Propellerhead Software
Address: Rosenlundsgatan 29C,
SE-118 63 Stockholm
Telephone: +46 8556 08400
Fax: +46 8556 08401
E-mail: info@propellerheads.se
Web: www.propellerheads.se

Remote™ Codec Developer Documentation

Remote is a protocol for communication between hardware control surfaces and software applications, developed by Propellerhead Software.

The information in this document is confidential and covered by the licensing agreement required to develop Remote Codecs. Do not distribute!

Contents

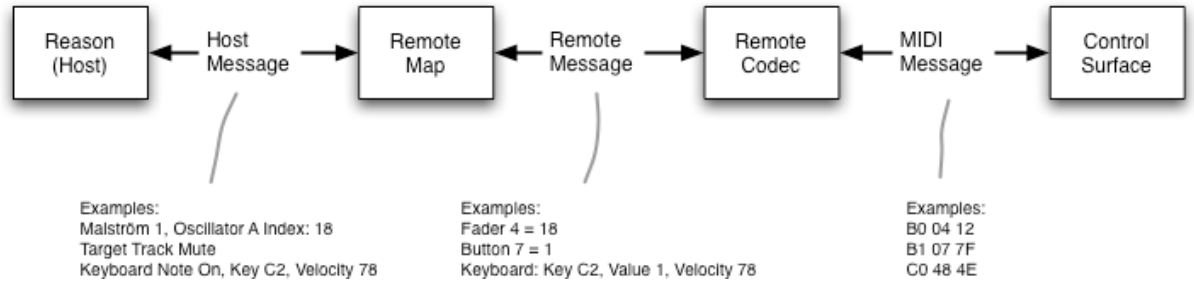
1 Introduction

This documentation describes how to write and dispatch all the material required for a complete Remote™ Codec, for Reason. The steps required are:

- Get familiar with Reason and the Remote technology, from a user's perspective. If you don't really understand how Reason works and how a user would best utilize it with a hardware control surface, you will have problems judging how to best design the integration between the two. Information about how Remote works from a user's perspective can be found in the Reason Operation Manual.
- Learn Lua. To write a Codec you don't need to be a Lua expert, but you need to master the basics. References to more information about Lua can be found later in this document.
- Read this documentation. It describes all the steps and includes example implementations for some imaginary control surfaces. You may also want to study other released Codecs for further examples. One such example included with Reason 3.0.2 is the Codec for the Frontier Design Group TranzPort, see http://frontierdesign.com.
- Write the actual Codec files.
- Test the Codec using the Codec Test application included in the SDK.
- Write a Remote Map file.
- Test your Codec and mapping with Reason, on both platforms (Mac OS and Windows). Remember that the final quality of the user experience is your responsibility, not Propellerhead Software's.
- Create installers for Mac OS and Windows. Although the codec itself is cross platform compatible, you need to create separate installers for each operating system.
- Release the Codec!

Please note that if you are also the manufacturer of the hardware control surface that you are implementing for, Propellerhead Software is interested in distributing your codec via our web page. Please contact Propellerhead Software for more information.

2 Definitions and terminology



The picture above shows the major parts of the Remote system.

2.1 Terminology

2.1.1 Control Surface

A **Control Surface** is any kind of hardware controller (for example a musical keyboard) that can send or receive **MIDI messages**. (In fact, although Remote is based on MIDI messages, the carrier may be USB or FireWire.) Control surfaces have buttons, faders, a keyboard, etc, that send unique MIDI messages when changed. A button, for example, usually sends a three byte message where one byte contains the button ID and another indicates if its being pressed or released.

2.1.2 Host

The **host** is the software that is being controlled by the control surface. As of this writing the only Remote host existing is Propellerhead Reason, but the Remote technology is defined to be host independent in order to ensure future compatibility.

2.1.3 Remote Messages

One of the tasks of a Codec is to translate the MIDI messages from the control surface into standardized **Remote messages**, which are named and categorized into well-defined types. The button MIDI message in the example above would typically be translated into a Remote message of type Button, with a name “Button 7” and the value 1 if pressed, 0 if released.

2.1.4 Remote Map

The Remote message is then sent into the **Remote Map**, where it’s translated into a **host message**. Where the codec is generic to the control surface and can theoretically be used with any host, the Remote Map is specific to *both* the control surface *and* the host. In other words, mapping files that you write for your codec and for Reason can only be used with Reason and that codec. Furthermore, when Reason gets updated, it is more than likely that you will have to update your Remote Map file, but the codec may remain the same.

The host message created by the Remote Map addresses the Remote message to a specific device and parameter in Reason, for example “Subtractor 1: Filter 2 On”. The value is automatically scaled to fit the target parameter.

2.1.5 Control Surface Items and Remotable Items

A parameter in the host that can be controlled by Remote is called a **Remotable Item**. One remotable item is controlled by exactly one **Control Surface Item** (a knob, fader or other hardware control on the control surface).

The relation between a Control surface item and its Remotable item is always one-to-one, you can’t control two remotable items with one control surface item, or vice versa.

2.2 What is a Codec?

2.2.1 Codec Tasks

A codec has three main tasks.

1. It defines all the elements on the control surface - the control surface items. The definitions give the items names and sorts them into categories, which help Remote handle input from a wide variety of surfaces.
2. It translates MIDI input messages to standardized Remote messages, which are associated with the control surface items.
3. Lastly, it translates the state of the Remote host back to MIDI when dealing with two-way control surfaces.

2.2.2 Multiplicity

When Reason is launched, it scans all installed codecs to find all supported models of all control surfaces connected to the computer.

Many different control surfaces can be used at the same time; the user can mix and match as required. Technically, each surface is identified by the **manufacturer** and **model** name.

In addition to this, one codec can support many control surfaces, to allow for combinations of surfaces that operate as one unit (a main unit and an "extender", for example). Still, each surface requires its own Remote map. The map specifies which remotable items are controlled by the control surface, depending on which Reason device is being controlled.

2.2.3 The Files

A codec is actually more than one file, a package consisting of at least three files:

1. The index file, with extension ".luacodec". This defines the control surface name and some attributes. All files with this extension are scanned by Reason at startup to build up a list of available control surfaces.
2. The Lua source file, usually with the extension ".lua". This file holds the source for one or more of the supported control surfaces. This file can be compiled into byte code.
3. A picture file for each control surface model. The picture must be in PNG format, 96*96 pixels.

2.2.4 Where to Install?

The codec package should be installed in a folder named after the manufacturer. This folder should be installed in a folder in the following directories:

Mac OSX:

```
Library:Application Support:Propellerhead Software:Remote:Codecs:Lua Codecs
```

Windows:

```
C:\Documents and Settings\All Users\Application Data\Propellerhead Software\Remote\Codecs\Lua Codecs
```

In addition to this, there must be a Remote map file for each supported model, with the extension ".remotemap". Map files are installed in a folder named after the manufacturer in:
Mac OSX:
`Library:Application Support:Propellerhead Software:Remote:Maps`
Windows:
`C:\Documents and Settings\All Users\Application Data\Propellerhead Software\Remote\Maps`

More about installers later in this document.

2.2.5 Midicodec files

Reason 3.0 uses files with the extension `midicodec` to define codecs for some control surfaces. This format is superseded by the Lua codecs. Propellerhead will not support new development of midi codecs.

2.3 What is a Remote Map File?

To make a control surface and its codec able to communicate specifically with Reason, it needs a Remote Map file. This specifies which parameter in Reason should be controlled by which control on the surface, in any given situation or **scope**. For example, when the surface is directed to a Subtractor, you may want "Knob 1" to control the Filter Frequency parameter, but when it's directed to a Mixer, you may want the same knob to control a volume fader, etc.

There are three categories of scopes:
1. Scopes for individual device types (Subtractor, Mixer, etc.). These have the highest priority.
2. The "Reason Document" scope. This includes items such as transport functions, setting the MIDI input focus to another sequencer track, Undo, etc. This has the second priority.
3. The "Master Keyboard" scope. Includes keyboard, pitch bend and other items, for playing instrument devices. This has the lowest priority.

3 The SDK

Included in the package is:
- This documentation.
- Reason Remote Support documentation.
- The Remote logo in eps format.
- Examples of Remote Maps.
- Windows and Mac OS version of the following applications:
    - **Lua Compiler**. If you want to compile your Lua code.
    - **Lua Interpreter**. Allows you to program Lua interactively and to run Lua programs.
    - **CodecTest**, an application developed by Propellerhead Software in order to test and or debug codecs and remote maps.

4 About Lua

A codec is written using the Lua language, which is a “powerful light-weight programming language designed for extending applications”. We chose Lua because it’s platform independent, a codec written on the Macintosh will instantly work on the PC. Lua is also an easy language, compared to C or C++. There is much less overhead in writing a Lua codec.

Remote defines how the Lua codec should be structured. There are a number of functions that the codec should implement. One function is called when a control surface instance is created (when the host connects to it), another function is called for each incoming MIDI message, etc. Some of these functions are mandatory, others are optional. The initializing function, for example, must be implemented because it defines all control surface items.

So the host sets the stage for the codec writer, and also supplies some utility functions for common definitions or operations. Lua is a full-fledged programming language, allowing you to create a complex and powerful Codec if required. However, normally you will not need to dive that deep, since most codecs will only use a few tables defining the control surface items and how to translate the messages they send.

This document does not describe the Lua language. To find out more about Lua please visit http://www.lua.org. Remote uses Lua version 5.0. The reference manual can be found at http://www.lua.org/manual/5.0/.

