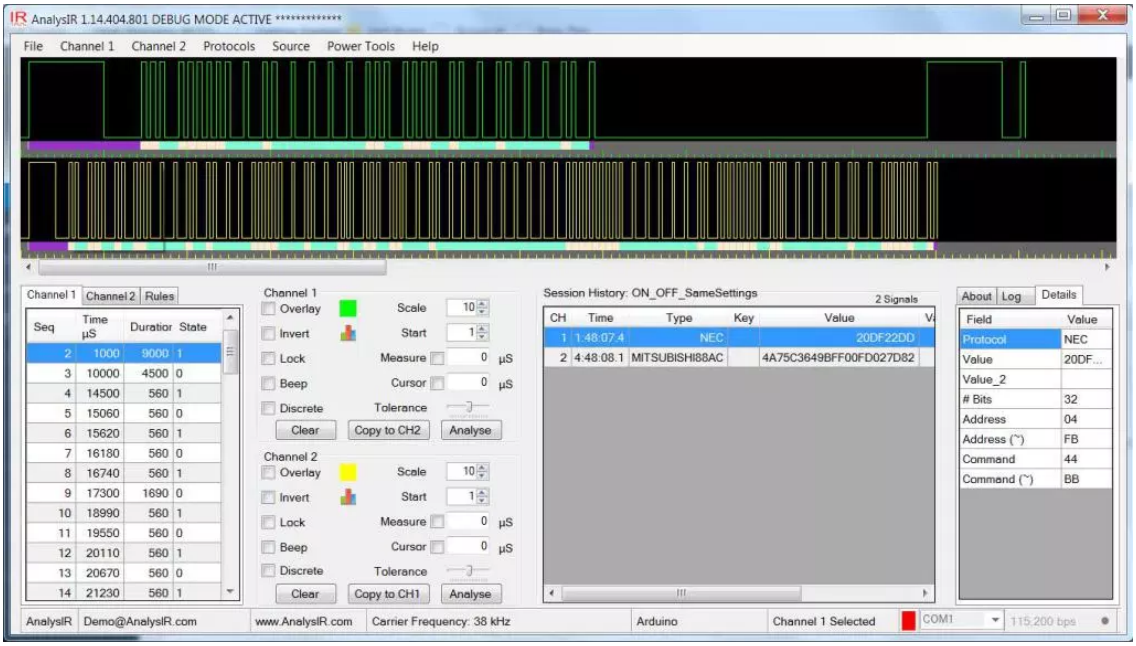
AnalysIR Blog

All about infrared remote control, IR decoding and more

Simple Infrared PWM on Arduino, Part 3 – Hex IR Signals

September 1, 2015 AnalysIR Blog Air Conditioner, AnalysIR, backer, Carrier frequency, Infrared, IR, NEC, Remote control

In Part 1 of this series, we demonstrated how to send signals using soft or Simple Infrared PWM on Arduino. In our Part 2 post we looked at sending RAW IR signals – specifically a RAW NEC signal and a longer RAW Mitsubishi Air Conditioner signal using soft PWM. We have since improved the PWM method shown in **Part 1 Part 2** to provide better performance and improve portability. In this Part 3, we will take the signals from Part 2 and show how to send them using their binary (or Hex) representation, which can save lots of SRAM in many projects, particularly when dealing with longer AC signals.



Original NEC 32-bit and Mitsubishi 88-bit Signals captured using AnalysIR

The image above shows the 2 signals we will be using, from Part 2. The first is an NEC 32 bit signal from an LG TV, and includes one NEC repeat header. The second is a signal from a Mitsubishi Air Conditioner with 88 bits of data. In this post, we will provide the source code showing how to send these signals using their Hex representation and for comparison we also include the less efficient methods for sending them as RAW encoded signals. Please note that the easiest way to get the Hex representation of a signal is by using a tool like **AnalysIR** *(or libraries like IRremote IRLib which may work for a limited set of signals up to 32 bits, which unfortunately excludes most AC signals)*

Rationale

The reason for publishing this series includes:

- See the rationale provided in Part 2, plus …
- Sending from the Hex representation, allows users to save valuable MCU resources such as SRAM.

