The RedRat-X

Integration Guide

Contents

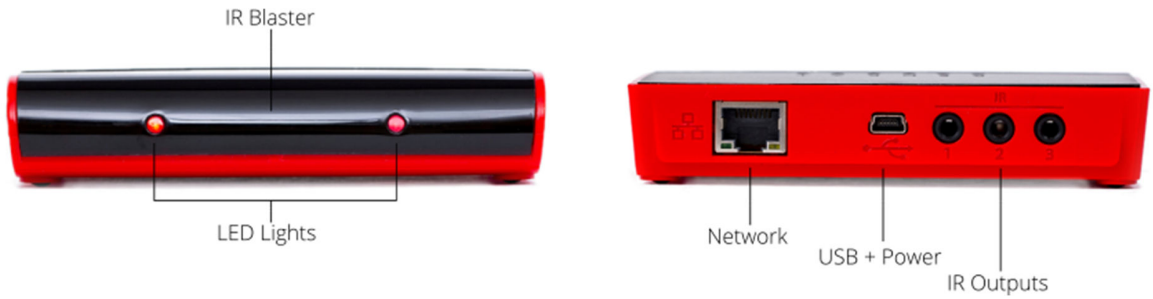
1 Introduction 3
2 Overview of the RedRat-X 3
2.1 Front 3
2.2 Rear 3
3 RedRat Applications 4
3.1 RedRat Device Manager 4
3.2 Signal Database Utility 4
3.3 RedRat Scheduler 4
3.4 TestManager 4
3.5 RedRat Hub 4
3.6 RedRat Control 5
4 Platform Support 5
5 RedRat Hub - Sending Commands from Applications 5
6 Using the RedRat SDK 6
6.1 RedRat-X Discovery 6
6.2 Simple Communication Tasks 6
6.3 IR Signal Output 7
6.3.1 Using the Front IR Blaster 7
6.3.2 Using the Rear IR Ports 8
6.4 IR Signal Input 8
6.4.1 The Wide-Band Detector 8
6.4.2 The Narrow-Band Detector 9
6.5 Handling USB Plug Events 10
7 Porting RedRat3-II Applications to the RedRat-X 11
7.1 USB Device Discovery 11
7.2 RedRat-X Connect and Disconnect 12
8 Using RF Modules 12

1 Introduction

The RedRat-X is the latest RedRat device for automating control of TVs, STBs and other audio-visual equipment. It combines features of both our irNetBox networked product and the USB RedRat3-II device, plus provides additional support for Bluetooth and RF4CE remote controls through the use of the correct RF module.

Whether you are a new RedRat user or need to replace the use of older RedRat devices with a RedRat-X, this guide is intended to help integrate it into your software infrastructure, or to write new software applications from scratch.

2 Overview of the RedRat-X



2.1 Front

IR Blaster: A high-powered IR blaster, positioned behind the centre of the front panel. The power output is controllable by software in steps from 0 (off) to 100 (max).

LEDs: Multi-coloured LEDs indicate the operational state of the unit, please see the *User Guide* for details. The LED colour and brightness are also adjustable via software control, so applications can use this to indicate the state or provide feedback.

2.2 Rear

USB: The RedRat-X is powered via its USB port. This can be connected to a computer, and in which case USB will also be used to control the unit.

Network: Ethernet control can be used in a very similar manner to the irNetBox. The device needs to be powered via the USB port using an external PSU. For full EMC compliance (CE and FCC), RedRat are able to supply a PSU which has been tested with the RedRat-X.

IR Output Ports: Three IR output ports are available for use with stick on IR emitters for individual control of STBs. The output power for each port is separately controllable, from 0 (off) to 100 (max). Each port can be set to operate in current drive mode (default for driving IR LEDs) or in voltage drive mode (for interfacing with IR distribution systems).

3 RedRat Applications

We provide a number of applications for use with RedRat devices which meet the requirements of many RedRat customers. An overview is provided here, but more detailed information about each application can be found on our website – www.redrat.co.uk.

