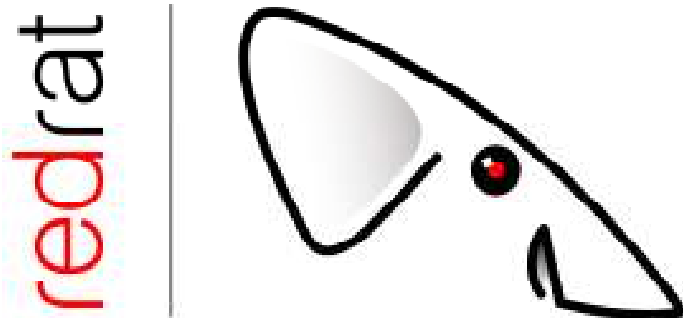


Using the RedRat API in .NET Applications

Chris Dodge – RedRat Ltd

25 May 2009

For the RedRat SDK V2.21



INTRODUCTION	5
PREREQUISITES AND SDK INSTALLATION	5
APPLICATION DEVELOPMENT AND DEPLOYMENT USING THE SDK	5
Runtime Versions	5
THE OBJECT MODEL OVERVIEW	6
AN INTRODUCTION TO REMOTE CONTROL SIGNALS	6
Modulated Remote Control Signals	7
Layer 1 – Main and Repeat Signals	7
Layer 2 – Signal Envelope	7
Layer 3 – Carrier/Modulation Frequency	7
IrDa-Like Signals	8
USING THE REDRAT API	8
Overview of the RedRat3s Functionality	8
Example Code Using the TestRemote Application	9
Finding a RedRat3	9
Discovering Information about the RedRat	10
IR Signal Input – Learning Mode	11
Outputting a Remote Control Signal	12
Program Termination	13
Using a Signal Database - The SignalDecoder Example Application	13
Loading a Signal Database	14
Using the Database to Decode Signals	14
More About Handling Incoming Remote Control Signals	15
Database Signal Lookup for Output	17
ADDITIONAL INFORMATION FOR THE DEVELOPER	17
RedRat Lookup	17
RedRat Type Discovery	18
Concurrency	19
PC Power Modes	19
Standby-Mode	19

Hibernation 20

API DETAILS 20

```
Interface RedRat.IRedRat 20
    event EventHandler LearningSignalIn; 20
    event EventHandler RCDetectorSignalIn; 20
    DeviceInfo DeviceInformation { get; } 20
    LocationInfo LocationInformation { get; set; } 20
    string FirmwareVersion { get; } 20
    void Blink() 20
    void GetModulatedSignal(int timeout) 20
    void CancelSignalInput() 21
    void OutputModulatedSignal(ModulatedSignal outSig) 21
    void OutputModulatedSignal(ModulatedSignal outSig, bool
    cacheData) 21
    double EndOfSignalTimeout { get; set; } 21
    int LengthMeasurementDelta { get; set; } 21
    int ModFreqPeriodsToMeasure { get; set; } 21
    double LengthToMilliSec(uint length) 21
    uint MilliSecToLength(double millis) 21
    ushort ModFreqToVal(double modFreq) 21
Interface RedRat.RedRat3.IRedRat3 22
    int MaxNumLengths { get; set; } 22
    int SignalMemorySize { get; set; } 22
    void OutputIrDaPacket() 22
    bool RCDetectorEnabled { get; set; } 22
    bool RCInputOneShot { get; set; } 22
Class RedRat.USB.USBDevice 22
    public static USBDevice GetInstance(String devName) 22
    public void CloseDriver() 23
    public string DeviceName { get; } 23
    public USBDeviceInfo DeviceInformation { get; } 23
    public void Reset() 23
Class RedRat.RedRat3.USB.USBRedRat3Impl 23
    public static string[] FindRedRat3s() 23
    public static new RedRat3USBImpl GetInstance(String devName) 23
Class RedRat.SignalEventArgs 23
    public SignalEventAction Action { get; } 23
    public IrDaPacket IrDaPacket { get; } 23
    public ModulatedSignal ModulatedSignal { get; } 23
    public Exception Exception { get; } 24
Enumeration SignalEventAction 24
    MODULATED_SIGNAL 24
    IRDA_PACKET 24
    EXCEPTION 24
    RC_DETECTOR_ENABLED 24
    RC_DETECTOR_DISABLED 24
    SIGNAL_ADDED 24
    SIGNAL_REMOVED 24
    SIGNAL_UPDATED 24
```

Class RedRat.DeviceInfo	24
public string Company { get; }	24
Class RedRat.USB.USBDeviceInfo	25
public string ProductName { get; }	25
public VersionInfo ProductVersion { get; }	25
public string SerialNo { get; }	25
public uint SerialNumberAsUInt { get; }	25
public ushort VendorID { get; }	25
public ushort ProductID { get; }	25
Class RedRat.LocationInfo	25
public string Name { get; set; }	25
public string Description { get; set; }	25
Class RedRat.RedRat3.RedRat3LocationInfo	25
public uint SerialNo { get; }	25
Struct RedRat.Util.VersionInfo	25
public uint Major { get; }	26
public uint Minor { get; }	26
Class RedRat.IR.IRPacket	26
public string Name { get; set; }	26
public string Description { get; set; }	26
public byte[] UID { get; }	26
public static bool UIDCompare(byte[] uid1, byte[] uid2)	26
Class RedRat.IR.ModulatedSignal	26
public const int EOS_MARKER	26
public double ModutationFreq { get; set; }	26
public double[] Lengths { get; set; }	26
public byte[] SigData { get; set; }	26
public int NoRepeats { get; set; }	27
public double IntraSigPause { get; set; }	27
Struct ToggleBit	27
Class RedRat.RedRat3.RedRat3ModulatedSignal	27
Class RedRat.IR.IrDaPacket	28
Class RedRat.IR.IrDaPacket.SubPacket	28
Class RedRat.RedRat3.RedRat3IrDaPacket	28
Class RedRat.IR.ProntoModulatedSignal	28
Class RedRat.AVDeviceMngmt.AVDeviceDB	28
Class RedRat.AVDeviceMngmt.AVDevice	29
Class RedRat.AVDeviceMngmt.SignalKey	30

Introduction

Many PC applications that deal with media, media management or home automation tasks require infrared remote control input and output. RedRat products have been designed for use in such applications. This document describes the APIs available for use by application developers.

The RedRat API is provided as a .NET assembly (dll), making development in the .NET environment straightforward. There is also a COM interface on top of the .NET assembly to support VB6 and VC++6 development – please see the document “RedRat COM API”.

From version 1.10 onwards, the SDK also supports the *RedRat4*, sold under the product name *irNetBox* - a networked, multi- IR output device. Although the RedRat4 API is very similar to that of the RedRat3, details have not been included in this documentation.

Prerequisites and SDK Installation

- The .NET Framework 2.0 and development environment (SDK, VisualStudio .NET or other).
- The RedRat SDK – download from <http://www.redrat.co.uk/SDK>
- RedRat hardware. This is actually optional, but only minimal application testing will be possible without hardware. To use this SDK, you need version 0.14 or newer of the RedRat3 firmware. If you want to use IrDa-like signal output functionality, you will need version 0.17 or later of the firmware.

Application Development and Deployment Using the SDK

The SDK is intended for use by application developers, but once an application is to be released the developer can use the RedRat runtime to distribute with their application.

Runtime Versions

An application developed with an SDK version has to be distributed the runtime of the same version number for correct operation. If this is not done, then the user will typically receive an error message stating that the .NET assembly file of correct version cannot be found.

