

LightwaveRF 433MHz Protocol Notes

Intro

This document gathers together notes on the RF protocol used to communicate by 433MHz to the devices like dimmers and switches from control units like the remote controls and Wifi link. It does not cover the UDP protocol into the wifi link as this is well documented elsewhere.

The notes come from material previously produced by GeekGrandad

<http://geekgrandad.wordpress.com/2013/01/03/lightwaverf-arduino-library/>, Benjie

<http://www.benjegillam.com/2013/02/lightwaverf-rf-protocol/>, and SoMakelt

https://wiki.somakeit.org.uk/wiki/LightwaveRF_RF_Protocol

I have also done my own experimentation and produced Arduino type libraries for TX and RX using 433MHz modules which can be found on Ebay very cheaply. The libraries can be found on github at <https://github.com/roberttidey/LightwaveRF>

The notes here are organised in a bottom up approach, rf modulation and bit encoding, message encoding, message transmission, commands and parameters. There is then a section describing the libraries.

RF modulation and bit encoding

Modulation

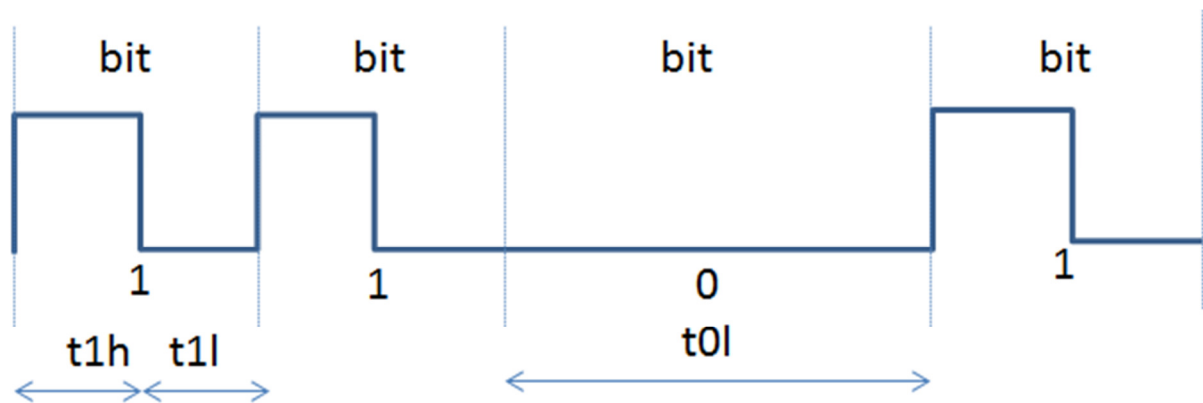
The transmissions take place at 433.29MHz and the basic modulation is simple AM modulation or OOK (on-off keying). A high turns the transmitter fully on and a low turns the transmitter fully off.

There are two common types of RX module. The more common is a simple super-regenerative receiver where an oscillator at 433MHz is regularly 'quenched' and allowed to build up again. A received signal changes the rate at which oscillations build up and effectively gives a high gain. A second type is a more classical superhet receiver using a built to purpose receiver ic. This is potentially more stable but also tend to be made available with controlled bandwidth which can affect the received timings a little bit.

Bit encoding

There are several ways of interpreting the bit encoding mechanism. The way I choose here seems to be the easiest to understand. A binary 1 is represented by a burst of RF followed by a short burst of silence. A binary 0 is represented by an extended burst of silence. To ensure that the data can be recovered reliably there must be a decent number of transitions in the data and the method used here is to do message encoding to ensure there is never more than a single 0 in a sequence.

The timing is illustrated here.



The timings I have measured are a good match with previous results.

Using the super-regen receiver to capture the timings and averaging over about 1000 transitions I get

$t1h = 293\mu\text{Sec}$, $t1l = 280\mu\text{Sec}$, and $t0l = 990\mu\text{Sec}$

The superhet receiver gives slightly different results of

$t1h = 328\mu\text{Sec}$, $t1l = 251\mu\text{Sec}$, and $t0l = 982\mu\text{Sec}$

The difference is probably due to a smaller bandwidth on the superhet making it slower to transition from on to off and therefore stretching the high periods a bit.

Transmitting bit encoding

This is fairly straightforward as a 1 bit can be encoded as a on period followed by an off period, and a 0 bit can be encoded as a longer off period.

Receiving bit decoding

This is little bit more complicated as there is no obvious transition between a 1 bit and a 0 bit.

However, there is never more than one 0 bit in the sequence so the decoding can be done just by looking at the duration of the low periods. A short low ($t1l$) represents a 1 bit, an extended low ($t1l + t0l$) represents a 10 sequence.

