Firmware walkthrough

I/R Protocol

The Humax YouView box uses what has become called the NEC protocol.

You can find references to it at for example https://www.sbprojects.net/knowledge/ir/nec.php

The Humax uses 'extended addressing' which is not referred to in all web documents. Initially, the two addressing bytes were complements of each other as a crude error-checking mechanism. With only 256 addresses, and many many vendors jumping on the bandwagon, it was decided to dispense with that check, and use all 16 bits as device address, allowing 65,536 different types of device.

For a useful graphical representation of the protocol see also https://techdocs.altium.com/display/FPGA/NEC+Infrared+Transmission+Protocol (but that reference, which is quite old now, does not mention extended addressing.

As you can see, the I/R light emitted by a real remote control is modulated at 38kHz. This modulation is stripped off by the equipment's I/R receiver.

Thus as we are not actually using I/R to control the device, we don't have to worry about the 38kHz modulation of the I/R light – we just send in baseband, pulling the pin low when we want the device to think the modulated infrared is active, and releasing it when we want it to believe the I/R is inactive.

The address bytes for the Humax YouView box are 0x0008

If your device also uses the NEC protocol, you should be able to use the firmware by adjusting the #defines around line 84 to 94.

If you device does NOT use the NEC protocol, you will need to redesign large parts of the firmware

Hardware operation and set-up

Pin RA2 is set up as a simple input, and is pulled low when power is present on the power sense pin.

Pin RA4 is initially set up as an input, but it is also configured as an 'Open Drain' output, and the output port bit is written permanently low. Therefore, to indicate "infra red present", we briefly re-configure the pin as an output.

Finally, we also configure 'Interrupt on change' (IOC) for both rising and falling pulses.

Timer 0 is set up as an 8-bit timer with the clock derived from the microcontroller's built in 32MHz clock, and with a 16* prescaler. The timer is set to create an interrupt every 210 clock pulses, which gives an output very close to 9,500 Hz – which is one quarter of the 38kHz carrier. We choose this frequency because the protocol bit timings are expressed in exact numbers of carrier cycles, and this makes it easier.

So to summarise, the controller is set up with both a regular timer tick interrupt, AND interrupts on every edge – rising or falling – of the incoming infrared stream.

Firmware structure

Almost all the work is done in the Interrupt Service routine (ISR) – for some programmers, this will look terrifying – after all, ISRs are meant to be minimal aren't they? In practice, the PIC has a simple instruction set and we have time to execute 32,000,000/9500 i.e. almost 3,400 instructions between timer interrupt ticks – and that's plenty (the whole program is only 854 bytes long, and that includes preset data!). You will also note that whilst there are a lot of lines of code, that's largely because there are long switch() statements which use a state variable to select the appropriate part of the coding. There are no wrapped loops or subroutine calls – it's all straight in-line.

Furthermore, we aren't expecting this tiny little brain to be doing anything else.

The 'task level' (i.e. non-interrupt) has two simple jobs:

- Set up the I/O lines and the timer (lines 174 to 208)
- Run round in a tight loop waiting for incoming I/R frames (lines 211-240)
    - If something relevant happens then transmit an appropriate I/R frame

Transmission of an I/R frame waits for anything happening at the receiver to stop, then disables the 'interrupt on change' detection, an cues a transmission by setting some variables, including the transmit state to start the Transmit state machine on its journey synthesizing the required output I/R frame

The ISR can be considered in two main parts

- Timer interrupt handling (lines 287-354)
    - Transmit state machine is driven by this, kicked off by setting TxState to tsxSTART
    - A couple of variables associated with the receiver are incremented on each timer interrupt
- IOC interrupt handling (lines 368-495)
    - Receive state machine is driven by this

I have attempted to write the code in as easy to read a manner as possible, and have avoided undue optimization.

As it stands, I have got some debugging handles in place which could be removed if you are getting short of space (presently using 42% of program memory).Cull any line which starts 'lastError = '

Things you may need to change

I have used the frames 000822DD (ie extended address 0008 – Humax, with command 0x22 which inverted is 0xDD) as my 'Power On' command, and 000832CD as 'Power Off' – see lines 84-92.

You will probably be familiar with the "Philips Pronto Hex Code" method of communicating I/R patterns. These two commands are described by the following paragraphs – which can be cut and pasted directly in to any universal Remote Control device or system which supports Philips Pronto Hex.

Power On command 0x22

```
0000 006D 0022 0002 0156 00AA 0016 0014 0016 0014 0016 0014 0016 0014 0016 0014
0016 0014 0016 0014 0016 0014 0016 0014 0016 0014 0016 0014 0016 0014 0016 003F
0016 0014 0016 0014 0016 0014 0016 0014 0016 0014 0016 003F 0016 0016 0016 0014
0016 0014 0016 003F 0016 0014 0016 003F 0016 003F 0016 0016 0016 003f 0016 003F
0016 003F 0016 0014 0016 003F 0016 0719 0156 0055 0016 0E38
```

Power Off command 0x32

```
0000 006D 0022 0002 0156 00AA 0016 0014 0016 0014 0016 0014 0016 0014 0016 0014
0016 0014 0016 0014 0016 0014 0016 0014 0016 0014 0016 0014 0016 0014 0016 003F
0016 0014 0016 0014 0016 0014 0016 0014 0016 0014 0016 003F 0016 003f 0016 0014
0016 0014 0016 003F 0016 0014 0016 003F 0016 003F 0016 0016 0016 0016 0016 003F
0016 003F 0016 0014 0016 003F 0016 0719 0156 0055 0016 0E38
```

You will almost certainly need to change these for different device types, and there is plenty of information out there on the web which describes the format. A good starting point is 'remotecentral.com/pronto' [I am aware that you can no longer buy Pronto, but its legacy will probably live on for a very long time!]

Rebuilding the firmware

You will need MPLAB X, and the appropriate compiler which is XC8, both of which can be downloaded from the Microchip site.

You will also need a PIC programming tool such as the PicKit3 (obsolete, but available in 'knock-off' form from online sources) or PicKit4 (quite a bit more expensive, but a current product with support).

It is beyond the scope of this document to set out the full detail of rebuilding the project, but the summary steps (for Windows) are:

- Install MPLAB-X and XC8, referring to Google to learn how to inform MPLAB about XC8
- From the main menu in MPLAB-X IDE choose File > New Project > Microchip Embedded > Standalone project
- Select the Device family as advanced mid-range 8-bit
- Start typing the part number in the 'Device' box, then select it and click 'Next'
- Choose your debug tool (or leave that till later)
- Choose your language (i.e. XC8) then Next
- Set your project location and hit save.

With that done, in the file tree at the left hand side, right-click the 'Source Files' branch, and 'Add existing item, then select youview.c from wherever you've saved it to your hard disk.

You now need to refer to 'getting started' guides on how to connect the programmer to your chip and program it