The Object Model Overview

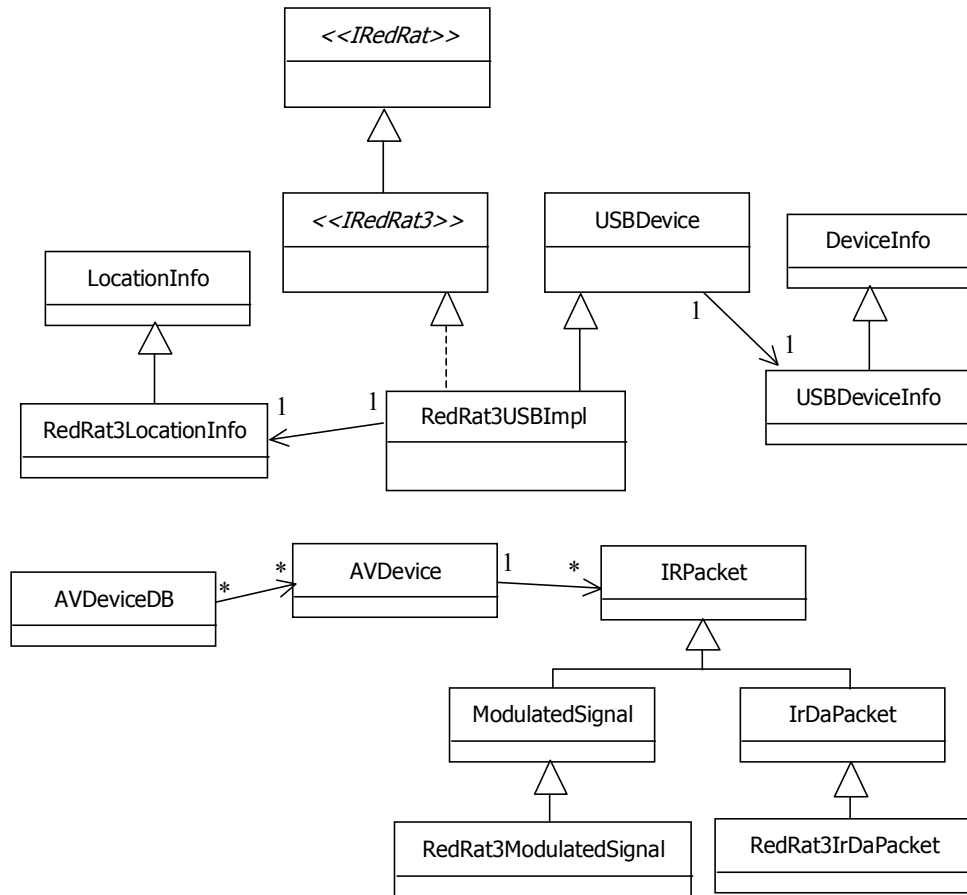


Figure 1. Overview of RedRat Object Model.

The model has been designed to support all RedRat infrared input and output devices, so may look a little more complex than necessary, however it is not difficult to use. Where possible try to use the top level interface, e.g. *IRedRat* instead of *IRedRat3*.

RedRat3USBImp is the main object with which the application developer will interact. As the *RedRat3* is a USB device, a lot of its hidden behaviour is as a *USBDevice*, i.e. it inherits USB data marshalling and interaction code from the *USBDevice* object. However, most of the time one is just interested in using it as an object of type *IRedRat3*, that is the IR I/O facilities.

The *AVDeviceDB* and *AVDevice* objects are for representing audio/visual devices, which is a useful way of managing collections of IR signals supporting exchange, centralized DBs and signal decoding.

An Introduction to Remote Control Signals

Although it is not necessary to understand a great deal about remote control signals to develop applications using RedRat products, some information on IR signals may make aspects of the API clearer, especially objects that represent the signals.

Modulated Remote Control Signals

The vast majority of remote control signals are in this family, having a carrier wave usually (though not always) in the frequency range 36kHz to 40kHz. Figure 2 shows the layers in such a signal. The RedRat does not attempt to interpret or discover the coding scheme used in the signal (e.g. shift/biphase coded, space coded, RC5, RCMM, REC-80 etc.) as knowledge of this is not needed for recognition or reproduction.

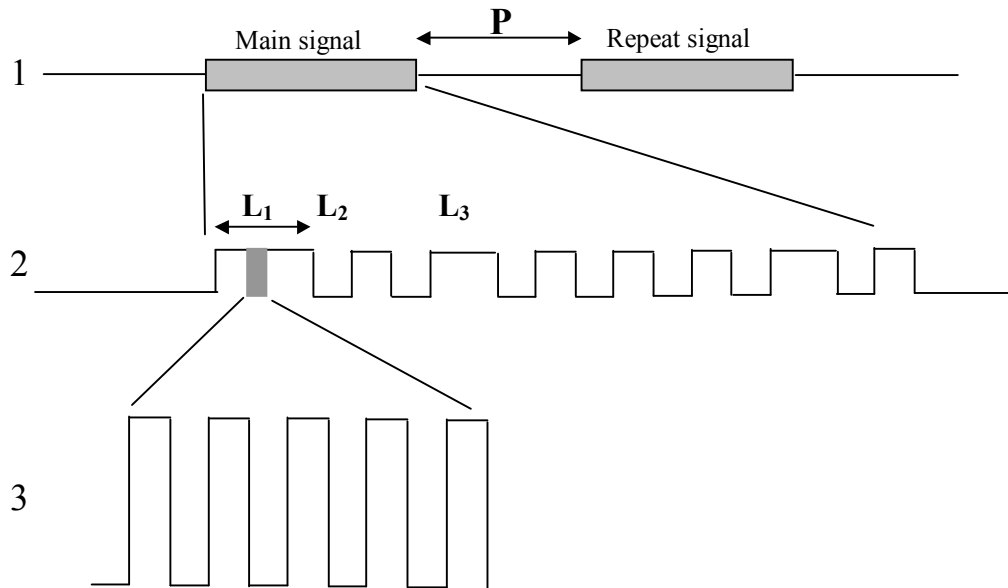


Figure 2. Layers in a Modulated Remote Control Signal.

Layer 1 – Main and Repeat Signals

These two sections to a signal are most commonly used to indicate the duration of a button press on a remote; the main signal being sent once and the repeat signal sent repeatedly until the button is released. There are of course variations, such as a 3 part signal (button down, hold and release) or repeating the whole main/repeat section. The value **P** is the inter-packet pause length.

Layer 2 – Signal Envelope

This is the basically the sequences of pulses and gaps that carry the signal information. When the RedRat samples the signal, it builds up an alphabet of the *lengths* of the pulses and gaps, L_1 , L_2 , L_3 etc. The actual signal data is then a series of numbers which are a lookup into the length array, i.e. 123333 is a sequence of length 1, length 2, length 3, length 3 etc.

Layer 3 – Carrier/Modulation Frequency

Each IR pulse is actually a rapidly switching signal, usually around the 36kHz to 40kHz allowing detectors to filter out background IR from this signal, so giving good transmission range and reliability. Standard remote control detectors strip out the carrier frequency and in doing so also alter the actual length values. This does not impact signal recognition, but in many cases are not sufficiently accurate for reliable reproduction. The RedRat samples the raw signal, giving accurate L values and a good measurement of the carrier frequency.

IrDa-Like Signals

Some set-top boxes use an IrDa-like remote control transmission protocol which offers some advantages over the modulated signal type described above, such as a higher data rate, supporting multiple handsets and timestamping signals. Figure 3 shows part of such a signal, comprising a series of sub-packets (15 at least) separated by quite large gaps.

