TREND MICRO | research

Hacking LED Wristbands as a Learning Opportunity to Jump on RF Security

Technical Brief

By Jonathan Andersson and Federico Maggi

Radio frequency (RF) technology is used in various settings and, considering its importance and impact, is a perennial subject of security research. In fact, early this year, we published a research on RF remote controllers used for heavy-duty equipment. We believe that it's of paramount importance that security researchers develop skills in this field, especially since a growing number of devices associated with the internet of things (IoT) and industrial internet of things (IIoT) are and will be based on RF communication. This is the main motivation behind this educational report: providing a simple yet complete research target to get started with.

RF is a common technology that can also be found in many nonindustrial applications. These include applications in the entertainment industry, where even if human lives and sensitive data are far from being at risk, security issues may be exploited to damage the image of an individual or organization directly involved in or otherwise attached to a production. For example, the reputation of the organizer of a certain event and the manufacturer or vendor of devices used in it may be called into doubt if the devices are compromised and abused to cause disruption or other undesirable outcomes. And in this scenario, the devices in question can be something as simple and accessible as LED wristbands, which are the research target covered by this report.

LED wristbands are usually given out to guests in clubs or attendees at concerts, and they're also used as part of the lighting systems of large shows and events, including broadcast ones where hundreds of thousands of people may be watching or attending. These wristbands can be remotely controlled manually — by a DJ, for example, who can set the LED lights to change colors, flash, or stay on in time with the song being played — or be integrated in a fully automated lighting control system. All in all, a LED wristband is a simple and harmless device used to set the ambiance and enhance the experience of an event. Despite its simplicity, though, it contains most of the basic components that are present in an IoT device: a microcontroller unit (MCU), a radio transceiver, some inputs, and some outputs (the LEDs, in this case).

In May, we came across such wristbands at the Hack In The Box Amsterdam post-conference event, where we presented our research on industrial RF remote controllers. And being the security enthusiasts that we are, we proceeded to conduct a spontaneous research on-site. The following is a blow-by-blow account of our experience.

Initial reconnaissance

We need to record some legitimate transmission in order to have a repeatable testing dataset. To that end, we need to locate the transmitter. We spot an antenna on the DJ stage and immediately see a box, which looks like the transmitter. According to its label, it's working at 869 MHz. We also notice on the label two hexadecimal (hex) numbers, 0x95 and 0x04, whose meaning will be clear later in this report. There is no brand on the wristbands, but we quickly find the vendor website by doing a web search for "Drome wristband," given the hint on the transmitter box.

Even on real targets, digging up information about a vendor is very important. Not only can it reveal relevant technical details, but it can also help in understanding the impact of security issues, if any: Is the vendor widespread? Are there other vendors that sell the same, rebranded products? Who are the customers of such vendors? And so on. These are questions that should be part of the initial reconnaissance.

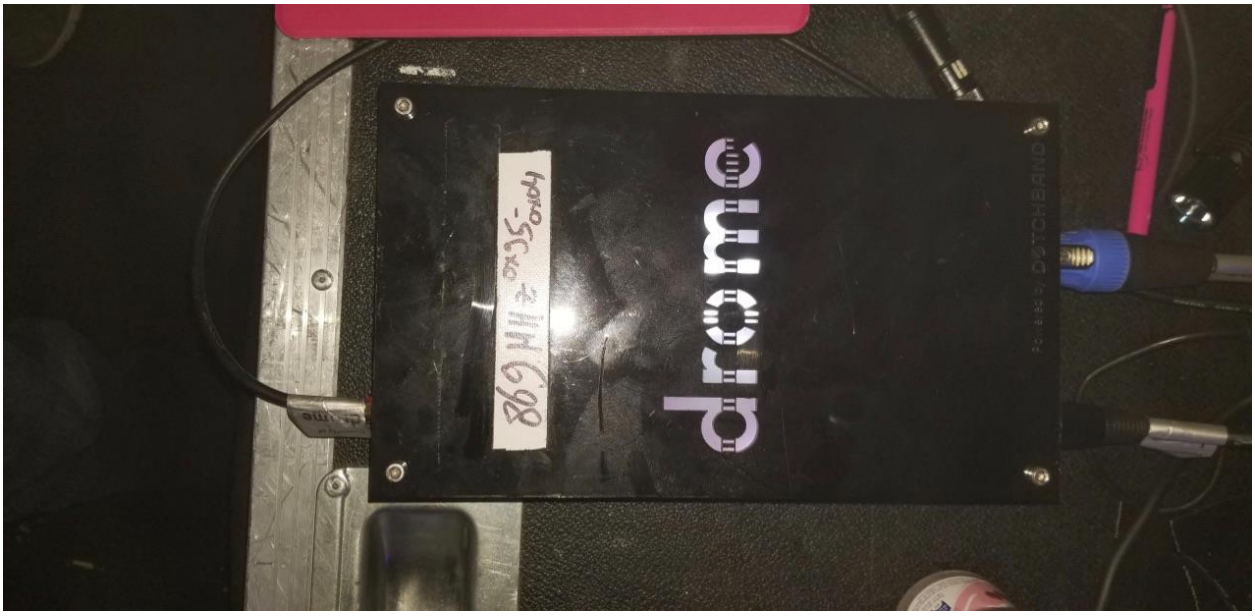


Figure 1. Transmitter for the LED wristbands

Figure 2. The control panel connected to the Drome transmitter module, which transmits signals to the LED wristbands

Chip identification

While inspecting the vendor's website, we find out about DMX512 (Digital MultipleX 512), an industry protocol used in such applications as pyrotechnic movements of theatrical effects and, more commonly, stage lighting and effects. In large areas, over-the-wire DMX512 is simply inapplicable, for both practical and technical reasons: Not only are long cables expensive and difficult to deploy, but they also cause transmission errors. Thus, there are secondary data links that are used to transmit DMX512 packets, and wireless data links are clearly of interest. For example, the giant observation wheel known as the Singapore Flyer uses wireless DMX to control the lighting. In our humble case, the wristband itself is a simple DMX512-over-RF receiver, connected to a DMX512-to-RF transmitter (which in our case is on the DJ stage).

Figure 3. The LED wristband (top) and a detail of the RF receiver (bottom)

The wristbands are fairly disposable: They're quite cheap, and since we're not required to return them to the organizer, we can even throw them away after the event. So it makes sense that there's little more than a simple printed circuit board (PCB) inside the main component of one of these wristbands, embedded in which is a 4 mm. by 4 mm. radio chip. As seen in Figure 3, it's a CC113L receiver, which is the receiver-only version of the well-known CC1101 transceiver. (We are no strangers to the CC11x family of semiconductors, as we encountered CC1120 in our industrial radio research.)

SDR Captures

Having confirmed the use of RF technology, we think it's viable to analyze further. By this time, the party is almost over, but a few nerds — including us — are still on the dance floor, getting more and more bored. So we decide to spice up the party a little bit by bringing in our software-defined radio (SDR) equipment: a BladeRF SDR. The goal is to make as many RF captures in various conditions (on, off, constant on, specific flash patterns, etc.) as possible. This is essential since diverse samples will allow us to compare different packets and reveal the overall packet structure. Especially when looking at real RF systems, taking as many captures as possible under all pertinent conditions is an important step. It's reasonable to assume that a packet's content determines the light's colors and flash speed or its displayed "effects." In real systems, there may be buttons to press or similar inputs that influence the data exchanged over the air.

We enlist the help of the DJ, who is justifiably intrigued by our equipment, in putting on various commands or conditions on the wristband, so that we can record the resulting packets in a more controlled way, rather than just at random. Having software tools such as Universal Radio Hacker (URH)

makes the difference in these situations. It allows us to easily take quick captures, name them, and repeat the procedure for different conditions.

At this point, we've already known the frequency but not much else about the signal. We haven't been able to determine if the wristband uses frequency-shift keying (FSK), on-off keying (OOK), or another modulation scheme, but at this point it's not yet crucial to find out. Before we can actually start capturing, we first need to guess the bandwidth and confirm the "exact" carrier frequency, of which we have a hint from the label. We use gr-fosphor to inspect the spectrum and the waterfall; the high resolution of gr-fosphor makes it a great reconnaissance tool. After finding the carrier frequency and bandwidth, we tune the BladeRF slightly toward the left to avoid any DC noise — a technique called offset tuning — and start to capture.

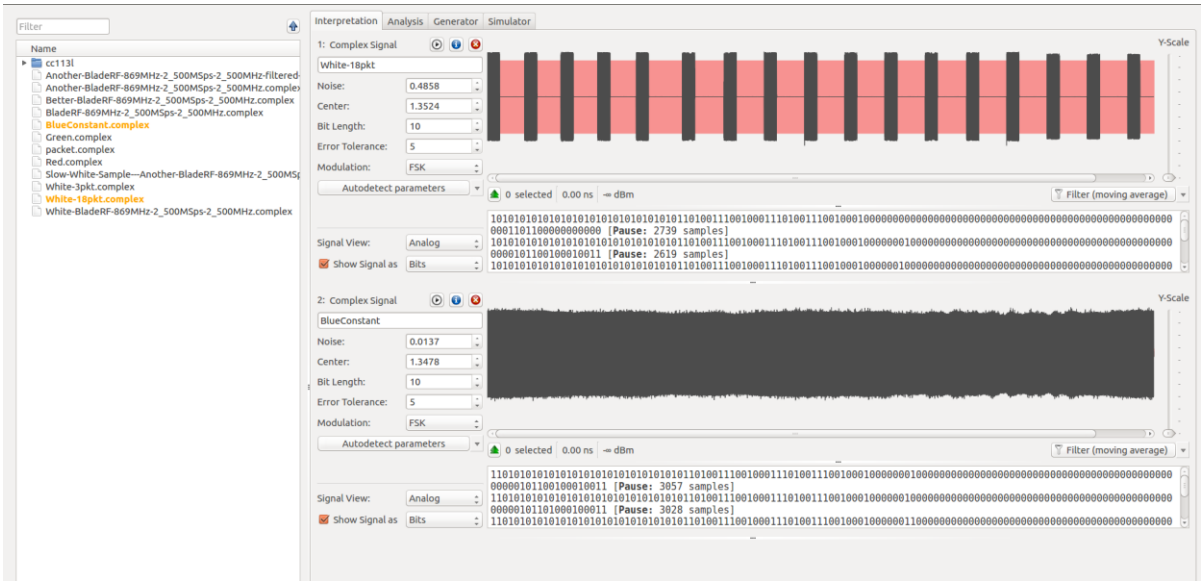


Figure 4. Our capture of the RF signals using SDR

Testing for replay attack vulnerability

With the captured signals, it's possible to test for replay attack vulnerability. Moving away from the more powerful transmitter, we replay the different signals corresponding to specific effects to see if nearby wristbands will respond. Sure enough, some wristbands begin to flash different colors, making their wearers stand out from the crowd and allowing us to relish the interesting reactions of the non-security-oriented folks.