5 Examples

5.1 In Control

Let’s start implementing the “In Control” surface codec. Automated Computer Music Equipment’s In Control model has a keyboard, pitch bend and modulation wheel, four faders, four encoders and four buttons:

While developing a codec, you should put the work in your user settings folders, instead of in the system folders as described in the previous section. We create the folder:
Mac OSX:

```
Users:<username>:Library:Application Support:Propellerhead Software:Remote:Codecs:Lua Codecs:ACME
```

Windows:

```
C:\Documents and Settings\<username>\Application Data\Propellerhead Software\Remote\Codecs\Lua Codecs\ACME
```

5.1.1 InControl.luacodec

First we create the `InControl.luacodec` file. It contains one function that lists all supported control surfaces, like this:

```
version={0,0,1}

function remote_supported_control_surfaces()
   local surfaces =
      {
         { manufacturer="ACME", model="In Control",
            source="InControl.lua", picture="InControl.png",
            in_ports={ {description="In Port"} },
            out_ports={{description="Out Port", optional=true}},
            has_keyboard=true
         },
      }
   return surfaces
end
```

The function returns an array containing the supported models. The `manufacturer` and `model` fields are the keys that identify one model. They're used to find the right Remote map, in preferences in Reason, etc. The `source` field points out the Lua source code file for the model. The `picture` field contains the file name of a 96*96 PNG file, which will be shown in the control surface setup dialogs. In Control is connected using two MIDI ports, we define these using the `in_ports` and `out_ports` fields. The out port is optional because it's only needed for auto-detection.

5.1.2 In Control surface items

Create the source file `InControl.lua`. In this file we start by defining all the control surface items. This is done in the `remote_init()` function:

```
function remote_init()
   local items={
      {name="Keyboard", input="keyboard"},
      {name="Pitch Bend", input="value", min=0, max=16383},
      {name="Modulation", input="value", min=0, max=127},
      {name="Fader 1", input="value", min=0, max=127},
      {name="Fader 2", input="value", min=0, max=127},
      {name="Fader 3", input="value", min=0, max=127},
      {name="Fader 4", input="value", min=0, max=127},
      {name="Encoder 1", input="delta"},
      {name="Encoder 2", input="delta"},
      {name="Encoder 3", input="delta"},
      {name="Encoder 4", input="delta"},
      {name="Button 1", input="button"},
      {name="Button 2", input="button"},
      {name="Button 3", input="button"},
      {name="Button 4", input="button"},
   }
   remote.define_items(items)
-- to be continued
```

We create an array containing one entry for each control surface item. Each item is given a unique name. The input type of the item defines how it works.
A "keyboard" input item works as a standard MIDI keyboard, it sends note on/note off, with note number and velocity.

The pitch bend, mod wheel and faders are “value” inputs, which means they always send absolute positions. Because of this we must also define their minimum and maximum values.
Encoders are limitless. They are “delta” inputs, sending positive or negative position changes.
The buttons send one message when pressed and one when released.

The `items` array is passed to the `remote.define_items()` function.

5.1.3 Automatic input handling

Remote has a utility for automatic input handling. By registering patterns for matching MIDI messages, you can let Remote handle the translation automatically. This pattern matching utility is quite versatile; all the items on the In Control surface can be handled automatically.
Registering auto inputs is also done in `remote_init()`:

```
-- remote_init() continued

   local inputs={
      {pattern="b? 40 xx", name="Fader 1"},
      {pattern="b? 41 xx", name="Fader 2"},
      {pattern="b? 42 xx", name="Fader 3"},
      {pattern="b? 43 xx", name="Fader 4"},
      {pattern="e? xx yy", name="Pitch Bend", value="y*128 + x"},
      {pattern="b? 01 xx", name="Modulation"},
      {pattern="9? xx 00", name="Keyboard", value="0", note="x", velocity="64"},
      {pattern="<100x>? yy zz", name="Keyboard"},
      {pattern="b? 50 <???y>x", name="Encoder 1", value="x*(1-2*y)"},
      {pattern="b? 51 <???y>x", name="Encoder 2", value="x*(1-2*y)"},
      {pattern="b? 52 <???y>x", name="Encoder 3", value="x*(1-2*y)"},
      {pattern="b? 53 <???y>x", name="Encoder 4", value="x*(1-2*y)"},
      {pattern="b? 60 ?<???x>", name="Button 1"},
      {pattern="b? 61 ?<???x>", name="Button 2"},
      {pattern="b? 62 ?<???x>", name="Button 3"},
      {pattern="b? 63 ?<???x>", name="Button 4"},
   }
   remote.define_auto_inputs(inputs)
end
```

We create an array for all pattern-matching entries, which is passed to the `remote.define_auto_inputs()` function. For each incoming MIDI event, Remote will search this array from the top down, until it finds a pattern matching the event. It then extracts values from parts of the MIDI event and builds a Remote message using these values. Each entry translates to a Remote message for one specific control surface item, specified by the `name` field. There may be many pattern matches for one control surface item.

The pattern string represents a MIDI message in hexadecimal notation. Let’s look at the first entry in `inputs`. It matches MIDI messages “`b? 40 xx`”, where “?” is a wild card character (we ignore the MIDI channel). `xx` is the new position of the fader. The auto-input system uses the value of the x variable when translating to a Remote message.

The pattern for “Pitch Bend” has two variables, because the range of pitch bend is 14 bits. We must therefore extract variables x and y and write a custom value translation expression: `value=”y*128 + x”`.

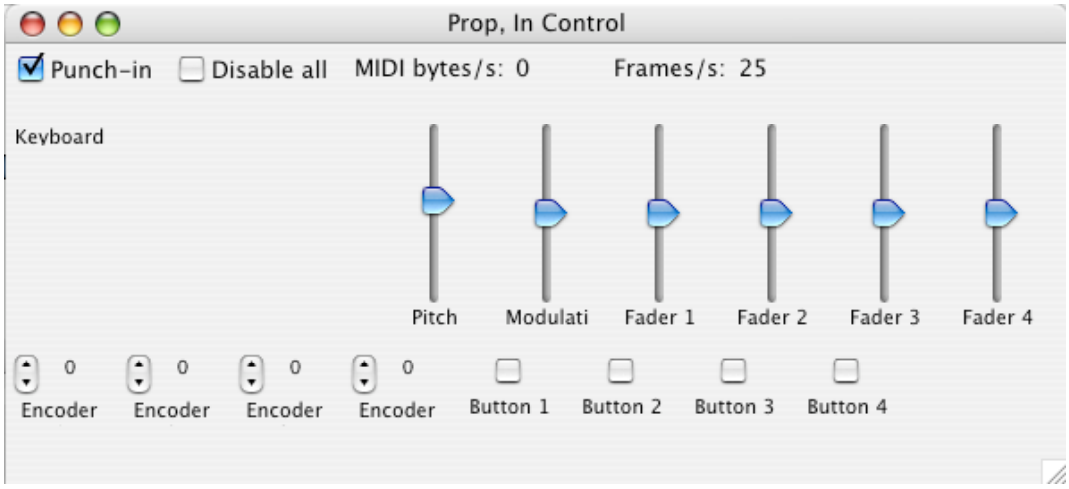
Keyboards sometimes require two pattern matches. The second match in our example, `"<100x>? yy zz"`, is the generic pattern for a standard MIDI keyboard message. The `"<100x>"` pattern is a bit-wise match (instead of hexadecimal). It will match a note on message (hex 9) and a note off message (hex 8), setting x to 1 if note on and 0 if note off. The extracted y variable is the note number and z is velocity. The auto input system will automatically use the variables in this way, so we need not specify any expressions.
The first keyboard match is a special case used for keyboards that send note on with zero velocity instead of note off. We specify expressions that explicitly set the value to 0 (note off) and the velocity to 64.

The patterns for the encoders, `"b? 50 <???y>x"`, extract the amount of the position change into x and the direction of the change into y. The y bit is 1 if the encoder is turned left. This requires a special expression to convert into delta values.

For the buttons we simply extract the last bit, which indicates if it's a press or release message.

5.1.4 Codec Test

When we have come this far we can test our codec using the Codec Test program. Select "Choose Codec File" from the "Test" menu, and find InControl.luacodec using the dialog that opens. Then open the "Create Control Surface" dialog from the "Test" menu and select Model and MIDI ports. This opens a test window for the control surface:



The window shows all the control surface items, using different GUI controls. With this you can test if the codec works. It's better to use the Codec Test instead of Reason, because it shows all control surface items. To test with Reason one needs a Remote map, which is an extra source of errors you don't want to test at this stage. Always test all control surface items before shipping a codec.

The controls are on purpose quite abstract (because the items they represent have abstract types).
A keyboard is shown as a text field, where the latest note on/off message is displayed. Value input items (Pitch, Mod and the Faders) are shown as sliders.

Delta inputs are shown as up/down buttons in combination with a text field that shows the accumulated value (of all delta changes).
Buttons are shown as checkboxes.

5.1.5 Auto-detection

The In Control model supports the standard MIDI identity request. This can be used to auto-detect the surface, by implementing the `remote_probe()` function:

```
function remote_probe()
   return {
      request="f0 7e 7f 06 01 f7",
      response="f0 7e 7f 06 02 56 66 66 01 01 ?? ?? ?? ?? f7"
   }
end
```

This function returns a table with fields describing MIDI patterns for the request and response messages. In the response message we use wild cards to filter out the last four data bytes, which contain version number. (Since we support all versions of this control surface).

Remote uses these messages when auto-detecting, to scan all available MIDI ports for this control surface. Use the “Auto Detect” menu in the Codec Test program to test this function. (Make sure you “Choose Codec File” first, the auto-detection is run for all models supported by the chosen codec.)