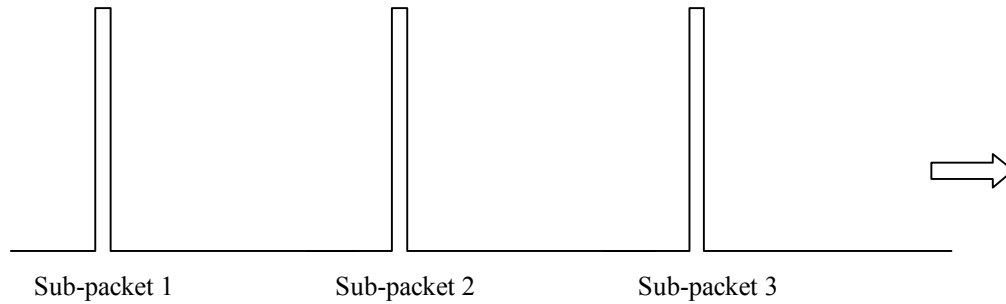


Figure 3. Part of an IrDa-like Signal

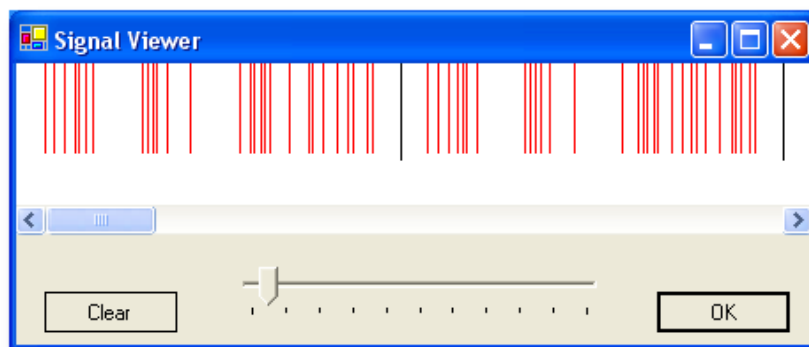


Figure 4. Sub-packet structure of an IrDa-like Signal.

Each sub-packet is a series of very short pulses (a couple of uS) with varying separation. Figure 4 shows the structure of the first two sub-packets of an signal, the large inter-sub-packet pause having been removed and replaced by the black vertical lines.

From firmware V0.17 and SDK V0.17 the RedRat3 can output IrDa like signals.

Using the RedRat API

Overview of the RedRat3s Functionality

The RedRat3 is designed to be a universal remote control input/output device for the PC, meaning that it has the ability to:

1. **Output** infrared remote control signals
2. Accurately record or **Learn** remote control signals for reliable output. This uses a short range IR detector (1 to 2m) and gives full information about the input signal.

3. **Detect** remote control signals from long range (10m or more), which allows control of the PC with any remote control. The data that is input from this detector is not generally accurate enough for reliable output of the signal.

The application developer has to clearly understand the difference between points 2 and 3 above, i.e. all data collected for signal output has to come from the *learning* detector. In fact, it is recommended that all databases of signal data collected for either recognition or output purposes are input through the *learning* detector.

Input signals are passed from the RedRat dll up to application code via events. In general, the application will setup event delegates to wait for incoming signals, the application code then responding appropriately when they arrive. The incoming signal events do not contain just IR signal data, but may also contain other information, such as an exception raised by the hardware or the fact that a detector has been disabled from elsewhere in the program. There are subtle differences in the way the incoming signals from the two detectors should be handled, which is discussed in the examples below.

Each RedRat3 has a serial number, which can be used by application programs to identify a particular RedRat3 in the situation where there is more than one attached to a computer (or on a network). There is support in the RedRat dll for the association of additional descriptive information with a serial number, the information being stored in the registry, keyed under the RedRat3 serial number. This allows the application code to find a RedRat3 using some meaningful name, such as LivingRoom, HiFiControl etc.

Example Code Using the TestRemote Application

Use of the RedRat API is demonstrated here using the *TestRemote* sample delivered as part of the SDK. *TestRemote* is a very basic graphical remote control in which program complexity has been kept to the minimum, but it exercises most of the RedRat3s functionality.

It displays a set of buttons, each of which can have a name and remote control signal associated with it.



Finding a RedRat3

```
protected void OpenRedRat3() {
    try {
        // Find the no. of RR3s connected.
        LocationInfo[] devices = RedRat3USBImpl.FindRedRat3s();
        if (devices.Length > 0) {

            // Just take the first device found.
            redRat3 = (IRedRat3)devices[0].GetRedRat();
        }
    }
}
```

```

    } else {
        MessageBox.Show("No RedRat3 devices found.", "TestRemote Warning",
            MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
    }

    } catch (Exception ex) {
        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
            MessageBoxIcon.Exclamation);
    }
}

```

The method `RedRat3USBImpl.FindRedRat3s()` returns a list of all RedRat3s found, as an array of *LocationInfo* objects. In the *TestRemote* application, we only take the first RedRat3 in the list.

The *LocationInfo* object contains PC local information about the RedRat hardware, such as its name, description and the type of RedRat hardware. It also has the *GetRedRat()* method, which returns an instance of the actual RedRat object to be used for IR input/output.

Discovering Information about the RedRat

The *RedRat3 Info* box under the *Help* menu item of *TestRemote* prints in a very simple fashion information about the RedRat3 as shown below:



This is done with the following code:

```

// Obtain and display info about the RR3.
string msg = null;
if (redRat3 == null) {
    msg = "No RedRat3 connected.";
} else {
    // Obtain information from RR3 to display....
    USBDeviceInfo devInfo = (USBDeviceInfo)redRat3.DeviceInformation;
    StringBuilder sb = new StringBuilder();
    sb.Append(devInfo.ToString() + "\n");
    sb.Append("Hardware Version: " + devInfo.ProductName + "." +
        devInfo.ProductVersion + "\n");
    sb.Append("Firmware Version: " + redRat3.FirmwareVersion);
    sb.Append("\nSerial Number: " + devInfo.SerialNo);
    sb.Append("\nLocation: " + redRat3.LocationInformation);
    msg = sb.ToString();
}
MessageBox.Show(msg, "RedRat3 Info.", MessageBoxButtons.OK,
    MessageBoxIcon.Information);

```

Information about the physical USB hardware is returned by the `IRedRat.DeviceInformation` attribute. Firmware version information is read directly from the RedRat3 using the

`IRedRat.FirmwareVersion` attribute, and finally information about the physical position is returned in a `LocationInfo` object returned by the `IRedRat.LocationInformation` attribute.

IR Signal Input – Learning Mode

The aim here is to learn an IR signal for output at a later stage. Being event driven, it is slightly more complex than the above examples, but is somewhat more elegant and perhaps easier to use than a non-event driven approach, which would put the onus of multithreaded code development on the developer.

In `TestRemote`, an IR signal is learnt from the button properties dialog box (`ButtonPropertiesDialog`), which pops up when the user clicks on a button with the right mouse button:



When the `Learn IR` button is clicked, the following actions need to be taken:

1. Tell the RedRat3 to enable IR signal input from the learning detector
2. Create an event delegate and add to the RedRat3 instance.
3. Create a method of terminating the signal input in case the user wants to cancel this action, such as a cancel dialog box.

These steps are shown in the code below:

```
// 1. Tell the RR3 we want to input a demodulated signal (learn)
rr3.GetModulatedSignal(0);

// 2. Create modal dialog that allows us to cancel the operation if required.
signalInputDialog = new SignalInputDialog(rr3);

// 3. The dialog box has to handle the input signal event from the RR3.
rr3.LearningSignalIn += new EventHandler(signalInputDialog.SignalDataIn);

// 4. Popup dialog....
signalInputDialog.ShowDialog(this);

// 5. Once finished, get the event object
SignalEventArgs siea = signalInputDialog.SignalInEvent;
```

Note that the event delegate is a method in the `SignalInputDialog`. This does not have to be the case, but it allows a simple method of managing the modal `SignalInputDialog` box, either the user presses the Cancel button in the box, or an input signal event arrives, and in both cases the `SignalInputDialog` will close and return control to button editor dialog.

In the dialog box, the following code is the actual event delegate, receiving the incoming IR signal event object, and then closing the dialog box.

```
public void SignalDataIn(object sender, EventArgs e) {
    if (e is SignalEventArgs) {
        sigInEvent = (SignalEventArgs)e;

        if (this.InvokeRequired) {
            CloseCallback cb = new CloseCallback(this.Close);
            this.Invoke(cb, null);
        }
        else {
            this.Close();
        }
    }
}
```

The *SignalEventArgs* object can indicate several things, using the *Action* property. The code below shows the main checks to be done; for an exception, an actual IR signal data or the user having closed the dialog box.

```
try {
    ...

    // Have input data of some kind...
    if (siea.Action == SignalEventAction.MODULATED_SIGNAL) {
        irPacket = siea.IRPacket;
        changed = true;
    }

    // We have had some error from the comms with the USB device,
    // so read exception and throw it.
    else if (siea.Action == SignalEventAction.EXCEPTION) {
        throw siea.Exception;
    }

    // The user has pressed cancel
    else if (siea.Action == SignalEventAction.INPUT_CANCELLED) {
        // Don't need to do anything here.
    }

} catch (Exception ex) {
    string msg = ex.Message;
    if (ex.InnerException != null) {
        msg += ": " + ex.InnerException.Message;
    }
    MessageBox.Show(msg, "Communication Error",
        MessageBoxButtons.OK, MessageBoxIcon.Exclamation);
} finally {
    // Remove listener...
    if (signalInputDialog != null) {
        rr3.LearningSignalIn -= new EventHandler(signalInputDialog.SignalDataIn);
    }
    signalInputDialog.Close();
}
```

The `finally` section is important here as it ensures that whatever the result, the *SignalDataIn* event delegate is removed and the *SignalInputDialog* is closed.