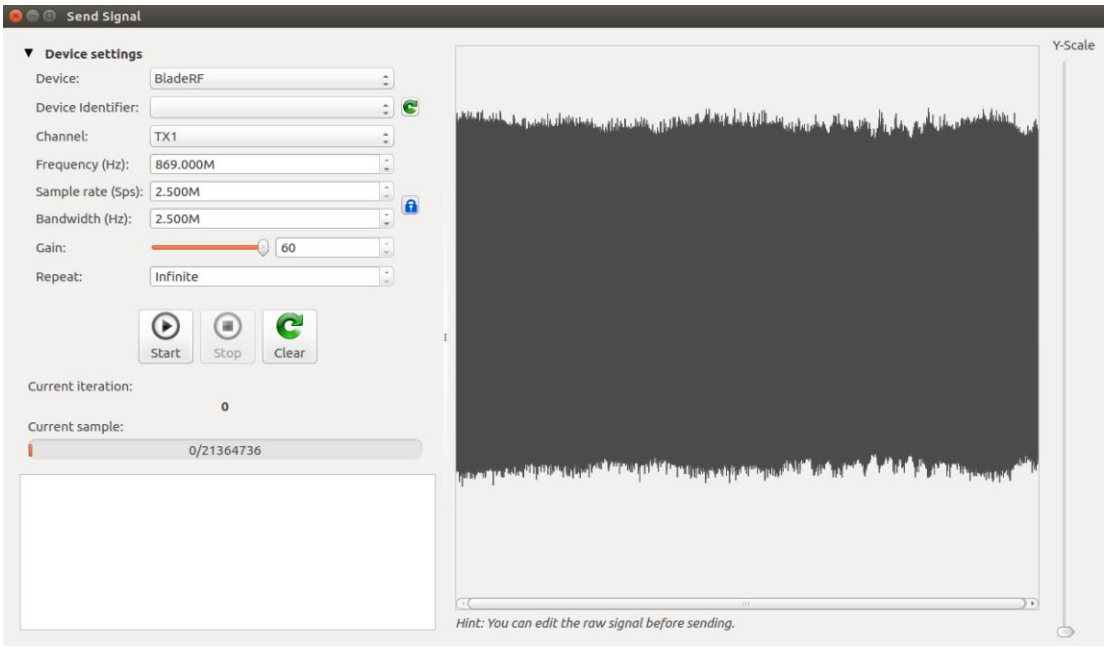


Figure 5. Replaying a signal

RF communication analysis

By this time, we have to leave the party because the club is about to close. Having enough captures and a working receiver with us, we're able to continue our research back in our labs. Using the captured signals, we can then begin a more in-depth analysis. To do so, we need to clean or filter the signals to isolate only the signal that we need. This section largely involves the different features of URH that help filter the desired packets.

Filtering or noise canceling

The signals we captured were not perfect, as we had no control over the RF-noisy environment. We had to purposely use offset tuning to avoid DC spikes. Thus, we need to re-center the signal.

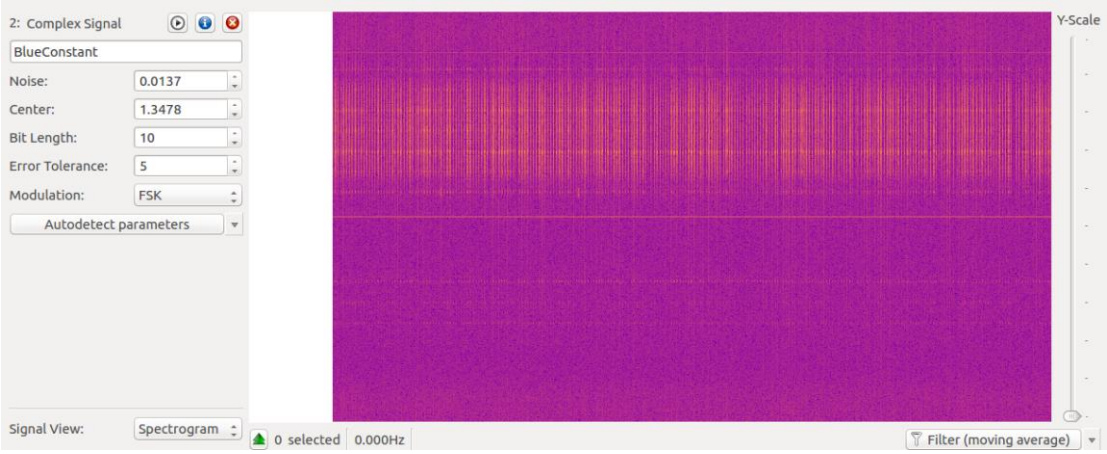


Figure 6. Spectrogram with the DC noise line showing in the middle.

In Figure 6, the solid orange line in the middle of the spectrogram is DC noise, and the signal is just above that line. The pattern looks like that of FSK, but at this point we cannot determine whether it's 2-FSK or 4-FSK. Because we can see more than two peaks, we are leaning toward 4-FSK, although this will still need further confirmation. This is another essential part of embedded-communication analysis: Always seek for multiple confirmations, even if they may sound redundant.

At this point, we can either apply a bandpass filter or use noise cancellation. These two options are fundamentally "the same" under the hood. Bandpass filtering is a feature of URH as well as many other RF analysis software tools. URH makes it exceptionally easy to apply: In the spectrogram view, we can choose the "band" or part of the signal that we can "pass." We make sure to keep our selection of the band wider than needed, as it's better to risk a bit of noise than to cap the signal. Applying the bandpass filter creates a copy of the signal that can pass the filter.

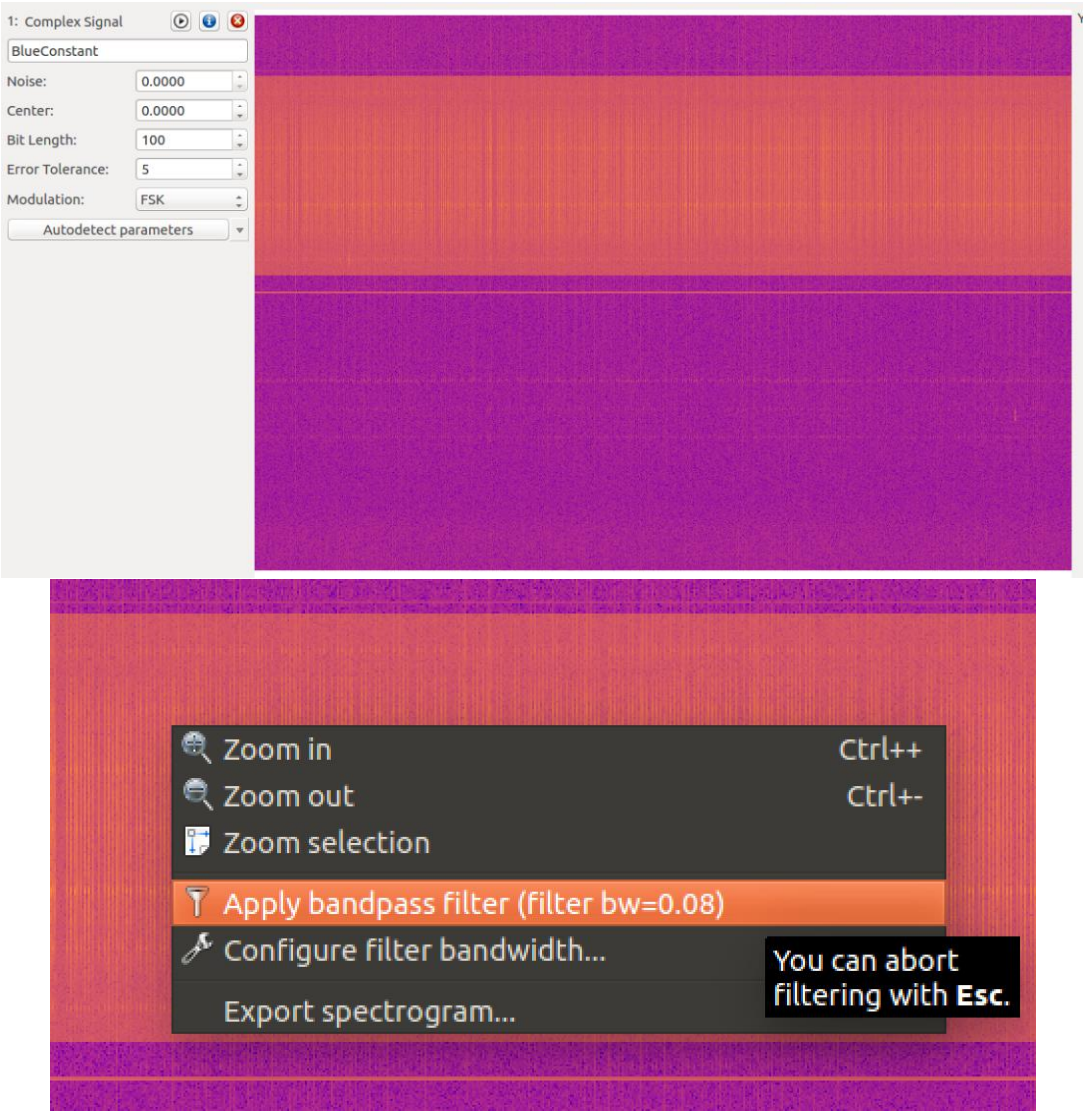


Figure 7. Using the bandpass filter

URH also has a de-noising feature. From the analog view, we're able to find the "silence" zone. Selecting this zone and setting it as "noise" makes the red area around the signal get narrower or ideally just a bit

larger than the “noise,” as seen in Figure 8. This method does not create a copy of the signal, but it’s still easy to copy and paste an entire signal in URH (memory permitting).

