutely certain all devices have been found, and there aren't still some left that just have very high latency. Thus, the scanning phase runs for a set amount of time (see IPSweepTimeout below). If this is set to 10 seconds, it will *always* run for 10 seconds. For testing, you can set this very high and then use the *Skip* button when all have been found.

   Very likely problem you'll bump into - *Most firewalls block incoming connections by default. If the correct IP addresses are coming up in the list and your device is logging that it's received the message and sent a pong, try disabling your firewall.*

2. **Linking** - Now that the confirmed *RemoteDevice* list is ready, a button is shown for each *RemoteObject* in the scene, and you are asked to link each *RemoteDevice* to its corresponding *RemoteObject*. For each *RemoteDevice* being linked, its IP address is shown, and an /id/ message is repeatedly sent to it - at the moment that means it will play a test sound.

int IPSweepTimeout

The number of seconds to wait after pinging every IP address on the subnet. This may need to be higher than you would expect - I've found giving around 10 seconds *just* manages to catch everything.

float IdentifyRepeatRate

The rate at which devices are pinged in the linking phase. This is basically how often they will play a sound to indicate which device you are currently assigning.

bool Search On Start

Launches the UI flow in Start()

bool Pause Timescale During UI

Sets Time.timeScale to 0 during the UI, then resets it to 1 afterwards. This can have issues if your game uses Time.timeScale in its own way.

bool Don'tSkipIPs

Normally, each IP is sent a /ping/ message and then only those that indicate they are RemoteDevices by returning a /pong/ message are accepted, with all other IPs discarded. If Don't Skip IPs is enabled, ALL IPs are let through, including those that have not responded with a /pong/ message.

UI references

There are five serialised fields for UI elements. These are preconfigured in settings and you shouldn't need to touch them, unless you're implementing your own UI. I recommend simply tweaking the existing UI to your liking, but either way here's what each reference refers to:

- **UI IP List Panel** - The parent UI panel on which new IP address entries are shown.
- **UI IP Prefab** - The prefab to be created in the UI IP List. Must have a RemoteIdentificationIPUI component.
- **UI Object Panel** - The parent UI panel on which each RemoteObject is shown.
- **UI Object Prefab** - The prefab to be created in the UI Object Panel. Must have a RemoteObjectIdentificationUIItem component.
- **UI Heading Text** - The main UI Text (TMP) to show the current status.

I haven't provided explanations on how to configure every element of the UI, so please see the existing prefabs for an example of how these should be implemented.

Remote Components & Modules

RemoteComponent (parent class)

All *RemoteComponent*s require a *RemoteObject* also to be attached to the *GameObject*.

Properties

protected string moduleKeyword

This is the name of this module as referred in slash commands. In other words, this is the first argument of the slash command. For RemoteAudioSource, this is set to "audio", meaning slash commands will always start with:

/audio/...

As you'll see in the template, this MUST be set in your implementation of *RemoteComponentAwake*. If you do not, your commands will not run correctly. You can technically modify it in realtime to "switch" modules, although this is not recommended and untested.

protected RemoteObject remoteObject

The *RemoteObject* this *RemoteComponent* is attached to. This is retrieved using *GetComponent* in *Awake*.

Avoid using *GetComponent* to retrieve the *RemoteObject* in your implementations.

protected bool fallbackMode (read-only property)

Returns *remoteObject.fallbackMode* (see above for more information).

Methods

protected RemoteComponentAwake()

This runs at the end of *Awake*, and any code placed here in your implementation can safely assume that references like *remoteObject* are set up.

ActivateFallback()

Manually activates fallback mode on this module ONLY.

DeactivateFallback()

Manually deactivates fallback mode on this module ONLY.

Components' ActivateFallback/DeactivateFallback methods generally shouldn't be used, even with autoFallbackMode disabled - Please look into RemoteObject.UpdateFallbackMode() if you are implementing a custom system using Fallback Mode.

SendCommand(string func, string[] args)

Sends a command using the given function and arguments.

e.g. (from *RemoteAudioSource*)

> *SendCommand("play", sound.AsArgs());*

For single-argument commands, you should **NOT** use e.g.

> *SendCommand("volume", new string[] {0.5f.ToString());*

Instead, use the following method...

SendCommand(string func, string arg)

It's quite common you'll only need one argument. This implementation will automatically handle converting *arg* into a single-element string array. This should clean up this scenario quite a lot compared to the above implementation.

e.g. (functionally equivalent to above)

> *SendCommand("volume", 0.5f.ToString());*

RemoteComponentTemplate and Implementing Custom Components

You can write your own RemoteComponents. These usually would utilise custom modules you've written for the Python side of the codebase, which is explained below.

In nearly all cases where you are using the existing modules (*audio* and *arduino*), you should simply write a standard *MonoBehaviour* script which references a *RemoteAudioSource* and/or *RemoteArduino* component, as shown in the examples and setup guide. However, there may be some extraneous cases where you find it useful to write your own components using existing modules, so I will provide some direction here on what to do in this case.

In *RemoteObject > Scripts > Components*, you'll find *RemoteComponentTemplate.cs*, which gives you a starting point for creating your own RemoteComponents.

