

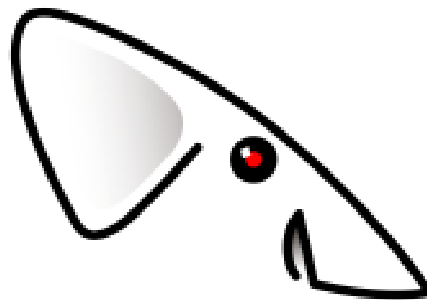
Using the RedRat COM API

Chris Dodge – RedRat Ltd

01 February 2008

For the RedRat SDK V2.06

redrat



Introduction	3
Prerequisites and SDK Installation.....	3
Changes.....	3
From SDK V2.04 on.....	3
From SDK V1.10 on.....	3
From SDK V1.00 on.....	3
In SDK V0.17.....	4
Using the COM API	4
Setting up the Program for Compilation.....	4
Runtime Initialisation and Finalization Tasks.....	5
Creating a RedRat3 Object.....	5
Simple RedRat3 Operations.....	6
Dealing with Errors.....	6
Using an IR Signal Database and Outputting an IR Signal	7
“FastPath” Signal Output	9
Signal Input via Events	9
Signal Input from the Learning Detector	11
Signal Recognition.....	11
The RedRat COM API.....	12
Interfaces.....	12
IRedRatLookup.....	12
ICCWRedRat3.....	12
ICCWRedRat4.....	13
ISignalDB.....	13
IRR3Events	14
IRR4Events	14
ISignalInEventArgs.....	15
Structs	15
CCWModulatedSignal.....	15
ToggleBit.....	15
CCWDoubleModulatedSignal.....	15
CCWIrDaSignal.....	15
CCWSubPacket	16
CCWRR3DeviceInfo	16
CCWRR3LocationInfo	16
CCWRR4DeviceInfo	16
CCWRR4LocationInfo	16
CCWAVDeviceInfo.....	16
Enumerations.....	17
SignalEventAction.....	17
AVDeviceType	17

Introduction

The core support code for RedRat devices has been written as a .NET assembly, however application code developed with VB6 and VC++6 needs to access this functionality via COM. This document outlines how to use this API, i.e. instantiating the COM objects, sending and receiving data from them, checking for errors etc. It does not however go into detail about the semantics of the function calls, e.g. what the function call *put_EndOfSignalTimeout()* means – please see the .NET API document for more information at this level.

The COM interface is created automatically by the .NET interoperability facilities in the Common Language Runtime (CLR), the result being a COM Callable Wrapper (CCW). The object structure in the .NET assembly is flattened for the CCW, so the interfaces presented are basically the aggregation of itself and all superclasses for a given class. Nonetheless, the function calls map pretty much one-to-one onto the .NET API.

Hopefully you will find that it will not be necessary to know much about .NET and mechanisms for COM interoperability to use the RedRat COM interfaces, but if you would like more information on .NET and COM interoperability, please see the book “.NET and COM – The Complete Interoperability Guide” by Adam Nathan, published by SAMS.

Prerequisites and SDK Installation

- The .NET framework 2.0.
- 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.
- Microsoft Visual C++ 6.0 or similar development environment.

Changes

This section lists the major changes impacting the use of the COM API.

From SDK V2.04 on

- All .NET code has been ported to .NET 2.0, so to use the COM API, .NET 2.0 or 3.0 will need to be installed on the PC. Microsoft Vista is also now supported.

From SDK V1.10 on

- The SDK has initial support for the *RedRat4*, sold under the product name *irNetBox*, some details of which are given at the end of this document.

From SDK V1.00 on

- The RedRat runtime for distribution with applications developed using the SDK now supports side-by-side installation, i.e. one can have multiple

versions simultaneously installed on one machine. Please see the main documentation for further details.

In SDK V0.17

- The COM interoperability facilities in version 1.1 of the framework are slightly more complete, so there are a few COM API changes from V0.17 of the SDK that reflect this. The main difference is in the use of variable length SAFEARRAYS when passing arrays of primitives or structs (UDTs) rather than fixed length arrays before.
- Support has been added for outputting IrDa-like IR signals that are used by some set-top boxes.
- A new “*fastpath*” signal output mechanism has been introduced to reduce the overhead of passing signal data from .NET to COM and back again for each output. This is detailed in the section on signal DBs and output.

Using the COM API

The following sections show how the COM API can be used from VC++ with the help of several snippets of code. This code is available in more complete form as a download from the SDK pages on the RedRat website.