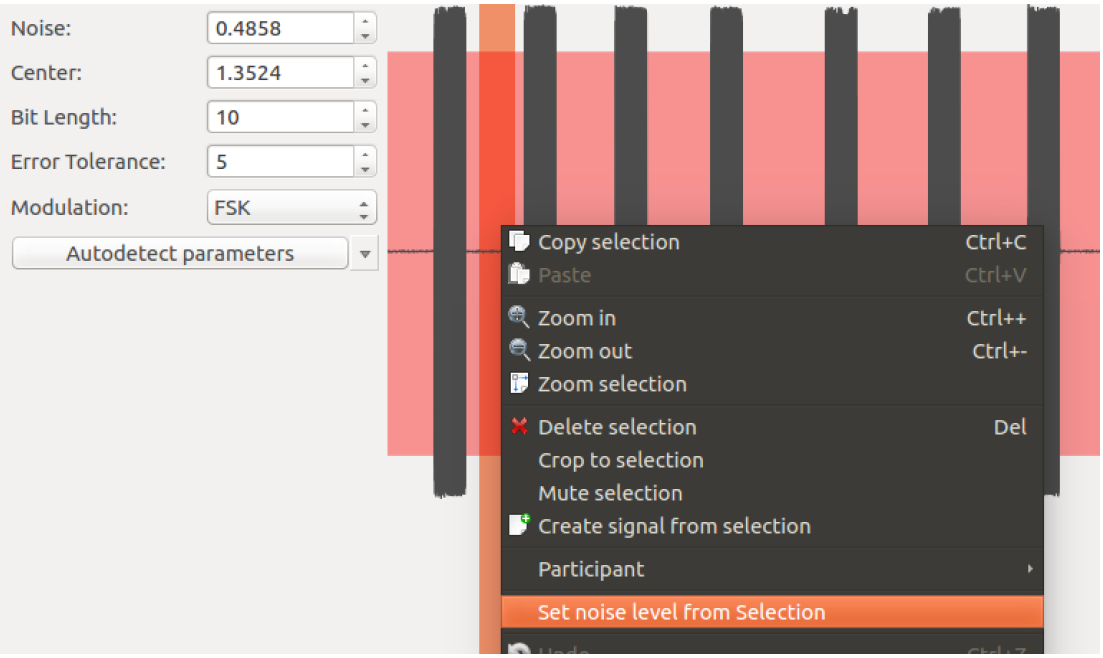
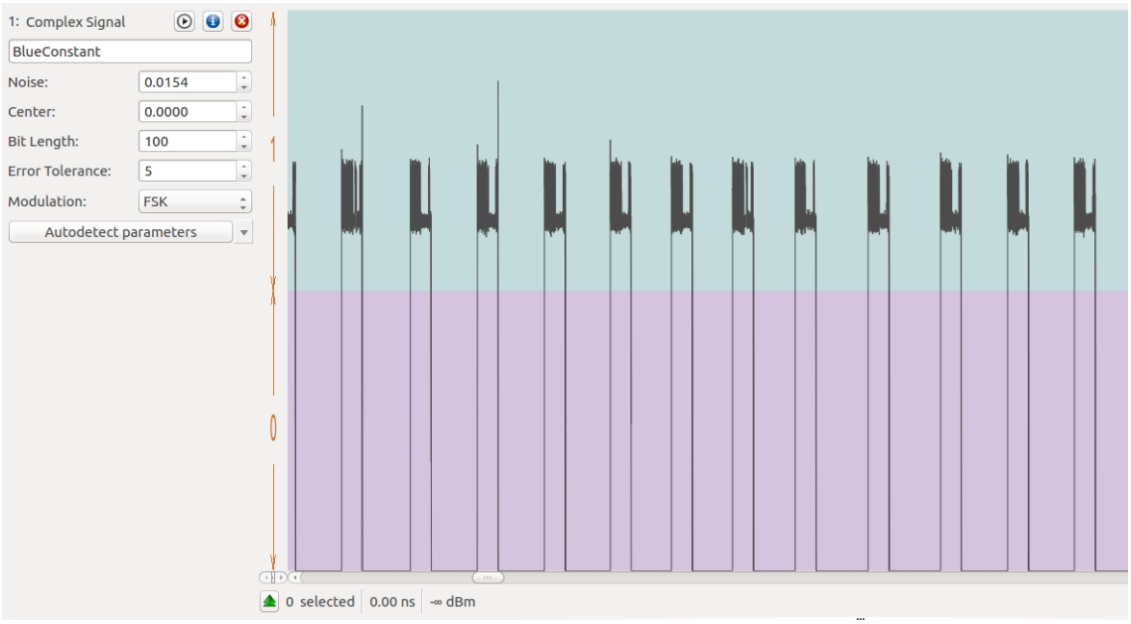


Figure 8. Using the noise canceling feature of URH

Demodulation

Since we’re dealing with a digital radio, we need to demodulate the analog signal to “see” the bits. In URH, we just switch to the “Demodulated” view, which results in the image in Figure 9.



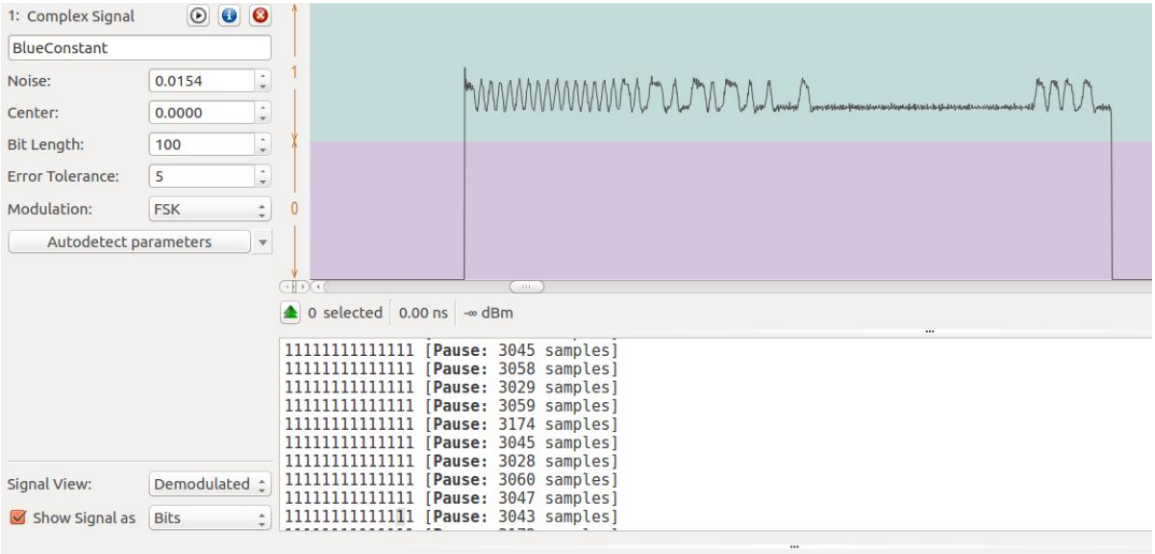


Figure 9. Using the demodulation feature of URH

We don't apply a bandpass filter in this example, so that the signal is still offset. Zooming in on this demodulated view reveals the packets in more detail, as seen in the bottom image of Figure 9. Since the center is still a bit offset, the demodulator interprets any sample of the signal above the threshold as 1. We just need to offset the threshold. A real (non-software) receiver does not require this adjustment since it's tuned to the exact carrier frequency and has built-in DC-noise cancellation subsystems, thereby making offset tuning unnecessary.

From the figure, it's also possible to see something that looks like a preamble at the beginning of each packet. The preamble is necessary in any digital packet radio communication to "wake up" the receiver and provide a reference to set the symbol rate. In other words, the initial part of the packet is known and has to be "101010 …" (unless configured differently).

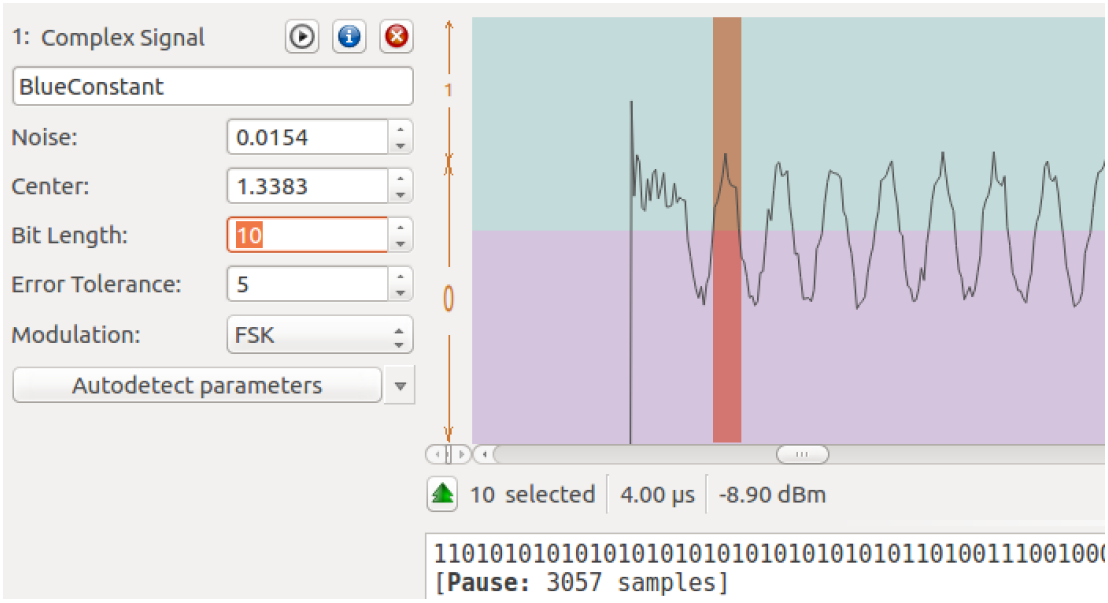


Figure 10. Setting a series of high-lows to the "1010" pattern

As seen in Figure 10, we use the preamble as a reference to set the symbol length (or bit length), assuming that one bit is one symbol in the alphabet. To do this, we select the shortest “pulse.” We find the shortest “symbol” in the preamble by choosing the point of the signal that crosses the threshold upward up to the point where the signal crosses back below the threshold.

A more precise way of estimating is to take multiple equally spaced symbols (e.g., 12, consisting of six 1s and six 0s) and to count their size in several samples and divide this number by the number of symbols. The symbol size is expressed in “number of IQ-samples” here, so if the sample rate is changed while capturing signals with the SDR, URH will show a different bit length. For this signal, we set the bit length to 10 samples.