Above in the RemoteComponent reference, you'll find details about different properties and functions available from a RemoteComponent-derrived script.

Here are the basic steps to creating a RemoteComponent:

1. Configure the *moduleKeyword*

This is the first word slash commands will use for this component. If you've made your own module, this should be your module's name. Audio uses "audio" (/audio/...)

and Arduino uses “arduino” (/arduino/...). You can change this in realtime but I strongly recommend avoiding this.

2. Implement functionality

Add whatever functions you would like to make your script work. In the case of Audio this is things like Play(). You can use coroutines and so on as normal. Make sure you use RemoteComponentAwake in place of the normal Awake function - Start can be used as normal. If you need to reference the attached RemoteObject, use *remoteObject*, you should never need to GetComponent for this.

When you’re ready to send a command to the *RemoteDevice*, use *SendCommand*. This takes a string for the function name, and then either a single string or a list of strings for the argument(s).

e.g. if your moduleKeyword is “*example*”;

- *SendCommand(“foo”, “bar”)*
  would send the command **/example/foo/bar/**
- *SendCommand(“foo”, new string[] { “bar”, “1”, “two” })*
  would send the command **/example/foo/bar/1/two/**

You can use RemoteAssets (described below) to simplify command calls when working with assets, especially when also using fallback mode.

3. Fallback mode

You’ll notice in the template premade functions for ActivateFallback and DeactivateFallback - these are called when fallback mode is enabled or disabled. In these you should perform the setup to prepare for switching modes. You can also leave them blank and your script should continue to work fine.

You can access the bool fallbackMode from anywhere in your script, which can be used to add fallback functionality.

RemoteAudioSource

RemoteAudioSources are designed to be used as a replacement for Unity’s AudioSources. They allow you to play audio clips through remote devices.

While I say this is designed to somewhat replicate a standard AudioSource, there are inherent differences in how the RemoteComponent version must work - it’s generally far more basic, and I’ve taken intentional liberties to make it very straight-forward to prototype and implement into existing projects easily.

Please note: Whilst latency can consistently be quite good in the right scenarios, it is not 100% consistent, so this component is not suitable for extremely timing-dependant scenarios. If you want to play a stereo or surround music track using distributed audio this is not the system for that. If you do want to play music or anything in stereo then I suggest using one device with a stereo output instead of trying to get two devices to play the left and right channels separately at the same time.

Setup

For this module to work, you'll need to ensure a sound library is configured on your devices and it contains the wanted files, which should in WAV format. Currently, there is no system to transfer audio files to remote devices automatically.

By default, sounds should be included in a folder named *sound_library* within the same directory as the main *remote_pi.py* script.

Important: sound_library **MUST** contain a file named *test.wav*. This will be played when the device is being identified (See *RemoteIdentificationHandler*). The default sound is very annoying so I recommend changing it :)

In future I hope to allow these features to be more configurable in the config.ini.

Sounds are played on devices through Pygame, a library for game development tools in Python. Any format Pygame supports should be workable long-term, although currently only WAV is tested working (mp3 does NOT currently work).

Properties

float volume (read only)

The default volume for this *RemoteAudioSource*. This is a value from 0.0 to 1.0 that represents how loud sounds will be played.

Note: Use SetVolume(float) to set the Audio Source's volume.

Methods

Play(RemoteAudioClip clip)

Plays the specified *clip* through this *RemoteObject*. You will need to set each *AudioClip* up as a *RemoteAudioClip* (see below). Also note that the specified clip must be included on the linked *RemoteDevice*'s sound library.

RemoteAudioClip

This is a RemoteAsset (derrived from ScriptableObject) that you'll need to configure for each clip you want to play remotely. It defines an audio clip's name on the remote device and associates it with an AudioClip.

You can can create a RemoteAudioClip in assets from the create menu under "RemoteObjectSystem".

AudioClip clip

The AudioClip for this RemoteSound. Ideally, this should be identical to the sound on the remote device(s). This is used to play the clip locally when a RemoteObject is in fallback mode. At some point, I also hope to allow automated file transfers.

string clipName

The name of the clip on remote devices. Do NOT include the file extension. This must be **identical** to the name of the sound files in your remote device's sound library.

Troubleshooting tips

- This system has only been extensively tested with WAV files, so I highly recommend using this format.
- Make sure the clip name on the RemoteSound asset is an EXACT match. This is case-sensitive.
- You currently cannot name multiple sounds identically. If you have multiple clips in your clips folder with the same name but different file extensions, then it is uncertain which of these will be chosen.
- If you plug a display into your remote device, you may find the audio outputs through the HDMI port rather than through the 3.5mm jack. You should be able to change this in your device's settings - however, a workaround for some devices is to keep the HDMI unplugged until the device has booted and started playing sounds. I hope to eventually provide a more elegant solution to this issue.

RemoteArduino

RemoteArduino allows you to directly control output pins on an Arduino (only tested with an Arduino Uno, but should be universally compatible). This is great for controlling LEDs, servos, and more. Note that, currently, there is no support for reading output pins (although you can remotely configure each pin's mode).

Setup

For this module to work, you will need to upload lekum's pyduino sketch to your Arduino, available here:
https://github.com/lekum/pyduino/blob/master/pyduino_sketch.ino.