- Using soft PWM as presented in this post also saves MCU resources such as Timers and allows users to use any pin for output, instead of the limited set of pins tied to a particular timer on the MCU.
- Much of the support requests on the Arduino forum relate to conflicts between libraries and the approach presented here will remove many of these conflicts. However, we would always recommend using hardware generated PWM for sending IR signals, where possible and a soft PWM approach for other situations.
- As before, we will post a link to the complete Arduino sketch at the bottom of this post.

This post may be a little on the long side for some, so feel free to jump to the end of the post to play with the source code immediately.

Simple Infrared PWM on Arduino

Definitions Setup

Both definitions and Setup remain largely the same as in Part 2. However, we have now added the 2 Hex representations along with the 2 RAW signals in buffers as below:

```
unsigned char carrierFreq = 0; //default
unsigned char period = 0; //calculated once for each signal sent in in
unsigned char periodHigh = 0; //calculated once for each signal sent i
unsigned char periodLow = 0; //calculated once for each signal sent in



unsigned long sigTime = 0; //used in mark & space functions to keep tr
unsigned long sigStart = 0; //used to calculate correct length of exis


//RAW NEC signal -32 bit with 1 repeat - make sure buffer starts with
unsigned int NEC_RAW[] = {9000, 4500, 560, 560, 560, 560, 560, 1690, 5

#define NEC_HEX_VALUE 0x20DF22DDL
#define NEC_BIT_COUNT 32

//RAW Mitsubishi 88 bit signal - make sure buffer starts with a Mark
unsigned int Mitsubishi88AC_RAW[] = {3172, 1586, 394, 394, 394, 1182,

unsigned char Mitsubishi88AC_Hex[] = {0x4A, 0x75, 0xC3, 0x64, 0x9B, 0x

void setup() {
 Serial.begin(57600);
 pinMode(txPinIR, OUTPUT);
}
```

Both of the RAW signals above were recorded with AnalysIR and automatically exported in the C language format above. You will also notice the sigTime sigStart variables defined. SigTime is used, in the mark space functions described below, to keep a track of elapsed time throughout the signal and greatly improves the fidelity of the transmitted signal, by effectively eliminating most of the delays in code loops and function calls. SigStart is for a very specific purpose when sending an NEC signal to keep track of the gap or space between the main signal and the first repeat signal.

You should also notice that the Hex representation of the 32 bit NEC signal only takes 4 bytes of SRAM plus one byte to record the number of bits used in the signal. This compares favourably to the 134 bytes of scarce SRAM used in storing the RAW version of the same signal or a mere 3.4% (5 vs 134 bytes). Similarly, the Hex representation of the 88 bit Mitsubishi AC signal only takes 11 bytes of SRAM . This compares favourably to the 358 bytes of scarce SRAM used in storing the RAW version of the same signal or a mere 3.1% (11 vs 358 bytes). It should now be obvious that using the Hex representation is a more sensible and efficient way to proceed, in most situations.

In the setup function, we initialise the Serial for sending notifications of when a signal is sent and set the txPinIR to OUTPUT. You can set any pin to output the IR signals by amending the associated pin definition.

