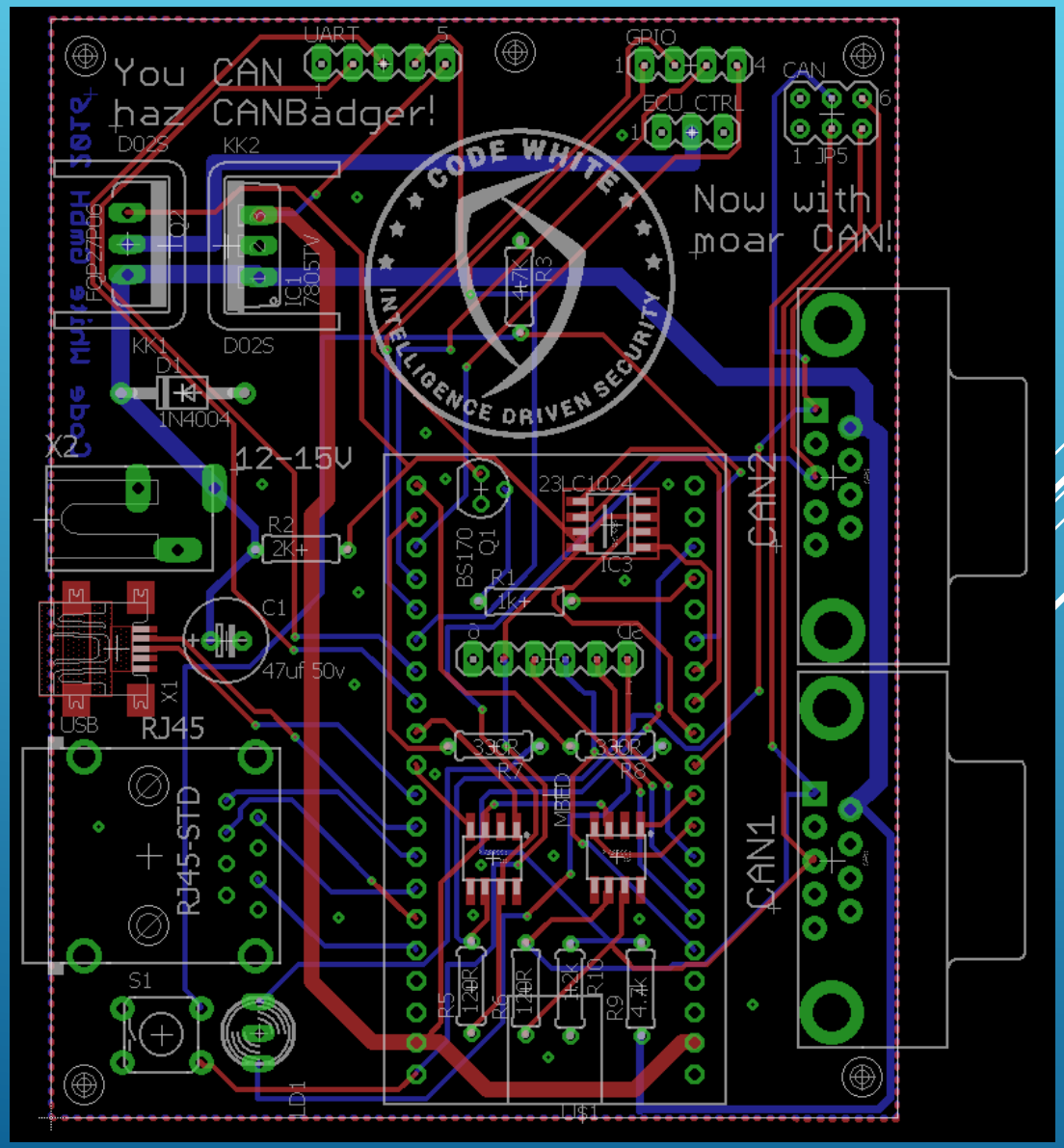
You CAN haz car secretz!



Javier Vazquez Vidal

Ferdinand Noelscher

Overview

1) CAN bus protocols
   - 1.1) UDS
   - 1.2) TP 2.0
   - 1.3) SIDs
   - 1.4) Diagnostics (PIDs)
2) CANBadger Features
3) CANBadger Standalone Mode
4) CANBadger USB mode
5) CANBadger GUI

CAN bus protocols: UDS

What is UDS?

- UDS stands for Unified Diagnostics Service
- It is present in all modern cars
- Provides access to the Services offered by ECUs
- It takes place on the top of the CAN protocol (Layer 4)
- Allows to perform transmissions of up to 256 bytes
- It is just a Transport Protocol

CAN bus protocols: UDS

What types of transmissions exist in UDS?

- Single frame transmissions (7 bytes or less payload length)
- Multiple frame transmissions (more than 7 bytes payload length)

CAN bus protocols: UDS

What kind of frames exist in UDS?