This sketch configures your Arduino to receive commands through the serial port. Once this is installed, plug your Arduino into one of your Raspberry Pi's USB ports, and you should be good to go. Make sure you restart the remote_pi script if it's already running, as the Arduino module is only set up when the script first starts.

Methods

DigitalWrite(int pin, int value)

Writes the specified pin with the specified value. The *value* may be either 0 for LOW or 1 for HIGH. Any values outside of this range may have unexpected results.

SetPinMode(int pin, RemoteArduino.PinMode pinMode)

Sets the specified pin to the specified mode. This can be Input, Output or InputPullup. The script is set up such that all pins should be configured as outputs by default.

There is currently no way to read data from an input pin, so there's limited utility to this method.

Troubleshooting tips

- There is a chance you may have issues getting the Pi to pick up the Arudino. Try changing USB ports and restarting the script. I was surprised to have it work first-try every time for me, so I can't give much more advice beyond that.
- If it seems like the Arduino isn't giving out enough power (e.g. weak LED brightness, no functionality), there's a chance some pins are not set to output. Use SetPinMode to ensure that each pin you're using is set to output.

Note: *This module has no fallback mode.*

Draft modules (not functional)

There are a couple of modules that have classes but with no implementation. These have been commented out and may be implemented fully at a later date.

- **RemoteGPIO** - this would allow direct manipulation of the Pi's GPIO pins.

    - This has been shelved for now as the Arduino implementation is simpler and theoretically more powerful for a GPIO system.
- **RemoteInput** - this is a broad module that would allow input to the Pi from various sources (USB controllers, buttons, mouse/keyboard, etc.) to be read by the host PC.
    - Note: Currently, the networking backend is just shooting signals from the host PC to devices, so getting this to work would require an additional network connection, and I chose to avoid this and instead focus on the unique output possibilities for now.

RemoteAsset

This is a simple ScriptableObject-derived abstract class designed to formally link assets on the host PC with assets on your devices.

In future, it may include functionality to send assets directly to devices, although this is not currently possible.

Currently this is only used for *RemoteAudioClip*, however you're free to implement your own version for whatever assets your project might need, e.g. RemoteSprite, etc.

The ***AsArgs()*** function returns an array of strings to be used to reference the asset from a command call. For example, with *RemoteAudioClip*, this is just the clip name. You'll need to implement this yourself. See the existing *RemoteAudioClip* class for an example of how this works.

Remote Device (Python) Documentation

The Python codebase is currently a lot less polished than the Unity implementation. In general, you'll only need to tackle this section if you're trying to use something other than Unity to control devices or you're writing your own modules.

Controlling devices with other applications should be fairly straightforward, given a good understanding of networking. I will also provide instructions for anyone keen to create their own modules; however, the format for a lot of this may dramatically change, and your modules may break in future.

Networking

Devices running the remote package respond to UDP messages directed to their IP on port 32019 or an alternate port specified in config.ini. They should be on the same LAN - technically, you might be able to get it working over the internet (obviously with much higher latency), but this is not at all the intended use and has not been tested. I also have not considered security at all in this system, so tread carefully if you go down this route.

Modules & Commands reference

Parameters denoted with a * are optional. The final / on a command is optional and will make no difference to the result.

At this stage, the script may crash when incorrectly formatted commands are received, so be careful. Some commands are more volatile than others.

Global commands

/poke/

Does nothing - however, the system receives it and will print a message to indicate this. This is used for the poking system (see RemoteManager in the Unity documentation above).

/ping/[source_ip]/[source_port]/

Ping message. Sends a return /pong/ message to the given IP and port. Note the /pong/ does not include an IP or port - you should be able to get this from the receiver's socket data.

To be honest, I could definitely have avoided the need for including the source IP and Port in the command and may alter this in future, although keeping the option to manually specify a return address.

If you're having issues receiving the pong message from this command and the logs all look right, check your host PC's firewall, as it may block incoming connections by default.

/id/

"Identifies" this device, e.g. for pairing purposes. Each module's /id/ command is called. At the moment, all this will do is play a test sound from the audio module.

/audio/

This module allows you to play sounds from remote devices. This corresponds with the RemoteAudioSource Unity component.

The Python library PyGame is used to play sounds. In the config file you can set the SampleRate and Buffer as you like.

/audio/play/[sound_name]/[*volume]/

Plays the specified sound from the sound_library. Do *not* include the file extension. Volume is a decimal number from 0 to 1 to set the volume for the clip to play at. If the volume argument is not included, it will play at the default volume.

/audio/volume/[value]/

This sets the default volume level for sounds to play at.

/audio/id/

Plays the sound named "test". Equivalent to /audio/play/test/

/arduino/

This uses the pyduino library to send commands to a connected Arduino.

/arduino/dwrite/[pin]/[value]/

Sets the given pin number to the given digital value. The value should be 0 or 1 - corresponding to LOW and HIGH, respectively.

/arduino/awrite/[pin]/[value]/

This command has not yet been implemented.

/arduino/pinmode/[*pin]/[*mode]/

Sets the given pin number to the given pin mode. The mode value can be:

- 'O' for Output
- 'I' for Input
- 'P' for Input Pullup

There's not much utility to anything other than output at the moment. If no mode argument is provided, the default is output. With no arguments, every pin is set to output.