Message encoding

The message encoding is nicely described by the SoMakelt reference. A message packet consists of 40 bits of data which is organised into 10 nibbles of data. Each nibble is represented by a start bit (1), followed by 8 tx bits. These tx bits are chosen to obey the rule that there is never more than one 0 bit in the sequence. The byte encoding of these nibbles is also chosen so there are always two 0's in each byte. This keeps the transmission time of each nibble the same.

Nibble	Byte	Hex
0x0	11110110	F6
0x1	11101110	EE
0x2	11101101	ED
0x3	11101011	EB
0x4	11011110	DE
0x5	11011101	DD
0x6	11011011	DB
0x7	10111110	BE
0x8	10111101	BD
0x9	10111011	BB
0xA	10110111	B7
0xB	01111110	7E
0xC	01111101	7D
0xD	01111011	7B
0xE	01110111	77
0xF	01101111	6F

This translation table can be used to both format data for transmission and to decode the real message content on reception.

The overall message has an additional start bit (1) and an additional end bit (1).

So, in summary, the 40 bits or 10 nibbles are transmitted as

MsgStart 1

10 times

Nibble start bit 1

4 bit nibble -> 8 bit encoding

MsgEnd 1

This gives a total of 92 bits of low level 1's and 0's encoded into actual transmission bits.

{Comment}

Although the chosen scheme here is simplistic, it is not as good as it could be. A more efficient self clocking encoding scheme could have either slowed the timing down making it more robust or have encoded more tx bits in the same type of packet. The latter could have given a more expandable message and could also have included error correction capability.

Message transmission

Each message packet of 92 bits consists of 72 1's and 20 0's (Each of the 10 translated nibbles has 2 0's). This means it takes $72 * (t_{1h} + t_{1l}) + 20 * t_{0l}$ time to transmit. This is about 61msec.

The transmitting device repeats this a number of times with a gap of about 10mSec in between each message. The repetition is intended to increase the chance of a message being received correctly

even if several get interfered with or lost. Devices also seem to need at least 2 repetitions to obey the message. This is to decrease the chance that a received corrupted message is acted upon.

The number of repetitions varies with the transmitting device. The simple remotes I have tested repeat the message 6 times taking about 400msec overall. The wifi link repeats the message 25 times taking about 1.7 seconds. I guess the reason for the difference is that a remote is likely to be used in fairly close proximity to the device whereas the wifi link has to work over a larger area.

Commands and parameters

As has been said the basic message consists of 10 4 bit nibbles of data. These are structured as

- 2 Nibbles – 1 byte parameter value (0-255)
- 1 Nibble - Command (0-15)
- 1 Nibble device (0-15). 15 reserved for mood control.
- 5 Nibbles of Transmitter ID
- 1 Nibble of Transmitter Sub ID (0-15)

The device, Transmitter ID and Sub ID are what are used to pair a controller with a device. This is an important aspect of the one way nature of the communication. The Controller / transmitters have no intrinsic way of directly addressing devices. Similarly a device does not have the concept of a device number or address. Instead during a pairing operation the device remembers the device number, Transmitter ID, and sub ID. It will then respond act on messages containing this information and ignore all others. Most devices seem to have the capability of remembering up to 6 ‘addresses’ and therefore responding to messages coming from 6 different sources. Note that a wifi link or equivalent master controller is regarded as a single ID combination in this respect. All UDP messages going into the Wifi link for a particular device will appear on the RF protocol to be coming from the same (wifi link) source irrespective of the phone, computer, etc sending commands to the wifi link.

The second interesting part of the ID combination is the Sub ID. This is used by the Wifi Link to effectively form a Room ID. The end devices themselves have no concept of rooms but commands into a wifi link are organised by rooms each with a number of devices. The device number becomes the device nibble (0-15) and the room number becomes the Sub ID (0-15). So, theoretically you can arrange devices in up to 16 rooms each with 16 devices for a total of 256 end devices. Other parts in the chain may limit this further, e.g. the smartphone app.

It is possible that other parts of the Transmitter ID could be used for further expansion in this direction but I have always seen the same 5 nibbles coming out of a particular wifi link.

That then leaves the Command in the message and the 8 bit parameter.

This seems to be the greyest area at the moment and although there is enough for basic control I am continuing to do some work on this and will edit this section.