- No ACK request, part of multiframe transmission(0x20)
- ACK request, part of multiframe transmission (0x10)
- No ACK request, single frame (0x00)
- ACK (0x30)
- NACK/Repeat transmission

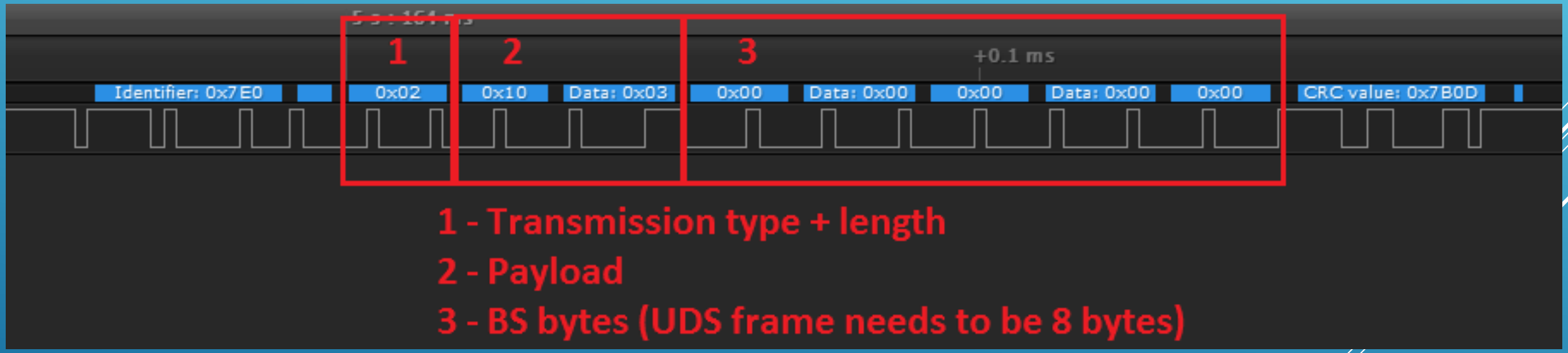
CAN bus protocols: UDS

How is an UDS transmission constructed?

- Transmission type is always stored in form of "first byte of the frame" & 0xF0. This means that it is determined by the 4 MSBs.
- If it's a single frame transmission, the first byte indicates transmission type and payload length (& 0x0F). The rest of the frame is the payload.
- If it's a multiframe, the first byte indicates if there is an ACK request (0x10) or not (0x20), plus a counter that goes from 0x00 to 0x0F and that is reset on every transmission. The transmission length is indicated by the second byte in the first frame of the transmission.

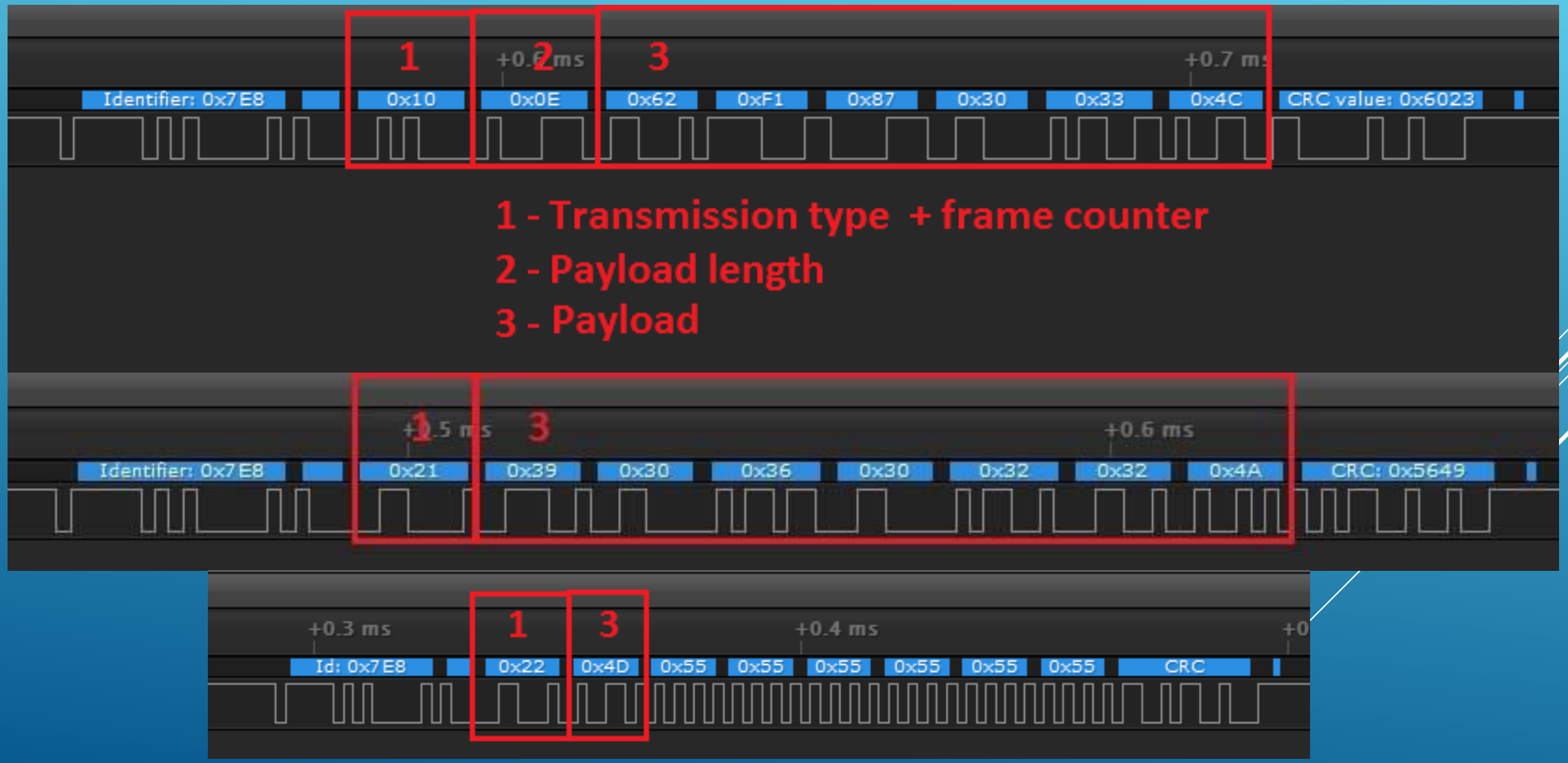
CAN bus protocols: UDS

Single frame transmission example



CAN bus protocols: UDS

Multiframe Transmission Example



CAN bus protocols: UDS

Did you notice?