Unless otherwise stated, these applications run on Microsoft Windows only (Windows 7 or greater) using the .NET Framework.

3.1 RedRat Device Manager

This is used for initial device installation, configuration and testing, for example to validate that communication between your computer and the device works and that it can then control the TVs/STBs required. The application should also be used periodically to check for firmware updates, which it can then automatically download from the RedRat website.

3.2 Signal Database Utility

RedRat devices are primarily designed for the input/output of infrared remote control signals, which need to be captured from the TV/STB's original remote control. The Signal DB Utility is used to do this, storing the IR data in files (XML format) for direct use in other RedRat applications and with the SDK. The IR data can also be exported in other formats for use in different applications or on different platforms.

3.3 RedRat Scheduler

If it is necessary to setup a set of remote control operations to happen at particular times, then the Scheduler application can be used to do this. Individual signals or short sequences of signals (macros) can be programmed to be sent at certain times or at regular intervals.

3.4 TestManager

A more advanced application for both interactive and script based control for banks of STBs.

With interactive control, graphical representations of STBs can be created and by simply selecting them, an on-screen remote control can be used to control them.

Script based control supports a simple in-built scripting language or python based scripting to automate repetitive procedures or tests.

3.5 RedRat Hub

A console/terminal server application which supports straightforward integration of RedRat devices with third party software. In effect, it looks after all communication with RedRat devices and IR signal management, accepting simple commands from client applications which are sent via a socket. Its use is discussed in more detail in section 5.

This application runs on Linux as well as Windows.

3.6 RedRat Control

Most of the applications described so far are intended for the output of IR signals to control TVs and STBs. RedRat Control operates differently in that it is designed to allow control of applications on a PC via infrared remote control. For example, if a media player is running, RedRat Control can be configured to respond to IR commands from a remote control handset and send appropriate instructions (play, pause etc) to the media player.

4 Platform Support

Historically, software for RedRat devices has been developed on Microsoft Windows using the .NET framework, so this is where there is still the best RedRat device support. However, many applications, particularly larger scale STB testing systems often use Linux, so we have now ensured that the RedRat-X is supported on Linux.

The RedRat core code (RedRat.dll) will run on both Windows and on Linux/Macs using Mono. Applications using the console/terminal window and not using any Windows-specific functionality will therefore also run using Mono. At present, Windows applications using a GUI interface (WinForms or WPF) will not run on Linux or Macs, though developing cross-platform GUI applications is a potential future area of development.

With the RedRat-X, both USB and network communication is supported on Windows and Linux.

5 RedRat Hub - Sending Commands from Applications

RedRat Hub provides perhaps the most straightforward mechanism for integrating RedRat devices with third party software.

When RedRat Hub starts, it loads any required IR datasets and then establishes communication with RedRat devices. Client applications then open a socket to RedRat Hub and send quite simple commands, such as:

```
ip="192.168.1.40" dataset="Sky+" signal="play" output="3"
```

This will send the IR signal *play* from the *SKY+* dataset via output 3 of the RedRat device found at the given IP address (irNetBox or RedRat-X).

```
name="RRX-1" output="4" dataset="SKY+" signal="play"
```

The second example sends the same signal via output 4 (blaster output) on the RedRat-X with the name “RRX-1”.

RedRat Hub is a very scalable application in that it can support a large number of client applications (in the 100s) concurrently sending commands to RedRat devices. It manages any resource contention, i.e. it will ensure that commands are sent to RedRat devices in a manner that they can handle.

Example client code is available for C#, Java, Python, Perl and PHP.

6 Using the RedRat SDK

The RedRat SDK is basically a .NET assembly (RedRat.dll) which contains all the low-level code for:

1. RedRat device discovery
2. Device communication, whether by Ethernet or USB
3. IR signal data management
4. Low-level RF module support

The API presented for application development is quite straightforward to use, and working with the RedRat-X should be quite familiar to those who have written code for other RedRat devices in the past.