After doing all this, our bitstream starts to look cleaner (11010101010101010101 ...). If we ignore the first 1, which is the effect of having not taken the capture in the best conditions, we can safely take that as a preamble, with the packet coming right after it.

Symbol interpretation and error correction

Switched to the hex view, URH shows the result seen in Figure 11. Typically, we see “0xAAAAAA ...”as the preamble. However, this is not the case for most of the samples.

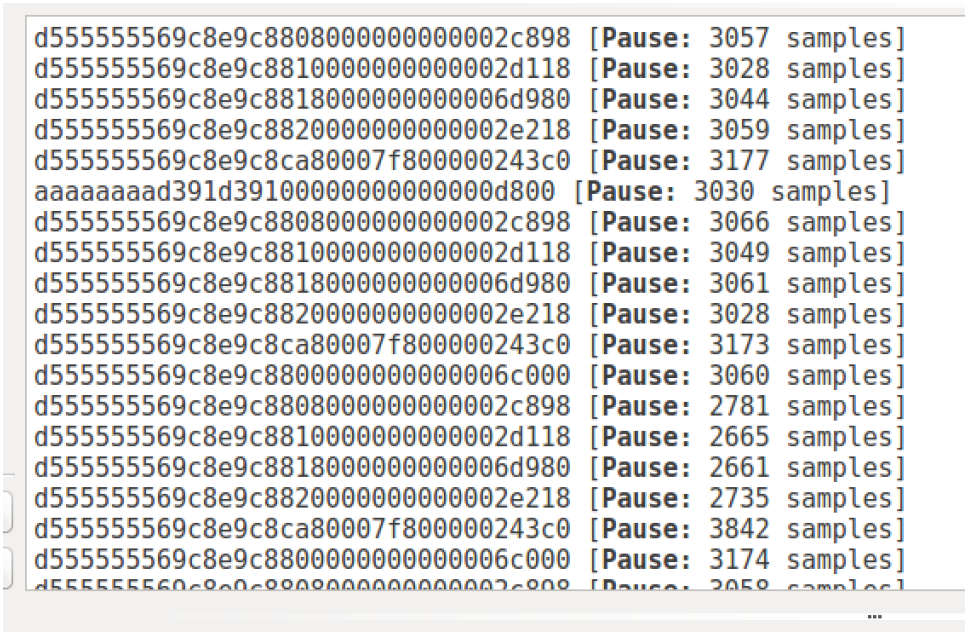


Figure 11. Hex view of the signals in URH

This result is strange, given that the sixth packet is decoded correctly — that is, showing the expected preamble. The challenge is to get all of the preambles decoded correctly. Two options are possible for remedying this problem, which can happen even when looking at other RF systems. We can either manually edit the signal and halve the length of the first “pulse” at the beginning of the preamble, or do this on the bitstream during post-processing. We choose the latter approach, although we describe how this can be done for a couple of packets. We just select the part of the signal that needs to be removed and delete it, as seen in Figure 12, thereby obtaining the signal in Figure 13.

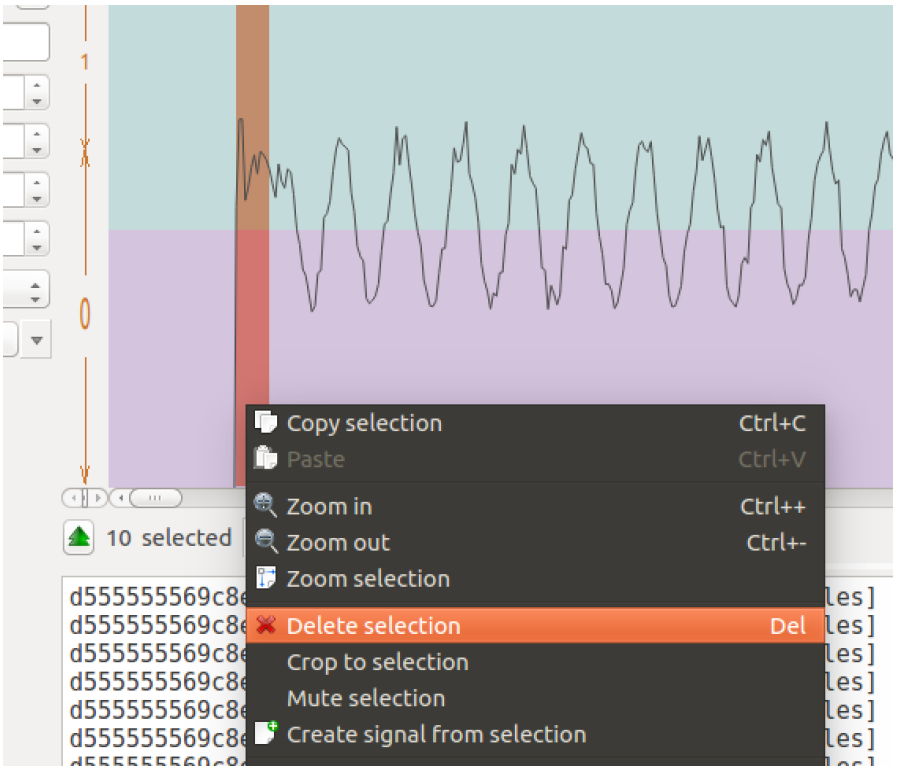


Figure 12. Removing part of the signal that is unnecessary

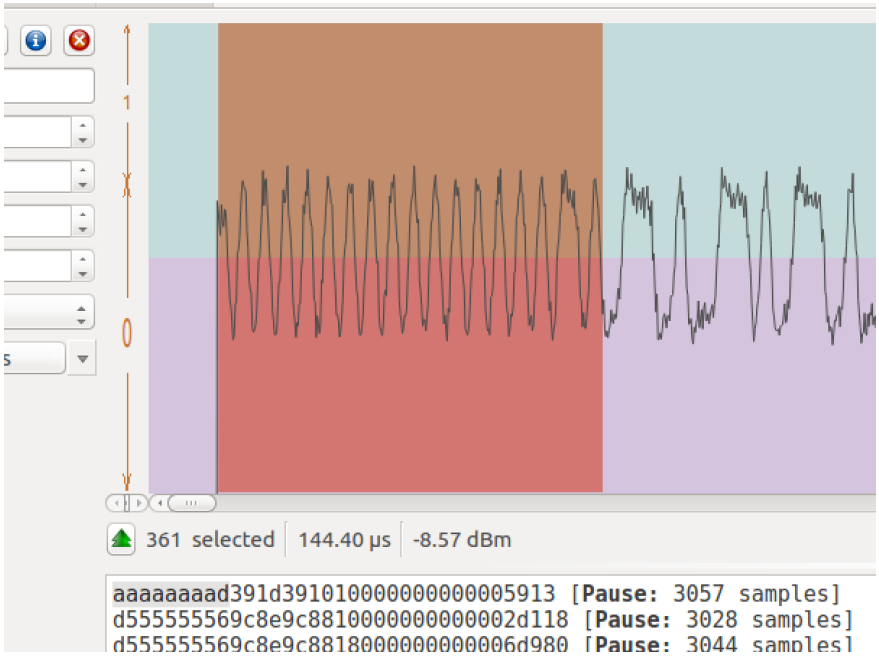


Figure 13. Preambles in the right length after removal of the unnecessary parts

To remove the extra 1 at the beginning of the preamble in post-processing, we go to the “Analysis” tab of URH, which offers a flexible decoding system and a good library of common decoders (e.g., Manchester, de-whitening).

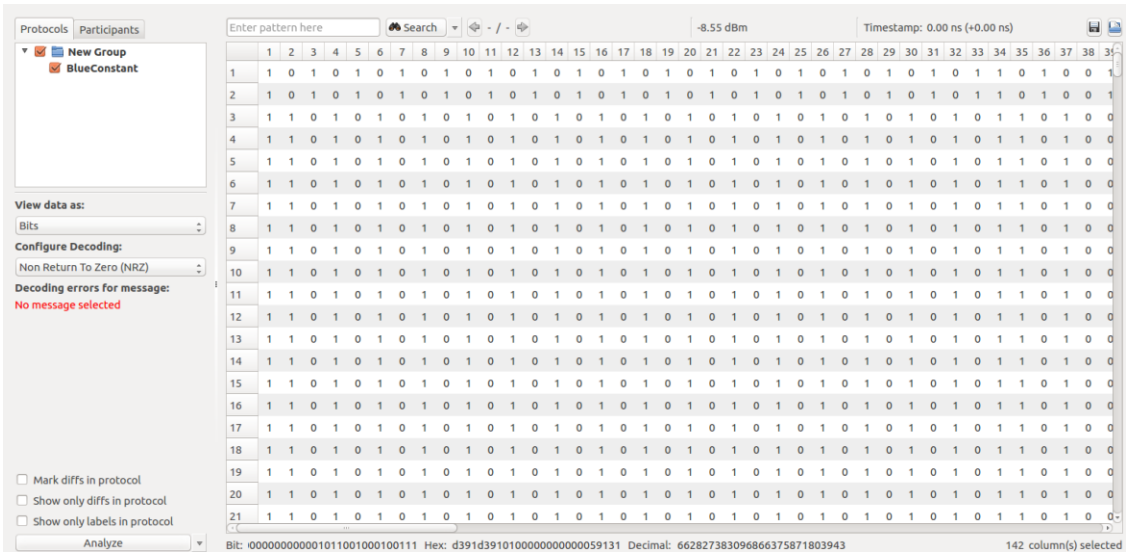


Figure 14. URH's decoding system

All of the packets contain a demodulation error in the first two bits of the preamble, except the first packet, which we fix manually on the "raw" signal. To fix the rest of the errors, we write a simple custom decoder, called "Fix preamble." To create it, we go to "Configure Decoding," as seen on the left side of Figure 14. This brings us to the window seen in Figure 15.