- When the BS bytes are 0x00, it means that the payload is native UDS
- When the BS bytes are 0x55, it means that the payload is KWP2000 over UDS (notice the Tester Address 0xF1 as well).

CAN bus protocols: TP 2.0

What is TP 2.0?

- TP stands for Tunneling protocol
- It is present in some ECUs and used for internal communications
- It is a port of the K-LINE protocol (KWP2000)
- It takes place on the top of the CAN protocol (Layer 4)
- Allows to perform transmissions of up to 256 bytes
- It is (again) just a Transport Protocol

CAN bus protocols: TP 2.0

What types of transmissions exist in Tp 2.0?

- Single frame transmissions (5 bytes or less payload length)
- Multiple frame transmissions (more than 5 bytes payload length)

CAN bus protocols: TP 2.0

What kind of frames exist in TP 2.0 transmissions?

- No ACK request, with following frames(0x20)
- ACK request, with following frames(0x00)
- No ACK request, last frame (0x30)
- ACK request, last frame (0x10)
- ACK (0xB0)
- NACK/Repeat transmission (0x90)

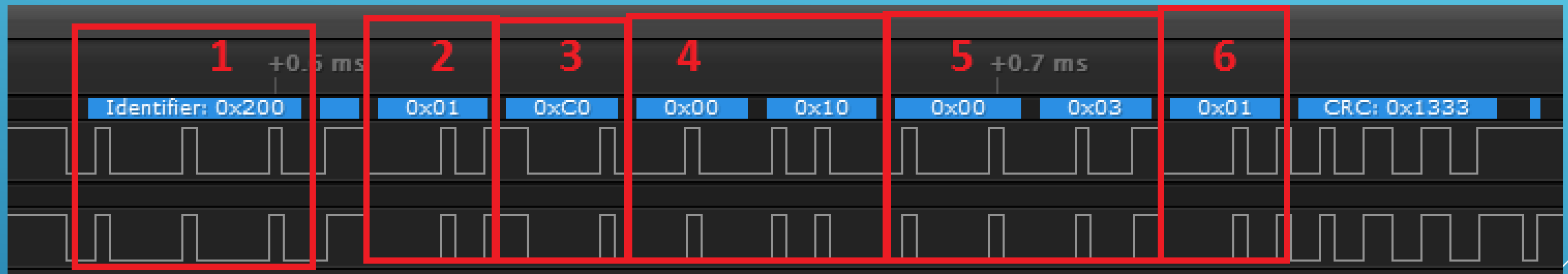
CAN bus protocols: TP 2.0

Channel negotiation

- Before starting communications, a Channel must be established
- Channel request ID is usually 0x200
- Module address is used to indicate target (1 byte)
- Reply ID from target module is formed by "Channel request ID + Target Module address"
- Channel is negotiated for an App
- IDs to be used will be provided by the Server (ECU)
- The negotiated IDs are not always the same

CAN bus protocols: TP 2.0

Channel negotiation: Request



1 - Channel request ID (0x200)

2- Target Module ID (0x01)

3 - Channel Setup request (0xC0)

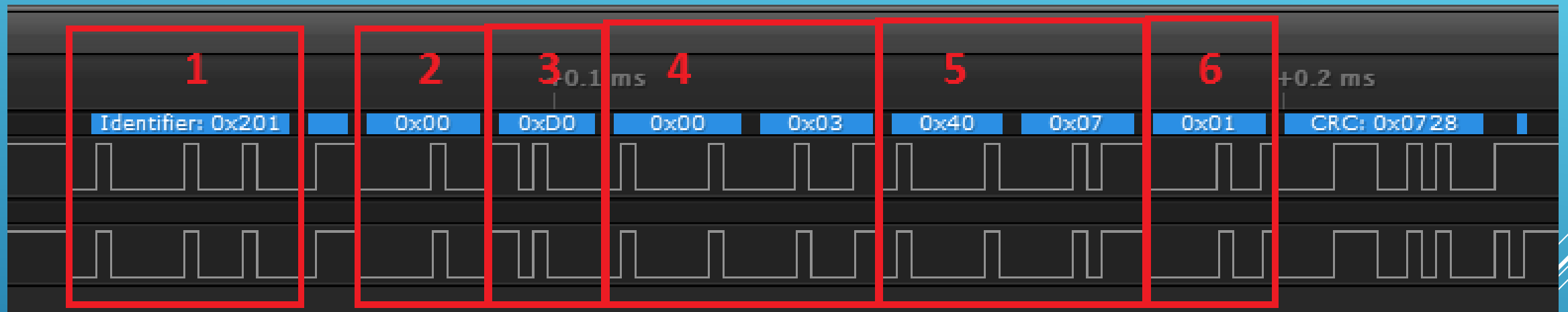
4- Requested Tester Address (0x1000). LSB first