Outputting a Remote Control Signal

This operation is very straightforward given a *ModulatedSignal* object:

```

try {
    // Send the ModulatedSignal object to the RR3
    redRat3.OutputModulatedSignal((ModulatedSignal)irPacket);
} catch (Exception ex) {
    MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
        MessageBoxIcon.Exclamation);
}

```

The above code simply shows outputting a signal via a RedRat3, and showing an error to the user if an exception is raised in the process. A couple of points to note are:

1. The exceptions returned from the RedRat3 object contain quite technical information, so are not necessarily suitable for direct presentation to the user as shown.
2. The object being used is `irPacket` (of type *IRPacket*) which is cast to an object of type *ModulatedSignal* before use. This may seem a little unnecessary; why not just use *ModulatedSignal* objects through the code? The main reason is that there will be at least one other type of *IRPacket* – *IrDaPacket* which is a different kind of remote control signal.

Program Termination

When a program is closed by the user, RedRat resources should be released so that RedRat hardware is left in a state easily usable by other applications. The two steps to be taken are:

1. Unregister any event delegates in the main program that are attached to RedRat event sources.
2. Call the method *RedRat3USBImpl.DisposeOfAll()*, for example in the *Dispose()* method of a Windows application.

It is advisable to do order the operations above as given as the *DisposeOfAll()* method may cause events to be sent from the RedRat code, which in turn will raise an exception if send to a windows form which is being disposed of.

Using a Signal Database - The SignalDecoder Example Application

Using a database of IR signals from several audio-visual devices (the *AVDeviceDB* object) can greatly simplify application development as it provides simple lookup and reference of signals, recognition/decoding of incoming signals and allows straightforward signal storage and exchange.

This application is designed primarily to show using the signal decoding facilities in the .NET assembly to recognize incoming IR signals and use them to control applications on the PC. At the end of the chapter is some sample code showing how to lookup signals in a signal database for output.

To be able to recognize and decode signals, they must all have been *learned* and stored in a signal database. It can be downloaded from the RedRat web site at: <http://www.redrat.co.uk/Util/SignalDBUtil> and should be used to create a signal database for any remotes you would like to use to control PC applications.

The *SignalDecoder* application puts up a single window as shown in Figure 5 which is to be moved around the screen with a remote control.

Right-clicking on the body of the window will bring up a menu item to allow the user to choose a signal database, which is then loaded. When the appropriate buttons on a remote are pressed when pointed at a RedRat3, the window will be moved up, down, left or right, depending on which buttons were pressed.

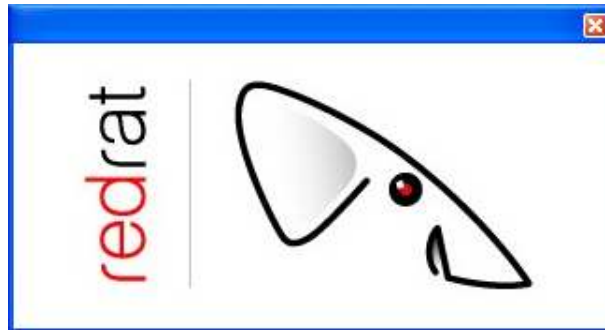


Figure 5. Main window of Signal Decoder Application

Setting up which buttons cause what actions is described in detail below.

Loading a Signal Database

```
// Read signal database from XML file.
AVDeviceDB newAVDeviceDB;
OpenFileDialog openFileDialog = new OpenFileDialog();
openFileDialog.Filter = "XML files (*.xml)|*.xml|All files (*.*)|*.*";
openFileDialog.RestoreDirectory = true ;

if (openFileDialog.ShowDialog() == DialogResult.OK) {
    string fName = openFileDialog.FileName;

    XmlSerializer ser = new XmlSerializer(typeof(AVDeviceDB));
    FileStream fs = null;
    try {
        fs = new FileStream((new FileInfo(fName)).FullName, FileMode.Open);
        newAVDeviceDB = (AVDeviceDB)ser.Deserialize(fs);
    } catch (Exception ex) {
        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK,
            MessageBoxIcon.Exclamation);
    } finally {
        fs.Close();
    }
}
```

The code above shows loading the signal database from an XML file to create an object of type *AVDeviceDB*. This object contains multiple *AVDevice* objects, each one representing a particular piece of AV equipment or remote.

Using the Database to Decode Signals

To obtain signals from a RedRat3, one needs to be discovered and an input signal event handler setup in a similar fashion to that described in the *TestRemote* example. The difference is that the event handler has to be hooked up to the *RCDetectorSignalIn* event and the RC detector enabled as shown below:

```
// Setup inputting signals from the RC detector.
rr3.RCDetectorSignalIn += new EventHandler(RCSignalInEventHandler);
rr3.RCDetectorEnabled = true;

// Application code here...

// Tidy up on program termination.
rr3.RCDetectorSignalIn -= new EventHandler(RCSignalInEventHandler);
rr3.RCDetectorEnabled = false;
```

The *SignalDecoder* example includes a full event handler for signals from the RC detector, which isn't listed here in full, instead we jump straight to the important code as far as signal decoding is concerned:

```
case SignalEventAction.MODULATED_SIGNAL:
    if (avDeviceDB != null) {
        // Decode signal if have DB.
        try {
            SignalKey sigKey = avDeviceDB.DecodeSignal(siea.ModulatedSignal);
            MoveWindow(sigKey);
        } catch (Exception ex) {
            Console.WriteLine(ex.Message + "\n" + ex.StackTrace + "\n");
        }
    }
    break;
```

This code section is part of a switch statement in the incoming signal event delegate, and if the incoming information is a *ModulatedSignal* (as opposed to an exception or some other notification), then the signal is passed directly to the *AVDeviceDB* object for decoding using the *DecodeSignal()* method call.

The *DecodeSignal()* methods returns a *SignalKey* object which contains information identifying the signal. In this example, it is passed to the *MoveWindow()* method to change the position of the window on the screen.

```
// Only want to respond to buttons on remote "CD"
if (sigKey.AVDevice.Name.Equals("CD")) {

    if (sigKey.Signal.Name.Equals("Play")) {
        // Move down
        AdjustStepSize(Direction.DOWN);
        curLoc.Y += (int)stepSize;

        // If window moves off screen, move back onto other side
        if (curLoc.Y > workingScreenSize.Height) {
            curLoc.Y = 0;
        }
        ...
    }
}
```

Using the *SignalKey* object is quite simple. We first look to check that it's the remote we want to respond to, then we check for the signal name. It is also possible use the signal UID to check for a recognized signal.

More About Handling Incoming Remote Control Signals

If you are developing an application that uses incoming remote control signals to initiate actions on the computer, then this section gives in-depth information on how the RedRat code manages the signals. By understanding this, the developer can give their application smooth and intuitive behavior when responding to remote control signal input.

The problem to be solved is the following. When reading incoming signals from the RedRat3, the action that each signal initiates on the computer cannot take too long or the RedRat will miss part or all of the next signal. (The RedRat does not have sufficient memory to queue signals internally.) The application developer may want to initiate a complex or protracted set of actions in response to a signal event and so cannot (and should not have to) complete these actions within the time between signals. Typically, the delay between a signal and its repeat on an prolonged remote control button press is around 30 to 150ms.