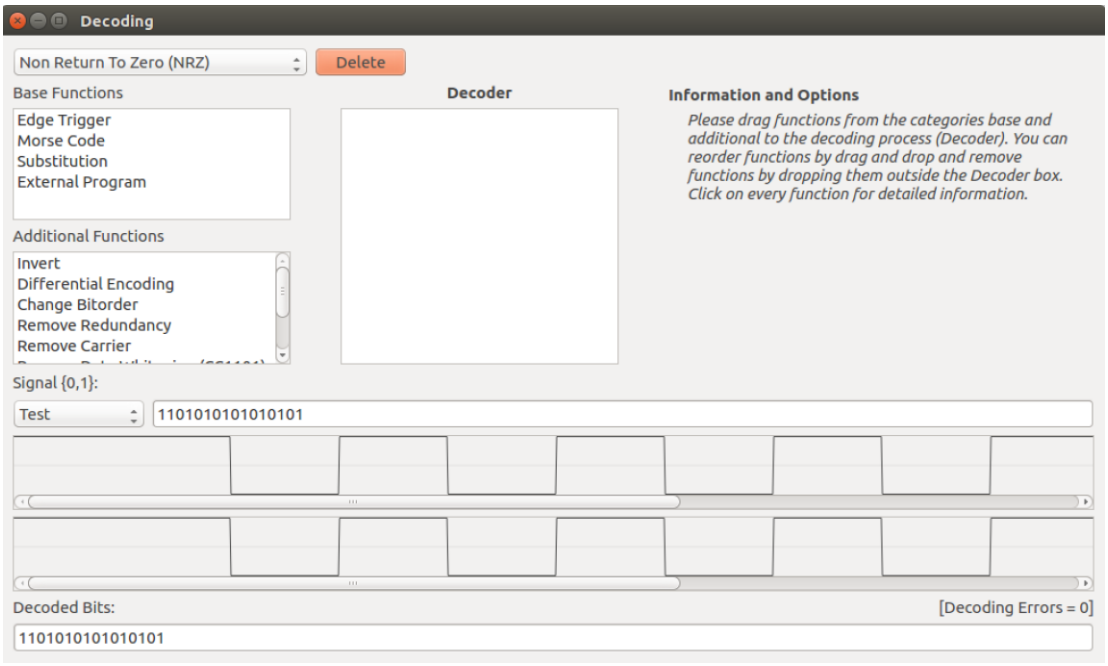


Figure 15. Writing a simple decoder in URH

We drag "Substitution" from the "Base Functions" list onto the "Decoder" area. We then enter a piece of preamble, one that includes the extra 1, in the "From" column. We copy the same preamble to the "To" column, this time without the extra 1. These steps are for a simple pattern substitution. To make sure that

it works as intended, one can enter a valid full packet (or just a longer preamble to be fixed) in the “Test” line seen in the middle of the screen and check if the “Decoded Bits” are an expected value.

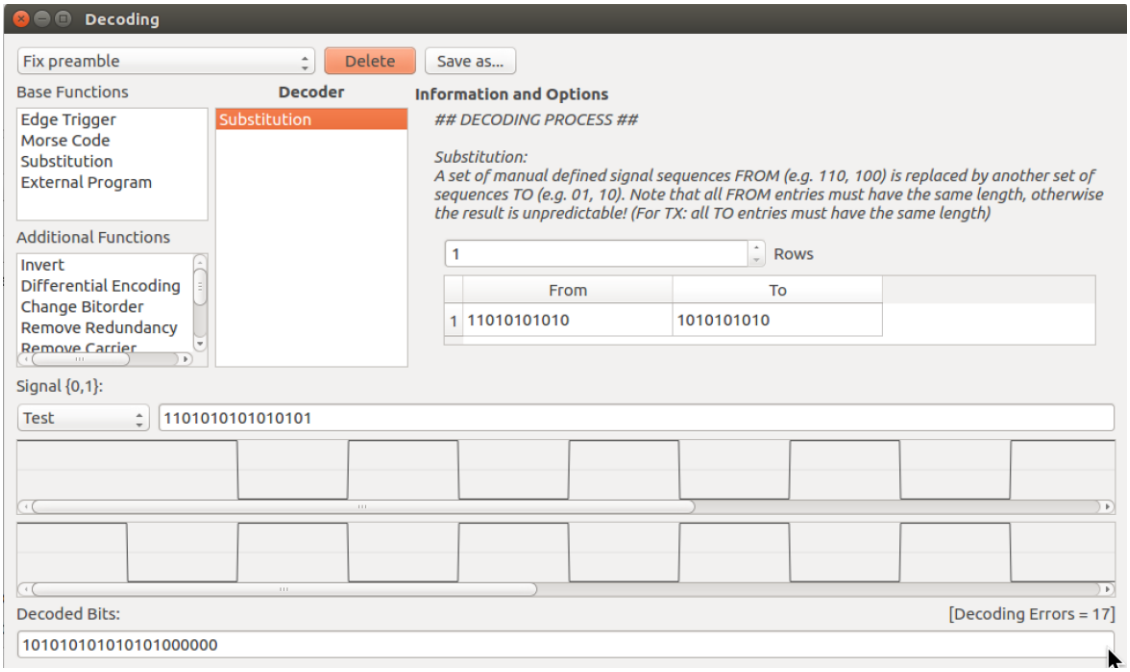


Figure 16. Simple pattern substitution in URH

We use this decoder for the rest of the packets.

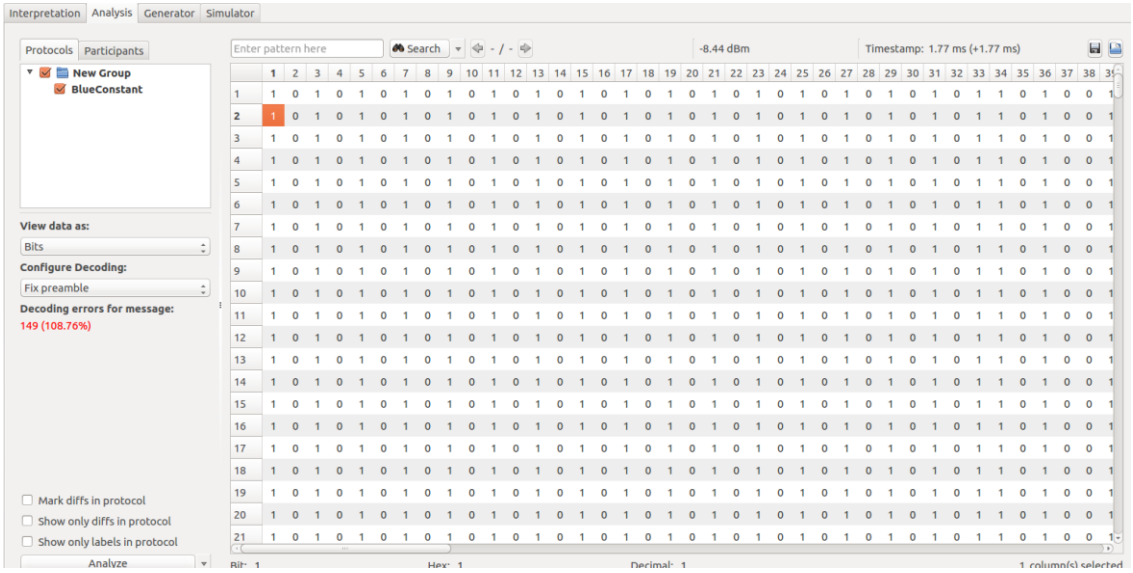


Figure 17. The fixed preambles

Finally, we take a look at the sync word, which is usually placed after the preamble for packet filtering purposes. The radio chip compares whatever comes after the preamble with the sync word value set in its registers. If the two match, the packet is accepted. If not, the packet is discarded. From its datasheet, we know that CC113L defaults to a double 16-bit sync word after the preamble (0xd391 0xd391), which is

exactly what we see. This confirms that we're using the right demodulation parameters and that we're going in the right direction.

Obtaining the packet structure through diffing

With the preambles fixed, we begin work on the packets in hex representation because it's more compact, and we proceed to comparing the packets to determine how they differ — a technique known as diffing. By checking "Mark diffs in protocol," as seen in Figure 18, we're able to immediately notice the structure. Even when using tools other than URH, this is a very common approach when reverse-engineering unknown communication protocols. By also checking "Show only diffs in protocol," we're able to notice that only two bytes changed: the byte after the sync word, which we've called "address," and another byte. Figure 18 also shows an important step in RF protocol analysis: labeling. Labeling packets with their semantic notations (blue, white, and red, in our case) and labeling each packet field using a different notation (color, in our case) is essential in figuring out the structure and patterns.

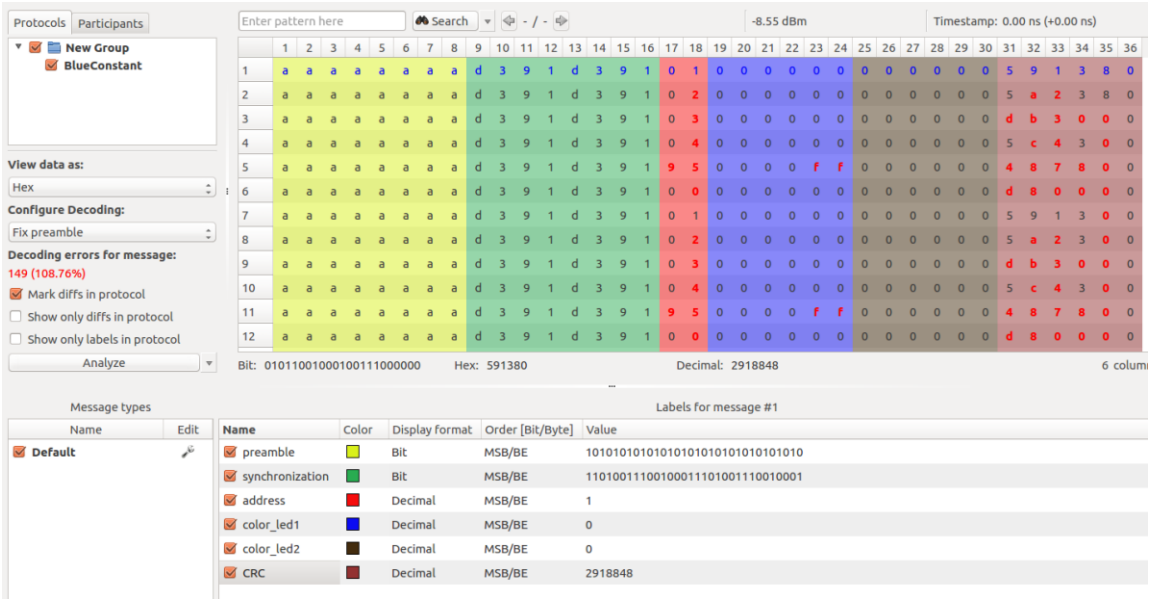


Figure 18. Marked differences in the protocol

Interestingly, the Drome transmitter box is labeled with "0x95", which we surmise is some form of address. We clearly see two types of packets: the "color packet" (address 0x95) and other "empty" packets (address 0x01 – 0x04). One interpretation of this is that this system can handle up to five independent "networks" of wristbands (from 1 to 4, plus 0x95). Another interpretation can be that those packets are synchronization "heartbeats," but it's hard to say at this point. The cyclic redundancy check (CRC) is a guess, but the datasheet comes in handy in this case.