There are three types of entity that form the RedRat COM API:

COM Objects: The actual instantiation of these objects is in the .NET layers and they are presented for use by COM code through an interface, e.g. *ICCWRat3*. They always inherit from the COM interfaces *IDispatch* and *IUnkown*. Some of these objects can be directly instantiated from COM code (those with a zero argument constructor), whereas some have to be returned from calls to other COM objects. Whichever way, the *Release()* function should be called on these objects once the application has finished with them.

Structs: These are pass-by-value data structures (UDTs), e.g. *CCWModulatedSignal*. Generally, the memory for structs is allocated in the COM application and passed to a COM object as an [OUT] parameter.

Enumerations: Contain a list of constants, e.g. *AVDeviceType*.

Setting up the Program for Compilation

References have to be made to the .NET and RedRat type libraries, and the necessary COM support included. The following code should be placed in your program, with appropriate adjustment for your environment, e.g. in *StdAfx.h*.

```
// Need this for CoInitializeEx defns. (DCOM)
#define _WIN32_DCOM

// Import the type libraries for the wrapper for the .NET assemblies.
// N.B. Your .NET Framework version may be different from that below.
#pragma warning(disable:4146)
#import "C:\\windows\\Microsoft.NET\\Framework\\v1.1.4322\\mscorlib.tlb"
no_namespace named_guids raw_interfaces_only
#pragma warning(default:4146)
```

```

import "C:\\Program Files\\RedRat\\RedRat SDK\\RedRat.tlb" no_namespace
named_guids raw_interfaces_only

#include <stdio.h>

// For the event handling
#include <objbase.h>
#include <atlbase.h>
ComModule _Module;
#include <atlcom.h>

```

Runtime Initialisation and Finalization Tasks

If your VC++ application does not already use COM, then the following code will be needed before attempting to use the COM API.

```

// Init COM.
HRESULT hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);
if (FAILED(hr))
{
    fprintf(stderr, "Error: Cannot initialize COM\n");
    return -1;
}

```

and at program termination:

```

// Close COM for this thread
CoUninitialize();

```

Creating a RedRat3 Object

A RedRat3 is presented to COM through the *ICCWRat3* interface, however it is not possible to directly instantiate an implementation of this interface. Instead it has to be looked up through the *IRedRatLookup* interface, so the first task is to create an instance of the object implementing this interface, then use that to return an *ICCWRat3*.

```

IRedRatLookup* rrLookup;
ICCWRat3* rr3;

// Create a COM RedRatLookup object
hr = CoCreateInstance(CLSID_RedRatLookup, NULL, CLSCTX_SERVER,
                    IID_IRedRatLookup, reinterpret_cast<void**>(&rrLookup));
if (FAILED(hr))
    fprintf(stderr, "Call to CoCreateInstance failed");
else
{
    // Obtain ref to the first RedRat3, i.e. RedRat3-0.
    BSTR rrName = SysAllocString(OLESTR("RedRat3-0"));
    hr = rrLookup->GetRedRat3(rrName, &rr3);
    SysFreeString(rrName);

    if (FAILED(hr))
        fprintf(stderr, "RRLookup->GetRedRat3 failed.");
    else
    {
        //
        // == USE THE REDRAT3 HERE ==
        //
        // ... then when finished release
        rr3->Release();
    }
    rrLookup->Release();
}

```

A couple of points to note are:

- Strings passed to and from COM are double-byte (Unicode) strings, so in the example above, the RedRat3 name (*RedRat3-0*) had to be created as a BSTR.

- The *Release()* function should be called on all COM objects once the program is finished. This decrements the reference count for the object, allowing the system to free memory when the object is no longer in use.

Simple RedRat3 Operations

The code below shows perhaps the simplest RedRat operation, blinking the visible LED:

```
// Blink the visible LED
HRESULT hr = rr3->Blink();
if (FAILED(hr))
    fprintf(stderr, "\nRedRat3 blink failed.", hr);
else
    printf("Blink ok\n");
```

This slightly more involved example obtains the firmware version number for output:

```
// Print the firmware version
BSTR fwVer;
hr = rr3->get_FirmwareVersion(&fwVer);
if (FAILED(hr))
    fprintf(stderr, "rr3->get_FirmwareVersion() failed.");
else
    // wprintf for BSTR
    wprintf(L"\n -> Firmware Ver: %s\n", fwVer);
if (fwVer) SysFreeString(fwVer);
```

Dealing with Errors

So far, the error reporting in the code has been very basic. Richer error information is available from the following sources:

- The HRESULT value returned by each call on a COM object.
- If an error occurs in the .NET code layers, then an exception is raised which is managed by the .NET CLR (Common Language Runtime) and can then be read.