One approach would be to ignore all incoming signals until the signal event processing has completed. In practice however, even when the event processing takes less time than the gap between signals, OS scheduling of other tasks frequently results in an insufficient response time for reading incoming signals. Running the whole of the signal event dispatching mechanism (plus whatever actions it initiates) at high-priority is not appropriate use of the computer's resources.

The actual solution is shown in Figure 6. Reading signal data from the RedRat is decoupled from the signal event dispatching using a FIFO queue. The RedRat read thread runs at high priority, doing very little work so consumes no noticeable CPU resources, but reads pretty much 100% of incoming signals. Consuming the signal events by the application can take as long as required, and even though it may lead to a build up of signals to be processed, no data will not be lost.

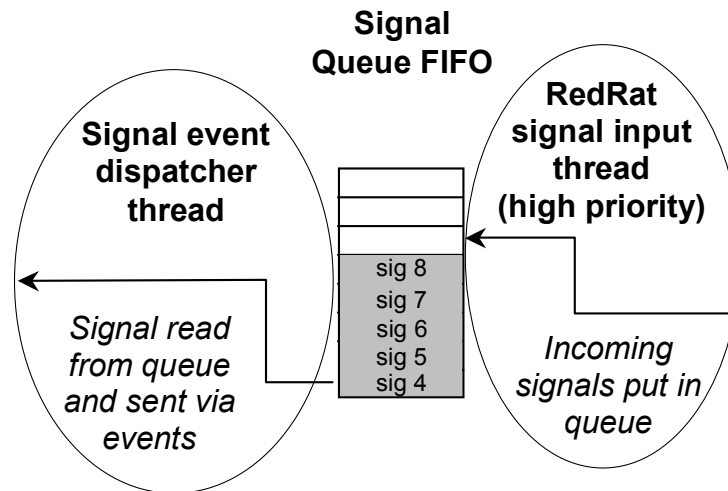


Figure 6. FIFO queue for signal input from the RedRat3 remote control detector.

The following code, when used in the event handler for RC signal input events, prints out the size of the signal queue and then processes each event very slowly by sleeping for 400ms each time.

```

case SignalEventAction.MODULATED_SIGNAL:
    Console.WriteLine("Have signal. Queue size is: " + siea.QueueSize);
    Thread.Sleep(400);
    break;

```

In some situations, one will want to process the signal backlog as long as it isn't too large, for example when the user is pressing Vol+ or Vol- on the remote. In other situations, one does **not** want to process the backlog. A good example is when "Skip Forward A Track" is pressed on a remote that is used to control a media player on a PC. A typical button press will result in several signals being sent, however acting on all these signals will cause the media player to jump several tracks. In this case, the application should clear the queue once it has moved on one track as shown below:

```

// Only want to respond to buttons on remote "CD"
if (sigkey.AVDevice.Name.Equals("CD")) {
    if (sigkey.Signal.Name.Equals("Skip+")) {
        mediaPlayer.NextTrack();
        Thread.Sleep(500);
        if (sender is IRedRat3) {

```

```

        // Clearing the queue.
        ((IRedRat3)sender).ClearRCSignalInQueue();
    }
}

```

When the instruction to skip forward a track is received, this action is carried out on the media player, and then a pause is inserted for half a second. Following the pause, all signals are cleared from the queue. This code thus gives the intuitive control when moving through the play list, i.e. rapid discrete button presses steps forwards or backwards quickly, but an extended button press won't accidentally shoot them to the end of the play list.

If a high level of custom or adaptive control is needed, then as each signal event is also given a timestamp on signal input, this can be used to give information such as the absolute time between successive signals.

Database Signal Lookup for Output

The example code below is not from the *SignalDecoder* example, but shows how to obtain a particular signal from the signal database for output.

```

ModulatedSignal sig = (ModulatedSignal)avDeviceDB.GetIRPacket("CD", "Pause");
rr3.OutputModulatedSignal(sig);

```

Instead of a device and signal name, a UID can also be used to lookup a signal:

```

ModulatedSignal sig = (ModulatedSignal)avDeviceDB.GetIRPacket(uid);

```

Additional Information for the Developer

RedRat Lookup

There are various types of RedRat hardware, connected in different ways to the computer or network as shown in the table below:

	USB Comms	TCP/IP Comms
RedRat3	Yes	
RedRat3-II	Yes	
irNetBox¹	Yes	Yes
irNetBox-II¹	Yes	Yes

Note 1: The irNetBox(-II) has to be pre-configured for either USB or TCP/IP comms; it is not capable of both.

To find or lookup RedRat hardware that is either directly attached to your PC or available on the network, use the following static method:

```

LocationInfo[] lis = RRUtil.FindRedRats();

```

After a few seconds (the time taken for a network broadcast), it will return a list of USB and TCP/IP devices discovered.

If you know that you will only want RedRat3s then the following can be used:

```
LocationInfo[] lis = RRUtil.FindRedRats(RRUtil.RedRatTypes.REDRAT3);
```

The enumeration *RRUtil.RedRatTypes* is as follows:

```
[Flags]
public enum RedRatTypes {
    REDRAT2 = 1,
    REDRAT3 = 2,
    IRNETBOX = 4,
    REMOTE = 8,           // Using .NET remoting
    ALL_LOCAL = IRNETBOX | REDRAT3 | REDRAT2,
    ALL = REMOTE | IRNETBOX | REDRAT3 | REDRAT2,
}
```

The *Flags* attribute indicates that the main values can be combined, as shown in the *ALL_LOCAL* and *ALL* enumeration members.

RedRat Type Discovery

In an application there may be situations where the code is given a *LocationInfo* object, but it needs to determine the exact details of the hardware associated with *LocationInfo* object. It is possible to look at the subclass type of *LocationInfo*, which will indicate whether the hardware is a RedRat3 or irNetBox (RedRat4).

For an exact description, the *LocationInfo.RRType* property can be used, which returns a *LocationInfo.RedRatType* enumeration value. The values are:

```
[Flags]
public enum RedRatType : int {
    Unknown = 0x0000,
    Default = 0x0001,           // The default RR to use.
    RedRat2 = 0x0002,
    RedRat3 = 0x0004,
    RedRat4 = 0x0008,

    USB = 0x0100,             // USB Device
    IP = 0x0200,             // IP Device
    RedRat4Output = 0x0400,  // Describes a single output port on a RedRat4.
    Remote = 0x0800,         // Uses remoting

    MK_II = 0x1000,          // MK_II version
}
```

These can be combined, so *LocationInfo.RRType* may return a value such as this:

```
RedRat3_II_USB = RedRatType.RedRat3 | RedRatType.MK_II | RedRatType.USB
```

LocationInfo also contains a set of utility test methods to help when checking the *RedRatType*:

```
LocationInfo.IsRedRat3(RedRatType rrt);
LocationInfo.IsRedRat4(RedRatType rrt);
LocationInfo.IsUSB(RedRatType rrt);
LocationInfo.IsIP(RedRatType rrt);
LocationInfo.IsMKI(RedRatType rrt);
LocationInfo.IsMKII(RedRatType rrt);
LocationInfo.IsRemote(RedRatType rrt);
```

To test directly for *RedRatType* values (and others not covered by the utility methods), the following can be used:

```

if ((rirt & RedRatType.RedRat3) == RedRatType.RedRat3) {
    // Do something with RR3
}

```

Concurrency

From version 0.12 of the SDK (requiring version 0.14 or greater of the RedRat3 firmware) it is possible for multiple programs on a PC to simultaneously use a RedRat3. RedRat3 use by multiple threads in a single program has also been greatly improved.

There are three main operations that the RedRat3 performs:

1. **Learn Signal** - Learn a new remote control signal (using the short-range detector).
2. **Output Signal** - Output a remote control signal to control A/V equipment.
3. **Input Signal** - Collect IR signals from remotes for PC control applications (using the long-range remote control detector).

If two or more threads/processes attempt the same operation, the behavior is shown in Table 1.