5.10.1 Packet Format

The format of the data packet can be configured and consists of the following items (see Figure 5-6):

- Preamble
- Synchronization word
- Optional length byte
- Optional address byte
- Payload
- Optional 2 byte CRC

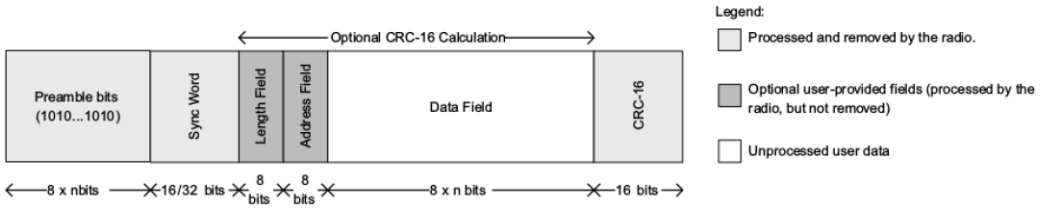


Figure 5-6. Packet Format

Figure 19. Packet format, according to the datasheet

According to the datasheet, there should be two bytes of CRC-16. We have three bytes, meaning we have to find out what the last byte is, which we mark as “unknown.”

At this point, we realize that we’ve analyzed only messages encoding red, green, and blue. We’ve been doing this because we’ve assumed that “white” is encoded as 0xffffff. Instead, we find that the payload structure is not 100% correct. The “white” is encoded as 0xffffff + 0xff (on the second, smaller LED, probably not an RGB). Accounting for this, we revise our initial field labeling and proceed to the next step.

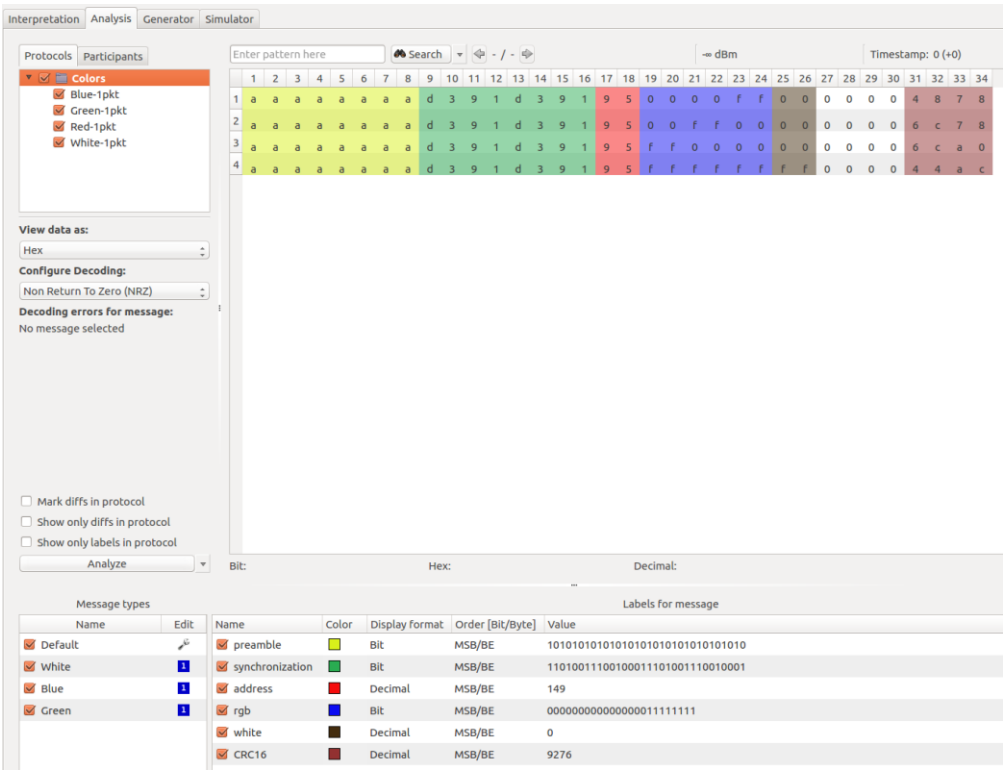


Figure 20. Accounting for the white messages encoding white light

At this point, we can already design a transmitter in URH, since we know almost all of the details. The final missing piece is the CRC-16 implementation. Doing a web search for “CC1101 CRC implementation” (recall that CC113L is the receiver-only version of CC1101), we find out that there may be slightly different implementations. Indeed, testing with the default CRC-16 IBM, we aren’t able to match the CRC-16 value.

Using libscrc, we brute-force the polynomial using a known packet’s CRC-16 value ('6ca0'), as seen below.

```
[x for x in [('{0:x}'.format(libscrc.ibm('\x95\xff\x00\x00\x00\x00\x00', i)), hex(i)) for i in
range(0xffff+1)] if x[0] == '6ca0']
```

This reveals that to generate the CRC-16, we should use the polynomial below.

```
'{0:x}'.format(libscrc.ibm(payload, 0xffff))
```

With this final step, we’re able to generate arbitrary packets with valid CRC-16 values that the receiver accepts.

Embedded protocol analysis

To design a precise transmitter, we need to know the exact radio parameters, which is why we need to dig deeper into the radio configuration. Again, multiple confirmations are very helpful in strengthening our knowledge of an embedded system.

Typical for embedded packet radios, the radio chip is connected to the MCU via the serial peripheral interface (SPI). Although there are embedded radios that function in the opposite way, the most common design is that the MCU is the SPI master (M) while the radio is the SPI slave (S).

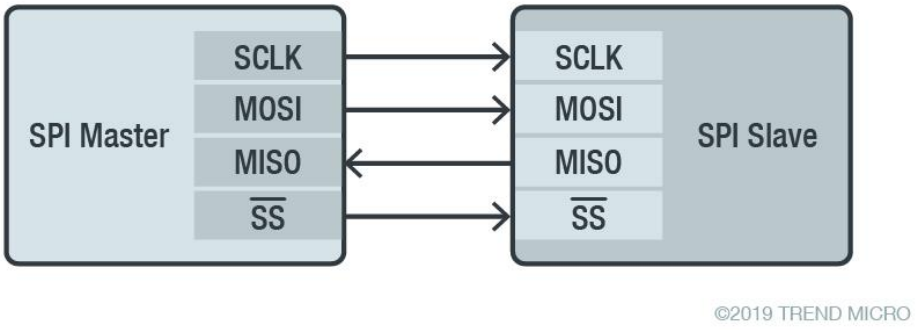


Figure 21. Typical MCU and SPI functions of embedded radios

Regardless of the specific code running on the MCU, at some point there must be a routine that sets up the radio. This is where the MCU sends SPI values and commands to the radio to make it ready to receive, transmit, or be in whatever state it needs to be to start functioning. This usually involves setting values onto certain registers and sending a series of commands, also known as “strobes.” Intercepting

such SPI communication is very helpful in knowing how the radio is set up, including — most importantly — the modem parameters (e.g., frequency, modulation, frequency deviation, and bandwidth).

Tapping into the SPI bus

As previously mentioned, the radio chip is enclosed in a 4 mm. by 4 mm. package. We don't have test clips for that size, so we have to do some soldering prior to connecting a logic analyzer. In principle, a logic analyzer works like an SDR: Instead of taking samples of a signal from an antenna, it takes samples of a signal measured between the ground and one or more other electrodes. With appropriate software, the samples are interpolated and the digital signal is reconstructed for subsequent analysis.

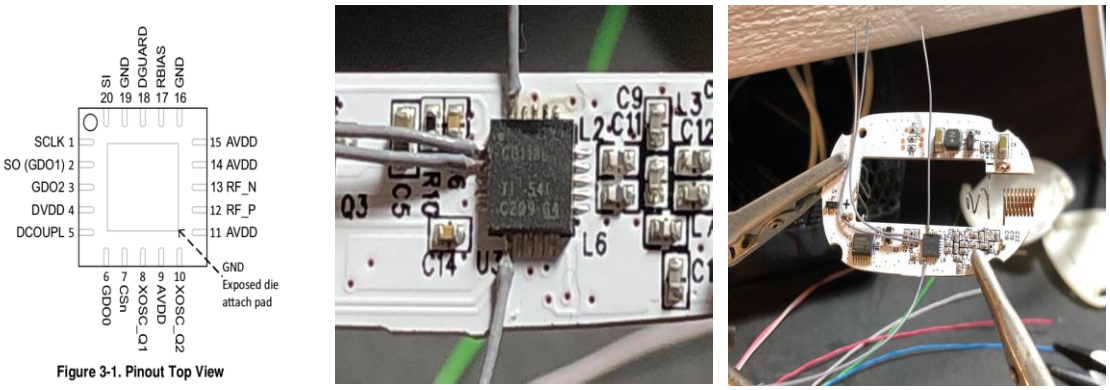
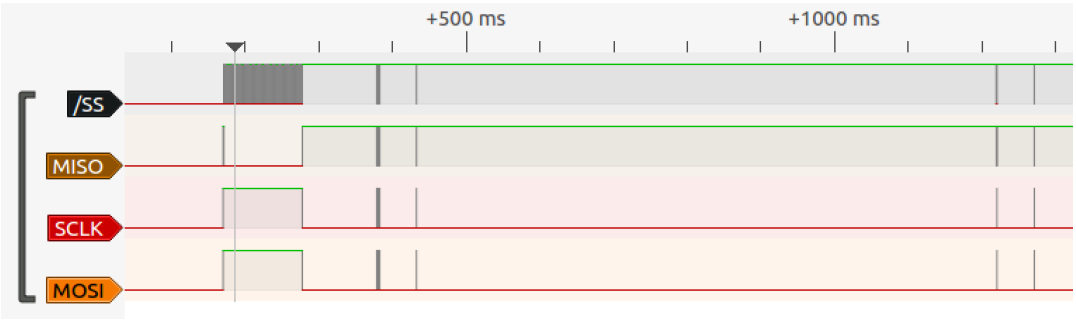


Figure 22. Detailed view of the radio chip and soldering to break out the SPI pins

We then hook up a simple logic analyzer — which, for starters, is more than enough to decode simple protocols — supported by Sigrok, an open-source signal analysis software. We first take a quick look with Sigrok's pulse view, enabling only the four channels we need, and set it to a 2 MHz sample rate. With the wristband powered off (no battery), we start to capture signals — which are all flat, of course. While capturing, we connect the battery to the wristband, making sure not to do this intermittently as it will spoil the SPI capture.