/arduino/id/

No implementation.

Writing a custom module

This is an area that's likely to change significantly over time, as I've realised there are some elements of how this is implemented that won't scale well. So whilst I am providing advice on how to create a custom module, it's likely the backend of this whole system will dramatically change in time and I can't guarantee modules will work for many iterations.

Setting up your module

In modules.py, you'll find both of the existing modules along with a couple draft stubs of some other modules. You can add to this script a class derived from remote_module. This parent class is very simple, you'll just need to make sure you include the following:

- **__init__(self, cfg)**
  - This is the constructor, as with any Python class. Include your setup code here. cfg is a copy of the parsed cfg file which you can use to set up your module, for an example of how to do this see my implementation in rmod_audio.
- **parse_command(self, args)**
  - This is the main input for a module. When a command is received that calls for your module, parse_command will be called. Args is the list of arguments, including the function name.
  - You'll see in my implementation, I use my parse_command to check args[0] for the command name and divert to a corresponding

function, providing the rest of the arguments as args[1:]. I plan to make this similar to the way all modules work by default in future.
  - Example: The command “/audio/play/snare/” is received. The main script will call the audio module’s parse_command function with [“play”, “snare”] as arguments. The audio module’s class sees the first argument is “play” and calls its play function with [“snare”] as the argument.
- **id(self)**
  - This can be used implement some basic functionality that identifies the remote device. For example the audio module plays a test sound. You can just pass this if you don’t want your module to make use of this.

Other than that, you can add whatever you want and set up your module however you’d like. You can also import more libraries to the modules script if needed.

Adding your module to the main script

This is the most borked bit about how the system works at the moment. In remote_pi.py you’ll need to create the module and add it to a modules list.

Around line 42 you’ll find the existing module setup. You’ll need to create your module and then add it to the MODULES dictionary under the string for the command name for your module. For example:

```
foo = modules.your_module_class_name(cfg)
MODULES = {
   … [existing entries],
   “bar” : foo
}
```

This would be an appropriate setup for if you want to call your commands as: /bar/[function]/[args]

Setup Guide & Example

What you’ll need

- A host PC with Unity installed (Tested on 2021.3.17f1).
- One or more Raspberry Pis (other microcomputers may also work but have not been tested)

- Display, keyboard, and mouse to interact with Raspberry Pi during setup. (Alternatively, you can use SSH if you prefer)
- Wi-Fi router, such as a phone hotspot
- For audio
    - 3.5mm speaker (and cable) to output sound from the Raspberry Pi (wired headphones will work in a pinch). You may also be able to get HDMI audio working.
- For Arduino control
    - An Arduino

Setting up the Raspberry Pi

Download the Raspberry Pi package from here. Transfer it to your Raspberry Pi, and extract it somewhere convenient.

You'll need to run the file ***remote_pi.py*** using Python. If you're doing this from the command line, make sure your current working directory is in the root folder of the extracted package. You will likely need to use PIP to install all required dependencies.

Also make sure **the Raspberry Pi is connected to the same network as your host computer.** You can use a phone hotspot, your normal Wi-Fi, or any other connection. Currently only Wi-Fi has been tested, although you can change the default network adaptor in the config.ini.

For most applications, it'll be convenient to run the Pi headless. There are a number of ways to do this that differ depending on your usecase, for example using RC.LOCAL or other solutions to have the script run on startup, or using SSH to launch the script remotely.

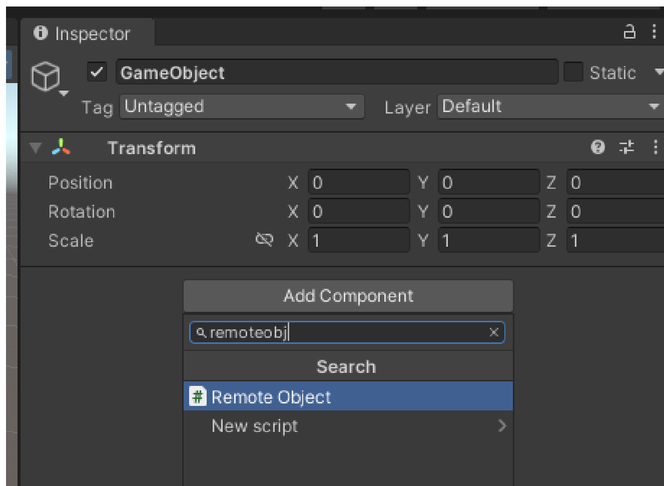
The log of the script is quite verbose, so it can be convenient to keep it running in a command window while testing the system out.

If when you launch the script there is an error about a port already being used, this may be because the script is already running in the background, or did not quit correctly. Make sure the script isn't running on startup, and reboot the device.