Table 1. Behavior when two or more threads/process simultaneously perform the same operation on a RedRat3

	Thread/Process One	Thread/Process Two
Learn Signal	Has exclusive access to the RR3 until signal has been input or action cancelled. This can be a long-lived operation.	Waits until process one is finished then gets exclusive access to the RR3.
Output Signal	Has exclusive access to the RR3 until output is completed. Usually a short-lived operation.	Waits until process one is complete then gets exclusive access to the RR3.
Input Signal	Exclusive access to the RR3. This may be indefinite, depending on the application.	Throws an exception informing the user that another thread/process has ownership of remote control signal input.

If two or more threads/process attempt different operations simultaneously, then the three operations are prioritized according to the list above, leading to the following rules:

1. If a thread/process is learning a signal, any signals to be output have to wait and input from the long-range detector is suspended until the signal learning has been completed or cancelled.
2. While the RedRat3 is outputting a signal, input from the long-range remote control detector is suspended until the signal output is complete.

PC Power Modes

Standby-Mode

When a computer enters standby mode, the RedRat3 is suspended, going into a low-power state. If the long-range remote control detector is enabled at when standby mode is requested, then the RedRat code will disable the detector and re-enable it when it comes out of standby. The application developer may therefore see a couple of additional RC detector enable and disable events.

Hibernation

Currently the device driver does not support hibernation mode. When the computer comes out of hibernation, the device driver will not re-enumerate the USB RedRat devices, so any RedRat3s are not discovered. Unfortunately the only way to reset the RedRat3 is to unplug it and plug back in. We hope to fix this at some point in the future.

API Details

Interface RedRat.IRedRat

Describes functionality common to RedRat infrared I/O devices. More specific interfaces extend this interface where necessary, for e.g. the RedRat has additional functionality, so *IRedRat* is extended by *IRedRat3*.

event EventHandler LearningSignalIn;

Event for incoming signal data from the short range, learning infrared data detector. The signature of the event delegate is standard (i.e. `public void SignalDataIn(object sender, EventArgs e)`) but the *EventArgs* object is of type *SignalEventArgs*. See the `GetModulatedSignal()` method for more information on setting up learning signal input.

event EventHandler RCDetectorSignalIn;

Event for signal data from the long range remote control detector. The details of the event are the same as for the *LearnignSignalIn* event.

Please Note: The signal data returned from this event is not suitable for IR signal output as it does not contain complete signal information. To collect data for signal output, please use the “learning” detector.

DeviceInfo DeviceInformation { get; }

Returns information about the physical device. Each type of physical device has a different set of non-overlapping descriptive properties (e.g. a COM port has a different set of descriptive information to a USB port), so the actual subclass returned depends on the physical device type. For example, the *RedRat3USBImpl* object returns USB device information – a *USBDeviceInfo* object.

LocationInfo LocationInformation { get; set; }

Returns information about the physical position of the device. This is intended for use in applications where multiple RedRats may be used, to support presenting readable information to the user on which RedRat is being used and where it is. This information is stored in the registry, and for the RedRat3 is keyed on the device’s serial number.

string FirmwareVersion { get; }

Attribute returning the version of the firmware running on the device in a readable form.

void Blink()

Both RedRat2s and RedRat3s have a red LED to give visual feedback of events, e.g. signal output. This method call blinks this red LED.

void GetModulatedSignal(int timeout)

Initiates input of a modulated IR remote control signal from the learning remote control detector. This method does not block, i.e. returns immediately - the actual signal is returned via the *LearningSignalIn* event.

The *timeout* (ms) parameter can be set if the application should only wait a certain amount of time before terminating learning signal input. If the timeout period is reached, the RedRat will no longer responding to IR signal input, and an event is sent to all *LearningSignalIn* event delegates informing them of input termination. A value of <0 can be passed for an indefinite timeout. Also see `CancelSignalInput()` for a programmatic method of terminating signal input.

void CancelSignalInput()

If the application is waiting for input from the learning IR detector, then this method will cancel the input mode. All listeners are informed of the cancellation in the *SignalEventArgs* object in the *LearningSignalIn* event.

void OutputModulatedSignal(ModulatedSignal outSig)

Given a modulated signal object (e.g. from a DB or previously captured from the learning IR detector), then this method call will output it via this device.

void OutputModulatedSignal(ModulatedSignal outSig, bool cachedData)

Given a modulated signal object (e.g. from a DB or previously captured from the learning IR detector), then this method call will output it via this device. Each time a signal is output, the data for transfer is constructed, so by default the constructed data is cached. For situations when caching is not wanted, it can be disabled for the signal with the *cachedData* parameter.

double EndOfSignalTimeout { get; set; }

Gets and sets the value for the IR dead period at the end of a signal used by the RedRat to determine the end of a signal. Units are in ms. Please see the IR Signal Details document for an in-depth explanation. The default value is currently 150ms.

int LengthMeasurementDelta { get; set; }

The RedRat uses a set of lengths as the alphabet used to represent most IR signals. Due to the approximate nature of IR signal data, two supposedly identical values will be slightly different. This attribute controls the size of the length “fuzz”, i.e:

```
if ((length1 - length2 <= fuzz) {
    length2 = length1
}
```

The default value is currently 112 (units are hardware specific at the moment!). Please see the IR Signal Details document for further information.

int ModFreqPeriodsToMeasure { get; set; }

The carrier frequency is measured during the first pulse of the IR signal. The larger the number of periods used to measure, the more accurate the result is likely to be, however some signals have short initial pulses, so in some case it may be necessary for applications to adjust this value. Please see the IR Signal Details document for further information. Default value is currently 8 periods.

double LengthToMillisec(uint length)
uint MillisecToLength(double millis)

Utility methods for length conversion between ms and values used by the RedRat hardware. The actual value depends on the particular RedRat hardware used.

ushort ModFreqToVal(double modFreq)

Converts the carrier/modulation frequency into a value used by the RedRat hardware.

Interface RedRat.RedRat3.IRedRat3

extends RedRat.IRedRat

This interface adds RedRat3 specific functionality to the standard *RedRat* interface. The reason for presenting the RedRat3 as an interface is that it allows flexibility in the concrete implementation, even if the same type of hardware device is being used. For e.g., to communicate with a RedRat3 attached to a different machine, the remote interface will still implement *IRedRat3*, but manage delegation of method call across the network.

int MaxNumLengths { get; set; }

The maximum number of length values (the signal alphabet) that a can be used in a single signal. As memory on a RedRat3 is limited, there is a trade-off in memory usage between the signal data and the length array. The default value is 16 lengths. Please see the IR Signal Details document for further information.

int signalMemorySize { get; set; }

The amount of memory used to hold signal data. Default value is 512 bytes.

void OutputIrDaPacket()

An *IrDaPacket* is a different kind of IR signal, similar to IrDa data communication signals, and is used by some modern A/V devices as it offers some advantages over standard modulated IR signals. As it is very different, not all RedRat hardware can deal with these signals.

bool RCDetectorEnabled { get; set; }

Gets and sets whether the long-range remote control detector is enabled. When enabled, IR signals from this detector will be passed back to the application via the *RCDetectorSignalIn* event.

bool RCInputOneShot { get; set; }

By default, when the long-range RC detector is enabled, it will produce a continuous stream of IR data, e.g. when a remote control button is held down for several seconds. In some situations, this may not be desirable, such as if there are a large number of actions to initiate with each signal, so an application developer may prefer to read a signal one at a time. If the application sets `RCInputOneShot = true`, then `RCDetectorEnabled` is set to false after a single input signal and has to be explicitly re-enabled.

void ClearRCSignalInQueue()

Incoming signals from the RC detector are queued so that detection isn't interrupted by heavyweight event handling in the application. If a bunch of signals pile up in the queue, then the application may want to clear the queue so that it doesn't carry on processing signals long after the remote control input has stopped.

