## 17:04 Sniffing BTLE with the Micro:Bit

*by Damien Cauquil*

Howdy y'all!

It's well known that sniffing Bluetooth Low Energy communications is a pain in the bottom, unless you have specialty tools like the Ubertooth One and its competitors. During my exploration of the BBC Micro:Bit, I discovered the very interesting fact that it may be used to sniff BLE communications.

The BBC Micro:Bit is a small device based on a nRF51822 transceiver made by Nordic Semiconductor, with a 5 × 5 LED screen and two buttons that can be powered by two AAA batteries. The nRF51822 is able to communicate over multiple protocols: Enhanced ShockBurst (ESB), ShockBurst (SB), GZLL, and Bluetooth Low Energy (BLE).

Nordic Semiconductor provides its own implementation of a Bluetooth Low Energy stack, released in what they call a SoftDevice and a well-known closed-source sniffing firmware used in Adafruit's BlueFriend LE sniffer for instance. That doesn't help that much, as this firmware relies on BLE connection requests to start following a specific connection, and not on packets exchanged between two devices in an existing connection. So, I found no way to cheaply sniff an existing BLE connection.

In this short article, I'll describe how to implement a Bluetooth Low Energy sniffer as software on the BBC Micro:Bit that can follow pre-existing connection despite channel hopping. In cases where channel remapping is in use, it can sniff connections on which even the Ubertooth currently fails.

### The Goodspeed Way of Sniffing

The Micro:Bit being built upon a nRF51822, it ignited a sparkle in my mind as I remembered the hack found by our great neighbor Travis Goodspeed who managed to turn another Nordic Semiconductor transceiver (nRF24L01+) into a sniffer.[7] I was wondering if by any chance this nRF51822 would have been prone to the same error, and therefore could be turned into a BLE sniffer.

It took me hours to figure out how to reproduce this exploit on this chip, but in fact it works exactly the same way as described in Travis' paper. Since the nRF51822 is a lot different than the nRF24L01+ (as it includes its own CPU rather being driven by a SPI bus), we must change multiple parameters in order to sniff BLE packets over the air.

First, we need to enable the processor high frequency clock because it is required before enabling the RADIO module of the nRF51822. This is done with the following code.

```
NRF_CLOCK->EVENTS_HFCLKSTARTED = 0;
NRF_CLOCK->TASKS_HFCLKSTART = 1;
while (NRF_CLOCK->EVENTS_HFCLKSTARTED == 0);
```

Then, we must specify the mode, addresses, power and frequency our nRF51822 will be tuned to.

```
/* Max power. */
NRF_RADIO->TXPOWER = (
  RADIO_TXPOWER_TXPOWER_0dBm
  << RADIO_TXPOWER_TXPOWER_Pos);

/* Setting addresses. */
NRF_RADIO->TXADDRESS = 0;
NRF_RADIO->RXADDRESSES = 1;

/* BLE channels are not contiguous, so you
   need to convert them into frequency
   offset. */
NRF_RADIO->FREQUENCY =
  channel_to_freq(channel);

/* Set BLE data rate. */
NRF_RADIO->MODE = (RADIO_MODE_MODE_Ble_1Mbit
                   << RADIO_MODE_MODE_Pos);

/* Set the base address. */
NRF_RADIO->BASE0 = 0x00000000;
NRF_RADIO->PREFIX0 = 0xAA; // preamble
```

The trick here, as described in Travis' paper, is to use an address length of two bytes instead of the five bytes expected by the chip. The address length is stored in a configuration register called `PCNF0`, along with other extra parameters. The `PCNF0` and `PCNF1` registers define the way the nRF51822 will behave: its endianness, the expected payload size, the address size and much more documented in the nRF51 Series Reference Manual.[8]

The following lines of code configure the nRF51822 to use a two-byte address, big-endian with a maximum payload size of 10 bytes.