Command	Nibble	Parameter	meaning
Off	0	0-127	Switch a device off param usually 64
Off	0	128-159	Set a device level 0-31 (param-128)
Off	0	160-191	Decrease Brightness
Off	0	192-255	All off param usually 192
On	1	0-31	Switch a device On to last level
On	1	32-63	Set a device level 0-31 (param-32)
On	1	64-95	Set a device level 0-31 (param-64) Same effect as 0-31
On	1	96-127	Set a device level 0-31 (param-96) Same effect as 0-31
On	1	128-159	Set a device level 0-31 (param-128) Same effect as 0-31
On	1	160-191	Increase brightness
On	1	192-223	Set all to level 0-31 (param-192)
On	1	224-255	Set all to level 0-31 (param-224)
Mood	2	130-	Start mood n (param-129). (130=Mood1) Device = 15
Mood	2	2-	Define mood n (param-1). (2=Mood1) Device = 15

The commands have been labelled as Off and On as that is what the controllers tend to send. This certainly applies to switches but for dimmers the distinction is less and in practice a lot of the behaviour is determined by the parameter. For example one can set the level of a dimmer using the same parameter setting using either an off or on command.

Libraries and test programs

I have written two libraries for Arduino type processors (e.g. ATmega328) to control the RX and TX modules used for 433MHz transmitting and receiving. They are split into the two parts as often one only wants Tx or Rx. Of course if an application has both reception and transmission then both libraries can be included. The code for these is on the Github mentioned together with some test sketch programs.

TX library (LwTx)

The Tx library is based around a tick based interrupt service routine running a state machine going through the various phases of message transmission (start of message, start of nibble (byte), nibble byte, bits, message end, inter message gap and repeats. The interrupt is only active once a message has been posted to be sent and once the message has been sent the interrupt is turned off until the next message.

This technique means that the foreground application of the Arduino can post a message and then get on with other tasks and no low level timing has to be done in the foreground.

When the interrupt is active then it operates with a standard tick of 140uSec as this gives a good match to the bit timing requirements. Everything therefore happens at this rate or a multiple of it. So for example a TX high consists of two ticks (280uSec).

The routine has been written to be tunable. The basic tick rate can be adjusted and also the number of ticks associated with particular states. These tuning parameters can be set from the application.

The interface to the library consists of

- void lwtx_setup(int pin, byte repeats, byte invert, int uSec); Should be called at least once to initialise the library. Set the TX pin to be used to control the transmitter, the number of

message repeats to use, whether a high or low controls the transmitter on, and the basic tick rate (140)

- `void lwtx_setTickCounts(byte lowCount, byte highCount, byte trailCount, byte gapCount);` Tunes the number of ticks to be used for the various timing periods. Defaults are set internally so only needs to be called if tuning needed.
- `void lwtx_setGapMultiplier(byte gapMultiplier);` Normally gap is restricted to $255 * 140\mu\text{sec}$ (35.7mSec). This call allows long gaps to be set for experimentation. The gap delay set by tick counts is multiplied by this amount allowing up to 9.1secs
- `void lwtx_settranslate(boolean txtranslate);` Sets whether nibble to byte translation of message data is used. This defaults to 1 (on) and the application should then send in message data. If set to 0 (off) then translation is skipped and the app should feed in raw byte values. Only needed if playing with the encoding scheme.
- `boolean lwtx_free();` Used to detect if a message transmission is in progress. Should be checked for true before sending any message. If false then the app should postpone sending the message until it becomes free.
- `void lwtx_send(byte* msg);` Message sending interface. Normally a message consists of a 10 byte buffer.
- `void lwtx_setaddr(byte* addr);` Set address as 5 digit address to allow sending short commands
- `void lwtx_cmd(byte command, byte parameter, byte room, byte device);` Short commands, `lwtx_setaddress` should be called first.

This library can also save the basic device address in EEPROM. This is enabled by configuring the `EEPROMaddr` in the library to the start of where in the EEPROM to store the 5 byte device address.

Example programs

Two example sketches are on the GitHub.

The `LwTXTest` is a simple sketch that simply sends a fixed message repeatedly. You can change the message by editing the sketch.

The `LwTxTestEx` is a larger sketch which allows for controlling the transmissions sent using commands received on the serial port. Type 9 to get help. Commands allow setting the basic 5 digit address, for sending command messages (cmd+parameter+room+device), for setting the basic send characteristics (repeats, clock period), and for adjusting pulse widths and gaps. If less than the number of values are used with a command then the previous values apply. So, for example, sending command messages can consist of just command + parameter and they will then address the last device used.

RX library (LwTx)

The Rx library is based around a pin change interrupt service routine running a state machine going through the various phases of message reception (start of message, start of nibble (byte), nibble byte, bits, and message end).

This technique means that the foreground application of the Arduino can get on with other tasks and no low level timing has to be done in the foreground. Message reception happens asynchronously and the foreground just has to occasionally check for and handle new messages.

The pin change interrupt basically measures the interval from the last change and splits this interval into groups of delay for the different types of interval of interest. This is combined with the level of the pin to form a state machine 'event'. The event is then handled by the main message state machine.