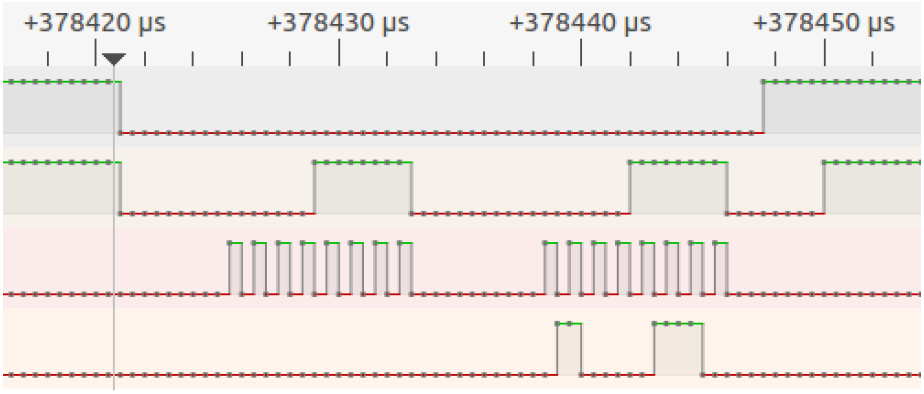


Figure 23. The signals as seen on Sigrok's pulse view

From the result shown in Figure 23, we can see, with a bit of zooming, the SS (slave select) signal going down to signal the beginning of an SPI transaction before going back to high, marking the end of the SPI transaction. We then try to decode the signal as a series of SPI transactions, without errors, as seen in Figure 24. With the shown good results, we can proceed and see if we can go beyond the SPI layer and interpret the higher-level CC113L proprietary protocol.

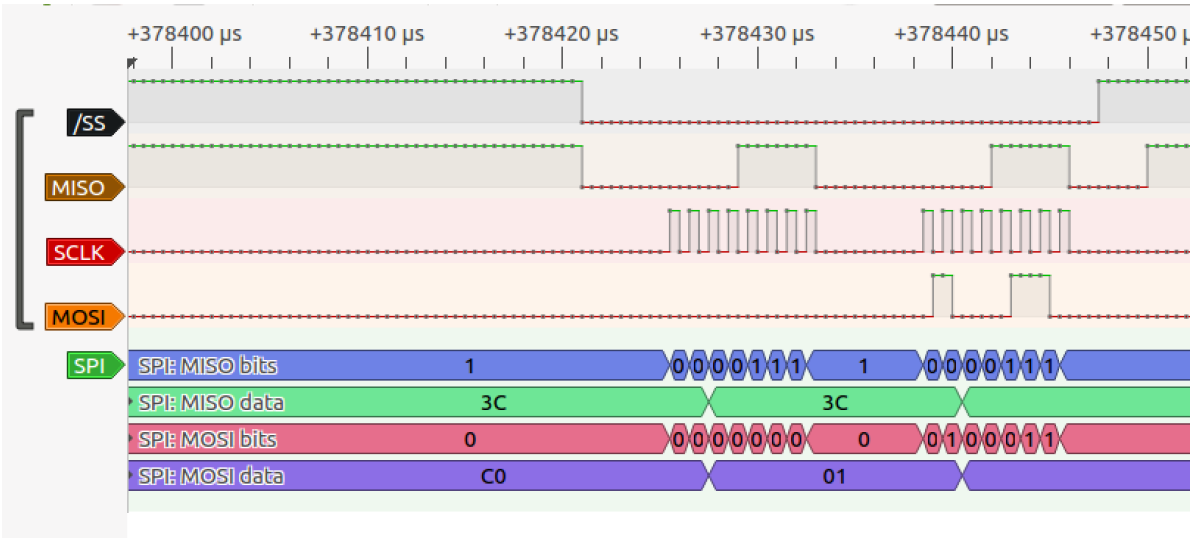


Figure 24. Decoding the SS signal to the SPI

Logic analyzer setup and CC1101/CC113L protocol decoding

When we looked into the CC1120 chip in our previous research, we had to write a custom decoder because no known implementations of the CC1120 SPI protocols were available. But Sigrok now includes a CC1101 SPI decoder based on the Sigrok API, which we use on the radio chip at hand. Logically, chip manufacturers tend to keep the same SPI protocol across various products. So with this assumption, CC113L should have the same SPI protocol as CC1101, or at least a very similar one. Upon testing, we find our assumptions to be correct.

To streamline the data collection, we switch the command line version of Sigrok as shown below:

```
$ sigrok-cli --continuous \
   -P spi:clk=SCLK:mosi=MOSI:miso=MISO:cs=SS,cc1101 -A cc1101 \
   --channels D0=SS,D1=MISO,D2=SCLK,D3=MOSI \
   --config "samplerate=2 MHz"
```

While doing this step, we have the wristband powered off. We then connect the wristband's battery, and after a few warnings, we see interesting data (Figure 25), which is followed by some strobe commands to set the radio in receive or idle mode (Figure 26).

```
user@ubuntu:~/Desktop/dromelights/cc113l/logic/sigrok-cli$ grep Write spi-cc1101-boot.txt
cc1101-1: Write: IOCFG2 (00) = 46
cc1101-1: Write: IOCFG0 (02) = 46
cc1101-1: Write: PKTLEN (06) = 07
cc1101-1: Write: PKTCTRL1 (07) = 0D
cc1101-1: Write: PKTCTRL0 (08) = 04
cc1101-1: Write: ADDR (09) = 95
cc1101-1: Write: FSCTRL1 (0B) = 12
cc1101-1: Write: FREQ2 (0D) = 21
cc1101-1: Write: FREQ1 (0E) = 71
cc1101-1: Write: FREQ0 (0F) = 7A
cc1101-1: Write: MDMCFG4 (10) = 2D
cc1101-1: Write: MDMCFG3 (11) = 3B
cc1101-1: Write: MDMCFG2 (12) = 93
cc1101-1: Write: MDMCFG1 (13) = 22
cc1101-1: Write: DEVIATN (15) = 62
cc1101-1: Write: MCSM0 (18) = 18
cc1101-1: Write: FOCCFG (19) = 1D
cc1101-1: Write: BSCFG (1A) = 1C
cc1101-1: Write: AGCTRL2 (1B) = C7
cc1101-1: Write: AGCTRL1 (1C) = 00
cc1101-1: Write: AGCTRL0 (1D) = B0
cc1101-1: Write: WORCTRL (20) = FB
cc1101-1: Write: FREND1 (21) = B6
cc1101-1: Write: FSCAL3 (23) = EA
cc1101-1: Write: FSCAL2 (24) = 2A
cc1101-1: Write: FSCAL1 (25) = 00
cc1101-1: Write: FSCAL0 (26) = 1F
cc1101-1: Write: FSTEST (29) = 59
cc1101-1: Write: PTEST (2A) = 7F
cc1101-1: Write: AGCTEST (2B) = 3F
cc1101-1: Write: TEST0 (2E) = 09
```

Figure 25. Recorded data after powering the wristband on and using the decoder

```
cc1101-1: Strobe SIDLE
cc1101-1: Strobe SFTX
cc1101-1: Strobe SNOP
cc1101-1: Strobe SFRX
cc1101-1: Strobe SRX
cc1101-1: Strobe SIDLE
cc1101-1: Strobe SPWD
cc1101-1: Strobe SIDLE
cc1101-1: Strobe SFTX
cc1101-1: Strobe SNOP
cc1101-1: Strobe SFRX
cc1101-1: Strobe SRX
cc1101-1: Strobe SIDLE
cc1101-1: Strobe SPWD
cc1101-1: Strobe SIDLE
cc1101-1: Strobe SFTX
cc1101-1: Strobe SNOP
cc1101-1: Strobe SFRX
cc1101-1: Strobe SRX
cc1101-1: Strobe SIDLE
cc1101-1: Strobe SPWD
cc1101-1: Strobe SIDLE
cc1101-1: Strobe SFTX
cc1101-1: Strobe SNOP
cc1101-1: Strobe SFRX
cc1101-1: Strobe SRX
cc1101-1: Strobe SIDLE
cc1101-1: Strobe SPWD
cc1101-1: Strobe SIDLE
```

Figure 26. Strobe commands to set the radio in receive or idle mode

We believe the firmware on the MCU is reusing “standard” library code to reset any CC11xx radio, because it’s sending SFTX probes (i.e., flush TX FIFO), which are not “available” in a receiver-only chip. Indeed, upon checking the datasheet, we see that the SFTX strobe isn’t listed on it. Since we’re using a decoder for a generic CC1101 (TX + RX), it’s decoded anyway. Interestingly, when we look at the SPI logs, we find out that the MCU sends the 0x3B strobe, which is SFTX for the CC1101, but is marked as “reserved” for the CC113L — confirming that different chips may use a common design.

Table 5-16. Command Strobes

Address	Strobe Name	Description
0x30	SRES	Reset chip.
0x31	Reserved	
0x32	SXOFF	Turn off crystal oscillator.
0x33	SCAL	Calibrate frequency synthesizer and turn it off. SCAL can be strobed from IDLE mode without setting manual calibration mode (MCSM0.FS_AUTOCAL=0)
0x34	SRX	In IDLE state: Enable RX. Perform calibration first if MCSM0.FS_AUTOCAL=1.
0x35	Reserved	
0x36	SIDLE	Enter IDLE state
0x37 - 0x38	Reserved	
0x39	SPWD	Enter power down mode when CSn goes high.
0x3A	SFRX	Flush the RX FIFO buffer. Only issue SFRX in IDLE or RXFIFO_OVERFLOW states.
0x3B - 0x3C	Reserved	
0x3D	SNOP	No operation. May be used to get access to the chip status byte.