[7] `unzip pocorgtfo17.pdf promiscuousnrf24l01.pdf # Promiscuity is the nRF24L01+'s Duty`

[8] `unzip pocorgtfo17.pdf nrf51.pdf`

```
// LFLEN=0 bits, S0LEN=0, S1LEN=0
NRF_RADIO->PCNF0 = 0x00000000;
// STATLEN=10, MAXLEN=10, BALEN=1,
// ENDIAN=0 (little), WHITEEN=0
NRF_RADIO->PCNF1 = 0x00010A0A;
```

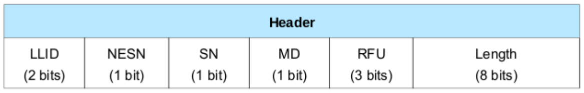
Eventually, we have to disable the CRC computation in order to make the chip consider any data received as valid.

```
NRF_RADIO->CRCCNF = 0x0;
```

## Identifying BLE Connections

With this setup, we can now receive crappy data from the 2.4GHz bandwidth and hopefully some BLE packets. The problem is now to find the needle in the haystack, that is a valid BLE packet in the huge amount of data received by our nRF51822.

A BLE packet starts with an access address, a 32-bit carefully-chosen value that uniquely identifies a link between two BLE devices, as specified in the Bluetooth 4.2 Core Specifications document. This access address is followed by some PDU and a 3-byte CRC, but this CRC value is computed from a CRCInit value that is unique and associated with the connection. The BLE packet data is whitened in order to make it more tamper-resistant, and should be dewhitened before processing. If the connection is already initiated, as it is our case, the PDU is a Data Channel PDU with a specific two-byte header, as stated in the Bluetooth Low Energy specifications.



| Header | | | | | |
|---|---|---|---|---|---|
| LLID<br>(2 bits) | NESN<br>(1 bit) | SN<br>(1 bit) | MD<br>(1 bit) | RFU<br>(3 bits) | Length<br>(8 bits) |

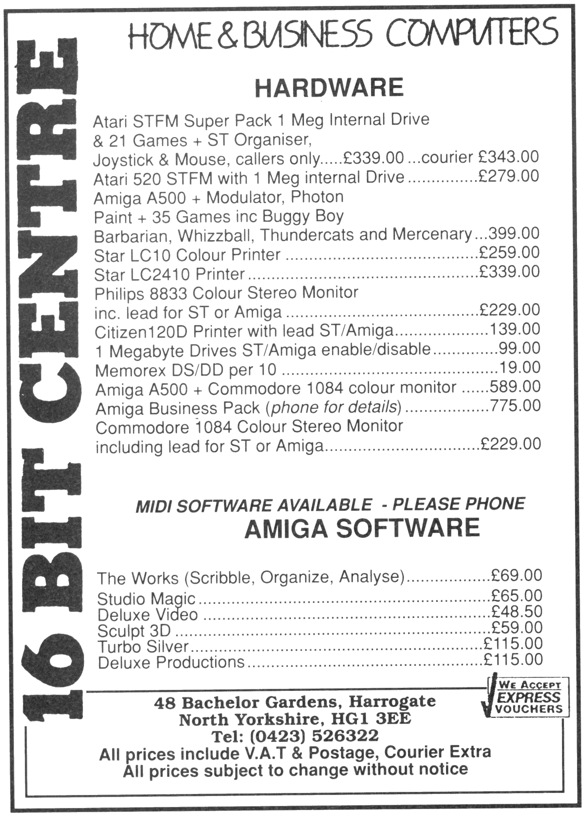
*Figure 2.13: Data channel PDU header*

When a BLE connection is established, keep-alive packets with a size of 0 bytes are exchanged between devices.

Again, we follow the same methodology as Travis' by listing all the candidate access addresses we get, and identifying the redundant ones. This is the same method chosen by Mike Ryan in its Ubertooth BTLE tool from WOOT13,[9] with a nifty trick: we determine a valid access address based on the number of times we have seen it combined with a filter on its dewhitened header. We may also want to rely on the way the access address is generated, as the core specifications give a lot of extra constraints access address must comply with, but it is not always followed by the different implementations of the Bluetooth stack.