Using the “blink” example code given above, then more useful error information could be given in the following way:

```
HRESULT hr = rr3->Blink();
if (FAILED(hr))
{
    PrintErrorMsg("RedRat3 blink failed.", hr);
    CreateExtendedErrorInfo(rr3);
    return;
}
```

The *PrintErrorMsg()* function is basically code in the demo application that looks up known/common HRESULT codes and prints out their meaning. More interesting is the *CreateExtendedErrorInfo()* function whose body is shown below:

```
ISupportErrorInfo* suppErrInfo;
IErrorInfo* errorInfo;
_Exception* exception;
_Type* type;

BSTR desc, exTypeName, stackTrace;

if (comObj)
{
    HRESULT hr = comObj->QueryInterface(IID_ISupportErrorInfo, (void*)&suppErrInfo);
    if (SUCCEEDED(hr))
    {
```

```

// Get the error info pointer
hr = GetErrorInfo(0, &errorInfo);
if (SUCCEEDED(hr))
{
// Get error description
hr = errorInfo->GetDescription(&desc);
if (SUCCEEDED(hr))
wprintf(L" Description:\t%s\n", desc);

// Obtain the .NET exception
hr = errorInfo->QueryInterface(IID__Exception, (void**)&exception);
if (SUCCEEDED(hr))
{
// Have exception object, so get info from it.
// - Exception type
hr = exception->GetType(&type);
if (SUCCEEDED(hr))
{
// Get type name
hr = type->get_ToString(&exTypeName);
if (SUCCEEDED(hr))
wprintf(L" Ex type:\t%s\n", exTypeName);
}

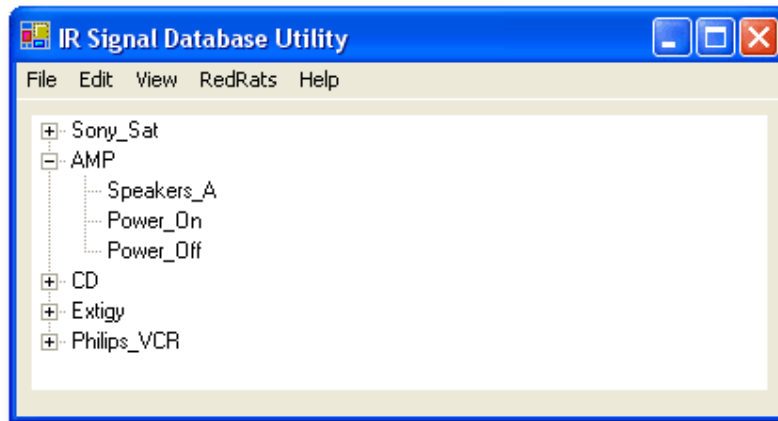
// Get exception stack trace
hr = exception->get_StackTrace(&stackTrace);
if (SUCCEEDED(hr))
wprintf(L" Stack trace:\t%s\n", stackTrace);
}
}
}
}
}
if (desc) SysFreeString(desc);
if (exTypeName) SysFreeString(exTypeName);
if (stackTrace) SysFreeString(stackTrace);
if (type) type->Release();
if (exception) exception->Release();
if (suppErrInfo) suppErrInfo->Release();
if (errorInfo) errorInfo->Release();

```

This code can be used with any COM object (i.e. object of type *IUnknown*), such as the *IRedRatLookup* or *ICCWRedRat3*.

Using an IR Signal Database and Outputting an IR Signal

An IR signal database is used to manage a set of audio/visual devices and the remote control signals associated with each device. Although the application developer does not have to use the RedRat support for IR signal management, it does provide facilities for signal management, import, export, exchange and recognition. To create a signal database, a .NET application called the "Signal Database Utility" is available for download from the RedRat website, the main screen shown below:



The AV device and signal data is stored in an XML file which can then be used in the code below (with minimal error reporting – see demo code for better error reporting).

```
// Create a COM signal DB for signal decoding.
ISignalDB* signalDB;
HRESULT hr = CoCreateInstance(CLSID_CCWSignalDB, NULL, CLSCTX_SERVER,
                             IID_ISignalDB, reinterpret_cast<void*>(&signalDB));

if (FAILED(hr))
    fprintf(stderr, "CoCreateInstance of type \"ISignalDB\" failed.");
else
{
    // Set filename of signal DB XML file
    BSTR dbFileName = SysAllocString(OLESTR("D:\\Temp\\DeviceDB.xml"));
    wprintf(L>Loading signal DB file: %s\\n", dbFileName);
    hr = signalDB->AddDeviceDB(dbFileName);
    if (FAILED(hr))
        fprintf(stderr, "Error loading signal DB file.");

    SysFreeString(dbFileName);
}
// == USE THE SIGNAL DB ==
// ... and release when finished
if (signalDB) signalDB->Release();
```