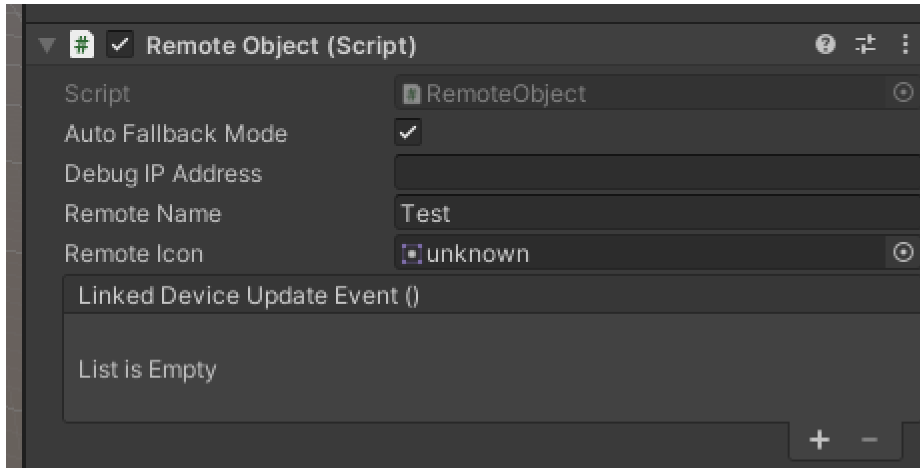
To exit the script, use the keyboard interrupt key, Ctrl+C.

Setting up the Unity Project

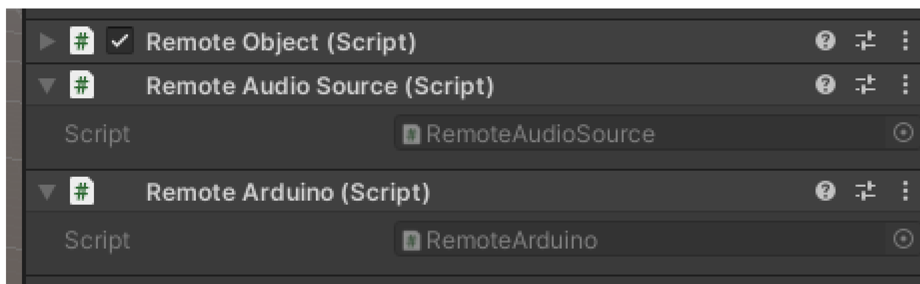
1. Create a new Unity project. To be absolutely certain of compatibility you'll want to use 2021.3.17f1 but the closest version you have installed should usually work fine.
2. Download and the Unity package and import it into your project.
3. Once the package has imported, navigate to *RemoteObject > Prefabs*. Drag a ***RemoteManager*** into the scene.
    - You may get a message about TextMeshPro, which is used for some of the default UI elements. Click Import TMP Essentials and close this window.
4. You will also need to add an *EventSystem* so that the UI works. Add a *UI > EventSystem* to the Hierarchy.
5. Add a GameObject to the scene.
6. On this GameObject, add a RemoteObject script.



7. On the RemoteObject script, set a name and an icon (you'll find a sprite called "unknown" from the RemoteObjectSystem package). Leave the other options as default.



8. Depending on which you want to use, add a *RemoteAudioSource* or a *RemoteArduino* script (I recommend implementing one first before trying both simultaneously).

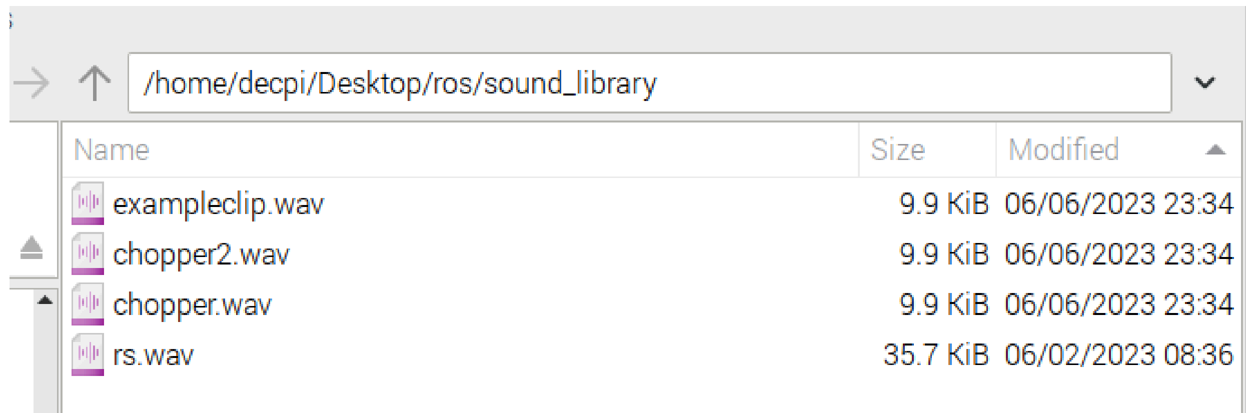


The following steps will differ between RemoteAudioSource and RemoteArduino.

Using audio

For this example we'll make the remote object play an audio clip when the Space key is pressed.