5.1.6 Complete

So the complete codec implementation for In Control (InControl.lua) is:

```
function remote_init()
   local items={
      {name="Keyboard", input="keyboard"},
      {name="Pitch Bend", input="value", min=0, max=16383},
      {name="Modulation", input="value", min=0, max=127},
      {name="Fader 1", input="value", min=0, max=127},
      {name="Fader 2", input="value", min=0, max=127},
      {name="Fader 3", input="value", min=0, max=127},
      {name="Fader 4", input="value", min=0, max=127},
      {name="Encoder 1", input="delta"},
      {name="Encoder 2", input="delta"},
      {name="Encoder 3", input="delta"},
      {name="Encoder 4", input="delta"},
      {name="Button 1", input="button"},
      {name="Button 2", input="button"},
      {name="Button 3", input="button"},
      {name="Button 4", input="button"},
   }
   remote.define_items(items)

   local inputs={
      {pattern="b? 40 xx", name="Fader 1"},
      {pattern="b? 41 xx", name="Fader 2"},
      {pattern="b? 42 xx", name="Fader 3"},
      {pattern="b? 43 xx", name="Fader 4"},
      {pattern="e? xx yy", name="Pitch Bend", value="y*128 + x"},
      {pattern="b? 01 xx", name="Modulation"},
      {pattern="9? xx 00", name="Keyboard", value="0", note="x", velocity="64"},
      {pattern="<100x>? yy zz", name="Keyboard"},
      {pattern="b? 50 <???y>x", name="Encoder 1", value="x*(1-2*y)"},
      {pattern="b? 51 <???y>x", name="Encoder 2", value="x*(1-2*y)"},
      {pattern="b? 52 <???y>x", name="Encoder 3", value="x*(1-2*y)"},
```

```
      {pattern="b? 53 <???y>x", name="Encoder 4", value="x*(1-2*y)"},
      {pattern="b? 60 ?<???x>", name="Button 1"},
      {pattern="b? 61 ?<???x>", name="Button 2"},
      {pattern="b? 62 ?<???x>", name="Button 3"},
      {pattern="b? 63 ?<???x>", name="Button 4"},
   }
   remote.define_auto_inputs(inputs)
end

function remote_probe()
   return {
      request="f0 7e 7f 06 01 f7",
      response="f0 7e 7f 06 02 56 66 66 01 01 ?? ?? ?? ?? f7"
   }
end
```

5.2 In Control II

In Control II is a more advanced model with two-way communication. It has the same number of elements, but the faders are motorized, the encoders have a ring of LEDs and the buttons have on/off LEDs. Let's see how we can modify our codec to add support for this model.

5.2.1 Support two models

First we add In Control II two the list of supported control surfaces, in the InControl.luacodec file:

```
function remote_supported_control_surfaces()
   local surfaces =
      {
         { manufacturer="ACME", model="In Control",
            source="InControl.lua", picture="InControl.png",
            in_ports={ {description="In Port"} },
            out_ports={{description="Out Port", optional=true}},
            has_keyboard=true
         },
         { manufacturer="ACME", model="In Control II",
            source="InControl.lua", picture="InControl2.png",
            in_ports={ {description="In Port"} },
            out_ports={ {description="Out Port"} },
            has_keyboard=true
         },
      }
   return surfaces
end
```

Simply add another entry in the `surfaces` array, and we're done. Note that both surfaces are supported by the same .lua source file, which makes sense in this case, since they are very similar. In other cases you may use one source file for each model.

5.2.2 Initialize

When Remote instantiates a control surface, it passes the wanted manufacturer and model to remote_init(). If you support only one model you can simply ignore these, like we did in the previous example. Now we support two, so we define the items differently depending on the model. For In Control II we define output type "value" for the advanced control surface items:

```
function remote_init(manufacturer, model)
```

```
   if model=="In Control II" then
      local items={
         {name="Keyboard", input="keyboard"},
         {name="Pitch Bend", input="value", min=0, max=16383},
         {name="Modulation", input="value", min=0, max=127},
         {name="Fader 1", input="value", output="value", min=0, max=127},
         {name="Fader 2", input="value", output="value", min=0, max=127},
         {name="Fader 3", input="value", output="value", min=0, max=127},
         {name="Fader 4", input="value", output="value", min=0, max=127},
         {name="Encoder 1", input="delta", output="value", min=0, max=10},
         {name="Encoder 2", input="delta", output="value", min=0, max=10},
         {name="Encoder 3", input="delta", output="value", min=0, max=10},
         {name="Encoder 4", input="delta", output="value", min=0, max=10},
         {name="Button 1", input="button", output="value"},
         {name="Button 2", input="button", output="value"},
         {name="Button 3", input="button", output="value"},
         {name="Button 4", input="button", output="value"},
      }
      remote.define_items(items)
   else
      local items={
         {name="Keyboard", input="keyboard"},
         {name="Pitch Bend", input="value", min=0, max=16383},
         {name="Modulation", input="value", min=0, max=127},
         {name="Fader 1", input="value", min=0, max=127},
         {name="Fader 2", input="value", min=0, max=127},
         {name="Fader 3", input="value", min=0, max=127},
         {name="Fader 4", input="value", min=0, max=127},
         {name="Encoder 1", input="delta"},
         {name="Encoder 2", input="delta"},
         {name="Encoder 3", input="delta"},
         {name="Encoder 4", input="delta"},
         {name="Button 1", input="button"},
         {name="Button 2", input="button"},
         {name="Button 3", input="button"},
         {name="Button 4", input="button"},
      }
      remote.define_items(items)
   end
-- to be continued
```

Note that the delta item now has a min and max value. This range corresponds to the number of LEDs in the ring around the encoder; if the output value is 0 the first LED is lit, if its 10 all are lit. This range is only used on the output value; the encoder input is still a delta change.

When Remote has instantiated one instance, it gets its own Lua environment, including the virtual machine, all item definitions, global variables, etc. If you have two control surfaces supported by the same source file, you can be sure that they are completely separated when instantiated, there are no common global variables between the two.

5.2.3 Automatic output handling

The input handling for In Control II is exactly the same as the first model, but the new model also supports two-way communication with some of the control surface items. The easiest way to handle output to control surface items is through auto outputs. They work like the auto input items, but they create MIDI messages instead of Remote messages. Auto outputs are registered in the `remote_init()` function:

```
-- remote_init() continued
   if model=="In Control II" then
      local outputs={
         {name="Fader 1", pattern="b0 40 xx"},
```

```
            {name="Fader 2", pattern="b0 41 xx"},
            {name="Fader 3", pattern="b0 42 xx"},
            {name="Fader 4", pattern="b0 43 xx"},
            {name="Encoder 1", pattern="b0 50 0x", x="enabled*(value+1)"},
            {name="Encoder 2", pattern="b0 51 0x", x="enabled*(value+1)"},
            {name="Encoder 3", pattern="b0 52 0x", x="enabled*(value+1)"},
            {name="Encoder 4", pattern="b0 53 0x", x="enabled*(value+1)"},
            {name="Button 1", pattern="b0 60 0<000x>"},
            {name="Button 2", pattern="b0 60 0<000x>"},
            {name="Button 3", pattern="b0 60 0<000x>"},
            {name="Button 4", pattern="b0 60 0<000x>"},
        }
        remote.define_auto_outputs(outputs)
    end
end
```

There can only be one auto output definition per control surface item. When the state of an item should be updated, Remote finds the output definition and creates an output MIDI message using the pattern field. The pattern may contain x, y or z placeholders for output variables.

The faders on the In Control II accept the same MIDI as they send, for example "`b0 40 xx`", where `xx` is the requested position of the fader. The default for x is the value of the control surface item. (Remote automatically scales the values of the host's remotable items to fit the ranges of the control surface items.)

The auto output definition for the encoders need a special expression to calculate the x if the MIDI message. If x is 0 all LEDs will be turned off, but we want the first LED to be on if the output value is 0, so we add one to the value. Then we multiply by the "enabled" variable to clear all LEDs if the control surface item is disabled.

The LEDs on the buttons are lit by the last bit of the MIDI message.

5.2.4 Auto-detection

The probe function is also changed to support both models. Remote will call the probe function once for each model, so we simply check which model is requested and return the proper request/response pair:

```
function remote_probe(manufacturer,model)
    if model=="In Control II" then
        return {
            request="f0 7e 7f 06 01 f7",
            response="f0 7e 7f 06 02 56 66 66 01 02 ?? ?? ?? ?? f7"
        }
    else
        return {
            request="f0 7e 7f 06 01 f7",
            response="f0 7e 7f 06 02 56 66 66 01 01 ?? ?? ?? ?? f7"
        }
    end
end
```

5.2.5 Complete

Here's the complete InControl.lua with support for the both control surfaces:

```
function remote_init(manufacturer, model)
    if model=="In Control II" then
```

```
    local items={
        {name="Keyboard", input="keyboard"},
        {name="Pitch Bend", input="value", min=0, max=16383},
        {name="Modulation", input="value", min=0, max=127},
        {name="Fader 1", input="value", output="value", min=0, max=127},
        {name="Fader 2", input="value", output="value", min=0, max=127},
        {name="Fader 3", input="value", output="value", min=0, max=127},
        {name="Fader 4", input="value", output="value", min=0, max=127},
        {name="Encoder 1", input="delta", output="value", min=0, max=10},
        {name="Encoder 2", input="delta", output="value", min=0, max=10},
        {name="Encoder 3", input="delta", output="value", min=0, max=10},
        {name="Encoder 4", input="delta", output="value", min=0, max=10},
        {name="Button 1", input="button", output="value"},
        {name="Button 2", input="button", output="value"},
        {name="Button 3", input="button", output="value"},
        {name="Button 4", input="button", output="value"},
    }
    remote.define_items(items)
else
    local items={
        {name="Keyboard", input="keyboard"},
        {name="Pitch Bend", input="value", min=0, max=16383},
        {name="Modulation", input="value", min=0, max=127},
        {name="Fader 1", input="value", min=0, max=127},
        {name="Fader 2", input="value", min=0, max=127},
        {name="Fader 3", input="value", min=0, max=127},
        {name="Fader 4", input="value", min=0, max=127},
        {name="Encoder 1", input="delta"},
        {name="Encoder 2", input="delta"},
        {name="Encoder 3", input="delta"},
        {name="Encoder 4", input="delta"},
        {name="Button 1", input="button"},
        {name="Button 2", input="button"},
        {name="Button 3", input="button"},
        {name="Button 4", input="button"},
    }
    remote.define_items(items)
end

local inputs={
    {pattern="b? 40 xx", name="Fader 1"},
    {pattern="b? 41 xx", name="Fader 2"},
    {pattern="b? 42 xx", name="Fader 3"},
    {pattern="b? 43 xx", name="Fader 4"},
    {pattern="e? xx yy", name="Pitch Bend", value="y*128 + x"},
    {pattern="b? 01 xx", name="Modulation"},
    {pattern="9? xx 00", name="Keyboard", value="0", note="x", velocity="64"},
    {pattern="<100x>? yy zz", name="Keyboard"},
    {pattern="b? 50 <???y>x", name="Encoder 1", value="x*(1-2*y)"},
    {pattern="b? 51 <???y>x", name="Encoder 2", value="x*(1-2*y)"},
    {pattern="b? 52 <???y>x", name="Encoder 3", value="x*(1-2*y)"},
    {pattern="b? 53 <???y>x", name="Encoder 4", value="x*(1-2*y)"},
    {pattern="b? 60 ?<???x>", name="Button 1"},
    {pattern="b? 61 ?<???x>", name="Button 2"},
    {pattern="b? 62 ?<???x>", name="Button 3"},
    {pattern="b? 63 ?<???x>", name="Button 4"},
}
remote.define_auto_inputs(inputs)

if model=="In Control II" then
    local outputs={
        {name="Fader 1", pattern="b0 40 xx"},
        {name="Fader 2", pattern="b0 41 xx"},
        {name="Fader 3", pattern="b0 42 xx"},
        {name="Fader 4", pattern="b0 43 xx"},
        {name="Encoder 1", pattern="b0 50 0x", x="enabled*(value+1)"},
        {name="Encoder 2", pattern="b0 51 0x", x="enabled*(value+1)"},
        {name="Encoder 3", pattern="b0 52 0x", x="enabled*(value+1)"},
        {name="Encoder 4", pattern="b0 53 0x", x="enabled*(value+1)"},
        {name="Button 1", pattern="b0 60 0<000x>"},
        {name="Button 2", pattern="b0 60 0<000x>"},
```

```
         {name="Button 3", pattern="b0 60 0<000x>"},
         {name="Button 4", pattern="b0 60 0<000x>"},
      }
      remote.define_auto_outputs(outputs)
   end
end

function remote_probe(manufacturer,model)
   if model=="In Control II" then
      return {
         request="f0 7e 7f 06 01 f7",
         response="f0 7e 7f 06 02 56 66 66 01 02 ?? ?? ?? ?? f7"
      }
   else
      return {
         request="f0 7e 7f 06 01 f7",
         response="f0 7e 7f 06 02 56 66 66 01 01 ?? ?? ?? ?? f7"
      }
   end
end
```