Once we found a valid access address, the next step consists in recovering the initial CRC value which is required to allow the nRF51822 to automatically check every packet CRC and let only the valid ones go through. This process is well documented in Mike Ryan's paper and code, so we won't repeat it here.

With the correct initial CRC value and access address in hands, the nRF51822 is able to sniff a given connection's packets, but we still have a problem. The BLE protocol implements a basic channel hopping mechanism to avoid sniffing. We cannot sit on a channel for a while without missing packets, and that's rather inconvenient.



[9] `unzip pocorgtfo17.pdf woot13-ryan.pdf`

```
function pickUniqueChannel(a_channelMap) :
  aa_sequences = generateSequences(a_channelMap)
  for channel in range (0..37) do :
    if (a_channelMap contains channel) then do :
      for increment in range (0..12) do :
        count = 0
        for i in range (0..37) do :
          if aa_sequences[increment][i] == channel then do :
            count = count + 1
            if count > 1 then do :
              break
            end if
          end if
        end for

        if count == 1 then do :
          return channel
        end if
      end for
    end if
  end for

  return -1
end function

function computeRemapping(a_channelMap) :
  a_remapping =  []
  j = 0
  for channel in range (0..37) do :
    if a_channelMap contains channel then do :
      a_remapping[j] = channel
      j = j + 1
    end if
  end for

  return a_remapping
end function

function generateSequences(a_channelMap) :
  aa_sequences = [][]
  remapping = computeRemapping(a_channelMap)
  for i in range (0..12) do :
    aa_sequences[i] = generateSequence(i+5, a_channelMap, a_remapping)
  end for
  return aa_sequences
end function

function generateSequence(increment, a_channelMap, a_remapping) :
  channel = 0
  a_sequence = []
  for i in range (0..37) do :
    if i in a_channelMap then do :
      sequence[i] =  channel
    else
      sequence[i] = a_remapping[ channel modulo size of a_remapping]
    end if

    channel = (channel + increment) % 37
  end for
end function
```

Figure 2. Hopping Algorithm

## Following the Rabbit

The Bluetooth Low Energy protocol defines 37 different channels to transport data. In order to communicate, two devices must agree on a hopping sequence based on three characteristics: the hop interval, the hop increment, and the channel map.

The first one, the hop interval, is a value specifying the amount of time a device should sit on a channel before hopping to the next one. The hop increment is a value between 5 and 16 that specifies the number of channels to add to the current one (modulo the number of used channels) to get the next channel in the sequence. The last one may be used by a connecting device to restrict the channels used to the ones given in a bitmap. The channel map was quite a surprise for me, as it isn't mentioned in Ubertooth's BTLE documentation.[10]

We need to know these values in order to capture every possible packets belonging to an active connection, but we cannot get them directly as we did not capture the connection request where we would find them. We need to deduce these values from captured packets, as we did for the CRC initial value. In order to find out our first parameter, the hop interval, Mike Ryan designed the simplest algorithm that could be: measuring the time between two packets received on a specific channel and dividing it by the number of channels used, i.e. 37. So did I, but my measures did not seem really accurate, as I got two distinct values rather than a unique one. I was puzzled, as it would normally have been straightforward as the algorithm is simple as hell. The only explanation was that a valid packet was sent twice before the end of the hopping cycle, whereas it should only have been sent once. There was something wrong with the hopping cycle.

It seems Mike Ryan made an assumption that was correct in 2013 but not today in 2017. I checked the channels used by my connecting device, a Samsung smartphone, and guess what? It was only using 28 channels out of 37, whereas Mike assumed all 37 data channels will be used. The good news is that we now know the channel map is really important, but the bad news is that we need to redesign the connection parameters recovery process.