Loop

For demonstration purposes, we continually send 6 Infrared signals with a 5 second gap in-between each one using each of the available carrier frequencies and demonstrating the functions to send the example NEC Mistsubishi88AC signals using Raw and Hex methods. The first one is an NEC signal from an LG TV and the second a Mitsubishi AC signal. You will also notice that we have added in dedicated functions for sending Raw signals, NEC Hex signals and Mitsubishi AC Hex signals. This provides a basis for extending to any other IR signal format required.

```
void loop() {
 //First send the NEC RAW signal defined above
 Serial.println(F("Sending NEC_RAW @ 56kHz"));
 sendRawBuf(NEC_RAW, sizeof(NEC_RAW) / sizeof(NEC_RAW[0]), 56);
 delay(5000); //wait 5 seconds between each signal (change to suit)

 //Next send the Mitsubishi AC RAW signal defined above
 Serial.println(F("Sending Mitsubishi88AC_RAW @ 40kHz"));
 sendRawBuf(Mitsubishi88AC_RAW, sizeof(Mitsubishi88AC_RAW) / sizeof(Mi
 delay(5000); //wait 5 seconds between each signal (change to suit)

 //Next send the NEC_HEX_VALUE signal defined above
 Serial.println(F("Sending NEC_HEX_VALUE @ 38kHz"));
 sendHexNEC(NEC_HEX_VALUE, NEC_BIT_COUNT, 1, 38);
 delay(5000); //wait 5 seconds between each signal (change to suit)

 //Next send the Mitsubishi88AC_Hex signal defined above
 Serial.println(F("Sending Mitsubishi88AC_Hex @ 36kHz"));
 sendHexMITSUBISHI88AC(Mitsubishi88AC_Hex, sizeof(Mitsubishi88AC_Hex)
 delay(5000); //wait 5 seconds between each signal (change to suit)

 //Next send the NEC_HEX_VALUE signal defined above
 Serial.println(F("Sending NEC_HEX_VALUE @ 33kHz"));
 sendHexNEC(NEC_HEX_VALUE, NEC_BIT_COUNT, 1, 33);
 delay(5000); //wait 5 seconds between each signal (change to suit)

 //Next send the Mitsubishi88AC_Hex signal defined above
 Serial.println(F("Sending Mitsubishi88AC_Hex @ 30kHz"));
 sendHexMITSUBISHI88AC(Mitsubishi88AC_Hex, sizeof(Mitsubishi88AC_Hex)
 delay(5000); //wait 5 seconds between each signal (change to suit)
}
```

As we now send the signals using a dedicated function for each protocol or method, it is also possible to pass the required carrier frequency as a parameter, which can be any of 30, 33, 36, 38, 40 or 56 kHz. The individual functions will be described below.

SendRawBuf

As can be seen from the Loop function we call sendRawBuf with 3 parameters:

- The buffer containing the timings for the signal. This can be automatically generated by AnalysIR or alternatively via IRremote or IRLib, with some limitations.
- The size of the buffer, which can be automatically calculated by the compiler if you use the approach shown.
- The carrier frequency at which to generate the IR signal.

```
void sendRawBuf(unsigned int *sigArray, unsigned int sizeArray, unsigne
 if (carrierFreq != kHz) initSoftPWM(kHz); //we only need to re-initial
 sigTime = micros(); //keeps rolling track of signal time to avoid impa
 for (int i = 0; i < sizeArray; i++) {
  mark(sigArray[i++]); //also move pointer to next position
  if (i < sizeArray) { //check we have a space remaining before sending
```

```
  }
 }
 }
```

The flow is quite simple. Initialise the settings for soft PWM, if different from the frequency requested. Initialise sigTime to keep track of elapsed time throughout the signal, thus eliminating code execution delays. Finally, iterate through the buffer, always starting with a mark, sending marks spaces in pairs. There is a special case for not sending a trailing space unless specifically requested. Normally, sending a trailing space is meaningless as it just represents the idle time between signals.

SendHexNEC

As can be seen from the Loop function we call sendHexNEC with 3 parameters:

- The Hex representation of the signal (32-bit), which can be obtained from AnalysIR or other libraries.
- The number of bits from this 32 bit value to send (Most significant bit first).
- The number of repeats to send afer the initial main signal.
- The carrier frequency at which to generate the IR signal. NEC signals are defined @ 38kHz, but there is nothing to stop vendors using alternative frequencies.
- This function is designed to send a maximum of 32 bits.

```
void sendHexNEC(unsigned long sigCode, byte numBits, unsigned char rep
/* // A basic 32 bit NEC signal is made up of:
 1 x 9000 uSec Header Mark, followed by
  1 x 4500 uSec Header Space, followed by
 32 x bits uSec ( 1- bit 560 uSec Mark followed by 1690 uSec space; 0
 1 x 560 uSec Trailer Mark
 There can also be a generic repeat signal, which is usually not necce
 */
#define NEC_HEADER_MARK 9000
#define NEC_HEADER_SPACE 4500
#define NEC_ONE_MARK 560
#define NEC_ZERO_MARK 560
#define NEC_ONE_SPACE 1690
#define NEC_ZERO_SPACE 560
#define NEC_TRAILER_MARK 560

 unsigned long bitMask = (unsigned long) 1 << (numBits - 1); //allows
  }
  // Last send NEC Trailer MArk
 mark(NEC_TRAILER_MARK);

 //now send the requested number of NEC repeat signals. Repeats can be
 /* A repeat signal consists of
 * A space which ends 108ms after the start of the last signal in this
 1 x 9000 uSec Repeat Header Mark, followed by
 1 x 2250 uSec Repeat Header Space, followed by
 32 x bits uSec ( 1- bit 560 uSec Mark followed by 1690 uSec space; 0
 1 x 560 uSec repeat Trailer Mark
 */
 //First calcualte length of space for first repeat
 //by getting length of signal to date and subtracting from 108ms

 if (repeats == 0) return; //finished - no repeats
 else if (repeats > 0) { //first repeat must start 108ms after first s
 space(108000 - (sigTime - sigStart)); //first repeat Header should st
 mark(NEC_HEADER_MARK);
 space(NEC_HEADER_SPACE / 2); //half the length for repeats
 mark(NEC_TRAILER_MARK);
 }

 while (--repeats > 0) { //now send any remaining repeats
 space(108000 - NEC_HEADER_MARK - NEC_HEADER_SPACE / 2 - NEC_TRAILER_M
 mark(NEC_HEADER_MARK);
 space(NEC_HEADER_SPACE / 2); //half the length for repeats
 mark(NEC_TRAILER_MARK);
```

```
  }

}
```

The relevant timings for the NEC protocol are show in the compiler define statements. The flow is quite simple. Initialise the bitmask and settings for soft PWM, if different from the frequency requested. Initialise sigTime to keep track of elapsed time throughout the signal, thus eliminating code execution delays. Initialise sigStart, which is used to calculate the gap between the main signal body and the first repeat, which should be 108 mSecs after the signal start. Subsequent repeats are easier to calculate because the timings are fixed. Then we send the following:

- The Header mark space
- The mark and space pairs representing binary '1' and binary '0' bits. For NEC this is usually a total of 32 bits.
- The trailing mark to signify the end of the main signal body.
- If there are no repeats required, we are finished and return.
- If there are repeats requested, the first one gets special treatment to ensure that timings are aligned. The length of the space immediately after the trailer is 108 mSecs minus the length of the main signal body. This ensures that the first repeat is exactly 108 mSecs after the signal start. If there is only one repeat requested, we return at this point.
- Note that the length of the repeat 'space' is half the length of the main header 'space'.
- Subsequent repeats are easier to calculate as the timings are fixed at every 108 mSecs. These repeats are useful for VOL+ or VOL- type signal when being powered by battery and result in much longer battery life.

SendHexMITSUBISHI88AC

This function is similar to sendHexNEC, but quite a bit simpler, as follows:

- There are no repeats in this protocol
- It has 88 bits instead of 32 bits
- The timings are different

Otherwise, the make-up is similar and can be used as a template for generating most AC signals using soft PWM. In this case the parameters passed are:

- A buffer containing the Hex representation of the signal (88-bit), which can be obtained from AnalysIR.
- The size of this buffer (Again note how we let the compiler calculate the size). In this case 11 bytes long.
- The carrier frequency at which to generate the IR signal. Most AC signals are 38 kHz, but some can vary with 33 kHz being the next most common for AC signals.
- Because of the 88 bit code it is not possible to pass the value as a standard variable, which is why the buffer is used.

```
void sendHexMITSUBISHI88AC(unsigned char *sigArray, unsigned int sizeA
 /* A basic 88 bit NEC-'like' signal is made up of:
 * 1 x 3172 uSec Header Mark, followed by
 * 1 x 1586 uSec Header Space, followed by
 * 32 x bits uSec ( 1- bit 394 uSec Mark followed by 1182 uSec space;
 * 1 x 9000 uSec Trailer Mark
 * There can also be a generic repeat signal, which is usually not nec
 */
#define MITSUBISHI88AC_HEADER_MARK 3172
#define MITSUBISHI88AC_HEADER_SPACE 1586
#define MITSUBISHI88AC_ONE_MARK 394
#define MITSUBISHI88AC_ZERO_MARK 394
#define MITSUBISHI88AC_ONE_SPACE 1182
#define MITSUBISHI88AC_ZERO_SPACE 394
```

```
#define MITSUBISHI88AC_TRAILER_MARK 394



 //
 if (carrierFreq != kHz) initSoftPWM(kHz); //we only need to re-initia
 sigTime = micros(); //keeps rolling track of signal time to avoid imp

 // First send header Mark & Space
 mark(MITSUBISHI88AC_HEADER_MARK);
 space(MITSUBISHI88AC_HEADER_SPACE);

 for (unsigned int i = 0; i < sizeArray; i++) { //iterate thru each by
 }

 }
 // Last send NEC Trailer MArk
 mark(MITSUBISHI88AC_TRAILER_MARK);
}
```

The flow is similar to sendHexNEC, except that the timings are different, there are no repeats and we iterate through the buffer one byte at a time to get the individual bits to send. This method should provide a good template for implementing any AC signal using minimum MCU resources.

InitSoftPWM

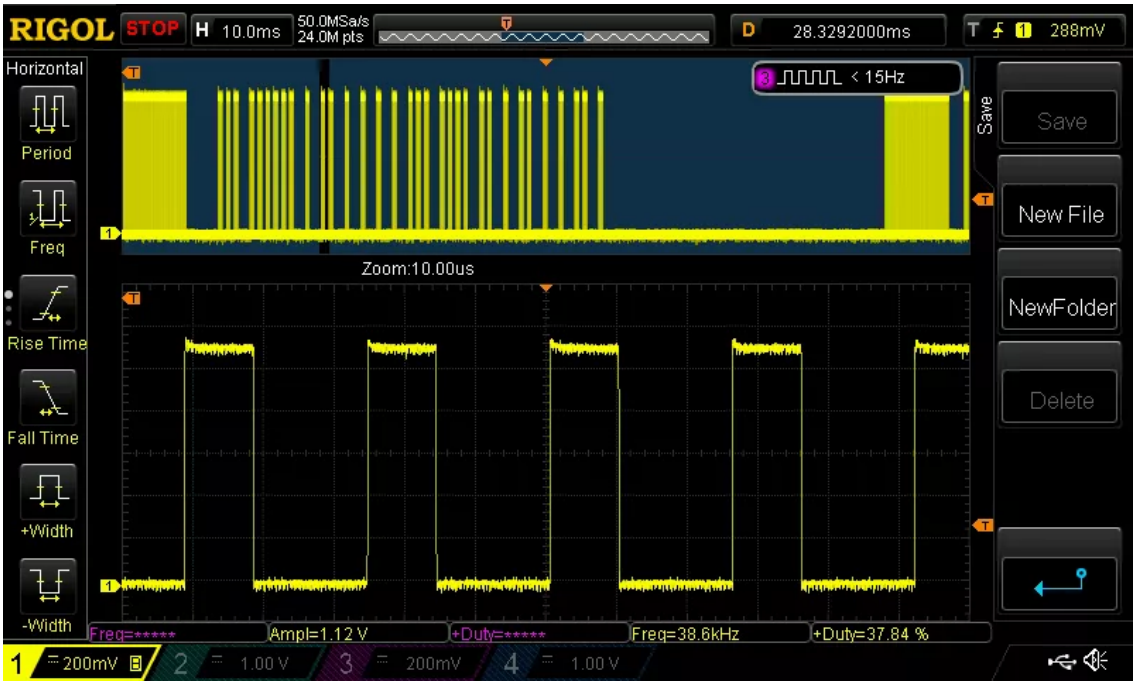
As the name suggests this function initialises the variable used to generate the soft PWM at the range of frequencies supported. (30, 33, 36, 38, 40 or 56kHz). The values have been measured using an oscilloscope and a 16 MHz Arduino and the results are quite impressive indeed. This function generates very workable frequencies and a duty cycle as near as possible to 33%. All frequencies are within 4% of target which is more than adequate for most IR receivers. The actual target frequencies and duty cycles measured at source are:

- **30kHz** – Carrier 29.8 kHz, Duty Cycle 34.52%
- **33kHz** – Carrier 32.7 kHz, Duty Cycle 34.64%
- **36kHz** – Carrier 36.2 kHz, Duty Cycle 35.14%
- **38kHz** – Carrier 37.6 kHz, Duty Cycle 36.47%
- **40kHz** – Carrier 40.6 kHz, Duty Cycle 34.96%
- **56kHz** – Carrier 53.8 kHz, Duty Cycle 40.86%

```
void initSoftPWM(unsigned char carrierFreq) { // Assumes standard 8-bi
 //supported values are 30, 33, 36, 38, 40, 56 kHz, any other value de
 //we will aim for a duty cycle of circa 33%

 period = (1000 + carrierFreq / 2) / carrierFreq;
 periodHigh = (period + 1) / 3;
 periodLow = period - periodHigh;
 // Serial.println (period);
 // Serial.println (periodHigh);
 // Serial.println (periodLow);
 Serial.println (carrierFreq);

 switch (carrierFreq) {
 case 30 : //delivers a carrier frequency of 29.8kHz & duty cycle of 3
 periodHigh -= 6; //Trim it based on measurementt from Oscilloscope
 periodLow -= 10; //Trim it based on measurementt from Oscilloscope
 break;

 case 33 : //delivers a carrier frequency of 32.7kHz & duty cycle of 3
 periodHigh -= 6; //Trim it based on measurementt from Oscilloscope
 periodLow -= 10; //Trim it based on measurementt from Oscilloscope
 break;

 case 36 : //delivers a carrier frequency of 36.2kHz & duty cycle of 3
 periodHigh -= 6; //Trim it based on measurementt from Oscilloscope
```

```
26    periodLow -= 11; //Trim it based on measurementt from Oscilloscope
27    break;
28
29    case 40 : //delivers a carrier frequency of 40.6kHz & duty cycle of 3
30    periodHigh -= 6; //Trim it based on measurementt from Oscilloscope
31    periodLow -= 11; //Trim it based on measurementt from Oscilloscope
32    break;
33
34    case 56 : //delivers a carrier frequency of 53.8kHz & duty cycle of 4
35    periodHigh -= 6; //Trim it based on measurementt from Oscilloscope
36    periodLow -= 12; //Trim it based on measurementt from Oscilloscope
37    Serial.println(periodHigh);
38    Serial.println(periodLow);
39
40    break;
41
42
43    case 38 : //delivers a carrier frequency of 37.6kHz & duty cycle of 3
44    default :
45    periodHigh -= 6; //Trim it based on measurementt from Oscilloscope
46    periodLow -= 11; //Trim it based on measurementt from Oscilloscope
47    break;
48    }
49  }
```

The flow is again quite simple. We first calculate the period for the selected carrier frequency to the nearest uSec. Next we calculate the nominal High/Low duration for a duty cycle of 33%. Then we use a select statement to set trim values for each frequency as verified with the oscilloscope. The trim values compensate for the delays in code execution and the 1 uSec granularity of the period value. Note these settings work very well with Arduinos running @ 16Mhz and sketches compiled using the Arduino IDE. They may not work as well with other platforms, compilers or different compiler optimisation settings.