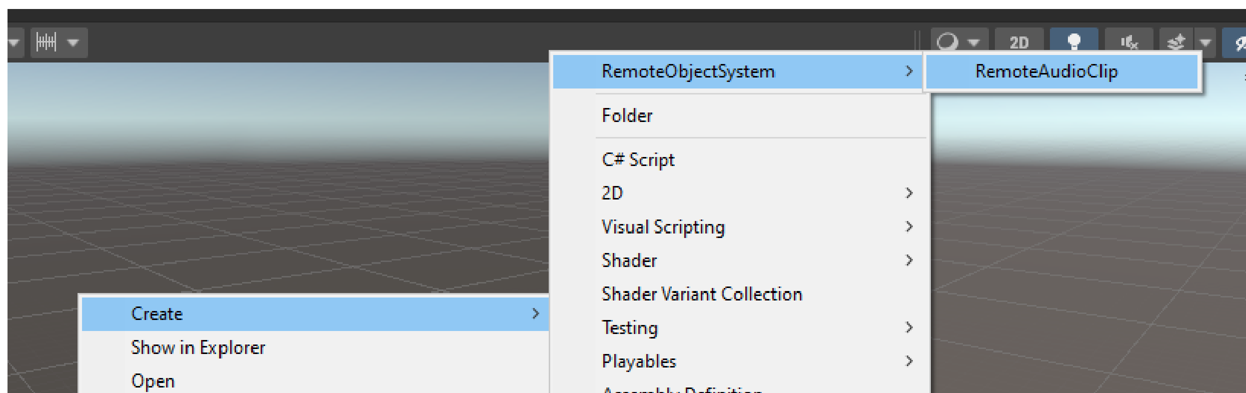
The first thing you'll need to do is make sure that you have the audio file you want to play (wav format) in the sound_library folder on the Raspberry Pi.



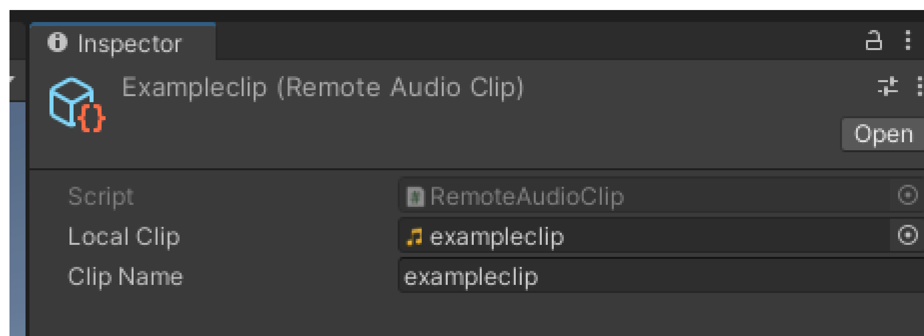
Take note of the name you've given your sound. For example in this case my sound is named "exampleclip".

Import the same audio file into your Unity project as normal (e.g. dragging it in).

Create a RemoteAudioClip by right clicking in the assets panel and selecting *Create > RemoteObjectSystem > RemoteAudioClip*.



On the RemoteAudioSource, set the Local Clip to the audio file you just imported. The Clip Name should EXACTLY match the name of the clip on your remote device - **do not include the file extension**.

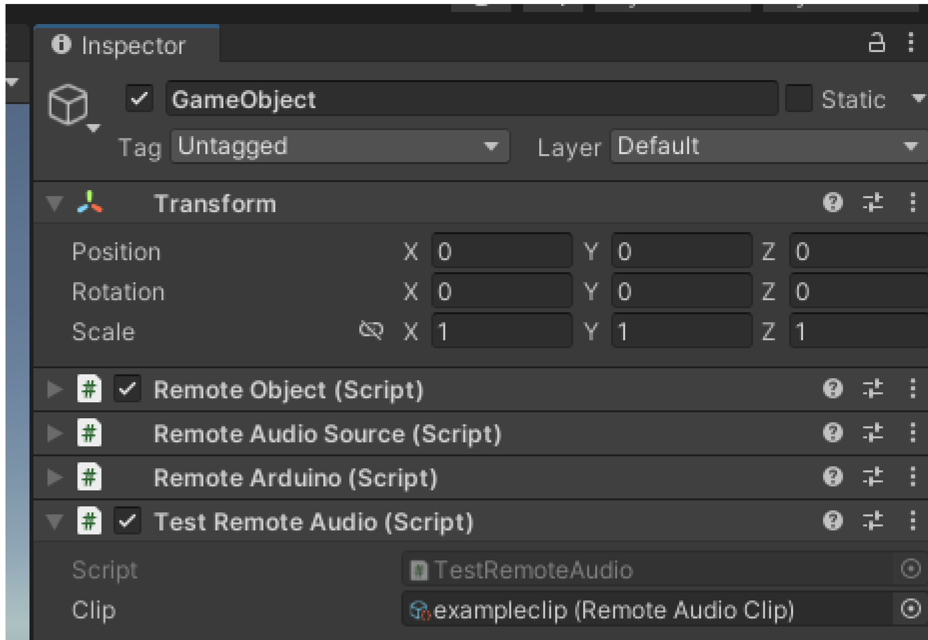


Now we'll create the script. Add a script to your assets. This script will go on the GameObject we added

See the following example code for a script that plays an audio clip when the Space key is pressed. This should be pretty straightforward - we get the RemoteAudioSource attached to the object and then play a clip from it when the space key is pressed.

```
using UnityEngine;

public class TestRemoteAudio : MonoBehaviour
{
    // A reference to the audio source of the object we want to play sound from.
    RemoteAudioSource remoteAudioSource;
    // A reference to the clip we want to play.
    [SerializeField] RemoteAudioClip clip;

    private void Awake() {
        // Get this object's RemoteAudioSource
        remoteAudioSource = GetComponent<RemoteAudioSource>();
    }

    private void Update() {
        // When the space key is pressed, play the clip.
        if (Input.GetKeyDown(KeyCode.Space)) {
            remoteAudioSource.Play(clip);
        }
    }

}
```

Add this to your GameObject with the RemoteAudioSource on it. Add the RemoteAudioClip asset we just set up as the clip.