## Improving Mike Ryan's Algorithm

First of all, we need to determine the channels in use by listening successively on each channel for a packet with our expected access address and a valid CRC value. If we get no packet during a certain amount of time, then it means this channel is not part of the hopping sequence. Theoretically, this may take up to four seconds per channel, so not more than three minutes to determine the channel map. This is a significant amount of time, but luckily devices generally use more than half of the available channels so it would be quicker.

Once the channel map is recovered, we need to determine precisely the hop interval value associated with the target connection. We may want our sniffer to sit on a channel and measure the time between two valid packets, but we have a problem problem: if less than 37 channels are used, one or more channels may be reused to fill the gaps. This behavior is due to a feature called "channel remapping" that

[10] `unzip pocorgtfo17.pdf ubertooth.zip; unzip -c ubertooth.zip ubertooth/host/doc/ubertooth-btle.md | less`

is defined in the Bluetooth Low Energy specifications, which basically replace an unused channel by another taken from the channel map. It means a channel may appear twice (or more) in the hopping sequence and therefore compromise the success of Mike's approach.

```
37 channels in use, no remapping:
{ 0, 1, 2, 3, ... , 27, 28, 29, 30,
  31, 32, 33, 34, 35, 36, 37}

28 first channels in use:
{ 0, 1, 2, 3, ... , 27, 0, 1, 2, 3,
  4, 5, 6, 7, 8}
```

A possible workaround involves picking a channel that appears only once in the hopping sequence, whatever the hop increment value. If we find such a channel, then we just have to measure the time between two packets, and divide this value by 37 to recover the hop interval value. The algorithm in Figure 2 may be used to pick this channel.

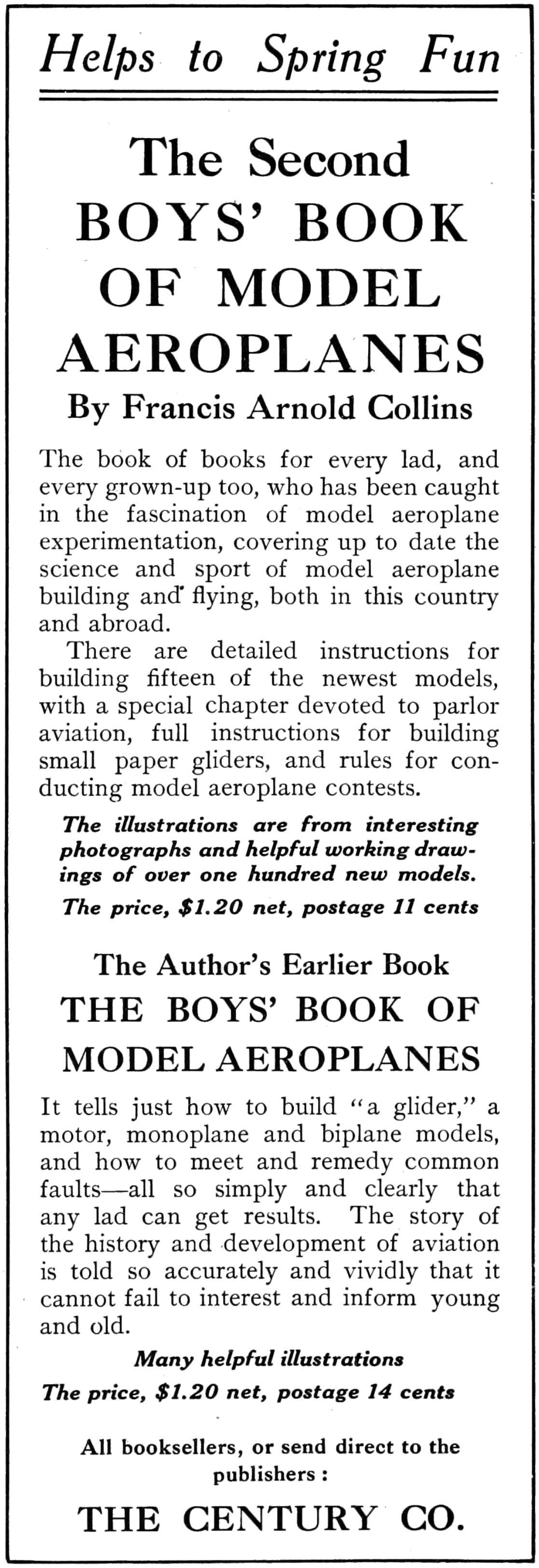
This algorithm finds a unique channel only if more than the half of the data channels are used, and may possibly work for a fewer number of channels depending on the hop increment value. This quick method doesn't require a huge amount of packets to guess the hop interval.