5.3 In Control Deluxe

In Control Deluxe is an advanced version of In Control II, with a display and a joystick. The new control items cannot be handled by the automatic input/output system, so we must write some Lua code for them.

5.3.1 New Surface Items

We choose to present the joystick as two control surface items, one vertical and one horizontal.
The LCD is presented as one control surface item. It has no input type, only output type “text”. (A codec can choose to divide an LCD into many sections, and define control surface items for each section. We won’t do that because our LCD is only 16 characters wide.)

```
function remote_init(manufacturer, model)
   assert(model=="In Control Deluxe")
   local items={
      {name="Keyboard", input="keyboard"},
      {name="Pitch Bend", input="value", min=0, max=16383},
      {name="Modulation", input="value", min=0, max=127},
      {name="Joystick X", input="value", output="text", min=0, max=127},
      {name="Joystick Y", input="value", output="text", min=0, max=127},
      {name="Fader 1", input="value", output="value", min=0, max=127},
      {name="Fader 2", input="value", output="value", min=0, max=127},
      {name="Fader 3", input="value", output="value", min=0, max=127},
      {name="Fader 4", input="value", output="value", min=0, max=127},
      {name="Encoder 1", input="delta", output="value", min=0, max=10},
      {name="Encoder 2", input="delta", output="value", min=0, max=10},
      {name="Encoder 3", input="delta", output="value", min=0, max=10},
      {name="Encoder 4", input="delta", output="value", min=0, max=10},
      {name="Button 1", input="button", output="value"},
      {name="Button 2", input="button", output="value"},
      {name="Button 3", input="button", output="value"},
      {name="Button 4", input="button", output="value"},
      {name="LCD", output="text"},
   }
   remote.define_items(items)
-- continued
```

Since we'll be writing custom code for the joystick and LCD, we save the index to these items in global variables. The "item index" is used in all Remote functions to refer to a control surface item (instead of using their names, which would mean passing around a lot of strings).

```
-- remote_init() continued

   g_lcd_index=table.getn(items)
   g_joystick_x_index=4
   g_joystick_y_index=5

-- auto inputs and outputs just like In Control II
```

The auto inputs and outputs are the same as In Control II.

5.3.2 The Joystick

When Remote receives incoming MIDI for a control surface, it calls `remote_process_midi()` (if the codec has defined that function). The function is called for each MIDI event. We use it to handle our joystick's messages:

```
g_last_joystick_x=-1
g_last_joystick_y=-1

function remote_process_midi(event)
   ret=remote.match_midi("f0 11 22 xx yy f7",event)
   if ret~=nil then
      if g_last_joystick_x~=ret.x then
         local msg={ time_stamp=event.time_stamp, item=g_joystick_x_index, value=ret.x }
         remote.handle_input(msg)
         g_last_joystick_x=ret.x
         remote_on_auto_input(g_joystick_x_index)
      end
      if g_last_joystick_y~=ret.y then
         local msg={ time_stamp=event.time_stamp, item=g_joystick_y_index, value=ret.y }
         remote.handle_input(msg)
         g_last_joystick_y=ret.y
         remote_on_auto_input(g_joystick_y_index)
      end
      return true
   end
   return false
end
```

The joystick sends a MIDI sysex containing both the vertical and horizontal position. We use the utility function `remote.match_midi()` to compare the incoming `event` with the joystick sysex. If there is a match the function returns a table containing the extracted values for x and y.
The new x and y values are compared with the previous values, stored in global variables. If they have changed we create a Remote message and call `remote.handle_input()` to pass it on to the host. So one incoming MIDI event will sometimes be translated into two Remote messages, which is why we had to write custom code to handle the joystick.

A Remote message is a table containing the index to the sending control surface item, the new value and the time stamp of the message. The time stamp should never be changed! It's a high precision, non-portable value used to record automation with high accuracy. Just copy the `time_stamp` field from the MIDI event.

If we find a match for the event we return `true`, which means the event was used. Otherwise we return `false`, and Remote will try the event with the auto inputs.

See the section on feedback below for an explanation of the `remote_on_auto_input()` call.

5.3.3 LCD output

Remote output is handled through two functions, `remote_set_state()` and `remote_deliver_midi()`. The first is called regularly with info about control surface item changes:

```
g_is_lcd_enabled=false
g_lcd_state=string.format("%-16.16s"," ")
g_delivered_lcd_state=string.format("%-16.16s","#")

g_feedback_enabled=false

function remote_set_state(changed_items)
   for i,item_index in ipairs(changed_items) do
      if item_index==g_lcd_index then
         g_is_lcd_enabled=remote.is_item_enabled(item_index)
         new_text=remote.get_item_text_value(item_index)
         g_lcd_state=string.format("%-16.16s",new_text)
      end
   end
end
```

`changed_items` is an array of indexes to the control surface items that have changed. This array can contain indexes to all items that have output, even those registered as auto outputs. We are only interested in the LCD so we loop through the array looking for `g_lcd_index`. If the LCD has changed we call `remote.is_item_enabled()`, to check if it's enabled (which means it's in use, mapped to something in the host). Then we call `remote.get_item_text_value()` to get the new text for the LCD. The new "enabled state" and text are stored in global variables.

We use the function `string.format()` to make sure `g_lcd_state` always is 16 characters, left justified. The return from `get_item_text_value()` can be of any length, depending on how the control surface item is mapped, so we must do fill our LCD display.

The global variable `g_delivered_lcd_state` holds the last text we sent to the LCD. The host calls the function `remote_deliver_midi()` when it's time to send MIDI. Our delivery function looks like this:

```
function remote_deliver_midi()
   local ret_events={}
   local new_text=g_lcd_state
   if g_delivered_lcd_state~=new_text then
      assert(string.len(new_text)==16)
      local lcd_event=make_lcd_midi_message(new_text)
      table.insert(ret_events,lcd_event)
      g_delivered_lcd_state=new_text
   end
   return ret_events
end
```

It's quite simple: compare the new g_lcd_state with the delivered state; if they differ we create an LCD update event. The LCD event is a MIDI sysex, created by the following function:

```
local function make_lcd_midi_message(text)
   local event=remote.make_midi("f0 11 22 33 10")
   start=6
   stop=6+string.len(text)-1
   for i=start,stop do
      sourcePos=i-start+1
      event[i] = string.byte(text,sourcePos)
   end
   event[stop+1] = 247          -- hex f7
   return event
end
```

It calls `remote.make_midi()` to create the beginning of the sysex (for an In Control Deluxe LCD update event). Then it adds the text string by copying the byte for each character, and finally adds the end-of-sysex marker.

5.3.4 Set state vs deliver MIDI

Why do we use the dual functions `remote_set_state()` and `remote_deliver_midi()`? Well, they usually do two quite different things. `remote_deliver_midi()` represents the MIDI cable; it should be used to translate variables to MIDI events. You may call the variables it translates the "machine state". These variables correspond exactly to the physical elements on the control surface; an LCD with 16 characters is represented by a 16-character string.

The control surface state, on the other hand, is not always exactly the same as the physical machine state. An LCD display may be presented as more than one text item and on top of that show song position, temporary parameter feedback, etc. By presenting alternative control surface items for the LCD we can use the Remote Map to decide which items should be shown, depending on the current remote targets.

`remote_set_state()` is the place where all these items and temporary texts are merged into the machine state. This function gets information about items that have changed, and can use this to decide what should be shown the given moment.

This makes a clean design, with clear responsibilities. `remote_deliver_midi()` will always remain the same for one control surface, since the physical control surface elements are still updated using the same MIDI messages. To add functionality or control surface items, we only need to change `remote_set_state()`, which we'll see in the next section when we add parameter feedback.