When you press play, you should find that when you press space, the audio clip plays from your computer's audio. This is because we haven't linked the RemoteObject to a device and it is has automatically been placed in *fallback mode*. (You can disable auto fallback mode from the RemoteObject if this is not appropriate for your project).

Continue on to "Connecting with the Raspberry Pi" below.

Using Arduino

We'll create a simple script that turns on and off the LED in port 13 (which should have an associated test LED on the board itself). This avoids having to use a breadboard at this stage.

The first thing you'll need to do is set up your Arduino. Get lekum's Pyduino sketch from here: https://github.com/lekum/pyduino/blob/master/pyduino_sketch.ino. Upload this to your Arduino board.

Connect your Arduino by USB to your Raspberry Pi. ***You'll need to restart the remote_pi.py script for it to register the Arduino has been connected.***

Back in Unity, we'll create a new C# script for testing the Arduino.

See the following example code - this will turn the LED at port 13 on when the 1 key is pressed and off when the 0 key is pressed.

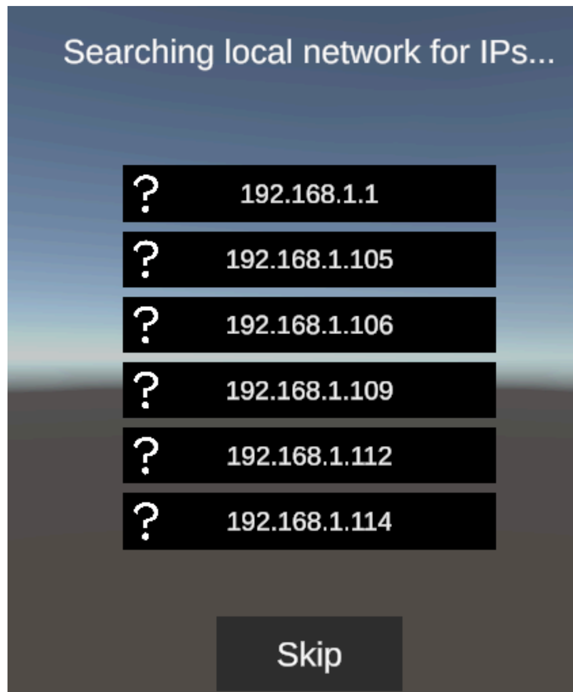
```csharp
using UnityEngine;

public class TestRemoteArduino : MonoBehaviour
{
    // A reference to the object controlling the Arduino.
    RemoteArduino remoteArduino;

    private void Awake() {
        // Get this object's RemoteArduino component.
        remoteArduino = GetComponent<RemoteArduino>();
    }

    private void Update() {
        // Make sure pin 13 is configured for output.
        if (Input.GetKeyDown(KeyCode.R)) {
            remoteArduino.SetPinMode(13, RemoteArduino.PinMode.Output);
        }
        // When the 1 key is pressed, turn the LED on.
        if (Input.GetKeyDown(KeyCode.Alpha1)) {
            remoteArduino.DigitalWrite(13, 1);
        }
        // When the 0 key is pressed, turn the LED off.
        if (Input.GetKeyDown(KeyCode.Alpha0)) {
            remoteArduino.DigitalWrite(13, 0);
        }
    }
}
```

Add this script to the GameObject you already added the RemoteArduino script to earlier.

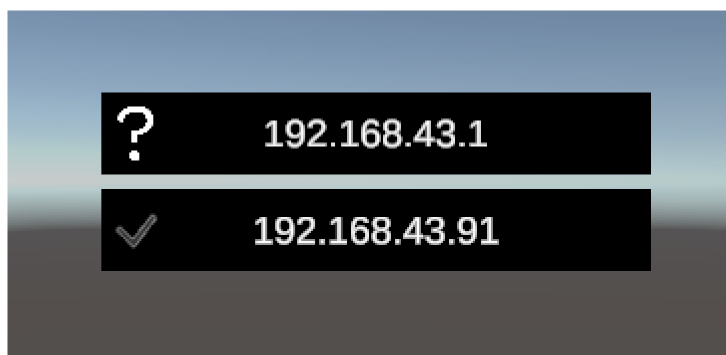
Connecting with the Raspberry Pi

Now we'll connect the game to your Raspberry Pi. Make sure your Pi is running the remote_pi.py script as described above. Also ensure both the Pi and your PC are on the same network.

In play mode, press the F9 key (if nothing happens, check your Remote Manager has Debug Keys Enabled set to true). A screen should appear that slowly fills with all IP addresses found in the network.



One of the IPs should should show a tick on it - this is confirmed to be your Raspberry Pi.