The last parameter to recover is the hop increment, and Mike's approach is also impacted by the number of channels in use. His algorithm measures the time between a packet on channel 0 and channel 1, and then relies on a lookup table to determine the hop increment used. The problem is, if channel 1 appears twice then the measure is inaccurate and the resulting hop increment value guessed wrong.

Again, we need to adapt this algorithm to a more general case. My solution is to pick a second channel derived from the first one we have already chosen to recover the hop interval value, for which the corresponding lookup table only contains unique values. The lookup table is built as shown in Figure 3.

Eventually, we try every possible combination and only keep one that does not contain duplicate values, as shown in Figure 4.

Last but not least, in Figure 5 we build the lookup table from these two carefully chosen channels, if any. This lookup table will be used to deduce the hop increment value from the time between these two channels.



```
function generateLUT(aa_sequences, firstChannel, secondChannel) :
  aa_lookupTable = [][]
  for increment in range (0..12) do :
    aa_lookupTable[increment] = computeDistance(aa_sequences, increment,
                                                firstChannel, secondChannel)
  end for
end function

function computeDistance(aa_sequences, increment, firstChannel, secondChannel) :
  distance = 0
  fcIndex = findChannelIndex(aa_sequences, increment, firstChannel, 0)
  scIndex = findChannelIndex(aa_sequences, increment, secondChannel, fcIndex)
  if (scIndex > fcIndex) then do :
    distance = (scIndex - fcIndex)
  else do :
    distance = (scIndex - fcIndex) + 37
  end if

  return distance
end function

function findChannelIndex(aa_sequences, increment, channel, start) :
  for i in range (0..37) do :
    if aa_sequences[increment] [( start + i) modulo 37] == channel then do :
      return ((start + i) modulo 37)
    end if
  end for
end function
```

Figure 3. Channel Lookup Table

```
function pickSecondChannel(aa_sequences, a_channelMap, firstChannel) :
  for channel in range (0..37) do :
    if a_channelMap contains channel then do :
      lookupTable = generateLUT(aa_sequences, firstChannel, channel)
      duplicates = FALSE
      for i in range (0..11) do :
        for k in range (i+1 .. 12) do :
          if lookupTable[i] == lookupTable[k] then do :
            duplicates = TRUE
          end if
        end for
      end for

      if not duplicates then do :
        return channel
      end if
    end if
  end for

  return -1
end function
```

Figure 4. Picking the Second Channel

```
function deduceHopIncrement(aa_sequences, firstChannel, secondChannel,
                            measure, hopInterval) :
  channelsJumped =  measure / hopInterval
  LUT = generateHopIncrementLUT(aa_sequences, firstChannel, secondChannel)
  if LUT[channelsJumped] > 0 then do :
    return LUT[channelsJumped]
  else do :
    return -1
  end if
end function

function generateHopIncrementLUT(aa_sequences, firstChannel, secondChannel) :
  reverseLUT = generateLUT(aa_sequences, firstChannel, secondChannel)
  LUT = []
  for i in range (0..37) do :
    LUT[i] = 0
  end for
  for i in range (0..12) do :
    LUT[reverseLUT[i]] = i+5
  end for

  return LUT
end function
```

Figure 5. Deducing the Hop Increment

## Patching BBC Micro:Bit

Thanks to the designers of the BBC Micro:Bit, it is possible to easily develop on this platform in C and C++. Basically, they wrote a Device Abstraction Layer[11] that provides everything we need except the radio, as they developed their own custom protocol derived from Nordic Semiconductor ShockBurst protocol. We must get rid of it.