NEC 32 bit IR signal generated using soft PWM on the A.IR shield, with zoom to PWM

Mark

```
1  void mark(unsigned int mLen) { //uses sigTime as end parameter
2   sigTime += mLen; //mark ends at new sigTime
3   unsigned long now = micros();
4   unsigned long dur = sigTime - now; //allows for rolling time adjustme
```

```
  if (dur == 0) return;
  while ((micros() - now) < dur) { //just wait here until time is up
  digitalWrite(txPinIR, HIGH);
  if (periodHigh) delayMicroseconds(periodHigh);
  digitalWrite(txPinIR, LOW);
  if (periodLow) delayMicroseconds(periodLow);
  }
 }
```

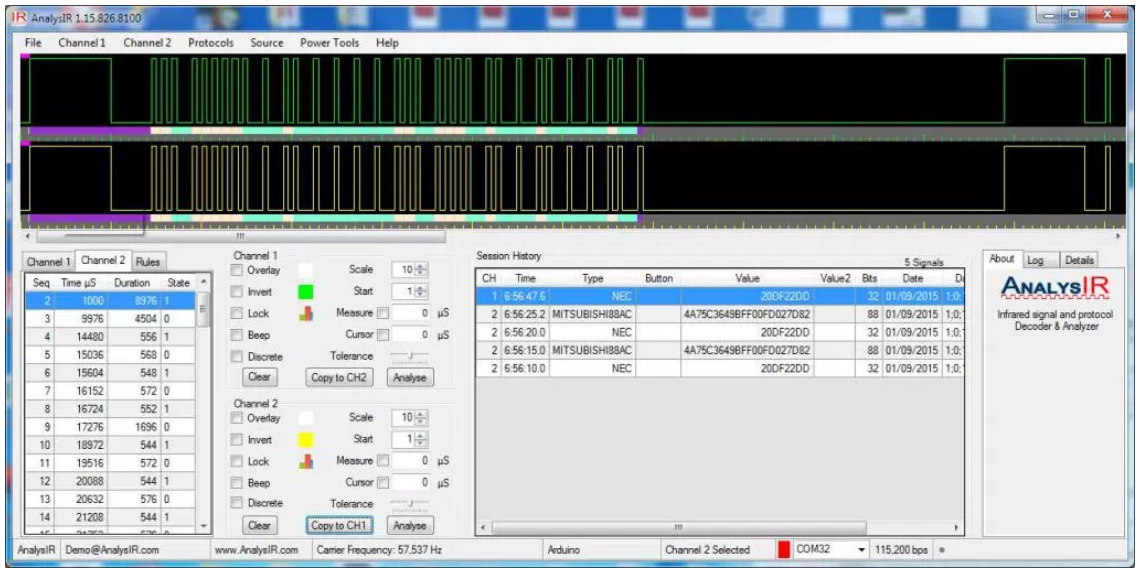
This method for sending a mark is very similar to that introduced in Parts 1 2. Refer to **Part 2** for more details.

Space

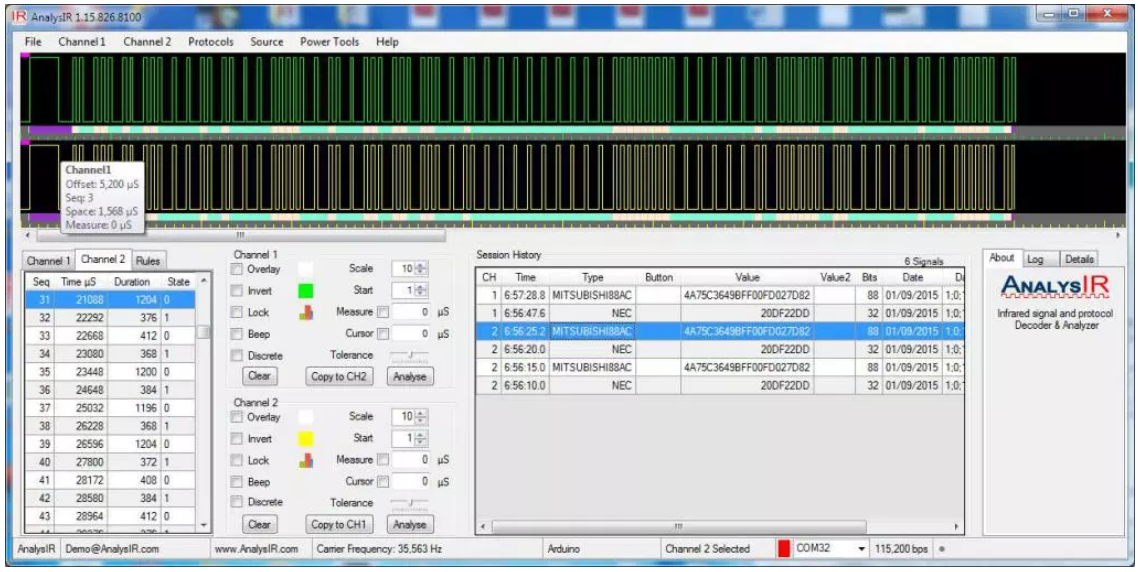
```
void space(unsigned int sLen) { //uses sigTime as end parameter
 sigTime += sLen; //space ends at new sigTime
 unsigned long now = micros();
 unsigned long dur = sigTime - now; //allows for rolling time adjustmen
 if (dur == 0) return;
 while ((micros() - now) < dur) ; //just wait here until time is up
}
```

