Virtual Pronto Remote plugin

Document Version 2:
8 Feb 2021
By A-Lurker

License

IR protocol info

This plugin translates IR button codes, using some information provided by infra red protocol (IRP) definitions, to produce pronto codes. Only some of the IRPs are implemented and are based on the information in the IRP site here. There's about 130 protocols to look at:

http://www.hifi-remote.com/wiki/index.php/DecodeIR

The site also explains the IRP notation occasionally mentioned in the plugin code:

http://www.hifi-remote.com/wiki/index.php/IRP_Notation

IRP PROTOCOLS
IR protocols attempted are listed below – not all will work – in particular some of the Kaseikyo family. Only about half have been tested. NEC is the default. Search for IR codes here: http://www.remotecentral.com/cgi-bin/codes/ http://irdb.tk/find/

and here: https://github.com/probonopd/irdb/tree/master/codes		
IRP protocol	Manufacturers/comments	
DENON	Denon, Sharp - similar to NEC	
JVC	similar to Mitsubishi	
Kaseiko family	Denon, JVC, Panasonic & other Japanese manufacturers. These can be tried (possibly some will not work). IRP protocol: PANASONIC PANASONIC2 DENON-K JVC-48 JVC-56 KASEIKYO KASEIKYO56	
MITSUBISHI	similar to JVC	
NEC2	Canon, Denon, Hitachi, JVC, NEC, Onkyo, Pioneer, Samsung, Sharp, Toshiba, Yamaha, and many other Japanese manufacturers	
NECx2	As above	
RC5	Philips and other manufacturers in Europe	
RC6 family	RC6-0-16	16 bits
	RC6-0-20 used by Sky TV	20 bits
	RC6-0-32 Windows Media Center, Kodi	32 bits
	MCE same as RC6-0-32: Windows Media Center & Kodi	
RCA	RCA	
SHARP	Denon, Sharp - similar to NEC	
SONY12	Serial Infra-Red or wired Control System (SIRCS)	
SONY15	Serial Infra-Red or wired Control System (SIRCS)	
SONY20	Serial Infra-Red or wired Control System (SIRCS)	
Raw style formats:		
GC100	Global Caché IR code format	
PRONTO	yes: you can even send your old pronto codes	
RAW	string of mark/spaces in microseconds	

Test bed

Plugin tested using openLuup running on a RasPi 3 with AltUI using the BroadLink Mk2 plugi, working with a RM mini Broadlink device, as the IR transmitter and also the GC100

plugin:

https://github.com/a-lurker/Vera-Plugin-BroadLink-Mk2

Installation

Under AltUI:

While showing the 'Devices' page hit the '+Create' button. In the appropriate entry boxes, insert the following:

- A device name you decide upon eg 'Media IR remotes'
- D_VirtualProntoRemote!.xml
- I_VirtualProntoRemote1.xml

Save the changes.

Restart the Luup Engine
Restart AltUI

The new device is created in 'no room'.

Not working: keep trying till functional! Also refer to the DebugEnabled flag further below.

JSON file info

The plugin uses a file called “irRemoteCodes.json”. In that file you can configure your own “Virtual Remotes”. eg

```
{
  "Sony remote" :
  {
    "IRemitter" : {"Device" : "164", "ServiceIdx" : "2"},

    "Model" : "Sony TV 15 bit IR codes",

    "Encoding"   : {
        "Protocol"  : "Sony15",
        "Device"    : "84",
        "Subdevice" : "-1",
        "LSBfirst"  : true,
        "Repeats"   : "1"
    },

    "Functions"  : {
      "Power On" : {"Fnc" : "46", "Note" : "Discrete on"}

      LIST MORE BUTTON CODES HERE

    }
  }
}
```

Refer to the json file contents and the web page provided by the plugin. The file can be in plain or compressed form ie “irRemoteCodes.json.lzo” (as per in Vera) and the plugin will handle it as required. Use a json validator after making modifications:

https://jsonlint.com/

Values can be entered as decimal or hexadecimal eg 0xa8

Remote names

You can name each remote as you wish, as long as it's valid json. Non English alphas may cause problems. In the JSON above, the remote has been named: `"Sony remote"` When sending codes you just refer to this remote name in the action.

Function aka button names

You can name each Function (remote button) as you wish, as long as it's valid json. Non English alphas may cause problems. In the JSON above, the remote's Power On Function/button has been named: `"Power On"`. When sending codes you just refer to this Function (remote button) name in the action.

IR code endian order

The values in the file can be in LSB or MSB first order. A flag is used to indicate which way round the values are: `"LSB_first:"` true/false

Many IR codes are transmitted LSB first. When decoded, from say interpreting an oscilloscope waveform, they are typically seen and written out in MSB first order. Consequently many web sites list decoded codes “back to front”. This flag can be used to make use them without having to reverse them. It defaults to LSB first = true.