Class RedRat.USB.USBDevice

This class contains general fairly low-level USB management and marshalling code. It is probably not necessary for the application developer to need to use this class directly, rather its RedRat3 specialization – *RedRat3USBImpl*.

public static USBDevice GetInstance(String devName)

There should only be one instance of this class per USB device, so the constructor is not public, rather a reference to an instance is obtained through this method. The parameter *devName* is the USB device name, e.g. “RedRat3-0”, “RedRat3-1” etc.

public void closeDriver()

Explicitly closes the driver handle open for this device. Generally not needed by application developers, but in some situations it is useful to explicitly close the driver before setting the object to null as one is not sure when the garbage collector will perform finalization tasks.

public String DeviceName { get; }

Returns the USB device name associated with this object, e.g. “RedRat3-0”.

public USBDeviceInfo DeviceInformation { get; }

Returns the USB device information about this object.

public void Reset()

Resets the USB device’s firmware. This is not the same as unplugging the device and then plugging it back in as there are some initialization tasks which only take place on power-on. Instead this method just re-starts the device firmware.

Class RedRat.RedRat3.USB.USBRedRat3Impl

extends RedRat.USB.USBDevice

implements interface RedRat.RedRat3.IRedRat3

This class implements the interface/behavior described by the *IRedRat3* interface in a USB device. Apart from the implementation of the interface, this class only provides two additional methods of interest here. Please see the superclasses for full interface details.

public static String[] FindRedRat3s()

The first step to using a RedRat3 is to find what RedRat3 are available with this static method. The return is a set of names, e.g. “RedRat3-0”, “RedRat3-1” etc.

public static new RedRat3USBImpl GetInstance(String devName)

As with the USBDevice object, we only want one instance of this class per RedRat3, so the constructor is not publicly available. To obtain an instance, use this method which will return the instance for the given device name, or create a new instance if it doesn’t already exist.

Class RedRat.SignalEventArgs

The event arguments for incoming IR signal events – see the interface *IRedRat* for details on the events. While the main purpose of this object is to pass the incoming IR data to event delegates, it must also notify the delegates of other situations, e.g. an error or change in state of signal input on the RedRat. The type of action is described by the enumeration *SignalEventAction* – see below.

public SignalEventAction Action { get; }

Returns the action (or reason) for this event, see *SignalEventAction* below.

public IrDaPacket IrDaPacket { get; }

Returns the *IrDaPacket* object captured by the RedRat3 or null if object does not contain this kind of data.

public ModulatedSignal ModulatedSignal { get; }

Returns the *ModulatedSignal* object captured by the RedRat3 or null if object does not contain this kind of data.

public Exception Exception { get; }

If there has been an exception, then this returns it. Null otherwise.

public int QueueSize { get; set; }

Signals from the long-range remote control detector are put into a FIFO queue, which is simultaneously emptied by an event dispatch thread. If these events are consumed at a slower rate than the incoming signal rate, then the queue will fill up. Its size can be monitored with this attribute.

public DateTime Timestamp { get; }

Time at which this event was created. For signal input, this is (pretty much) the time the signal was received.

Enumeration SignalEventAction

Used to describe the reason for signal events, i.e. *SignalEventArgs* object. Most often this is due to incoming IR signals from a RedRat, but can also be due to changes in signal data, databases etc. Values are:

MODULATED_SIGNAL

This event args object contains new *ModulatedSignal* data.

IRDA_PACKET

This object contains new *IrDaPacket* data.

EXCEPTION

Indicates to the event delegate that an exception has been thrown. There are two situations in which this may happen: 1) a legitimate cancellation of IR signal input mode and 2) an actual error from all code and hardware layers below this point (USB I/O code, device driver, system or RedRat3).

RC_DETECTOR_ENABLED RC_DETECTOR_DISABLED

These are only applicable to the long-range remote control detector and are used to indicate that it has been enabled or disabled. Why is this needed? For some applications the RedRat3 is used to control the PC and is then in permanent “listening” mode, waiting for input on the long-range detector. However, when it has to perform another operation, for example output an IR signal, then the RC detector is disabled for the duration of this operation. Application code needs to be notified of this so it can take appropriate action, e.g. re-enable the RC detector.

SIGNAL_ADDED SIGNAL_REMOVED SIGNAL_UPDATED

Indicates to the event delegate that a signal has been added to an *AVDevice* or *AVDeviceDB*. One reason to use this event is to update GUIs that may be displaying signal databases.

Class RedRat.DeviceInfo

Abstract class intended for use as superclass for hardware specific device information. As this type of information is so hardware dependant, there is not much commonality that can be factored into this class, so subclasses contain full information.

public string Company { get; }

Returns the name of the hardware manufacturer company.

Class RedRat.USB.USBDeviceInfo

extends RedRat.DeviceInfo

Subclass of *DeviceInfo* containing USB specific hardware details.

public string ProductName { get; }

Display name of product, e.g. “RedRat3”.

public VersionInfo ProductVersion { get; }

Returns the version of the product hardware. See below for information on the *VersionInfo* struct.

public string SerialNo { get; }

Returns the serial number of the hardware. This is of the form 1234ABCD, i.e. 8-digit ASCII represented hexadecimal number.

public uint SerialNumberAsUInt { get; }

Returns the serial number as an unsigned integer.

public ushort VendorID { get; }

public ushort ProductID { get; }

Returns the USB vendor ID and product ID for the USB device. These numbers are used by the OS for loading the correct driver.

Class RedRat.LocationInfo

Holds physical location information for a RedRat, e.g. where it is physically located. Subclasses for certain types of hardware give more specific features and map this information onto some uniquely identifying feature of the hardware.

public string Name { get; set; }

Readable name for the RedRat, such as the equipment it controls or its location – “HiFi” or “LivingRoom”. This name should be used by application programs when looking up RedRats.

public string Description { get; set; }

Longer descriptive text for the location of the RedRat.

Class RedRat.RedRat3.RedRat3LocationInfo

extends RedRat.LocationInfo

Maps location information onto a RedRat3 device. The RedRat3 serial number is used as the key to store the location information in the local computer’s registry.

public uint SerialNo { get; }

Returns the serial number of the RedRat3.

Struct RedRat.Util.VersionInfo

Very simple struct to hold major and minor version information.

```
public uint Major { get; }
public uint Minor { get; }
```

Class RedRat.IR.IRPacket

Superclass for all types of IR signal. As different types vary quite considerably, this class does not currently have any attributes.

```
public string Name { get; set; }
    Common name of the signal, e.g. "Play", "Stop", "Mute" etc.
```

```
public string Description { get; set; }
    Free text description of signal if required.
```

```
public byte[] UID { get; }
    Unique identifier for an IRPacket. This is used in signal databases and situations in which
    persistent references to signals are required. It is not a hash code of internal state, so cannot be
    used to deduce any information about the signal. Is intended for use with the System.Guid
    structure.
```

```
public static bool UIDCompare(byte[] uid1, byte[] uid2)
    Used to compare the identity of two IRPackets using their UID. Returns true if uids are the
    same.
```

Class RedRat.IR.ModulatedSignal

extends RedRat.IR.IRPacket

The vast majority of infrared remote control signals fall into this category. The RedRat2 only these types of signals, and currently RedRat3 firmware is only setup to manage these kind of signals. Generally it is not necessary for the application developer to understand the details of IR signals, but if data manipulation or conversion routines are developed, then understanding is necessary – please refer to the *IR Signal Details* document.

```
public const int EOS_MARKER
    Value in the signal data array marking the separation between the main and repeat signals. See
    the SigData attribute.
```

```
public double ModulationFreq { get; set; }
    The signal carrier or modulation frequency. Units of Hz.
```

```
public double[] Lengths { get; set; }
    An IR signal is constructed from periods of IR activity separated by periods of no activity. The
    lengths of these periods (not the sequence) are collected together into an alphabet from which
    the signal sequence can be described. The lengths are the alphabet, and the units are ms.
```

```
public byte[] SigData { get; set; }
    The sequence of length values (i.e. lookup into the Lengths data) creating an IR signal. A signal
    is normally constructed from a main signal, followed by a repeat signal, all contained in this
    data array. The separation between the two blocks of data can be identified by the
    EOS_MARKER.
```

```
public int NoRepeats { get; set; }
```

When a remote control button is held down for more than a very brief period, the usual behavior is that the repeat signal section is repeatedly transmitted. This attribute contains the number of re-transmissions of the repeats section.

```
public double IntraSigPause { get; set; }
```