The *ISignalDB* object is created directly using the COM *CoCreateInstance* call, the filename is then passes to it which it then loads. This can then be used to lookup signals for output:

```
void outputIRSignal(ICCWRat3* rr3, ISignalDB* signalDB)
{
    if ((rr3)&&(signalDB))
    {
        CCWModulatedSignal* outSig = new CCWModulatedSignal();
        BSTR devName = SysAllocString(OLESTR("CD"));
        BSTR sigName = SysAllocString(OLESTR("Pause"));

        HRESULT hr = signalDB->GetSignal(devName, sigName, outSig);
        if (FAILED(hr))
            fprintf(stderr, "Failed to lookup signal in signal DB.\\n");
        else
        {
            hr = rr3->OutputModulatedSignal(outSig);
            if (FAILED(hr))
                fprintf(stderr, "Failed to output IR signal.\\n");
            else
                wprintf(L"\\n >>> Output signal: %s, %s\\n\\n", devName, sigName);
        }
        if (devName) SysFreeString(devName);
        if (sigName) SysFreeString(sigName);
    }
}
```

```

    if (outSig) delete outSig;
}
}

```

“FastPath” Signal Output

The steps for signal output above require that a .NET signal object is converted to a COM object and then converted back to a .NET object for output. Although the amount of work is not that significant given the frequency of IR signal output (maximum 3 to 4 per second), it is work that can be avoided if required. An alternative method of signal output is shown below:

```

rr3->SetSignalDB(signalDB);

BSTR devName = SysAllocString(OLESTR("CD"));
BSTR sigName = SysAllocString(OLESTR("Pause"));

HRESULT hr = rr3->OutputSignal(devName, sigName);
if (FAILED(hr))
{
    PrintErrorMsg("\nFailed to output IR signal.", hr);
    CreateExtendedErrorInfo(rr3);
}
else
{
    wprintf(L"\n >>> Output signal: %s, %s\n\n", devName, sigName);
}
if (devName) SysFreeString(devName);
if (sigName) SysFreeString(sigName);

```

The first step is to pass the signal database object to the RedRat3 that will use it for output. Following this, any signal in the DB can be output by passing the device and signal name to the *OutputSignal()* method on *ICCWRat3*.

Signal Input via Events

Incoming IR signals are delivered to the application via events, so if the application is to deal with them, a handler has to be setup. There are two events:

- Incoming IR signals from the long-range IR detector that are then used to initiate program actions, i.e. remote control of your application. The signal recognition facilities described in the next section could be useful for this.
- Learning/recording IR signals with the short-range IR detector for output later or use with recognition. As this is a one-off operation, and can be done with the signal database utility it is not covered here, however the principles are very similar to the use of the long-range detector.

The events are passed to the program via “connection points”, using ATL facilities and needing some fairly cryptic stuff in the code! In the demo code, the *IREventHandler* class deals with this, extracts of which are show below, but please see the demo code for more details.

The basic principle of using the *IREventHandler* is shown below (for input from the long-range remote control detector):

```

// Create an event handler object
IREventHandler* sigIn = new IREventHandler(rr3);

// == Enable RC signal input ==
rr3->put_RCDetectorEnabled(TRUE);

printf("waiting for signal input (press <RETURN> to terminate):\n");
getchar();

```

```

// == IMPORTANT ==
// == Disable RC signal input ==
rr3->put_RCdetectorEnabled(FALSE);

delete sigIn;

```

Once the event handler object is created, the RedRat3 remote control detector is enabled. Incoming signals are handled in the *IREventHandler* object, so this thread can simply sit and wait (using *getchar()* here) until remote control input should be terminated.

Important: The remote control detector should be disabled before program termination. If this is not done, the .NET layer remains awaiting IR signal data input and will not terminate properly, and the RedRat3 will appear to hang when the program is next started. Hopefully this will be sorted out soon!

So how is the event handler connected to the “connection point”, and how are the incoming signal events handled?