RedRat.dll version 4.15 or greater is needed for full RedRat-X support. It runs on both Windows (using the .NET framework V4.5.1 or above) and on Linux (using Mono).

Most of the code examples shown below are taken from the *RedRatX-Demo* project in the RedRat samples solution, which can be downloaded from the SDK section on the website.

6.1 RedRat-X Discovery

Device discovery takes place by:

- Enumerating USB devices
- A network search using a UDP broadcast.

There are a number of ways to do this, or do selective searches, but perhaps the most straightforward way to find all RedRat-Xs is:

```
var rrxLis = RRUtil.FindRedRats( LocationInfo.RedRatType.RedRatX );
```

A list of generic `LocationInfo` objects are returned, which contain details about the RedRat device. This search will take a few seconds to return as it waits for responses from the network broadcast. If only USB devices are going to used, then it can be made almost instantaneous by limiting the search:

```
var rrxLis = RRUtil.FindRedRats( LocationInfo.RedRatType.RedRatX_USB );
```

Assuming that there is at least one element in the list, the actual RedRat-X object for the first element in the list is obtained so:

```
var rrx = RedRatX.GetInstance( rrxLis[0] )
```

The `rrx` object is of type `RedRatX` regardless of whether it is connected via USB or via the network.

6.2 Simple Communication Tasks

Before any operations with the actual device can take place, the code must first connect with the RedRat-X. This code below shows connecting to the RedRat-X, obtaining the firmware version, blinking the front LEDs 5 times and then disconnecting:

```
rrx.Connect();

// Use the RRX a bit
Console.WriteLine( $"Using RedRat-X '{rrxLi.Name}'" );
Console.WriteLine( $"Firmware version: {rrx.FirmwareVersion}" );

// Blink the lights a few times
Console.WriteLine("Blinking the LEDs...");
for ( var i = 0; i < 5; i++ )
{
	rrx.Blink();
	Thread.Sleep( 750 );
}

rrx.Disconnect();
```

6.3 IR Signal Output

The RedRat-X has four IR outputs, which in code are referred to as outputs 1 to 4:

Output Number	Function
1	Output port 1 on back of case
2	Output port 2 on back of case
3	Output port 3 on back of case
4	Front IR blaster

6.3.1 Using the Front IR Blaster

To make the API's use similar to that of the RedRat3-II, if no output ports are enabled and the simple synchronous `OutputModulatedSignal` method is used, this will transmit the signal via the front blaster.

```
// Output an IR signal via the blaster
rrx.OutputModulatedSignal( sig );
Console.WriteLine( "IR out via blaster using sync API" );
```

The async API can also be used where higher IR output throughput is required and concurrent IR transmission is needed from the rear IR ports and the blaster. The code below will output an IR signal 5 times from the front blaster:

```
// Using the ASYNC API, all ports need to be explicitly set
var opList = new OutputList { new OutputPort( RedRatX.IR_BLASTER_PORT, 60 ) };
var mutex = new ManualResetEvent(false);

for (var i = 0; i < 5; i++)
{
	rrx.OutputIRPacketAsync(GetSignal(), opList, 100, null, ar =>
	{
		Console.WriteLine($"IR out via blaster using async API.");
		mutex.Set();
	});

	mutex.WaitOne();	// Wait for IR output completion
	Thread.Sleep(750);	// Wait a bit longer
}
```

In the above example, the `RedRatX.IR_BLASTER_PORT` constant gives the port number for the blaster. The value 60 passed to the `OutputPort()` constructor is the power to use, in this case 60% of maximum.

Another point to note with the async API is that the `OutputIRPacketAsync()` method returns immediately so a mutex is used to block thread execution until execution is complete. The last parameter to the `OutputIRPacketAsync()` method is a callback that is executed on IR signal completion, and in this case is used to set the mutex so that the waiting thread can continue.

6.3.2 Using the Rear IR Ports

These work in an identical fashion to irNetBox output ports. The code blow uses the sync API to send an IR signal via output 2 at 30% power:

```
var opList = new OutputList { new OutputPort( 2, 30 ) };
rrx.EnabledIROutputs = opList;
rrx.OutputModuledSignal( sig );
```

Using the async API is identical to the IR blaster example, except the rear IR ports are selected rather than the front blaster.