5 - Requested Server Address (0x300). LSB first

6 - App ID (0x01)

CAN bus protocols: TP 2.0

Channel negotiation: Response



1 - Reply ID (0x200 + 0x01)

2 - Tool ID (0x00)

3 - Positive response to Channel Setup request (0xD0)

4 - ID to be used by Server (0x300). LSB first

5 - ID to be used by Tester (0x740). LSB first

6 - App ID (0x01)

CAN bus protocols: TP 2.0

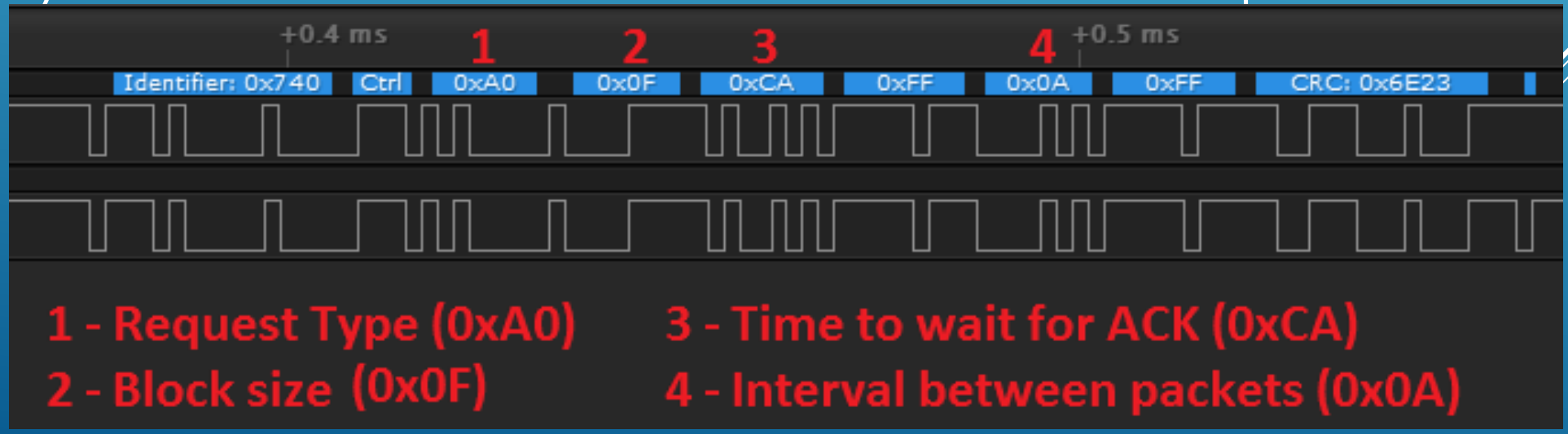
Channel Parameters

- They are negotiated right after the channel is set up
- They are requested ever 500ms – 1000ms to keep the channel alive
- The related requests are:
  - Parameters request (0xA0)
  - Parameters response (0xA1)
  - Channel test (0xA3)
  - Channel break (0xA4)
  - Channel disconnect (0xA8)

CAN bus protocols: TP 2.0

Channel Parameters message structure

- Byte 0 indicates type of request
- Byte 1 indicates Block size (Number of packets to send before expecting an ACK)
- Byte 2 indicates the time to wait for an ACK in ms
- Byte 4 indicates the interval between two packets in ms