The constructor of *IREventHandler* contains the code to connect to the event source:

```

// Attach to the event connection points.
HRESULT hRes = DispEventAdvise((IUnknown*)comRR3);

```

Once no more IR signal events are needed, then the application should disconnect from the event source (found in the destructor of *IREventHandler*):

```

// Unattach from connection points.
DispEventUnadvise((IUnknown*)comRR3);

```

The actual event handler function is:

```

//
// Input from long-range remote control detector.
//
HRESULT __stdcall IREventHandler::RCSignalIn(const ICCWRedRat3* sender,
                                             struct ISignalInEventArgs * e)
{
    // This event could indicate an input signal, or that the detector
    // has been enabled/disabled.
    SignalEventAction action;
    e->get_Action(&action);
    switch (action)
    {
        case SignalEventAction_RC_DETECTOR_ENABLED:
            printf(" --> In event: RC detector enabled.\n");
            break;

        case SignalEventAction_RC_DETECTOR_DISABLED:
            printf(" --> In event: RC detector disabled.\n");
            break;

        case SignalEventAction_MODULATED_SIGNAL:
            printf(" --> In event: Have RC signal.\n");
            // If we have a signal DB, then see if we recognize the signal.
            break;

        case SignalEventAction_EXCEPTION:
            printf(" --> In event: Have exception from signal input.\n");
            break;

        default:
            // Ignore all other event types.
    }
}

```

```

        break;
    }
    return S_OK;
}

```

The `ISignalInEventArgs` object contains all event information, and the code above uses the *action* attribute to print out information about the event.

Techy Note: The .NET design guidelines suggest that an event should have parameter 1 of type *Object* and parameter 2 of type *EventArgs* or subclass thereof. This turns out to not be very helpful for COM clients as early-binding type information is not available, so we have used custom event arguments here.

Signal Input from the Learning Detector

The principle involved in learning signals for recognition or output is similar to the above, but for clarity an example of setting this up is included here.

```

IREventHandler* sigIn = new IEventHandler(rr3);

// Pass ref to signalDB to event handler so it can add new signals.
sigIn->SetSignalDB(signalDB);

// Request input signal (learning detector)
BSTR sigName = SysAllocString(OLESTR("NewSignal"));
sigIn->SetSignalName(sigName); // Give a name for the new signal
sigIn->SetWaiting(true); // Setup waiting for input
printf(" waiting 6s for signal input...\n");
rr3->GetModulatedSignal(6000); // Setup input and ...

while (sigIn->GetWaiting() // wait until signal or timeout.
{
    sleep(10);
}
SysFreeString(sigName);

// No longer need event handler
delete sigIn;

```

The above code uses the demo *IREventHandler* object. Basically the *GetModulatedSignal()* method call initiates the process of signal input from the “learning” detector, but the event handler receives the signal and adds it to the database, so it is setup with the database to use (*SetSignalDB()*) and the name of the signal when added to the DB (*SetSignalName()*).

This thread has to wait until the signal is input or it times-out, which is achieved here by checking the event handler *waiting* flag using the *GetWaiting()* method.

Signal Recognition

The incoming signals can be compared against the signal DB, so allowing signal recognition.

```

case SignalEventAction_MODULATED_SIGNAL:
    printf(" --> In event: Have RC signal: ");
    // If we have a signal DB, then see if we recognize the signal.
    if (signalDB)
    {
        CCWModulatedSignal modSig;
        BSTR sigKey;
        HRESULT hr = e->get_ModulatedSignal(&modSig);
        if (FAILED(hr))
            fprintf(stderr, "Failed to read input IR signal\n");
        else

```

```

    {
        hr = signalDB->DecodeSignal(modSig, &sigKey);
        wprintf(L"Decoded signal->%s", sigKey);
    }
}
break;

```

The *DecodeSignal* function gives a *BSTR* containing the device and signal name if the signal has been recognized (*sigKey* in the above code). The information is of the form: “CD, Play”, i.e. the two fields separated by a comma.

The above code is actually part of the incoming signal event handler, so as the signal arrives it is recognized and then an appropriate action can be initiated.

The RedRat COM API

The RedRat COM API is documented below, in IDL generated from the type library. Please use the API listing below in conjunction with the .NET documentation for descriptions of the function calls.

Interfaces

IRedRatLookup

This is the entry point for discovery of RedRat3 and obtaining a RedRat3 object to interact with (ICCWRat3).

```

interface IRedRatLookup : IDispatch {
    HRESULT FindRedRat3s([out, retval] SAFEARRAY(BSTR)* pRetVal);
    HRESULT GetRedRat3(
        [in] BSTR rr3Name,
        [out, retval] ICCWRat3** pRetVal);
    HRESULT FindRedRat4s([out, retval] SAFEARRAY(BSTR)* pRetVal);
    HRESULT GetRedRat4(
        [in] BSTR rr4Name,
        [out, retval] ICCWRat4** pRetVal);
};