The routine is not directly tunable but the delay interval categories can be easily changed in the code if required.

Filtering of messages can also take place in the library. A repeat count allows for only reporting messages that occur at least this number of times in quick succession and they will only be reported once. A timeout also controls the repeat window.

Pairing is also supported. The library can accept pairing info in the form of room, device and 5 digit base address. It can also be put into an auto-pairing mode and will then add the address data from the next received message into its pairing information if the command is on (cmd=1). If the command is an off (cmd=0) then it will try to remove the pair. It will then only report messages that match one of its paired address. Multiple pairs up to a maximum of 10 are supported. There are two additional controls associated with pairing. PairEnforce controls what happens if no pairs are present. If false then any message received is reported. If true then messages are not reported until a pair is defined. PairBaseOnly controls how a pair is defined. If false then a pair is the transmit address + room + device associated with the message. This would be normal if using the library to make a single end point device. If true then only the 5 nibble transmit address is used and messages will be reported for any room and device. This can be useful if making intermediate devices like a repeater.

The routine can also gather stats on the timing of the pulses giving average and min, max values for the durations of interest.

The interface to the library consists of

- `void lwrx_setup(int pin);` Must be called at least once to set the pin used for reception which must be pin 2 or 3 as these are supported by change detect interrupts.
- `void lwrx_settranslate(boolean translate);` Sets whether nibble to byte translation is used. Defaults to translate on.
- `boolean lwrx_message();` Returns whether a message has been received or not. Should be called periodically to determine whether to handle a message.
- `boolean lwrx_getmessage(byte* buf, byte len);` Returns the message into a buffer of at least len bytes. Len controls what information is reported. It can take values of 2, 4 or 10. If 10 then the full message packet is returned. 4 returns command, parameter, room and device. 2 returns just command and parameter. 2 and 4 are intended to be used primarily when pairing is used. `translate` determines whether the returned message is raw data or translated nibbles but this only applies to 10 byte returns.
- `void lwrx_setfilter(byte repeats, byte timeout);` This sets the number of repeats for a message to be regarded as valid. It can also set the timeout for these repeats. Use a repeat count of 0 to get all packets reported.
- `byte lwrx_addpair(byte* pairdata);` This adds pairing data. This is a 8 byte buffer containing device, dummy, 5 base address, room. It returns the number of active pairs defined.
- `byte lwrx_getpair(byte* pairdata, pairnumber);` This puts the pairing data for one pair into the buffer in the same format as `addpair`. It also returns the current paircount. If the `pairnumber` exceeds the paircount then the buffer is not updated and just the paircount is returned. So this may be called with a high `pairnumber` to get just the paircount.
- `void lwrx_setPairMode(boolean pairEnforce, boolean pairBaseOnly);` This sets the two pairing mode controls. `pairEnforce` true means that messages only get reported once pairing is established. `pairEnforce` false will report all messages if no pairing is defined. `pairbaseOnly` defines what constitutes a pair. If true then only the basic Tx address is used, If false then it uses the Tx address plus the device and room number associated with the pair.
- `void lwrx_makepair(byte timeout);` This puts the receiver into pairing mode for timeout in 100ms units. The first valid message received has the pair data added to the pairing pool. After the timeout period it reverts to normal mode.
- `void lwrx_clearpairing(byte* pairdata);` This clears out any previously added pair data.
- `void lwrx_packetinterval();` This returns the number of milliseconds since the last packet was received. It may be used to determine when the reception is quiet as repeat packets may be being received after a message is reported due to the repeat filtering.

- `boolean lwrx_getstats(unsigned int* stats);` Returns an array of 9 values of stats consisting of average, max and min values of high times, short low times and long low times. The average values are multiplied by 16. Min and max are directly uSecs.
- `void lwrx_setstatsenable(boolean rx_stats_enable);` Controls whether stats are collected. Disabling also resets the stats counters.

This library can also save the pairing data in EEPROM. This is enabled by configuring the `EEPROMAddr` in the library to the start of where in the EEPROM to store the paircount (1 byte) followed by 8 byte entries for each pair. The pair information is held in translated raw tx byte form to make comparisons quicker.

Example programs

One example sketch is on the [GitHub](#).

The `LwRXTest` is a sketch that sends any message it receives down the serial line. It shows the elapsed time and the message nibbles.

It can respond to a number of commands received on the serial line. 9 will give a little help menu. Commands allow for controlling the message reporting format, adding and clearing pair data and controlling repeat filtering. Commands also allow displaying the stats and resetting and disabling the gathering of stats.

Compile time macros allow for the log messages responding to serial commands to be controlled. Turning these off by commenting out the `#define` feedback means the sketch is smaller and the output will be primarily message traffic.