5.3.5 Parameter feedback

If your control surface has a display, you should consider using it to show parameter feedback. The feedback shows the name and value of the parameter being changed, when the user moves a control surface item. If a control surface has displays besides all its control items you could show their values all the time, but In Control Deluxe has only one display so we'll show temporary feedback for the last touched item.

We don't want to show any values changed only by the host, for example automated values, so we add a function to store the index to the last item that sent us input:

```
g_last_input_time=-2000
g_last_input_item=nil

function remote_on_auto_input(item_index)
   if item_index>3 then
      g_last_input_time=remote.get_time_ms()
      g_last_input_item=item_index
   end
end
```

The function `remote_on_auto_input()` is called by remote after it has handled an auto input item. Its parameter is the index of the control surface item. We store this index and the current time in global variables.
Control surface items at index 1, 2 and 3 (keyboard, pitch and mod wheel) are performance parameters. It makes no sense to show feedback for them.

We also call this function from our `remote_process_midi()` function, to get feedback on the Joystick as well.

In `remote_set_state()` we add code that checks our global variables and writes the feedback on the LCD display:

```
g_feedback_enabled=false

function remote_set_state(changed_items)
   for i,item_index in ipairs(changed_items) do
      if item_index==g_lcd_index then
         g_is_lcd_enabled=remote.is_item_enabled(item_index)
         new_text=remote.get_item_text_value(item_index)
         g_lcd_state=string.format("%-16.16s",new_text)
      end
   end

   local now_ms = remote.get_time_ms()
   if (now_ms-g_last_input_time) < 1000 then
      if remote.is_item_enabled(g_last_input_item) then
         local feedback_text=remote.get_item_name_and_value(g_last_input_item)
         if string.len(feedback_text)>0 then
            g_feedback_enabled=true
            g_lcd_state=string.format("%-16.16s",feedback_text)
         end
      end
   elseif g_feedback_enabled then
      g_feedback_enabled=false
      if g_is_lcd_enabled then
         old_text=remote.get_item_text_value(g_lcd_index)
      else
         old_text=" "
      end
      g_lcd_state=string.format("%-16.16s",old_text)
   end
end
```

After we checked the `changed_items`, we get the current time and compare against the last input time. If it was less than a second ago, and the control surface item is enabled (used by the host), we write the feedback on the display. Our display has only one row so we call `remote.get_item_name_and_value()` which returns the name and value of the remotable item being changed, formatted by the host in a nice way. (For example "Volume: 111" or

"Freq: 12Hz"). Since we show the value for one second after the last input, the parameter feedback should be shown all the time while the user moves a fader, for example.

When the time is up, the display should once again show the value of the LCD control surface item. `remote_set_state()` is called at regular intervals even if no items have changed, so we write an else block that checks `g_feedback_enabled` to see if the display has just shown feedback then restores the LCD (machine) state.

5.3.6 Prepare and Release

Many control surfaces require some setting up before being used by Remote. If this setup can be done through MIDI, your codec should implement `remote_prepare_for_use()`. This function should return MIDI events that are sent when the surface is used:

```
function remote_prepare_for_use()
   g_delivered_lcd_state=string.format("%-16.16s","ACME In Control")
   return {
      remote.make_midi("f0 11 22 21 01 f7"),
      make_lcd_midi_message("ACME In Control")
   }
end
```

For In Control Deluxe, we send the enable LCD backlight sysex message and an LCD text message. (We also store this text message in `g_delivered_lcd_state`, since it should mirror the physical machine state.)

When Remote releases the control surface it calls `remote_release_for_use()`. We use it to display a "disconnected" text:

```
function remote_release_from_use()
   return {
      make_lcd_midi_message("Remote disconnect")
   }
end
```

These functions are often necessary to use for control surfaces that can be programmed to send different MIDI messages, to adapt the surface to different software programs or synths. Since Remote has the Map system that associates control surface items with remotable items, there is no need for programmable buttons and knobs. Remote only needs to setup the control items once when it instantiates, or takes control of, the control surface. Sometimes a surface also needs to be reset to a default state when released.

6 Codec Reference

6.1 Data types

6.1.1 MIDI event

Some of the functions in Remote use MIDI events. Each MIDI event is an array containing the MIDI bytes, starting at index 1. The array or table also contains the following fields:
`size` - the size of the MIDI event, in bytes.
`port` – an optional field specifying port number. This is the port an input message came from or the port where the message should be sent, if outgoing.
`time_stamp` – an exact time stamp on an incoming MIDI message. This value should never be changed, it must be passed back to Remote as is. An outgoing MIDI message should not have a `time_stamp` field. It will be sent immediately.

The following example creates a sysex message addressed to port two.

```
local event={ 240, 67, 77, 88, 247 }
event.size=5
event.port=2
```

6.1.2 MIDI mask

Some functions use a MIDI mask, to match against incoming MIDI events, or to create outgoing MIDI events. The MIDI mask is a string containing the MIDI data in hexadecimal form, for example "`f0 78 66 88 f7`".
A MIDI mask can contain question marks as wild card characters to match against anything: "`9? 69 58`".
The mask can contain the variables x, y and z, which are used to extract values from incoming messages, or to put values into outgoing messages: "`90 xx yy`"
The MIDI mask can be written in binary form or in mixed hexadecimal and binary form. Binary digits must be enclosed in brackets. This is useful to extract individual bits of a message, for example "`f0 00 7<00xx> f7`", which extracts the two lowest bits of the third byte.

6.2 Luacodec callback functions

6.2.1 remote_supported_control_surfaces()

This function must be put in the .luacodec file to list the control surface models supported by the codec. It should return an array, containing one entry for each model. Each entry must contain the following fields:
`manufacturer` – a string containing the manufacturer name.
`model` – a string containing the control surface model name.
`source` – the name of the Lua source file for this model.

`picture` – the name of the picture file for this model. It must be a 96*96 pixel PNG file.
`in_ports` – an array of properties for each input port this model requires. See below.
`out_ports` – an array of properties for each output port this model requires. See below.
There are also some optional attributes:
`has_keyboard` –set to true of this model has a keyboard.
`setup_info_text` –a string containing information about setting up this control surface. This string is shown in the control surface setup preferences in the host.
`setup_warning_text` – a string used to warn the user about some aspect of the control surface setup. It will be shown in an alert when the control surface is added or auto-detected, with an option for the user to cancel.
`setup_user_action_text` – this is a text instructing the user to setup the control surface. This should be a more direct instruction than the setup info text; it's shown in a dialog when the control surface is added or auto-detected.

Each entry in the `in_ports` and `out_ports` arrays contain:
`description` – a string describing the in port (if a control surface has multiple ports it is necessary to tell them apart somehow). This description is shown in the control surface setup in the host.
`optional` – an optional field, set to `true` if the port is optional.

Here's an example entry for a control surface connected through one out port, and two in ports (of which one is optional).

```
local supported_surfaces =
   {
      { manufacturer="ACME", model="Project Z",
         source="Z.lua", picture="Z.png",
         in_ports={
            {description="Keyboard In Port"},
            {description="Control In Port", optional=true}
         },
         out_ports={{description="Out Port", optional=true}},
         has_keyboard=true,
         setup_info_text="When using Z with Project Y, select memory bank 0.",
         setup_warning_text="This will erase any presets store in bank 0.",
         setup_user_action_text="Please select memory bank memory bank 0 on your Z"
      },
   }
```

6.3 Callback functions

These functions are called by Remote.

6.3.1 remote_deliver_midi(max_bytes, port)

This function is called at regular intervals when the host is due to update the control surface state. The return value should be an array of MIDI events.
`max_bytes` is the recommended maximum for the total size of all the returned events. It's calculated by the host, from the MIDI bandwidth and the output frequency. This max can be ignored. In that case the returned MIDI events will be buffered, and the host may skip some of the following calls to `remote_deliver_midi()` instead.
`port` is the index of the output port where this delivery is going to be sent. If the control surface has more than one out port, the host will call `remote_deliver_midi()` once for each port.

Auto output items are handled before this function is called. If there are many changes of auto-outputs they might fill the MIDI out buffer, which means `remote_deliver_midi()` will not be called until the buffer is cleared. This can be used as a simple priority scheme; control surface items that must be updated quickly, such as motorized faders, should be registered as auto-outputs.

6.3.2 remote_init(manufacturer, model)

`remote_init()` is called when a control surface instance is setup for use. Each instance of a control surface uses its own Lua environment. This function should be used to register all control surface items, using `remote.define_items()`. It can also be used to register automatic handling of input and output, with `remote.define_auto_inputs()` and `remote.define_auto_outputs()`. The `define-` functions can only be called from `remote_init()`. `manufacturer` and `model` are strings specifying the selected model.

6.3.3 remote_on_auto_input(item_index)

This function is called by Remote after an auto-input item message has been handled. The typical use is to store the current time and item index, for timed feedback texts. `item_index` is the index to the item.

6.3.4 remote_prepare_for_use()

Is called when a control surface is initialized. It should return an array of MIDI events, used to setup the control surface. If the control surface uses many output ports, the MIDI events should be addressed to the proper ports using the `port` field of each MIDI event, for example:

```
function remote_prepare_for_use()
   local retEvents={
      remote.make_midi("f0 66 66 66 14 0a 02 f7", { port=1 } ),
      remote.make_midi("f0 66 66 66 14 0a 02 f7", { port=2 } ),
   }
   return retEvents
end
```

6.3.5 remote_probe(manufacturer, model, prober)

This function is called when Remote is auto-detecting surfaces. `manufacturer` and `model` are strings specifying the model being auto-detected. This function is always called once for each supported model.
There are two ways to handle the auto-detection. The easiest way is to return a table containing a request message and a response mask. If remote gets such a return value it will use these messages to scan all MIDI ports automatically.
The request and response are strings containing MIDI masks. They may not contain variable references (x, y or z). The response mask can contain wild cards, which is useful to ignore version specific information from the control surface. A typical request/response looks like this:

```
return { request = "f0 7e 7f 06 01 f7",
   response = "f0 7e 7f 06 02 6b 6d 01 00 00 ?? ?? ?? ?? f7" }
```

This example shows the standard MIDI identity request and response messages. (In the response message, the version bytes are masked out using wild cards.) The request/response does not have to be MIDI standard; some control surfaces use other messages.