```

ICCWRat3

Represents a physical RedRat3 device.

```

interface ICCWRat3 : IDispatch {
    HRESULT MaxNumLengths([out, retval] long* pRetVal);
    HRESULT MaxNumLengths([in] long pRetVal);
    HRESULT SignalMemorySize([out, retval] long* pRetVal);
    HRESULT SignalMemorySize([in] long pRetVal);
    HRESULT RCDetectorEnabled([out, retval] VARIANT_BOOL* pRetVal);
    HRESULT RCDetectorEnabled([in] VARIANT_BOOL pRetVal);
    HRESULT ClearRCSignalInQueue();
    HRESULT DeviceInformation([out, retval] CCWRR3DeviceInfo* pRetVal);
    HRESULT LocationInformation([out, retval] CCWRR3LocationInfo* pRetVal);
    HRESULT LocationInformation([in] CCWRR3LocationInfo pRetVal);
    HRESULT FirmwareVersion([out, retval] BSTR* pRetVal);
    HRESULT Blink();
    HRESULT GetModulatedSignal([in] long timeout);
    HRESULT CancelSignalInput();
    HRESULT OutputModulatedSignal([in, out] CCWModulatedSignal outSig);
    HRESULT OutputProntoSignal([in] BSTR prontoSignalData);
    HRESULT OutputSignal([in] BSTR deviceName, [in] BSTR signalName);
    HRESULT SetSignalDB([in] ISignalDB* signalDB);
    HRESULT EndOfSignalTimeout([out, retval] double* pRetVal);
    HRESULT EndOfSignalTimeout([in] double pRetVal);
    HRESULT LengthMeasurementDelta([out, retval] long* pRetVal);
    HRESULT LengthMeasurementDelta([in] long pRetVal);
    HRESULT ModFreqPeriodsToMeasure([out, retval] long* pRetVal);
    HRESULT ModFreqPeriodsToMeasure([in] long pRetVal);
};

```

ICCWRedRat4

```
interface ICCWRedRat4 : IDispatch {
    HRESULT Connect();
    HRESULT Disconnect();
    HRESULT IsConnected([out, retval] VARIANT_BOOL* pRetVal);
    HRESULT EnableIROutput([in] long output);
    HRESULT DisableIROutput([in] long output);
    HRESULT SetAllIROutputsEnabled([in] VARIANT_BOOL onOff);
    HRESULT EnabledIROutputs([out, retval] SAFEARRAY(VARIANT_BOOL)*pRetVal);
    HRESULT EnabledIROutputs([in] SAFEARRAY(VARIANT_BOOL) pRetVal);
    HRESULT NoIROutputs([out, retval] long* pRetVal);
    HRESULT MaxNumLengths([out, retval] long* pRetVal);
    HRESULT MaxNumLengths([in] long pRetVal);
    HRESULT SignalMemorySize([out, retval] long* pRetVal);
    HRESULT SignalMemorySize([in] long pRetVal);
    HRESULT RCDetectorEnabled([out, retval] VARIANT_BOOL* pRetVal);
    HRESULT RCDetectorEnabled([in] VARIANT_BOOL pRetVal);
    HRESULT ClearRCSignalInQueue();
    HRESULT DeviceInformation([out, retval] CCWRR4DeviceInfo* pRetVal);
    HRESULT LocationInformation([out, retval] CCWRR4LocationInfo* pRetVal);
    HRESULT LocationInformation([in] CCWRR4LocationInfo pRetVal);
    HRESULT FirmwareVersion([out, retval] BSTR* pRetVal);
    HRESULT Blink();
    HRESULT GetModulatedSignal([in] long timeout);
    HRESULT CancelSignalInput();
    HRESULT OutputModulatedSignal([in, out] CCWModulatedSignal* outSig);
    HRESULT OutputIrDaSignal([in, out] CCWIrDaSignal* outSig);
    HRESULT OutputProntoSignal([in] BSTR prontoSignalData);
    HRESULT OutputSignal(
        [in] BSTR deviceName,
        [in] BSTR signalName);
    HRESULT SetSignalDB([in] ISignalDB* signalDB);
    HRESULT EndOfSignalTimeout([out, retval] double* pRetVal);
    HRESULT EndOfSignalTimeout([in] double pRetVal);
    HRESULT LengthMeasurementDelta([out, retval] long* pRetVal);
    HRESULT LengthMeasurementDelta([in] long pRetVal);
    HRESULT ModFreqPeriodsToMeasure([out, retval] long* pRetVal);
    HRESULT ModFreqPeriodsToMeasure([in] long pRetVal);
};
```