I removed all the useless code from this abstraction layer, the piece of code in charge of handling every packet received by the RADIO module of our nRF51822 in particular. I then substitute this one with my own handler, in order to perform all the sniffing without being annoyed by some hidden third-party code messing with my packets.

Eventually, I coded a specific firmware for the BBC Micro:Bit that is able to communicate with a Python command-line interface, and that can be used to detect and sniff existing connections. This is not perfect and still a work in progress, but it can passively sniff BLE connections. Of course, it may lack the legacy sniffing method based on capturing connection requests; that will be implemented later.

This tiny tool, dubbed `ubitle`, is able to enumerate every active Bluetooth Low Energy connections.

```
# python3 ubitle.py -s
uBitle v1.0 [firmware version 1.0]

[i] Listing available access addresses ...
[ - 46 dBm] 0x8a9b8e58 | pkts: 1
[ - 46 dBm] 0x8a9b8e58 | pkts: 2
[ - 46 dBm] 0x8a9b8e58 | pkts: 3
```

It is also able to recover the channel map used by a given connection, as well as its hop interval and increment.

```
# python3 ubitle.py -f 0x8a9b8e58
uBitle v1.0 [firmware version 1.0]

[i] Following connection 0x8a9b8e58 ...
[i] Recovered initial CRC value: 0x16e9df
[i] Recovering channel map.
[i] Recovered channel map: 0x1fffffffff
[i] Recovering hop interval ...
[i] Recovered hop interval: 48
[i] Recovering hop increment ...
[i] Recovered hop increment: 16
```

[11] `git clone https://github.com/lancaster-university/microbit-dal`

Once all the parameters recovered, it may also dump traffic to a PCAP file.