To auto-detect more complex control surfaces, for example those with multiple in- and out-ports, the automatic scanning cannot be used. In that case this function can be used to write code that scans the MIDI ports in any way necessary. The `prober` argument is a table containing properties and functions used to accomplish this. It has the following fields:
`prober.in_ports` – the number of MIDI in ports available on the system.
`prober.out_ports` – the number of MIDI out ports available on the system.
`prober.midi_send_function` – a function that sends MIDI to a port. Its first parameter is the destination port index (`1 - out_ports`), the second parameter is an array of MIDI events.
`prober.midi_receive_function` – a function that collects MIDI data received on a port. It takes as parameter the input port index (`1 - in_ports`), and returns an array of MIDI events received on that port.
`prober.wait_function` – a function used to wait for the response messages.

A typical probing algorithm loops through all outputs given by `out_ports`. For each output it sends some request messages using the `midi_send_function`, then waits a while before checking for answers on all `in_ports`, using the `midi_receive_function()`.

When this method is used the return value from `remote_probe()` should be an array containing a table for each control surface that was detected (since the user can connect several control surfaces of the same model). Each result table must contain to fields:
`in_ports` – an array of indexes to the in ports used by the detected control surface.
`out_ports` – an array of indexes to the out ports used by the control surface.

Here's en example probe function, searching for a control surface connected to two inputs:

```
function remote_probe(manufacturer,model,prober)
   local request_events={remote.make_midi("f0 7e 7f 06 01 f7")}
   local response="f0 66 66 06 02 41 62 01 00 00 03 ?? ?? ?? f7"

   local function match_events(mask,events)
      for i,event in ipairs(events) do
         local res=remote.match_midi(mask,event)
         if res~=nil then
            return true
         end
      end
      return false
   end

   local results={}
   for outPortIndex = 1,prober.out_ports do
      prober.midi_send_function(outPortIndex,request_events)
      prober.wait_function(50)
      local responding_ports={}
      for inPortIndex = 1,prober.in_ports do
         local events=prober.midi_receive_function(inPortIndex)
         if match_events(response,events) then
            table.insert(responding_ports,inPortIndex)
         end
```

```
      end
      if responding_ports[1]~=nil then
         local ins={ responding_ports[1] }
         if responding_ports[2]~=nil then
            -- Second in port is optional
            table.insert(ins,responding_ports[2])
         end
         local one_result={ in_ports=ins, out_ports={outPortIndex}}
         table.insert(results,one_result)
      end
   end
   return results
end
```

6.3.6 remote_process_midi(event)

This function is called for each incoming MIDI event. This is where the codec interprets the message and translates it into a Remote message. The translated message is then passed back to Remote with a call to `remote.handle_input()`. If the event was translated and handled this function should return `true`, to indicate that the event was used. If the function returns false, Remote will try to find a match using the automatic input registry defined with `remote.define_auto_inputs()`.

6.3.7 remote_set_state(changed_items)

`remote_set_state()` is called regularly to update the state of control surface items. `changed_items` is a table containing indexes to the items that have changed since the last call.

A common use for this function is useful to create a new “machine state”, mirroring the state of the whole control surface, at that point in time. The new machine is later used to compare with a delivered machine state when `remote_deliver_midi()` is called.
This function is called even if no items have changed, so it can be used to clear temporary feedback texts, etc.

6.3.8 remote_release_from_use()

Is called when a control surface is no longer used by the Remote host. It should return an array of MIDI events that reset the control surface. If the control surface uses many output ports, the MIDI events should be addressed to the proper ports using the `port` field of each MIDI event.

6.4 Utility functions

6.4.1 remote.define_auto_inputs(inputs)

Registers auto inputs. The parameter `inputs` is an array of MIDI matching items that define how Remote should interpret incoming MIDI. Each entry in the array defines one MIDI mask, which is associated with one control surface item. For each incoming MIDI message Remote searches this array, from the top down, until there is a match. When there is a match a Remote message for the associated control surface item is built and sent to the host. The values for the message are extracted from the MIDI message using expressions given in the auto input entry.

This function can only be called from `remote_init()`.

An entry in the auto input array must contain the following fields:
`name` – the name of the control surface item this entry is handling.
`pattern` – the MIDI mask for this entry. May contain wild-cards (?) and value references (x,y and z).
The following fields are optional:
`value` – an expression that calculates the value of the message. The default is `value="x"`.
`note` – expression for the note number of a keyboard message. The default is `note="y"`.
`velocity` – expression for the velocity of a keyboard message. The default is `velocity="z"`.
`port` – a port index, if matching messages from one specific input port.

The expressions are Lua scripts. They can call global functions define in the codec, but that should be avoided. In fact try to keep the expressions as simple as possible, to save processing time.

Keyboard messages can only be sent from control surface items of keyboard type. For all other types, the `note` and `velocity` values are ignored. The `value` of a keyboard message is `1` for note on and `0` for note off.

The `port` field is useful if the control surface is connected using multiple MIDI ports, to match against messages from one specific port. If no port is given, MIDI events from all ports will match.

An example with some common masks:

```
local inputs={
   -- Converts note on with 0 velocity to note off (special case of the next entry)
   {pattern="9? xx 00", name="Keyboard", value="0", note="x", velocity="64" },
   -- Note on/off. Needs no expressions because the default is value=x, note=y and vel=z
   {pattern="<100x>? yy zz", name="Keyboard" },
   -- Pitch bend is 14 bits. Calculate value using x and y.
   {pattern="e? xx yy", name="Pitch Bend", value="y*128+x" },
   {pattern="b? 01 xx", name="Modulation" },
   -- Last bit indicates button press or release.
   {pattern="b? 33 ?<???x>", name="Button 1" },
   -- This button only send MIDI when pressed, hence value=1.
   {pattern="f0 00 44 66 77 f7", name="Start Button", value="1" },
}
remote.define_auto_inputs(inputs)
```

Note that the first two masks are associated with the same control surface item. It's possible to have any number of masks for an item.
Also note that few of these examples need expressions to calculate the resulting message values. By picking out the exact bits used for the values, the default expression (`value="x"`) is used, which is faster than other expressions.

Control surface items handled by auto inputs need not be handled in `remote_process_midi()`. The auto inputs are usually handled much faster, so it should be the preferred method. `remote_process_midi()` can be used when the auto input mechanism is inadequate.

6.4.2 remote.define_auto_outputs(outputs)

Registers auto outputs. The parameter `outputs` is an array with MIDI mask items that define how Remote should create outgoing MIDI when the host's remotable items change.
Each entry in the array corresponds to one control surface item, and contains a MIDI pattern used to create a MIDI message.
This function can only be called from `remote_init()`.

An entry in the auto output array must contain the following fields:
`name` – the name of the control surface item this entry is handling.
`pattern` – the MIDI mask for this entry. May contain variable references (x,y and z).
The following entries are optional:
`x` – an expression that calculates x for the MIDI mask. Default is `x="value"`.
`y` – expression for y. Default is `y="mode"`.
`z` – expression for z. Default is `z="enabled"`.
`port` – a port index, if messages should be addressed to a specific output port.

The expressions may use the variables `value`, `mode` and `enabled`. `value` is the new value of the control surface item. `mode` is the current mode selected for the control surface item. `enabled` is `1` if the control surface item is mapped, `0` otherwise.

The expressions are Lua scripts. They can call global functions define in the codec, but that should be avoided. In fact try to keep the expressions as simple as possible, to save processing time.

The `port` field is useful if the control surface is connected using multiple MIDI ports, to send MIDI to a specific port. If not given the event is sent to port 1.

An example with some common masks:

```
local outputs={
   -- Use default expression x=value for mod wheel
   {name="Modulation", pattern="b0 01 xx" },
   -- Button with LED. Last bit lights LED.
   {name="Button 1", pattern="b0 33 0<000x>" },
   -- Fader 1 is 14 bits. Use expression with bit manipulation utils to calculate x and y.
   {name="Fader 1", pattern="e0 xx yy", x="bit.band(value,7)", y="bit.rshift(value,7)" },
   -- Encoder with LED ring. Switch off all LEDs when disabled.
   {name="Encoder", pattern="c0 33 0x", x="(value+1)*enabled" },
   -- Button with LED and modes. The x bit lights LED, y sets mode.
   {name="Button 2", pattern="b0 36 y<000x>", y="get_mode_bits(mode)" },
}
remote.define_auto_outputs(outputs)
```

Note the last entry, which calls a function to create the mode bits of the message.

Auto outputs are handled before `remote_deliver_midi()` is called. If the auto output events fill the MIDI output queue, `remote_deliver_midi()` will not be called until those MIDI events have been sent.
Auto outputs cannot be used for text output. Use `remote_set_state()` and `remote_deliver_midi()` for that.

6.4.3 remote.define_items(items)

Registers all control surface items. `items` is a array with one entry for each item. The item's index in the array is important. This index is later used in all other functions that refer to control surface items.
This function can only be called from `remote_init()`.

An entry in the array has the following fields:
`name` – the control surface item's name.
`input` – the input type of the item.
`output` – the output type of the item.
`min` – the minimum value of the item.
`max` – the maximum value of the item.
`modes` – an optional array of modes this item supports.

An item must define one of the input and output types, or both. Some types need a definition of min and max values. The following tables list the available types:

Input type	Description	Min	Max
"button"	Sends 1 when pressed, 0 when released	Always 0	Always 1
"value"	A fader or a knob	Any	Any
"delta"	An encoder; sends positive or negative delta values.	N/A	N/A
"keyboard"	One item for the whole keyboard. Sends note and velocity as extra data.	0 (note off)	1 (note on)
"noinput"	Only output, e.g. display or LED.	N/A	N/A

Output type	Description	Min	Max
"value"	A fader or a knob or a LED	Any	Any
"text"	A display.	N/A	N/A
"nooutput"	No output. The host never sets its state.	N/A	N/A

The input and output types can be combined. An encoder with LED rings, for example, has "delta" input type because it sends value changes, and "value" output type with min and max corresponding to the number of LEDs.