ISignalDB

Object used for managing a database of signals. Has to be loaded with a set of pre-captured signals in a device DB file – see the Signal DB Utility for creation of such a database file in XML format. The file data is passed in via the `AddDeviceDB([in] BSTR dbFileName)` function. An input signal is then passed to the decoder via the `decodeSignal` function which returns a `ccwSignalKey` struct identifying the signal.

```
interface ISignalDB : IDispatch {
    HRESULT AddDeviceDB([in] BSTR dbFileName);
    HRESULT SaveDeviceDB([in] BSTR dbFileName);
    HRESULT AddAVDevice([in] BSTR avDeviceName);
    HRESULT AddAVDevice(
        [in] BSTR avDeviceName
        [in] VARIANT_BOOL readWrite);
    HRESULT RemoveAVDevice([in] BSTR avDeviceName);
    HRESULT GetAVDeviceInfo(
        [in] BSTR avDeviceName,
        [out, retval] CCWAVDeviceInfo* pRetVal);
    HRESULT SetAVDeviceInfo([in] CCWAVDeviceInfo newAVDeviceInfo);
    HRESULT GetDeviceNames([out, retval] SAFEARRAY(BSTR)* pRetVal);
    HRESULT GetSignalNames(
        [in] BSTR avDeviceName,
        [out, retval] SAFEARRAY(BSTR)* pRetVal);
    HRESULT AddSignal(
        [in] BSTR devName,
        [in, out] CCWModulatedSignal* newSig,
        [in] VARIANT_BOOL overwrite);
    HRESULT AddDoubleModSignal(
        [in] BSTR avDeviceName,
```

```

        [in, out] CCWDoubleModulatedSignal* newSig,
        [in] VARIANT_BOOL overwrite);
HRESULT RemoveSignal(
    [in] BSTR avDeviceName,
    [in] BSTR signalName);
HRESULT GetSignal(
    [in] BSTR avDeviceName,
    [in] BSTR signalName,
    [out, retval] CCWModulatedSignal* pRetVal);
HRESULT GetIrDaSignal(
    [in] BSTR avDeviceName,
    [in] BSTR signalName,
    [out, retval] CCWIrDaSignal* pRetVal);
HRESULT GetDoubleModSignal(
    [in] BSTR avDeviceName,
    [in] BSTR signalName,
    [out, retval] CCWDoubleModulatedSignal* pRetVal);
HRESULT DecodeSignal(
    [in] CCWModulatedSignal inSig,
    [out, retval] CCWSignalKey* pRetVal);
HRESULT SetDebugOutput([in] BSTR textFile);
HRESULT SignalToXML(
    [in, out] CCWModulatedSignal* inSig,
    [out, retval] BSTR* pRetVal);
HRESULT XMLToSignal(
    [in] BSTR xml,
    [out, retval] CCWModulatedSignal* pRetVal);
HRESULT IrDaPacketToXML(
    [in, out] CCWIrDaSignal* inSig,
    [out, retval] BSTR* pRetVal);
HRESULT XMLToIrDaPacket(
    [in] BSTR xml
    [out, retval] CCWIrDaSignal* pRetVal);
HRESULT DecoderSMCreateDelta([out, retval] double* pRetVal);
HRESULT DecoderSMCreateDelta([in] double pRetVal);
HRESULT DecoderSMDecodeDelta([out, retval] double* pRetVal);
HRESULT DecoderSMDecodeDelta([in] double pRetVal);
};

```

IRR3Events

These events pass incoming signals from the RedRat3 to the application program.

```

dispinterface IRR3Events {
    properties:
    methods:
        [id(0x00000001)]
        void RCSignalIn(
            [in] ICCWRedRat3* sender,
            [in] ISignalInEventArgs* e);

        [id(0x00000002)]
        void LearningSignalIn(
            [in] ICCWRedRat3* sender,
            [in] ISignalInEventArgs* e);
};

```

IRR4Events

Used to pass incoming signals and other information to the application program from the RedRat4 object.

```

dispinterface IRR4Events {
    properties:
    methods:
        [id(0x00000001)]
        void RCSignalIn(
            [in] ICCWRedRat4* sender,
            [in] ISignalInEventArgs* e);

        [id(0x00000002)]
        void LearningSignalIn(

```

```

        [in] ICCWRedRat4* sender,
        [in] ISignalInEventArgs* e);
    [id(0x00000003)]
    void IROutputsChanged(
        [in] ICCWRedRat4* sender,
        [in] IIROutputChangedEventArgs* e);
};

```

ISignalInEventArgs

The events in IRR3Events pass two parameters, the first being the RedRat3 that caused the event, and the second this object containing information about the event.

```

interface ISignalInEventArgs : IDispatch {
    HRESULT Action([out, retval] SignalEventAction* pRetVal);
    HRESULT ModulatedSignal([out, retval] CCWModulatedSignal* pRetVal);
    HRESULT Exception([out, retval] _Exception** pRetVal);
    HRESULT TimeStamp([out, retval] DATE* pRetVal);
    HRESULT QueueSize([out, retval] long* pRetVal);
};