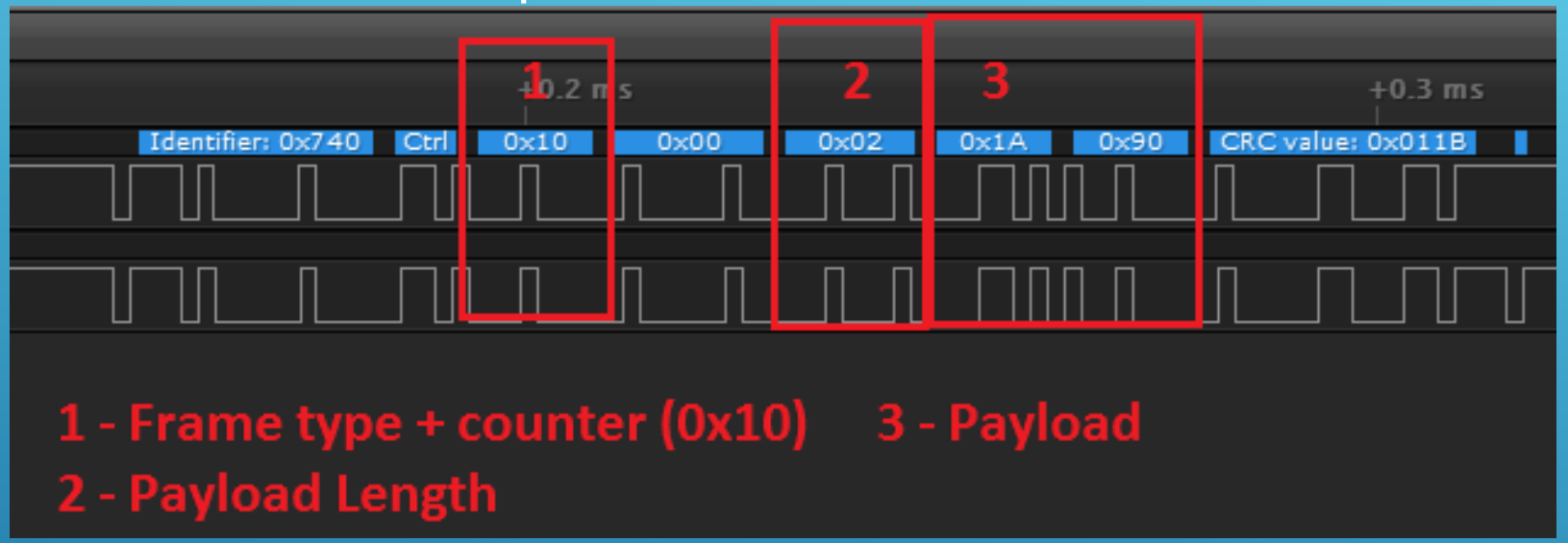
CAN bus protocols: TP 2.0

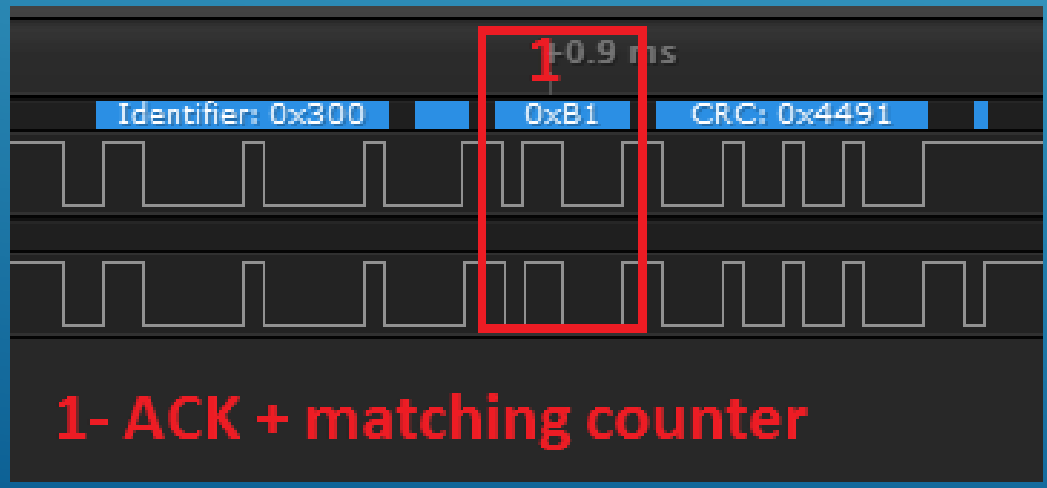
Transmission handling – Frame counters

- Each side of the channel has its own counter.
- Each counter is incremented by one with every frame sent, except for ACKs and NACKs
- ACKs are constructed by adding one to the counter of the other side, plus 0xB0.
- Counters go from 0x00 to 0x0F. Once they reach 0xF0, they are reset back to 0x00.

CAN bus protocols: TP 2.0

TP 2.0 transmission example





CAN bus protocols: SIDs

What is an SID?

- SID stands for Service ID.
- Since the target (ECU) is a Server, it provides Services!
- Most SIDs take parameters.
- There are standard parameters, and OEM specific ones (usually hidden).
- Access to some SIDs require SecurityAccess before requesting them.

CAN bus protocols: SIDs

Fun Facts about SIDs

- SID replies are always formed by (request + 0x40)
- They are the first byte of the payload
- SID 0x7F means “error”, and the reasons are passed as data
- SecurityAccess is required to access some SIDS. It is an authentication that is performed by the tester, to prove the Server that it has legit access to its functions.
- There are absolutely no checks in any data obtained by the Server, but there are some checks performed by the Server for some data sent by the tool.

CAN bus protocols: SIDs

Most interesting SIDs:

- Read Data by ID / Common ID(0x22)
- Write Data by ID / Common ID (0x2E)
- Diagnostics Session control / Start Diag Session (0x10)
- ECU Reset (0x11)
- Security Access (0x27)
- Read Memory by Address (0x23)
- Write Memory by Address (0x3D)
- Request Upload (0x35)
- Routine Control (0x31) and its KWP2000 variants

CAN bus protocols: SIDs

Read Data by ID / Common ID (0x22)

- Great for information gathering
- No SecurityAccess required
- Provides data such as VIN, HW platform, SW version…

Write Data by ID / Common ID (0x2E)

- Allows to modify the Information provided by the Server in the "Read Data by ID" SID.
- Some parameters require SecurityAccess, but not always ☺

CAN bus protocols: SIDs

Diagnostics Session Control (0x22 - UDS)
Start Diagnostics Session (0x10 – TP2.0)

- Allows to start an specific type of Diagnostics Session
- SecurityAccess might be required for some type of Sessions
- Each type of Diagnostics Session enables access to certain SIDs

ECU Reset (0x11)

- Resets the ECU to a Soft state, Hard state, or Ignition toggle state.
- It is specially useful when then ECU becomes blocked or unresponsive.
- It will just JMP 0 or wipe RAM, but will not restore default firmware or any data that is modified in FLASH or EEPROM.