This method for sending a space is very similar to that introduced in Parts 1 2. Refer to **Part 2** for more details.

Example Images



NEC captured in AnalysIR using soft PWM, RAW 32 bit signal above HEX signal below



Mitsubishi88AC signal captured in AnalysIR using soft PWM, RAW signal above HEX signal below

Conclusion – Part 3 of 3

In this part 3 we have taken real world signals and played them back using the code above and recorded them signals using AnalysIR. The images above are screenshots of the actual signals recorded using AnalysIR, with the Raw hex generated signal being almost identical. The main lesson from Part 3 is that using the Hex representation to send an Infrared signal is by far the most efficient method in most situations and using soft PWM does not mean lower quality signals when compared to hardware PWM.

What about other MCU platforms? This same method will work on any platform, with only a few minor adjustments and most importantly accurate calibration to generate the trim values for each platform.

The complete source code for the sketch can be downloaded here:
Simple_infrared_PWM_Send_RAW.ino

Update for Photon

We have also included a sketch version for the Photon here:
Photon Version of this sketch

Please feel free to use the code in this blog post without restriction. If you gain any benefit please link back to this article credit the Author where possible.

Part 1 of this 3 part series can be **found here.**
Part 2 of this 3 part series can be **found here.**

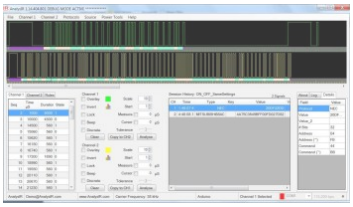
Get your own copy of AnalysIR here.

Share this:

Like this:

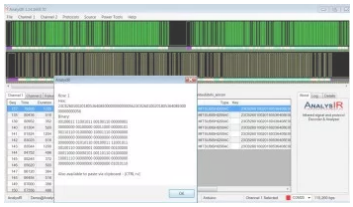
Loading...

Related



Simple Infrared PWM on Arduino, Part 2- RAW IR Signals
June 10, 2015
In "AnalysIR Blog"

Simple Infrared PWM on Arduino
May 12, 2015
In "AnalysIR Blog"



Reverse engineering the Mitsubishi AC Infrared protocol
January 6, 2015
In "AnalysIR Blog"

You might also like....

A.IR Shield Nano Python Script - Raspberry Pi (RPi)
Preview: A.IR Shield ESP8266/ESP32 Tx, a high-end IR Shield
Using AnalysIR with Flirc - Video tutorial
Preview: A.IR Shield ESP8266 TRx, a high-end IR Shield
Updated ESP8266 NodeMCU Backdoor uPWM Hack for IR signals
ESP8266 NodeMCU Backdoor uPWM Hack for IR signals

One thought on "Simple Infrared PWM on Arduino, Part 3 – Hex IR Signals"

Pingback: Backdoor uPWM Hack on Photon for Infrared signals

You must log in to post a comment.