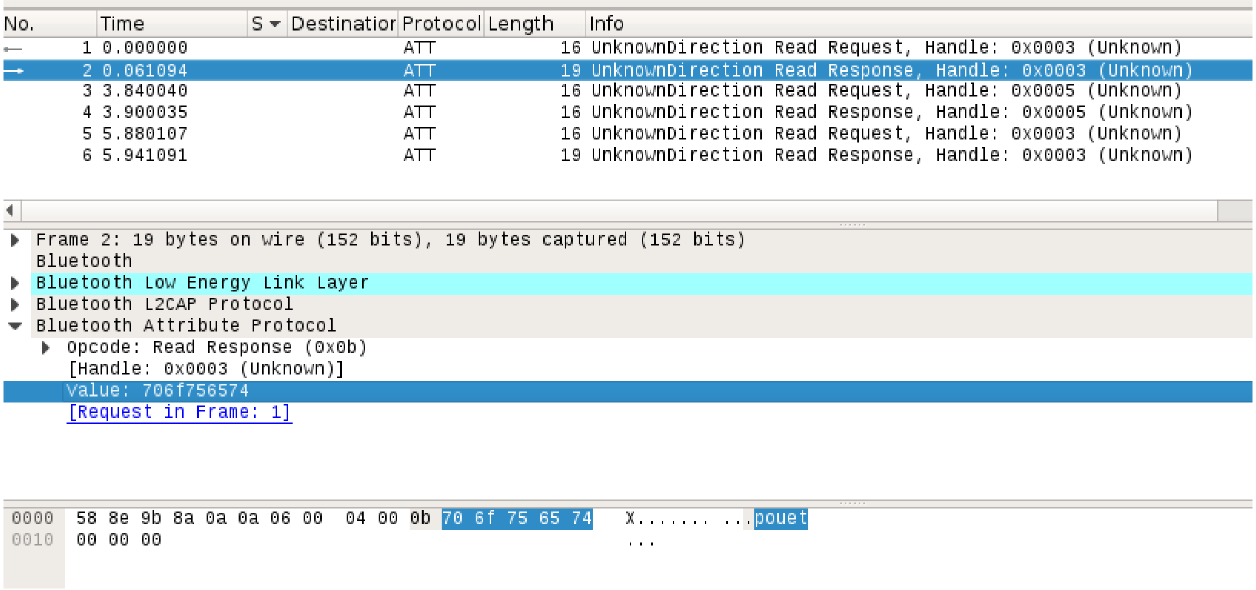
```
# python3 ubitle.py -f 0x8a9b8e58 \
  -m 0x1fffffffff -o test.pcap
uBitle v1.0 [firmware version 1.0]

[i] Following connection 0x8a9b8e58 ...
[i] Recovered initial CRC value: 0x16e9df
[i] Forced channel map: 0x1fffffffff
[i] Recovering hop interval ...
b'\xbcC\x06\x00X\x8e\x9b\x8a0\x00\xf1'
[i] Recovered hop interval: 48
[i] Recovering hop increment ...
[i] Recovered hop increment: 16
[i] All parameters successfully recovered,
    following BLE connection ...
LL Data: 02 07 03 00 04 00 0a 03 00
LL Data: 0a 0a 06 00 04 00 0b 70 6f 75 65 74
LL Data: 02 07 03 00 04 00 0a 05 00
LL Data: 0a 07 03 00 04 00 0b 00 00
LL Data: 02 07 03 00 04 00 0a 03 00
LL Data: 0a 0a 06 00 04 00 0b 70 6f 75 65 74
```

The resulting PCAP file may be opened in Wireshark to dissect the packets. You may notice the keep-alive packets are missing from this capture. It is deliberate; these packets are useless when analyzing Bluetooth Low Energy communications.

## Source code

The source code of this project is available on Github under GPL license, feel free to submit bugs and pull requests.[12]

This tool does not support dynamic channel map update or connection request based sniffing, which are implemented in Nordic Semiconductor's closed source sniffer. It's PoC||GTFO so take my little tool as it is: a proof of concept demonstrating that it is possible to passively sniff BLE connections for less than twenty bucks, with a device one may easily find on the Internet.



```
No.  Time      S  Destination Protocol Length Info
   1 0.000000             ATT        16 UnknownDirection Read Request, Handle: 0x0003 (Unknown)
   2 0.061094             ATT        19 UnknownDirection Read Response, Handle: 0x0003 (Unknown)
   3 3.840040             ATT        16 UnknownDirection Read Request, Handle: 0x0005 (Unknown)
   4 3.900035             ATT        16 UnknownDirection Read Response, Handle: 0x0005 (Unknown)
   5 5.880107             ATT        16 UnknownDirection Read Request, Handle: 0x0003 (Unknown)
   6 5.941091             ATT        19 UnknownDirection Read Response, Handle: 0x0003 (Unknown)

▶ Frame 2: 19 bytes on wire (152 bits), 19 bytes captured (152 bits)
  Bluetooth
▶ Bluetooth Low Energy Link Layer
▶ Bluetooth L2CAP Protocol
▼ Bluetooth Attribute Protocol
  ▶ Opcode: Read Response (0x0b)
    [Handle: 0x0003 (Unknown)]
    Value: 706f756574
    [Request in Frame: 1]

0000  58 8e 9b 8a 0a 0a 06 00  04 00 0b 70 6f 75 65 74   X....... ...pouet
0010  00 00 00                                           ...
```

[12] git clone https://github.com/virtualabs/ubitle-firmware || unzip pocorgtfo17.pdf ubitle.tgz