CAN bus protocols: SIDs

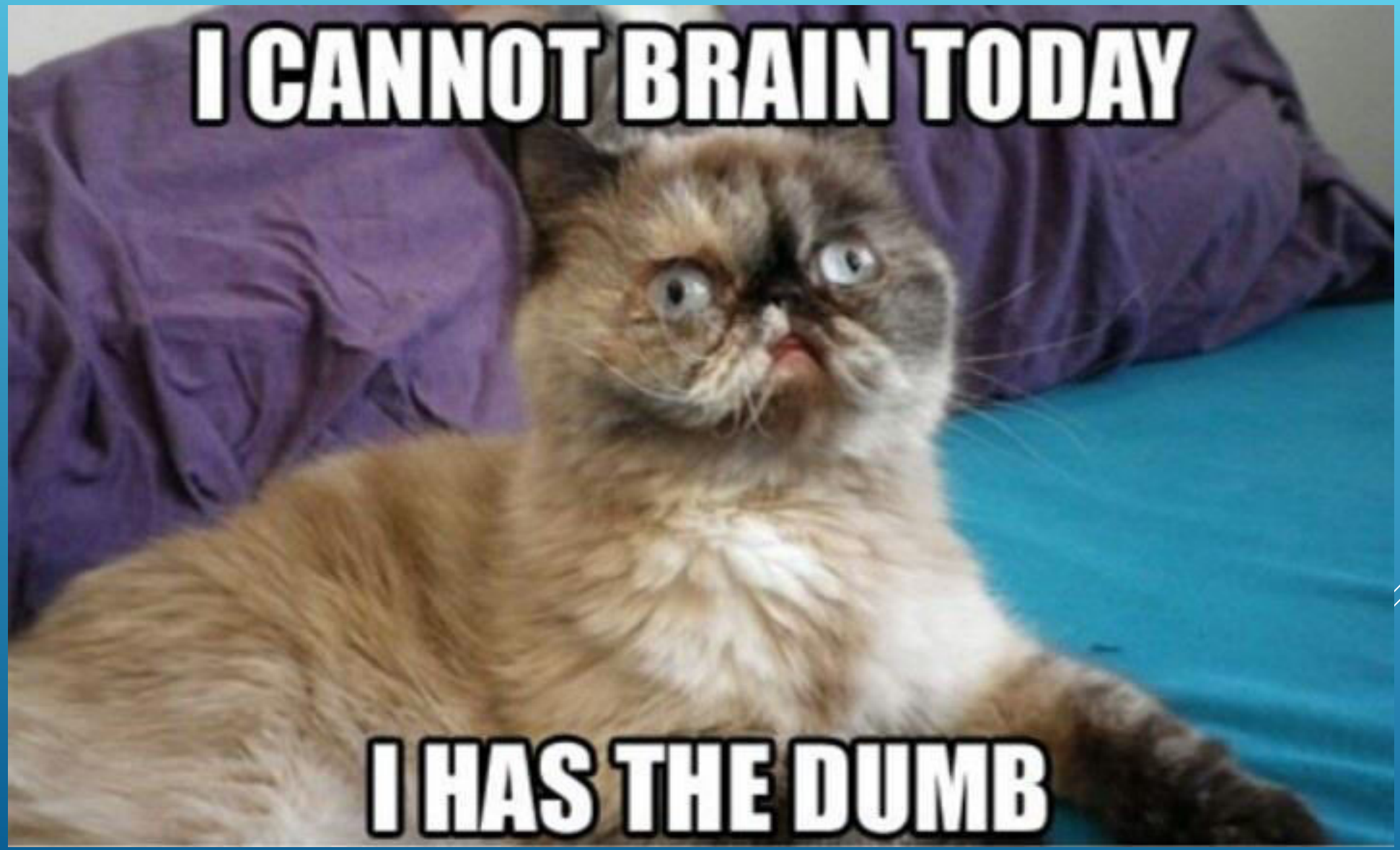
Security Access (0x27)

- Provides access to restricted SIDs if challenge is passed.
- It's based on a challenge-response.
- Bruteforcing is not an option, since the seed is random on every challenge.
- Key length is 4 bytes.
- Algorithm used changes from one ECU to another, an even from same one on different FW versions.

Read Memory by Address(0x23)

- Allows to read memory (Flash, RAM, EEPROM).
- Most of the times, requires SecurityAccess. But not always ;)
- Some ranges are restricted, and will not be readable even if SecurityAccess is provided.

Because randomness



CAN bus protocols: SIDs

Write Memory by Address(0x3D)

- Allows to write memory (Flash, RAM, EEPROM).
- Most of the times, requires SecurityAccess. But not always ;)
- Some ranges are restricted, and will not be writeable even if SecurityAccess is provided.

Request Upload(0x35)

- Pretty much like Read Mem by Address, but with additional options.
- Allows to read memory using encryption and/or compression for transfer

CAN bus protocols: SIDs

Routine Control(0x31)

- Allows to start, stop, and check results for routines
- A routine is either an standard (Erase Flash) or vendor specific (retrieve boot keys, unlock debug port, skip signature checking…) process that is run by the ECU.
- They usually require SecurityAccess, but again, not always.

CANBadger – First things first!

First things you should do:

- Format the SD on your computer (!) to FAT32
- Insert the SD
- Rename your CanBadger, if using Ethernet mode
- Verify that folders are correctly set up after first boot
- Unless you are using power from OBD/DB9 Connectors, make sure you have the12V plugged

CANBadger Features: Standalone mode

Standalone mode does not require a PC.

- MITM mode
- CAN Logger (Standard Mode)
- CAN Logger (Bridge Mode)
- Delete CAN log
- Emulation mode
- Replay mode
- Emulation data generation

CANBadger Features: USB mode

USB mode emulates a CDC device (USB to Serial).