Repeats

A repeat value can be assigned to the codes. '0' means no repeats. Repeats are often needed. In particular by Sony codes may need to be repeated once or in some cases two times for them to work.

IR emitter

```
"IRemitter"   : {"Device" : "164", "ServiceIdx" : "2"},
```

In this section of the json file you assign the Device ID of the physical IR emitter you are going to use. ie say a Broadlink device or say a Global Caché GC100 or similar.

The ServiceIdx is the service the IR emitter uses, to send a pronto code. Set as follows:

'1' = 'urn:micasaverde-com:serviceId:IrTransmitter1'

'2' = 'urn:a-lurker-com:serviceId:IrTransmitter1'

JSON parser

If the json file is large, it is highly recommended that 'cjson.lua' is used in preference to the Vera default 'dkjson.lua' as it is **substantially faster**. If cjson is made available, the plugin will automatically make use of it.

On a RasPi this (in theory) this should install it:

luarocks install cjson

Which places a symbolic link in:

/usr/lib/arm-linux-gnueabihf/lua/5.2/cjson.so

to:

/usr/lib/arm-linux-gnueabihf/liblua5.2-cjson.so.0.0.0

Misc format notes:

GC100

Very similar to pronto but uses integer values rather than hex values.

RAW

Just mark space pairs in microseconds eg

9024, -4512, 564, -564, 564, -564, 564, -1692, 564, -564 etc

The plugin will also accept the values with no minus signs eg:

9024, 4512, 564, 564, 564, 564, 564, 1692, 564, 564 etc

This format is not allowed, as plus signs can't precede numbers in JSON. eg

+9024, -4512, +564, -564, +564, -564, +564, -1692, +564, -564 etc

DENON-K

DENON-K is a member of the Kaseikyo family - see further below for Kaseikyo details. It's not overly clear how the IR format is really laid out. But here we assume the following: Device is aka "Genre 1", Subdevice is aka "Genre 2".

DENON-K - Kaseikyo format, 48 bits of data. OEM code is 84 dec, 50 dec:

84 dec = 0x54 = 0101, 0100 reversed = 0010,1010 see m0-m7
50 dec = 0x32 = 0011,0010 reversed = 0100,1100 see n0-n7

Denon OEM code = 84 dec, 50 dec																Parity				Genre 1			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0	1	0	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	0	*	*	*	*
m0	m1	m2	m3	m4	m5	m6	m7	n0	n1	n2	n3	n4	n5	n6	n7	0	0	0	0	d0	d1	d2	d3

Genre 2				Data										ID		Parity							
25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
s0	s1	s2	s3	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	p0	p1	p2	p3	p4	p5	p6	p7

On an 8 bit byte basis: exclusive-OR the 3rd byte and the 4th byte and the 5th byte:

3rd byte (17 - 24 bit)
4th byte (25 - 32 bit)
5th byte (33 - 40 bit)

JVC

The JVC protocol omits its lead in burst when it repeats. This can confuse manual IR decoding methods.

Kaseikyo

Widely used by Japanese manufacturers. Each manufacturer have their own OEM code that is embedded in the Kaseikyo code. See DENON-K above for an example. The usage of the remaining bytes then appears to be a dogs breakfast. Consequently not all Kaseikyo codes will work but the framework is there to add manufacturers. These can be tried:

PANASONIC - tested OK
PANASONIC2
DENON-K
JVC-48
JVC-56
KASEIKYO
KASEIKYO56

Pioneer

Pioneer have a system where by they extend the number of codes available by sending two codes in a row in quick succession. A function extension code is sent first, then the actual function code.

Example - turning on “Speakers A”:
1) Send extension function code: Pioneer, 165, -1, 89
2) Then send actual function code: Pioneer, 165, -1, 0

RC5

RC5 has a toggle bit, which could introduce problems.

RC6 family

RC6 (RC6-0-16) is a 16 bit sequence

RC6-6-20 (Sky TV) is a 20 bit sequence

RC6-6-32 (MCE) is a 32 bit sequence

The most significant bit is sent first (NEC and many others do the opposite).

Of interest here is MCE, which is the coding used by the Windows Media Center, Kodi, game consoles, XBoxes. eg

https://github.com/probonopd/irdb/search?q=mce

RC6 has a toggle bit, which could introduce problems. It's handled OK for RC6-6-32 (MCE) but not the others in this family.

Sony12,15,20

Repeats must be used. Typically setting repeats to '1' will work. Some AVRs definitely require the code to be sent three times (repeats = 2). Volume control codes may need more?