```

Structs

CCWModulatedSignal

```

struct tagCCWModulatedSignal {
    BSTR name;
    BSTR description;
    double modFreq;
    double intraSigPause;
    long noRepeats;
    SAFEARRAY(double) lengths;
    SAFEARRAY(unsigned char) sigData;
    long terminated;
    SAFEARRAY(ToggleBit) toggleData;
} CCWModulatedSignal;

```

ToggleBit

A modulated signal can have data bits that toggle with one signal to the next.

```

struct tagToggleBit {
    long bitNo;
    long len1;
    long len2;
} ToggleBit;

```

CCWDoubleModulatedSignal

When a single IR signal comprises two signals (sent alternately with every button press) then this held in this struct. Can only be used for reading from the signal DB and signal output, not signal creation.

```

struct tagCCWDoubleModulatedSignal {
    long terminated;
    BSTR name;
    BSTR description;
    CCWModulatedSignal signal1;
    CCWModulatedSignal signal2;
} CCWDoubleModulatedSignal;

```

CCWIrDaSignal

```

struct tagCCWIrDaSignal {
    BSTR name;
    BSTR description;
};

```

```
double interSubPacketPause;
long noSubPackets;
SAFEARRAY(CCWSubPacket) subPacketArray;
CCWIrDaSignal;
```

CCWSubPacket

Constituent part of an IrDa-like IR signal.

```
struct tagCCWSubPacket {
    SAFEARRAY(single) data;
} CCWSubPacket;
```

CCWRR3DeviceInfo

Contains information about the USB device hardware.

```
struct tagCCWRR3DeviceInfo {
    BSTR company;
    BSTR productName;
    long vendorID;
    long productID;
    long verMajor;
    long verMinor;

    BSTR serialNo;
} CCWRR3DeviceInfo;
```

CCWRR3LocationInfo

Used to hold information about the physical location of the device.

```
struct tagCCWRR3LocationInfo {
    long serialNo;
    BSTR name;
    BSTR description;
} CCWRR3LocationInfo;
```

CCWRR4DeviceInfo

Contains information about the RedRa4.

```
struct tagCCWR43DeviceInfo {
    BSTR company;
    BSTR productName;
    BSTR serialNo;
    long verMajor;
    long verMinor;
    BSTR ipAddress;
    BSTR macAddress;
} CCWRR4DeviceInfo;
```

CCWRR4LocationInfo

Used to hold information about the physical location of the device.

```
struct tagCCWRR4LocationInfo {
    BSTR macAddress;
    BSTR ipAddress;
    BSTR name;
    BSTR description;
} CCWRR4LocationInfo;
```

CCWAVDeviceInfo

Used to hold information about a particular AVDevice in the signal DB.

```
struct tagCCWAVDeviceInfo {
```

```
    LPSTR name;
    LPSTR manufacturer;
    LPSTR deviceModelNumber;
    LPSTR remoteModelNumber;
    AVDeviceType AVDeviceType;
} CCWAVDeviceInfo;
```

Enumerations

When using enumerations, it is recommended that the descriptive value is used rather than the number, for e.g. `SignalEventAction_SELF_DETERMINE` rather than the value 0.

SignalEventAction

When a signal in event is received, then this enumeration can be used to determine the reason for the event.

```
enum {
    SignalEventAction_SELF_DETERMINE,
    SignalEventAction_MODULATED_SIGNAL,
    SignalEventAction_DATA_PACKET,
    SignalEventAction_EXCEPTION,
    SignalEventAction_RC_DETECTOR_ENABLED,
    SignalEventAction_RC_DETECTOR_DISABLED,
    SignalEventAction_SIGNAL_ADDED,
    SignalEventAction_SIGNAL_REMOVED,
    SignalEventAction_SIGNAL_UPDATED
} SignalEventAction;
```

AVDeviceType

An AVDevice can be given one of the types listed below. This is intended to help (at some point in the future) with narrowing signal DB searching or UI customization based on device type.

```
enum {
    AVDeviceType_TV,
    AVDeviceType_VCR,
    AVDeviceType_DVD,
    AVDeviceType_CAMCORDER,
    AVDeviceType_DVD_RECORDER,
    AVDeviceType_PVR,
    AVDeviceType_CD_PLAYER,
    AVDeviceType_AMP,
    AVDeviceType_TAPE_DECK,
    AVDeviceType_TUNER,
    AVDeviceType_MINI_DISK,
    AVDeviceType_SAT_RECEIVER,
    AVDeviceType_SET_TOP_BOX,
    AVDeviceType_OTHER
} AVDeviceType;
```