- TP 2.0 Diag Session
  - Diagnostic Session Control
    - Switch to XX Session
    - Switch to Custom Session
    - Scan for available Session types
  - Read Data by ID
    - All data
    - Specific data
    - Custom ID
    - Scan for available IDs

CANBadger Features: USB mode

- TP 2.0 Diag Session
  - Security Access
    - Use known algorithm
      - Level 1
      - Level 3
    - Manual Authentication (currently not implemented)

  - ECU Reset
    - Hard
    - Ignition
    - Soft
    - Custom

CANBadger Features: USB mode

- TP 2.0 Diag Session
  - Memory operations
    - Request Upload
  - Terminate Current Session
- UDS Diag Session
  - Everything is the same, except:
  - Memory operations
    - Read Memory by Address

CANBadger Features: USB mode

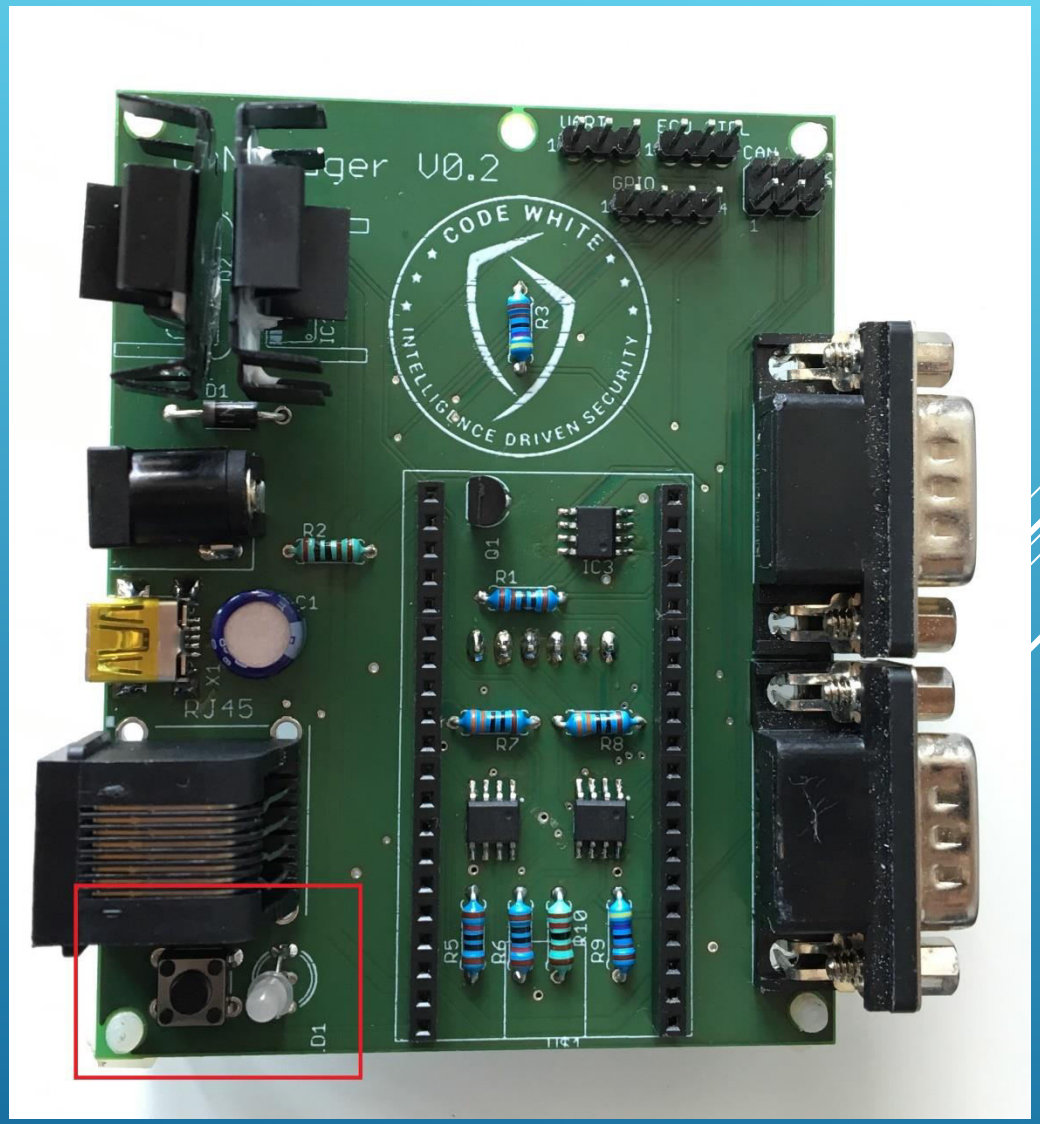
- Scan for TP 2.0 IDs (11-bit)
- Scan for UDS IDs (11-bit)
- Change CAN Speed
- Detect CAN Speed
- Protocol Analysis
  - Scan current traffic for know protocols
  - Logging
    - Log all CAN traffic on CAN1 to SD
    - Log UDS channel traffic on CAN1 to SD
    - Log TP 2.0 channel traffic on CAN1 to SD
    - Log all CAN traffic (Bridge mode) to SD
    - Log UDS channel traffic (Bridge mode) to SD
    - Log TP 2.0 channel traffic (Bridge mode) to SD
    - Manage logs

CANBadger Features: USB mode

- Protocol Analysis
  - Scan current traffic for know protocols
  - Logging
    - Manage logs
      - View log
      - Generate TP 2.0 log from CAN log
      - Generate UDS log from CAN log
      - Scan CAN log for protocols
      - Delete log
- MITM mode
- ECU Power
  - Toggle ECU CTRL power ON/OFF and check input voltage (PSU)

CANBadger Standalone Mode

- Operated by the push button.
- Status indicated by LED.
- Requires SD.
- No PC required



CANBadger Standalone Mode

How to use it?