Plugin general:

What the plugin will do:

1. Convert various protocols to pronto codes and transmit them.
2. You can configure your own virtual remotes.
3. The codes are stored in the json file as remotes.
4. Codes and there binary representations are listed on a web page.

What the plugin will not do:

1. convert all of the 130 or so IR protocols listed in the IRP web pages. The plugin is

not an IRP parser.

Potential problems:

1. In the IRP documentation and elsewhere, the words: "Button Code", "Function" and "Command" are pretty much synonymous. This can be a little confusing especially the word "Function" as it conflicts with the idea of "function" being a keyword in computer programs.
2. I propose not to engage in why some codes work or some don't work. The IR world is many and varied.
3. Make sure you can send a pronto code successfully with the BroadLink device (or whatever device you are using) first. Then try this plugin. The plugin logs the pronto codes it produces, so you should be able cross reference them against working pronto codes.
4. You may need to repeat a code to get any action. In particular Sony devices generally require one repeat (ie be sent twice) or more times.
5. The RC5 & RC6 toggle bits may not be handled as needed. MCE is handled OK and should work. If a function/button works once then stops, this may be the cause. You can send the function and then immediately send a dummy function (something that does nothing). The original function is reset and should work again.

IR code transmission

If the LED on your BroadLink device flashes when you transmit a code, the plugin is working. It’s a different matter whether you have the correct code or not. Some codes need replication and some codes do not. ie some TVs, etc require a code to be sent two or three time, in close succession, for them to work eg Sony.

All the variable values are strings. Identifiers are case sensitive except "Protocol".

Usage

Services:		
urn:a-lurker-com:serviceId:VirtualProntoRemote1		
All variable values are strings		
SendRemoteCode	Remote	The name you put in the json file eg 'Sony Remote'
	Function	Command you put in json file eg 'Power On'
SendIRPCode	Protocol	Standard IRP identifier eg 'Sony15'

	Device	IRP device number (string)
	Subdevice	IRP sub device number (string)
	Function	IRP function number (string)
	Repeats	Code repeats (string) '0' is no repeats
	IRdevice	The device that transmits the IR signal (string)
	IRserviceIdx	'1' = 'urn:micasaverde-com:serviceId:IrTransmitter1' '2' = 'urn:a-lurker-com:serviceId:IrTransmitter1'

Examples:

Send a button code:

```
-- this is the typical usage: remote info comes from the json file

-- enter the plugin id here
local deviceID = 162

-- as seen in your personalised json file: irRemoteCodes.json
-- eg protocol Sony12, device 1, subdevice -1, OBC 46
local remote  = 'Sony Remote'
local command = 'Power On'

-- this will turn on the Sony TV using the info in the json file
luup.call_action('urn:a-lurker-
com:serviceId:VirtualProntoRemote1', 'SendRemoteCode', {Remote =
remote, Function = command}, deviceID)

return true
```

Send a remote function / button code:

```
-- this allows you to try out an IRP code
-- codes from here:   http://irdb.tk/codes/

-- enter this plugin's id here
local deviceID = 210

-- and the IR transmitter plugin id - it's a string
local broadLinkDeviceID = '164'

-- eg protocol Sony12, device 1, subdevice -1, OBC 46
local protocol  = 'Sony12'
local device    = '1'
local subdevice = '-1'
local fnc       = '46'
```

```
-- this will turn on the Sony TV using its IRP code
luup.call_action('urn:a-lurker-
com:serviceId:VirtualProntoRemote1', 'SendIRPCode',
{
    Protocol     = protocol,
    Device       = device,
    Subdevice    = subdevice,
    Function     = fnc,
    Repeats      = '1',
    IRdevice     = broadLinkDeviceID,
    IRserviceIdx = '2'
}, deviceID)

return true
```

Searching for codes

Have a device that you don't know all the IR codes for? There is example code in GitHub that could be used to search for codes - see SearchForButtonCodes.lua. Need to be careful as you many send a manufacturer's “Hardware reset” code! Each code should be sent with a reasonable dwell time between transmissions, as the hardware many need time to respond to a code. Eg a power on code make time to come into action.

Debug

If DebugEnabled is set to '1' the debug info is enabled. Any debug commands will show up in the log file. Search on the string:

'VirtualProntoRemote debug:'

DebugEnabled should be set to '0' in normal operation.

Log file

You can use AltUI plugin to look at the log file:

See Misc–>Os Command–> Tail Logs Tab Note: you can change the 50 to say 500 to see more log info.

In openLuup use the console logs:

http://<openLuup_ip_address>:3480/console?page=log

Or use the infoviewer plugin (Vera only). Please provide a log file with any bug reports, otherwise the report may not be responded to.