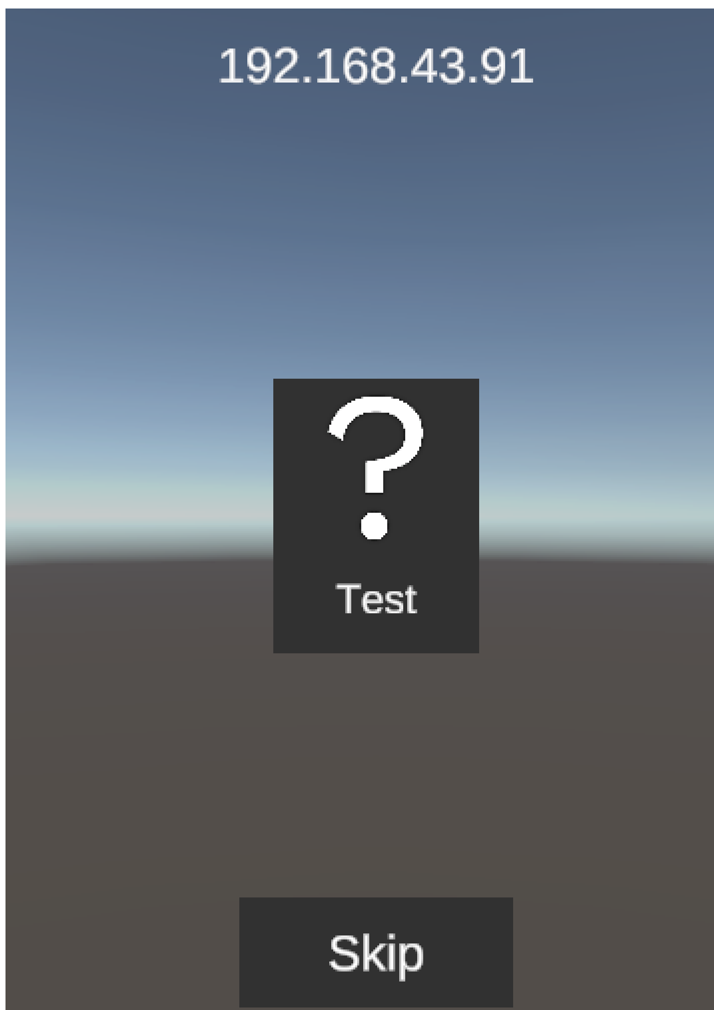


Troubleshooting: If no tick shows, here are some things to check;

- Disable or reconfigure your firewall.
- Check your Pi's Wi-Fi connection - if the signal is weak, you might need to move your Pi or use a Wi-Fi hotspot.
    - Also check the Pi's command line log - there may be a message about not being able to retrieve the IP address. This means you do not have a stable connection to a network.
- Some corporate Wi-Fi networks may not work with this system, although most home modems will work fine. Your Pi's IP address should have the same first 3 parts as your PC's IP address (e.g. 192.168.4.xxx)

- It could be a case where there is high network latency and your device is not being pinged fast enough. Increase IP Sweep Timeout on RemoteManager - this is the number of seconds the system waits for a response from each IP.
- If all else fails, check if your Pi's IP address is showing up in the list at all (remote_pi.py should print the device's IP shortly after starting). If it is, you can check "Don't Skip Ips" on the Remote Manager. This means you'll have to skip through each IP to get to your Pi's IP, but it should work as a last resort.

Assuming you have found your Pi, the next screen will show the IP address of your Pi and a button representing your RemoteObject in the Unity scene. If you have a speaker hooked up to your Raspberry Pi, you'll also hear it playing a test sound repeatedly. Click on the button with your RemoteObject's name on it to link the device and Object.



You should now find that the functionality we wrote earlier outputs to the Pi.

- **Audio;** Pressing Space will play the sound from the Pi

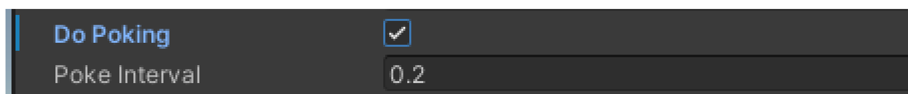
- **Arduino;** First press R to ensure pin 13 is set as output. Then press 1 or 0 to turn the onboard LED on or off.

From here you should have a basic foundation on how to make your own creations using the Remote Object System!

Troubleshooting

If it is not working, check your Pi's command line to see if there is an error. Also ensure that all connections (3.5mm cable to speaker, USB cable to Arduino, breadboard wiring) are connected properly.

You may notice some latency if you're using a phone hotspot or other low-power device. In Remote Manager, enable Do Poking to improve latency consistency.



Be absolutely certain your device and the host computer are using the same Wi-Fi network and are in the same subnet (i.e. first 3 fields of the IP address are identical). Again, double-check your firewall - I find even after manually allowing the Unity editor, I need to completely disable it to have the signals go through (this should be less of an issue for a built executable).

If your device randomly stops working, check it hasn't changed Wi-Fi network on its own. I've had this happen often, even from a 5 bar signal to a 1 bar signal. For now, the solution is to forget all Wi-Fi networks other than the one you're using.