Some control surface items have different modes, for example an LED or display that can show different colors. The `modes` field should contain an array of strings with the names of the different modes. Modes can be used freely by the codec implementer, as a means to change how a control surface item looks or behaves. It cannot, however, be used to change the attributes of the item, i.e. the input/output types or the min/max setting.
The mode of an item is set in the Remote Map, depending on the remotable item being controlled. A change of mode will trigger `remote_set_state()`, and the codec can query the current mode using `remote.get_item_mode()`.

Some example item definitions:

```
local items={
   -- Simple button (with item index 1)
   { name="Play Button", input="button" },
   -- Button with an LED
   { name="Record Button", input="button", output="value" },
   { name="Knob 1", input="value", min=0, max=127 },
   { name="Fader 1", input="value", output="value", min=0, max=16383 },
   { name="Data Wheel", input="delta" },
   { name="Encoder 1", input="delta", output="value", min=0, max=10 },
   { name="LCD 1", output="text", modes={ "Dark", "Backlight" } },
   { name="Peak Meter", output="value", min=0, max=20 },
}
remote.define_items(items)
```

6.4.4 remote.get_item_mode(item_index)

Returns the current mode of the control surface item given by its index in `item_index`. The return value is a number corresponding to a mode in the mode list defined for the control surface item. If the item has no modes the return is always 1.

6.4.5 remote.get_item_name(item_index)

Returns the name of the remotable item mapped to the given control surface item.

6.4.6 remote.get_item_name_and_value(item_index)

Returns a string containing the name and current value of the remotable item mapped to the given control surface item. This string may be formatted in any way that host program desires, for example "`Level: -3 db`" or "`Pan: -12`".

6.4.7 remote.get_item_short_name(item_index)

Returns a string containing the short version of the name of the remotable item mapped to the given control surface item. The max size of the short name is eight characters.

6.4.8 remote.get_item_short_name_and_value(item_index)

Returns a string containing the short version of the name-and-value of the remotable item mapped to the given control surface item. The max size of the short name-and-value is sixteen characters.

6.4.9 remote.get_item_shortest_name(item_index)

Returns a string containing the shortest version of the name of the remotable item mapped to the given control surface item. The max size of the shortest name is four characters.

6.4.10 remote.get_item_shortest_name_and_value(item_index)

Returns a string containing the shortest version of the name-and-value of the remotable item mapped to the given control surface item. The max size of the shortest name-and-value is eight characters.

6.4.11 remote.get_item_state(item_index)

Returns a table with the complete state of the given item. The table has the following fields:
`is_enabled` – true if the control surface item is mapped/enabled
`value` – the value of the item
`mode` – mode the current mode for the item
`remote_item_name` – the name of the remotable item mapped
`text_value` – the text value of the remotable item
`short_name` – the short version of the name (8 characters maximum)
`shortest_name` – the shortest version of the name (4 chars)
`name_and_value` – the name and value combined
`short_name_and_value` – the short version of name-and-value (16 chars)
`shortest_name_and_value` - the shortest version of name-and-value (8 chars)

Here's an example for a fader mapped to "Channel 7 Level" on Reason's Mixer:

```
is_enabled              true
value                   741
mode                    0
remote_item_name        "Channel 7 Level"
text_value              "92"
short_name              "Level"
shortest_name           "Lvl"
same_and_value          "Channel 7 Level 92"
short_name_and_value    "Level 92"
shortest_name_and_value "Lvl 92"
```

Note that the `value` field contains the level scaled to fit the control surface item, which in this example has the range 0-1023, while the `text_value` field shows the value of the remotable item, which in this case has the range 0-127.

6.4.12 remote.get_item_text_value(item_index)

Returns a string containing the text value of the remotable item mapped to the given control surface item. If the remotable item's output type is other than text, the string will contain its value converted to a suitable string, for example "`35%`", "`127`" or "`-3 db`".

6.4.13 remote.get_item_value(item_index)

Returns the value of the remotable item mapped to the given control surface item. The value is automatically scaled according to the range specified in the control surface item definition. If the item's output type isn't value the functions return value is unspecified.

6.4.14 remote.get_time_ms()

Returns the current time in milliseconds. The return value is a positive number starting at zero when the control surface is initialized.

6.4.15 remote.handle_input(msg)

This function should be called to let the Remote host handle translated input messages.
`msg` is a Remote message; a table containing the following fields:
`item` – the index of the control surface item sending the message
`value` – the value of the message
`note` – the note number for a keyboard message (0-127)
`velocity` – the velocity for a keyboard message (0-127)
`time_stamp` – the time-stamp of the message (copy from the MIDI event)

The value of the message is different depending on the control surface item's input type:

Input type	Value
"button"	Sends 1 when pressed, 0 when released
"value"	Sends the absolute value of the control surface
"delta"	An encoder; sends positive or negative value changes.
"keyboard"	1 for note on, 0 note off. Sends note and velocity as extra data.

This function can only be called from `remote_process_midi()`. The time-stamp should be copied as it is from the MIDI event supplied as parameter to that function. Do not modify the time-stamp! It's a high-precision, non-portable time representation.

`remote.handle_input()` may be called many times for the same MIDI event. The MIDI event may also be buffered, then translated and handled later, in combination with another event.

6.4.16 remote.is_item_enabled(item_index)

Returns true if the given control surface item should be enabled.

6.4.17 remote.make_midi(mask,params)

This is a utility function that creates a MIDI event from a string mask and variables.
`mask` should be a string containing the MIDI mask. It may contain variable references (x,y and z).
`params` should be a table containing the variable fields used in the mask, i.e. x, y or z. The `params` table can also contain a `port` field, if the message should be addressed to a specific output port. The default address is port 1.
The return value is a MIDI event table.

A couple of examples:

```
-- A sysex, adressed to the default port
local event1=remote.make_midi("f0 00 5f bb f7")
-- Compose MIDI event using "mode" and "value". Adressed to port two.
local event2=remote.make_midi("b0 0<00xx> yy", { x=mode, y=value, port=2 })
```

6.4.18 remote.match_midi(mask,event)

This is a utility function used to compare MIDI events with a string mask. The function can also extract values from parts of the MIDI event.

`mask` should be string with the MIDI mask. It can contain wild cards and value references (x, y or z).
`event` – should be a MIDI event table.
This function does not match against the `port` field of the MIDI event. The caller can do that if necessary.
If the event does not match the MIDI mask the function returns `nil`. If there is a match the return value is a table containing the extracted x, y and z values. (If the MIDI mask contains no value references, these fields are set to 0).

Two examples:

```
-- Match sysex, the two last data bytes are ignored.
local res=remote.match_midi("f0 66 77 ?? ?? f7",event)
if res~=nil then
   return true
end

-- Match with wild cards and value extraction
ret=remote.match_midi("b?<?xxx>?yy",event)
if ret~=nil then
   local msg={ time_stamp=event.time_stamp, item=5, value=ret.y*8+ret.x }
   remote.handle_input(msg)
   return true
end
```

6.4.19 remote.trace (str)

This is a debugging utility which prints `str` on the Log Window in the Codec Test program. It has no function when the codec is run outside the codec test environment, so it should be removed or commented out when a codec is finished.

6.5 Bitlib

Bitlib is a library of bit manipulation routines by Reuben Thomas, provided under the following conditions (quote):

```
bitlib is a C library for Lua 5.0 that provides bitwise operations.
It is copyright Reuben Thomas 2000-2003, and is released under the
MIT license, like Lua (see http://www.lua.org/copyright.html for the
full license; it's basically the same as the BSD license). There is no
warranty.
```

6.5.1 bit.bnot(a)

Returns the one's complement of a.

6.5.2 bit.band(w1,...)

Returns the bitwise and of the w's.

6.5.3 bit.bor(w1,...)

Returns the bitwise or of the w's.

6.5.4 bit.bxor(w1,...)

Returns the bitwise exclusive or of the w's.

6.5.5 bit.lshift(a,b)

Returns a shifted left b places.

6.5.6 bit.rshift(a,b)

Returns a shifted logically right b places.

6.5.7 bit.arshift(a,b)

Returns a shifted arithmetically right b places.

6.5.8 bit.mod(a,b)

Returns the integer remainder of a divided by b.

7 Remote Map Reference

For a surface with a codec to communicate with Reason, it needs a Remote mapping file. This specifies which parameter in Reason should be controlled by which control on the surface, in any given situation or scope. For example, when the surface is directed to a Subtractor, you may want "Knob 1" to control the Filter Frequency parameter, but when it's directed to a Mixer, you may want the same knob to control a volume fader, etc.

The Remote SDK includes two map templates, one for a control surface with eight knobs and one for a control surface with eight knobs, faders and buttons. These templates are useful as a starting point; they show the most useful mapping for a typical control surface and also demonstrate how to use mapping variations. For inspiration you can also look at the map files installed by Reason in the Application Support folder.

7.1 Scope

There are three categories of scopes:

1. Scopes for individual device types (Subtractor, Mixer, etc.). These have the highest priority.
2. The "Reason Document" scope. This includes items such as transport functions, setting the MIDI input focus to another sequencer track, Undo, etc. This has the second priority.
3. The "Master Keyboard" scope. Includes keyboard, pitch bend and other items, for playing instrument devices. This has the lowest priority.

An example of "priority": If some keyboard keys are defined for transport functions in the "Reason Document" scope, they will never reach the "Master Keyboard" scope - these keys will be used for transport functions only and cannot be used to play instrument devices. If these keys had also been defined in an individual device scope (e.g. for switches on a Subtractor Synth), these keys wouldn't reach the transport functions or the master keyboard when the control surface was directed to a Subtractor.

7.2 Groups

The mapping file can also define "groups". This is a way of creating "variations". For example: your 8-fader surface controls the first 8 channel faders on the mixer by default (the first group), but you can switch to another variation (group 2) and control fader 9-14. This switching can be done from within Reason, by defining "Keyboard Shortcut Variation" groups in the mapping file (this is the most common way of handling variation). You can also define groups and have the surface itself switch between them. However, this is best suited for surfaces with two-way MIDI communication (because then the surface can indicate the selected variation with LEDs, displays, etc.). See the examples below.