6.4 IR Signal Input

As with the RedRat3-II device, the RedRat-X has two IR detectors for different purposes as described below.

6.4.1 The Wide-Band Detector

This allows the device to sample the raw IR signal, so is used for IR signal capture/learning. It is fairly short range in operation, the recommended distance being between 1m and 2m. The irNetBox also has this detector for IR learning/capture.

```
var mutex = new ManualResetEvent( false );
EventHandler learningSignalInHandler = ( s, ea ) =>
{
	var siea = ea as SignalEventArgs;
	if ( siea == null ) return;
	if ( siea.Action == SignalEventAction.MODULATED_SIGNAL )
	{
		Console.WriteLine( $"IR signal:\n{siea.ModulatedSignal}" );
	}
	else if ( siea.Action == SignalEventAction.INPUT_CANCELLED )
	{
		Console.WriteLine( "Timeout when waiting for IR signal." );
	}
	mutex.Set();
};

// Hook up the incoming event handler just for this operation.
rrx.LearningSignalIn += learningSignalInHandler;

// Set the RR-X up for IR signal input and allow 10s for the RCU button press.
rrx.GetModulatedSignal( 10000 );
mutex.WaitOne();
rrx.LearningSignalIn -= learningSignalInHandler;
```

6.4.2 The Narrow-Band Detector

This is exactly the same kind of detector as is used in TVs and STBs, tuned to a particular IR signal modulation frequency, but with a long range of 10m or more. The default value of this detector's frequency is 38KHz as this is the most common remote control carrier frequency, however we can replace it with a detector at another frequency if requested. The most common carrier/modulation frequencies are 36KHz, 38KHz and 56KHz.

The intended use of this detector is to allow control of the computer, or applications on the computer, via a remote control handset. For example, if running a media player or STB interface simulation software, "sit-back" control from a sofa can provide a better experience.

Writing code to use the narrow band detector is quite similar to the wide-band, except that the detector is turned ON for the duration of the period it is in use. During this period, any number of input events may be received, depending on how often the RCU buttons are pressed or what other IR noise may appear as data.

```
EventHandler rcDetectorSignalInHandler = (s, ea) =>
{
	var siea = ea as SignalEventArgs;
	if (siea == null) return;
	if (siea.Action == SignalEventAction.MODULATED_SIGNAL)
	{
		Console.WriteLine($"IR signal:\n{siea.ModulatedSignal}");
		Console.WriteLine("Hit <RETURN> to stop RC signal input.");
	}
};

// Hook up the incoming event handler just for this operation.
rrx.RCDetectorSignalIn += rcDetectorSignalInHandler;

// Start the RR-X listening until <RETURN> is hit.
rrx.RCDetectorEnabled = true;
Console.WriteLine("Hit <RETURN> to stop RC signal input.");
Console.ReadLine();

rrx.RCDetectorEnabled = false;
rrx.LearningSignalIn -= rcDetectorSignalInHandler;
Console.WriteLine("RC signal input terminated.\n");
```

6.5 Handling USB Plug Events

If using the RedRat-X via USB, then USB plug/unplug events can be used by the application to update itself if devices are unplugged or new devices plugged in.

A USB unplug/plug handler is a typical C# event handler, which can be set up a number of ways, for example an event handler method or using a lambda expression, as shown in the example below:

```
// Create local plug/unplug handler
EventHandler<UsbPlugHandler.UsbPlugHandlerEventArgs>
                                    usbPlugHandler = ( sender, args ) =>
{
        Console.WriteLine( "Plug event... " + args );
        if ( args.plugEventType == UsbPlugHandler.DevicePlugEventEnum.Added )
        {
                // The plug event gives us the serial number of the device,
                // so use this to find the RR-X that has just been plugged in.
                var xLis = RRUtil.FindRedRats( LocationInfo.RedRatType.RedRatX_USB )
                                                .OfType<RedRatXUsbLocationInfo>();
                var xLi = usbLis.FirstOrDefault(
                                li => li.SerialNumber == args.deviceSerialNumber );
                Console.WriteLine( $"Plugged in device is '{xLi.Name}'" );
        }
};
```