Period of dead time between the main and the repeat signals. Units of ms.

```
public byte[] MainSignal { get; }  
public byte[] RepeatSignal { get; }
```

Derived attributes from the signal data - the signal data is usually split into a main and repeat signal sections.

```
public ToggleBit[] ToggleData { get; }
```

This data returns bits toggled or swapped every time a signal is sent – see information on the *ToggleBit* struct and documentation on IR signals for more information.

```
public static string ToXML(ModulatedSignal modSig)
```

Converts a modulated signal to an XML string for storage, exchange etc.

```
public static ModulatedSignal FromXML(string xmlData)
```

Converts an XML representation of an IR signal back to a *ModulatedSignal* object.

```
public static bool CompareSignal(ModulatedSignal sig1, ModulatedSignal sig1,  
                                double tolerance)
```

Simple method to compare two signals, returning **true** if they are the same to the given tolerance. The *tolerance* parameter is a percentage, a value of somewhere between 10 and 20 giving reasonable results.

```
public static bool MainRepeatIdentical(ModulatedSignal sig)
```

Checks whether the repeat signal is the same as the main signal.

```
public static bool SigDataEquivalence(byte[] data1, byte[] data2)
```

Utility method to test whether two signal data arrays are equal.

Struct ToggleBit

Some devices/remotes put so-called toggle bits in their signals, i.e. one or bits are swapped every time a signal is sent. While it is not always necessary to reproduce such toggle bits on signal output, it is important to know about them for signal recognition.

```
public int bitNo;
```

Position in the signal data to which this toggle bit applies.

```
public int len1, len2;
```

The two length values at the point in the signal represented by this *ToggleBit*. When a signal is output the first time, this position in the signal has length value *len1*, the second time *len2*, third time *len1* and so on.

Class RedRat.RedRat3.RedRat3ModulatedSignal

```
extends RedRat.IR.ModulatedSignal
```

Subclass of *ModulatedSignal* containing RedRat3 specific functionality, e.g. marshalling/de-marshalling of USB transfer data blocks into the appropriate *ModulatedSignal* variables.

Class RedRat.IR.IrDaPacket

extends RedRat.IR.IRPacket

Holds data that describes an IrDa-like remote control signal.

public double InterSubPacketPause { get; set; }

The period of time between sub-packets in a packet.

public SubPacket[] SubPackets { get; set; }

An array of objects, each one representing a signal sub-packet.

Class RedRat.IR.IrDaPacket.SubPacket

Represents a single sub-packet in an IrDa-like signal. A sub-packet is a series of pulses separated by varying periods given as float millisecond values.

public float[] PulseLengths { get; set; }

The set of lengths (ms) of the periods between pulses in this *SubPacket*.

Class RedRat.RedRat3.RedRat3IrDaPacket

extends RedRat.IR.IrDaPacket

Subclass of *IrDaPacket* containing RedRat3 specific functionality, e.g. marshalling/de-marshalling of USB transfer data blocks into the appropriate *IrDaPacket* variables.

Class RedRat.IR.ProntoModulatedSignal

extends RedRat.IR.ModulatedSignal

Subclass of *ModulatedSignal* constructed from a Pronto signal data string.

public ProntoModulatedSignal(string origProntoData)

Constructor, taking an input string of the form:

```
0000 006C 0000 0011 00A2 0014 003C 0014 003C 0014 0014 0014 0014  
0014 003C 0014 003C 0014 003C 0014 0014 0014 003C 0014 003C 0014  
003C 0014 0014 0014 003C 0014 0014 0014 0014 0014 0014
```

Class RedRat.AVDeviceMngmt.AVDeviceDB

Holds information for many audio-visual devices, i.e. a local database of IR signals. This database can be saved, exchanged with others, have single devices added (e.g. from the web), used for decoding signals or have signals looked up for output.

public void AddAVDevice(AVDevice newAVDevice)

Adds a new *AVDevice* to this database. Throws an *AVDeviceExistsException* if a device of this local name already exists in the DB.

public AVDevice GetAVDevice(string deviceName)

Returns the *AVDevice* object of the given name.

public void RemoveAVDevice(string devName)

Deletes the *AVDevice* with the given name from the database.

public string[] GetAVDeviceNames()

Return the names of the *AVDevice* objects currently contained in the signal DB.

public IRPacket GetIRPacket(string devName, string signalName)

Obtain the IR signal from the given device with the passed signal name.

public IRPacket GetIRPacket(byte[] uid)

Obtain the IR signal with the given signal 16 byte UID.

public SignalKey DecodeSignal(IRPacket irPacket)

If one has an input signal from either detector (usually the long-range detector though), one can use this method to find a signal match in this database. A *SignalKey* object is returned, which can be used for simple evaluation and initiation of actions.

public void CreateDecoder()

Inside this object, a signal decoder state machine is built to support reasonably efficient and reliable decoding. Normally the decoder state machine is constructed the first time the *DecodeSignal()* method is called, however this cause a holdup in the decoding operation while this takes place if a large database is used. An alternative is to explicitly construct the decoding state machine using this method, e.g. immediately after loading the database.

public double DecodersMCreateDelta { get; set; }

When constructing the decoding state machine, it is desirable to have equivalent patterns of state transition in the decoding state machine for similar signals (e.g. the first half of signals from the same remote if they all have the same starting code). However, when capturing/learning signals, they are seldom identical, so this value sets the latitude for the construction of decoding states – if two states at the same depth differ by less than this value they are merged. Default value is 0.05ms.

public double DecodersMDecodeDelta { get; set; }

When decoding a signal, direct comparison of a state's value with a signal data values is not possible due to variation in signal data each time a signal is input, so this value sets the latitude for equivalence of a value in a particular state. Default value is 0.5ms.

Class RedRat.AVDeviceMngmt.AVDevice

This object represents a single audio-visual device (often also meaning a single remote control) with a set of remote control signals. There is also some additional information which is designed to support publishing and finding the right information, e.g. via a web service rather than having to capture all the codes.

public enum AVDeviceType;

An enumeration defining the type of device, e.g. TV, VCR etc. This allows applications to behave appropriately, e.g. upon creation of a new *AVDevice* object of type CD, a set of default IR signals could be created.

public string Name { get; set; }

The local name for this device, such as LIVING_ROOM_TV or VCR2. This is then used by application code to identify signals within the context of RedRat hardware installation.

public string Manufacturer { get; set; }

Manufacturer of the AV device – used for global lookup.

```
public string DeviceModelNumber { get; set; }
```

Model number of the AV device.

```
public string RemoteModelNumber { get; set; }
```

Model number of the remote. Provides an alternative method of finding this data, e.g. from a web service.

```
public AVDeviceType DeviceType { get; set; }
```

The type of this particular AV device.

```
public IRPacket[] Signals { get; set; }
```

The set of signals associated with this AV device.

```
public void AddSignal(IRPacket newSignal, string name, bool overwrite);
```

Adds a new signal to this device. If *overwrite* is false, then a *SignalExistsException* is thrown if a signal with this name already exists. If *overwrite* is true, then the signal with that name is replaced if it exists.

```
public void RemoveSignal(string sigName);
```

Removes the signal with the given name from this *AVDevice* object.

```
public string[] GetSignalNames();
```

Returns the names of signals in this *AVDevice*.

```
public IRPacket GetSignal(string name);
```

Obtains the signal with the given name if it exists, otherwise returns null.

```
public IRPacket GetSignal(byte[] uid);
```

Obtains the signal with the given uid if it exists, otherwise returns null.

Class RedRat.AVDeviceMngmt.SignalKey

When a signal has been “decoded” or recognized, a *SignalKey* object is returned containing information about this signal.

```
public enum Status;
```

Can take values *KNOWN* or *UNKNOWN*.

```
public AVDevice AVDevice { get; }
```

The *AVDevice* to which this signal belongs.

```
public IRPacket signal { get; }
```

The actual signal represented by this key.

```
public Status keyStatus { get; }
```

Status of this *SignalKey*, i.e. does it represent a signal that is known or not?