7.3 Map File Format

This is how a Remote Mapping File looks like. The fields in the file are delimited by tabs, shown here as "->":

```
Propellerhead Remote Mapping File
File Format Version -> 1.0.0
```

Tells that this is a mapping file of a certain file format version.

```
Control Surface Manufacturer -> ACME
Control Surface Model -> In Control
```

Manufacturer and Model of the Control Surface. These must be the same as in the Codec File.

```
Map Version -> 1.0.0
```

Version nr of this mapping file, to keep track of changed maps.

```
Scope -> Propellerheads -> Master Keyboard
```

The Scope heading tells under which conditions the following mappings should be used. The mapping in the Master Keyboard scope is used when the control surface is designated "Master Keyboard" in Reason.

```
// -> Control Surface Item -> Key -> Remotable Item -> Scale -> Mode
```

These are the headings for the columns.
Control Surface Item is an item defined in the Codec.
Key is only used with the "Keyboard" Remotable Item - it's for defining an individual key for a special function.
Remotable Item is an item in the host that can be remote controlled. The document "Reason Remote Support.doc" included in the Remote SDK lists all available items in Reason.
Scale is only used with Delta-value controls - it allows you to scale the increment/decrement by a factor.
Mode is only used with control items that can show output data, e.g. LED meters or dials with LED rings. It determines how these devices should behave.

```
Map -> Keyboard -> ->  Keyboard
Map -> Pitch Bend Wheel -> ->  Pitch Bend
```

```
Map -> Modulation Wheel -> ->  Mod Wheel
Map -> Pedal ->  -> Damper Pedal
```

```
Scope -> Propellerheads -> Reason Document
```

The mapping in the Reason Document scope is used for non-device specific functions in Reason, such as the transport. For many control surfaces, this scope needn't be included in the mapping file - the user can manually map controls to transport functions, etc, if needed.

```
// -> Control Surface Item -> Key -> Remotable Item -> Scale -> Mode
Map -> Keyboard -> C7 -> Stop
Map -> Keyboard -> D7 -> Play
```

Examples of using the key column.

```
Scope -> Propellerheads -> Mixer 14:2
```

This is a scope for a specific device (the Mixer 14:2). It will be used when the control surface is directed to a Mixer device.

```
Define Group -> Keyboard Shortcut Variations -> Ch1-8 -> Ch9-14
```

This defines variations that can be selectable from within Reason. There can be up to ten variations, named as desired. In this case there are two, "Ch1-8" and "Ch9-14". If there are no variations you don't need to include the "Define Group" row.

```
// -> Control Surface Item -> Key -> Remotable Item -> Scale -> Mode -> Group
Map -> Data Entry Fader -> -> Master Level
```

Note the "Group" column, where you specify to which Group value (variation) a map belongs. This item is defined without a Group value - this means that the Data Entry Fader will control Master Level regardless of the selected keyboard shortcut variation.

```
Map -> Knob 1 -> -> Channel 1 Level -> -> -> Ch1-8
Map -> Knob 2 -> -> Channel 2 Level -> -> -> Ch1-8
Map -> Knob 3 -> -> Channel 3 Level -> -> -> Ch1-8
```

This maps are active when keyboard variation “Ch1-8” is active.

```
Map -> Knob 1 -> -> Channel 9 Level -> -> -> Ch9-14
Map -> Knob 2 -> -> Channel 10 Level -> -> -> Ch9-14
Map -> Knob 3 -> -> Channel 11 Level -> -> -> Ch9-14
```

This maps are active when keyboard variation “Ch9-14” is active.

```
Scope -> Propellerheads -> NN19 Digital Sampler
```

And so on, with scopes for all Reason devices that should be supported by the control surface.

7.3.1 Selecting variations from the surface itself

The example above shows how to use the "Keyboard Shortcut Variations" group. You could also have buttons on the surface that selects variations. This would be specified in the following way in the mapping file:

```
Define Group -> Channels -> Ch1-8 -> Ch9-14
```

Defines a group called "Channels", with two possible values: "Ch1-8" and "Ch9-14".

```
// -> Control Surface Item -> Key -> Remotable Item -> Scale -> Mode -> Group
Map -> Channel Left Button -> -> Channels=Ch1-8
```

```
Map -> Channel Right Button -> -> Channels=Ch9-14
```

On this control surface there are two buttons called "Channel Left Button" and "Channel Right Button". The "Remotable Items" here are "Channels=Ch1-8" and "Channels=Ch9-14", respectably. These are not really items in Reason, but functions in the mapping system itself (they become available because you have defined the group called "Channels" above).

As stated above, this is best used with surfaces with indicators or displays (with MIDI input), as this allows Reason to control the state of these indicators and show which variation is selected.

7.3.2 Mapping constant values to output items

On control surface with output item (e.g. LCD displays, meters) you can map constant values to these items if needed. Constant values can either be text strings or numbers. Some examples:

```
Map -> LCD Row 1 -> -> "Subtractor"
```

In this example the text string "Subtractor" is mapped to an LCD display. If the "Remotable Item" starts with a quote character, it's considered a text string.

```
Map -> Meter 4 -> -> 11
```

In this example the numerical value 11 is mapped to a meter. If the "Remotable Item" starts with a digit, it's considered a numerical constant.

7.3.3 If a control surface item isn't mapped in a variation

If there's no mapping for a control surface item in a variation, it becomes available to the "Reason Document" and "Master Keyboard" scopes (if it's mapped there). If you specifically want to lock it so that it's not available, map it to 0 like this:

```
Map -> Knob 3 -> -> 0
```

If the control surface item has output (e.g. a LED ring, a display or a motor fader) this will also send the value 0 to the item.

7.4 Naming and Location

A Mapping File for a surface must have the name specified under "Model" in the codec file, and the extension `".remotemap"`. It resides in the following location:
Mac OS X:
`Library:Application Support:Propellerhead Software:Remote:Maps`
Or
`Users::Library:Application Support:Propellerhead Software:Remote:Maps`
(for custom made map files, added by the user)

Windows:
`C:\Documents and Settings\All Users\Application Data\Propellerhead Software\Remote\Maps`
Or
`C:\Documents and Settings\\Application Data\Propellerhead Software\Remote\Maps`
(for custom made map files, added by the user)

Typically the "Maps" folder contains subfolders with the names of the surface Manufacturers, so that the mapping files are easy to find.

For example, the mapping file for our fictious "ACME In Control II" would be called `"InControlII.remotemap"` and reside in
`Library:Application Support:Propellerhead Software:Remote:Maps:ACME`
Or
`C:\Documents and Settings\All Users\Application Data\Propellerhead Software\Remote\Maps\ACME`

8 SDK Utilities

8.1 Codec Test

The Codec Test program has the following menu items:

8.1.1 Choose Codec File

The codec test program is intended for use when developing one Remote codec, so it does not scan the preference directories for installed codecs. Use this menu to select the codec you're working on. It opens a file selection dialog where you can select the ".luacodec" file.

8.1.2 Auto-detect

This menu tries to auto-detect all models for the loaded codec (opened with the Choose Codec File menu). It opens a test window for all models found.

8.1.3 Create Control Surface

Use this to manually open the control surface test window. It opens a dialog where you can select surface model, and connect to the proper MIDI ports.
When the surface test window opens, the Lua environment is created and the selected model is instantiated. It does the same things as when you check the "Use with Reason" check box in Reason's preferences.
Any errors encountered, in the Lua syntax or when the code is run, are reported in the Log Window.

8.1.4 Validate Map File

Use this item to validate your Remote map file. It opens a file dialog where you select the Remote map to validate. If any map errors are found they are displayed in the Log Window.
The codec for this control surface must be loaded by "Choose Codec File".

8.1.5 MIDI Test

Opens a simple MIDI test window. You can use it to send MIDI or listen to MIDI ports, to verify that the control surface sends and receives MIDI properly.

8.1.6 Show Log Window

Opens the log window.

8.2 Lua Compiler

This is a console program, `luac`, which compiles a lua source file into byte code. Compiling your Lua code is an option. The only benefit of doing it is to avoid exposing proprietary information to third parties and to disallow end users to modify the code. You may also choose to "open source" your codec and allow user to modify the codec.

8.3 Lua Interpreter

The Lua interpreter, `lua`, is a standalone Lua interpreter. It cannot be used to run your codecs, but it may be useful while learning Lua.

8.4 MIDI Tools

When developing a codec you should some kind of MIDI listener utility. This is useful to see exactly which MIDI messages are being sent, both from your codec and from the control surface. It helps when debugging and testing to find out if a bug is in your codec or in the control surface itself (or perhaps in the documentation of the hardware).

For the Mac there is the MIDI Monitor, which is free to download from:
http://www.snoize.com/MIDIMonitor/

For Windows we recommend MIDI-OX:
http://www.midiox.com/

9 Installation

9.1 Files

As mentioned in the introduction your codec files (.luacodec, source and picture files) should be installed in the following directory:
Mac OSX:

```
Library:Application Support:Propellerhead Software:Remote:Codecs:Lua Codecs:<your folder>:
```

Windows:

```
C:\Documents and Settings\All Users\Application Data\Propellerhead Software\Remote\Codecs\Lua Codecs\<your folder>\
```

And the Remote Map should be installed in:
Mac OSX:

```
Library:Application Support:Propellerhead Software:Remote:Maps:<your folder>:
```

Windows:

```
C:\Documents and Settings\All Users\Application Data\Propellerhead Software\Remote\Maps<your
folder>
```

9.2 Installers

The codec and map files are all portable but you need to create separate installers for Mac and Windows.

9.2.1 Mac

When creating Mac OS installs, we recommend that you use Apple's own PackageMaker, it is free and creates installers that Mac OS users are familiar with. You can find it on the Developer installation of OSX.

9.2.2 Windows

For Windows installations, we do not provide any special recommendations. Use any installer-creating application that you find suitable or are already familiar with.