Table 1. List of command strobes and their descriptions, according to the datasheet

Radio parameters

The last step is to decode the register settings into meaningful radio parameters. Configuring CC11xx chips can be complicated, so Texas Instruments, their manufacturer, provides a configuration tool called SmartRF Studio. This tool can derive the exact registry values to obtain the desired RF parameters, and vice versa. Inputting the register values into SmartRF Studio will show the RF parameters.

Figure 27 shows that we're right about our assumption that the wristband is using 2-FSK. GFSK (as seen in the same figure) is just the Gaussian version of it, with a smoother transition. We're also correct in the assumption that 0x95 is the address.

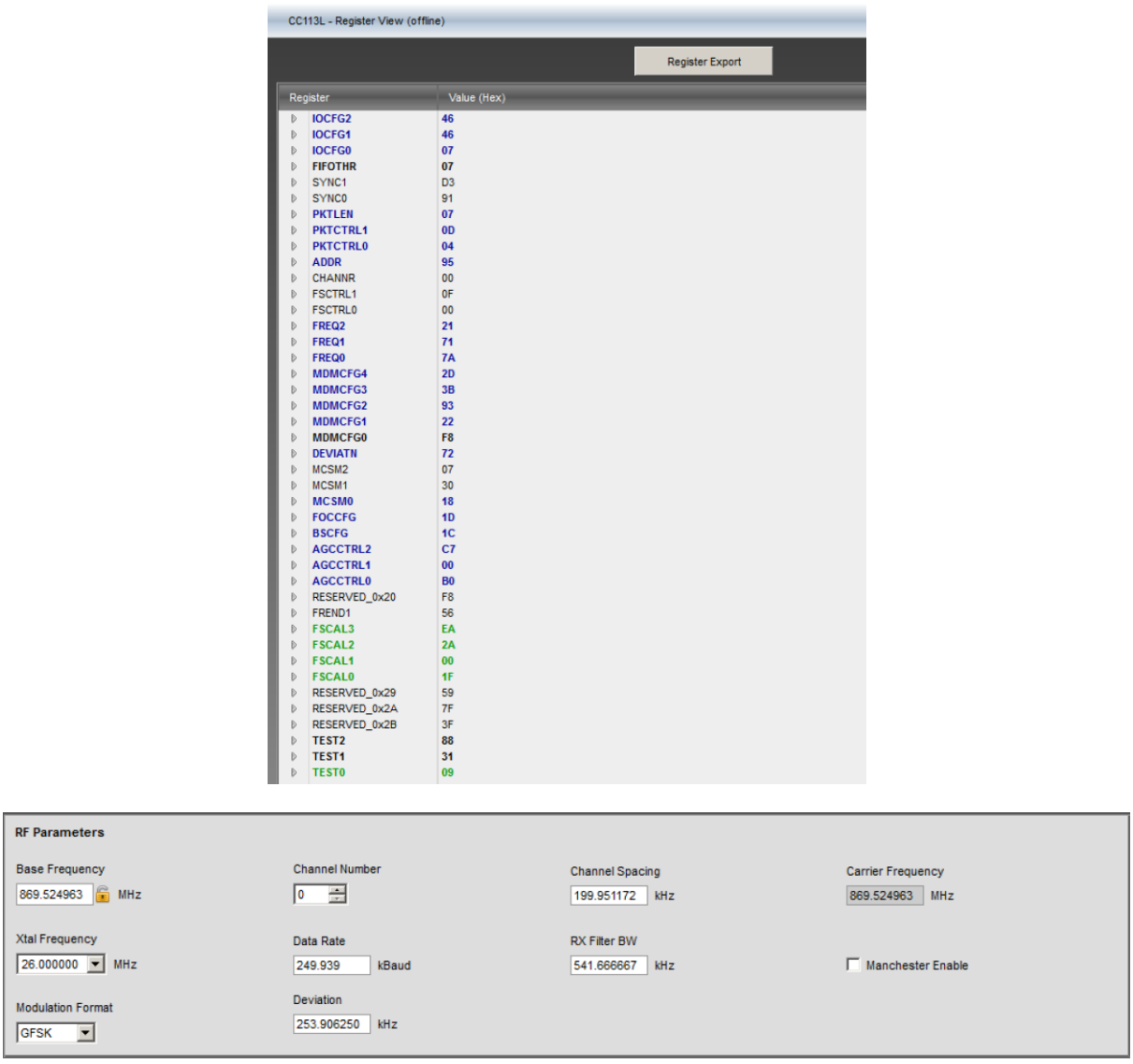


Figure 27. Using SmartRF Studio to show RF parameters

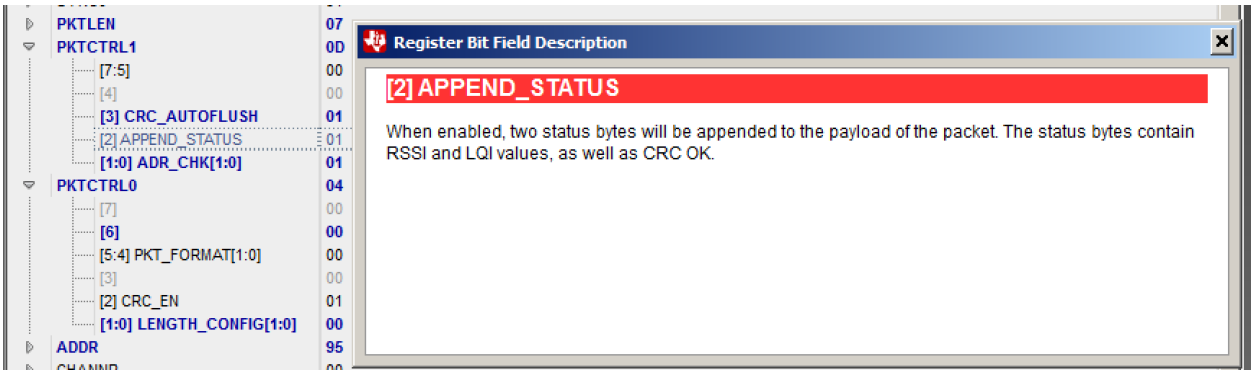


Figure 28. A closer look at the packet structure

A closer look at the packet structure, as shown in Figure 28, reveals that its length is equal to 0x07 bytes, with CRC enabled (auto-flushed on the receiver side so it won't register as part of the payload in the RX FIFO) and two status bytes appended by the receiver (stripped off if the packet is accepted so it also won't show up on the RX FIFO).

Overall, since there is still one byte that we're unable to see from our RF decoding, we've successfully reconstructed 99% of the packet structure.

Application data

We then check what SPI transactions we will record when the radio receives commands instead. To do this, we repeat the last procedure using the Sigrok command line front end Sigrok-cli.

```
user@ubuntu:~/Desktop/dromelights/cc113l/logic/sigrok-cli$ grep "Burst read" spi-cc1101-rx-blue.
txt  | grep "= 95"
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00 0F
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00 0F
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00 0F
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00 0F
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00 0F
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00 0F
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00 0F
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00
cc1101-1: Burst read: FIFO (3F) = 95 00 00 FF 00 00 00
```

Figure 29. Repeated procedure using Sigrok-cli

We purposely search only for packets matching the address. Most of the packets match what we see in URH, although the last byte is slightly off. This is because of the decoder: Since the fixed packet length is 0x07, the eighth byte (0x0F) should not have been there. We assume that it's a status byte value. Simply put, the decoder confuses the status byte returned on the MISO (master input, slave output) line with part of the RX FIFO burst transaction, or we haven't set the logic analyzer's sample rate high enough.

Transmitter design

Finally, we now have enough information to design our own transmitter. We use the following parameters, which we've collected and confirmed from the steps above:

- Modulation: GFSK (we can use FSK and it will work anyway)
- Precise carrier frequency: 869.524963 MHz
- Bandwidth: 541.666667 kHz
- Frequency deviation: 253.906250 kHz
- Data rate: 249.939 kBaud

We can use URH, GNU Radio, or any other SDR software to design our transmitter. Another way is to use an RF dongle featuring a CC1101 chip (e.g., YardStickOne, PandwaRF) or any other radio chip with enough bandwidth (e.g., RFQuack with the RFM69 shield), since we know how the registers are set.

Equipped with a reliable transmitter, we're able to forge packets and make the wristband flash lights in whatever way we want.

Conclusion and security recommendations

This was a very useful exercise that we decided to share as a learning experience. It led us to discover DMX512, a relevant industry protocol. It also allowed us to review and add to our knowledge of the CC11xx family. The use of the CC11xx family in a different setting and in a more novel and personal device gave us a more holistic understanding of it.

From a broader perspective, although attacks on LED wristbands may have a decidedly lower impact than those on industrial radio controllers, this application of RF technology is still worth looking into. This case demonstrates the ubiquity of RF technology and consequently its wide attack surface. The use of RF technology in controlling lights alone is applied to large events, like concerts and theatrical presentations, and instances of compromise involving it can ruin the experience of these events.

We did contact the vendor of the wristbands used in our research to disclose our findings, considering that, as previously mentioned, while small events may not offer any real opportunity to an attacker, large shows, especially broadcast ones, can. If the lighting system of an important show does not work as expected while thousands of people are watching, the reputation damage can be substantial.

Vendors, integrators, and users alike have a responsibility to ensure the security of devices. Vendors should continue to build on well-known, standard protocols. And they should uphold security from the design phase by building radio transceivers that support encryption capabilities and applying tamper-proof mechanisms that can hinder reverse engineering. Integrators and users should inspect the technical manuals of devices before purchasing and implementing them. In doing so, they should have a preference for vendors that offer dual-technology devices and use well-known, standard protocols.

Researchers, for their part, should keep on analyzing obscure RF protocols, because by no means should "obscurity" rhyme with "security." Researchers who are not familiar with these topics should not feel overwhelmed by the amount of knowledge needed to analyze embedded devices. As this report-cum-tutorial shows, there are plenty of opportunities to get started and learn.

TREND MICRO™ RESEARCH

Trend Micro, a global leader in cybersecurity, helps to make the world safe for exchanging digital information.

Trend Micro Research is powered by experts who are passionate about discovering new threats, sharing key insights, and supporting efforts to stop cybercriminals. Our global team helps identify millions of threats daily, leads the industry in vulnerability disclosures, and publishes innovative research on new threats techniques. We continually work to anticipate new threats and deliver thought-provoking research.

www.trendmicro.com