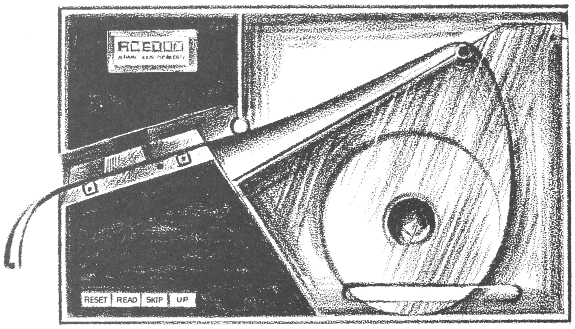
## 17:05 Up close and personal with Ethernet.

*by Andrew D. Zonenberg,*
*because real hackers don't need PHYs or NICs!*

If you're reading this, you've almost certainly used Ethernet on a PC by means of the BSD sockets API. You've probably poked around a bit in Wireshark and looked at the TCP/IP headers on your packets. But what happens after the kernel pushes a completed Ethernet frame out to the network card?

A PC network card typically contains three main components. These were separate chips in older designs, but many modern cards integrate them all into one IC. The bus controller speaks PCIe, PCI, ISA, or some other protocol to the host system, as well as generating interrupts and handling DMA. The MAC (Media Access Controller) is primarily responsible for adding the Ethernet framing to the outbound packet. The MAC then streams the outbound packet over a "reconciliation sublayer" interface to the PHY (physical layer), which converts the packet into electrical or optical impulses to travel over the cabling. This same process runs in the opposite direction for incoming packets.

In an embedded microcontroller or SoC platform, the bus controller and MAC are typically integrated on the same die as the CPU, however the PHY is typically a separate chip. FPGA-based systems normally implement a MAC on the FPGA and connect to an external PHY as well; the bus controller may be omitted if the FPGA design sends data directly to the MAC. Although the bus controller and its firmware would be an interesting target, this article focuses on the lowest levels of the stack.



### MII and Ethernet framing

The reconciliation sublayer is the lowest (fully digital) level of the Ethernet protocol stack that is typically exposed on accessible PCB pins. For 10/100 Ethernet, the base protocol is known as MII (Media Independent Interface). It consists of seven digital signals each for the TX and RX buses: a clock (2.5 MHz for 10Base-T, 25 MHz for 100Base-TX), a data valid flag, an error flag, and a 4-bit parallel bus containing one nibble of packet data. Other commonly used variants of the protocol include RMII (reduced-pin MII, a double-data-rate version, which uses less pins), GMII (gigabit MII, that increases the data width to 8 bits and the clock to 125 MHz), and RGMII (a DDR version of GMII using less pins). In all of these interfaces, the LSB of the data byte/nibble is sent on the wire first.

An Ethernet frame at the reconciliation sublayer consists of a preamble (seven bytes of `0x55`), a start frame delimiter (SFD, one byte of `0xD5`), the 6-byte destination and source MAC addresses, a 2-byte EtherType value indicating the upper layer protocol (for example `0x0800` for IPv4 or `0x86DD` for IPv6), the packet data, and a 32-bit CRC-32 of the packet body (not counting preamble or SFD). The byte values for the preamble and SFD have a special significance that will be discussed in the following section.

### 10Base-T Physical Layer

The simplest form of Ethernet still in common use is known as 10Base-T (10 Mbps, baseband signaling, twisted pair media). It runs over a cable containing two twisted pairs with 100 ohm differential impedance. Modern deployments typically use Category 5 cabling, which contains four twisted pairs. The orange and green pairs are used for data (one pair in each direction), while the blue and brown pairs are unused.

When the line is idle, there is no voltage difference between the positive (white with stripe) and negative (solid colored) wires in the twisted pair. To send a 1 or 0 bit, the PHY drives 2.5V across the pair; the direction of the difference indicates the bit value. This technique allows the receiver to reject noise coupled into the signal from external electro-