This is then wired up to the `UsbPlugHandler` object:

```
// Add plug/unplug handler
UsbPlugHandler.Instance.UsbPlugEvent += usbPlugHandler;

Console.WriteLine("Unplug + replug USB RedRat-X and then hit <RETURN>");
Console.ReadLine();

// Remove plug handler
UsbPlugHandler.Instance.UsbPlugEvent -= usbPlugHandler;
```

While the code is waiting for the user to hit <RETURN>, the RedRat-X can be unplugged and re-plugged a number of times to see the events as they are generated.

In a normal application, it is probably more useful for the plug/unplug handler to be setup at the start of the application and remain active for its lifetime.

7 Porting RedRat3-II Applications to the RedRat-X

Generally, the same interface is maintained across all RedRat devices, with additions to support new functionality as its added to newer products. This means that code developed for an older device should work with a newer one with minimal code modification.

When porting code developed for use with a RedRat3-II device to the RedRat-X, the primary modifications are in the way the device is discovered and the fact that explicit connect/disconnect steps are needed.

7.1 USB Device Discovery

The most straightforward method of finding RedRat USB devices of any type is using the RRUtil class:

```
var lis = RRUtil.FindRedRats( LocationInfo.RedRatType.USB );

if ( lis.Length == 0 )
{
	Console.WriteLine("No USB RedRat devices found.");
	return;
}
```

Where 'lis' is a list of `LocationInfo` objects, i.e. objects describing the device and its location.

Code developed for the RedRat3-II device will use the `IRedRat3` interface, which is also implemented by the RedRat-X. So by creating an object of this type, it can be used regardless of whether the underlying implementation is a RedRat3-II or RedRat-X:

```
IRedRat3 rr = null;
if ( LocationInfo.IsRedRatX( lis[0].RRType ) )
{
	rr = RedRatX.GetInstance( lis[0] );
}
else if ( LocationInfo.IsRedRat3( lis[0].RRType ) )
{
	rr = lis[0].GetRedRat() as IRedRat3;
}
else
{
	Console.WriteLine( "Unknown USB RedRat type." );
	return;
}
```

One point to note is that the way a `LoctionInfo` object is used to obtain the actual `IRedRat3` device object is slightly different. This is a result of the fact that RedRat-X code has to handle both USB and network communication mechanisms, so a bit of underlying complexity is hidden in the `GetInstance()` method.

7.2 RedRat-X Connect and Disconnect

Using the RedRat-X and RedRat3-II should be almost identical, apart from the connect/disconnect steps required when using the RedRat-X. This is also a result of the fact that the RedRat-X could be on the end of a network connection (like the irNetBox) so that a connection could be needed, even though in this case we know it's a USB device.

One simple way to introduce this into code is shown below:

```
if ( rr is IRedRatX )
{
	rr.Connect();
}

rr.Blink();
Console.WriteLine( $"Firmware version: {rr.FirmwareVersion}" );

if (rr is IRedRatX)
{
	rr.Disconnect();
}
```

8 Using RF Modules

The RedRat-X can host one RF module, currently either a Bluetooth (Classic and Low Energy) module or an RF4CE module.

As Bluetooth and RF4CE control of TVs and STBs is more complex than using infrared remote control, it is not done via APIs in the RedRat.dll, but using our "Toolbox" applications and HTTP REST API.

In addition, there is no generic standard for either Bluetooth or RF4CE in that each type of TV or STB has individual sets of advertising/discovery information, service tables, key report data etc. As a result, this information needs to be discovered or reverse engineered so that we can provide it as part of the solution for a particular STB, which usually means that we need to work with the STB before ensuring that we can support it.

Please contact RedRat (info@redrat.co.uk) for more information.