- Press the button, the LED will blink in red 3 times.
- Press the button the required number of times for the desired action
- The LED will blink each time you press the button
- Once you have selected the module, the LED will blink in green to indicate the selected one.
- After all these blinky show, the LED will remain orange, which means the CANBadger is working on the requested mode.
- If instead of orange, it blinks in red, it means that it was not possible to start the requested module.

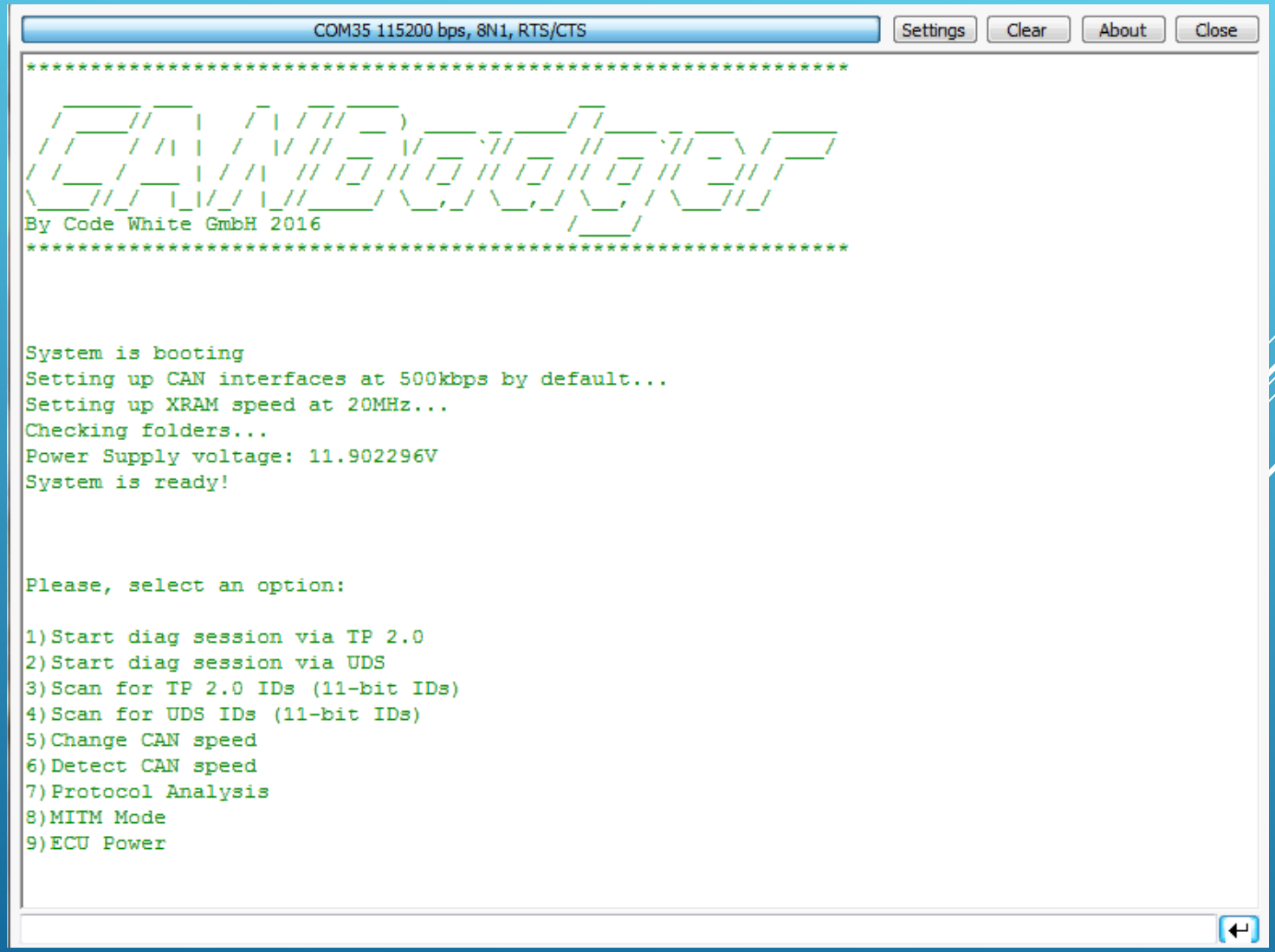
CANBadger Standalone Mode

Available modules (per number of button presses)

- 1)MITM mode – Will load rules from SD
- 2)CAN Logger (Bridge Mode) – Will dump all traffic to SD and bridge comms between CAN1 and CAN2
- 3)CAN Logger (Standard Mode)- Classic CAN logger, dumps all traffic on CAN1 to SD.
- 4)Delete CAN Log – Removes the CAN log from SD
- 5)Emulator mode – Loads emulation data from SD and begins to emulate.
- 6)Replay CAN – Loads specific standalone replay data from SD, and you can guess the rest (on CAN1)
- 7)Generate UDS emulator Data – Self descriptive, on CAN1

CANBadger USB Mode

- Operated via USB
- Emulates a CDC device (USB-Serial)
- Requires SD.
- Extended functionality
- Everything is done by the CANBadger firmware, only a Serial terminal is required.



CANBadger Ethernet Mode

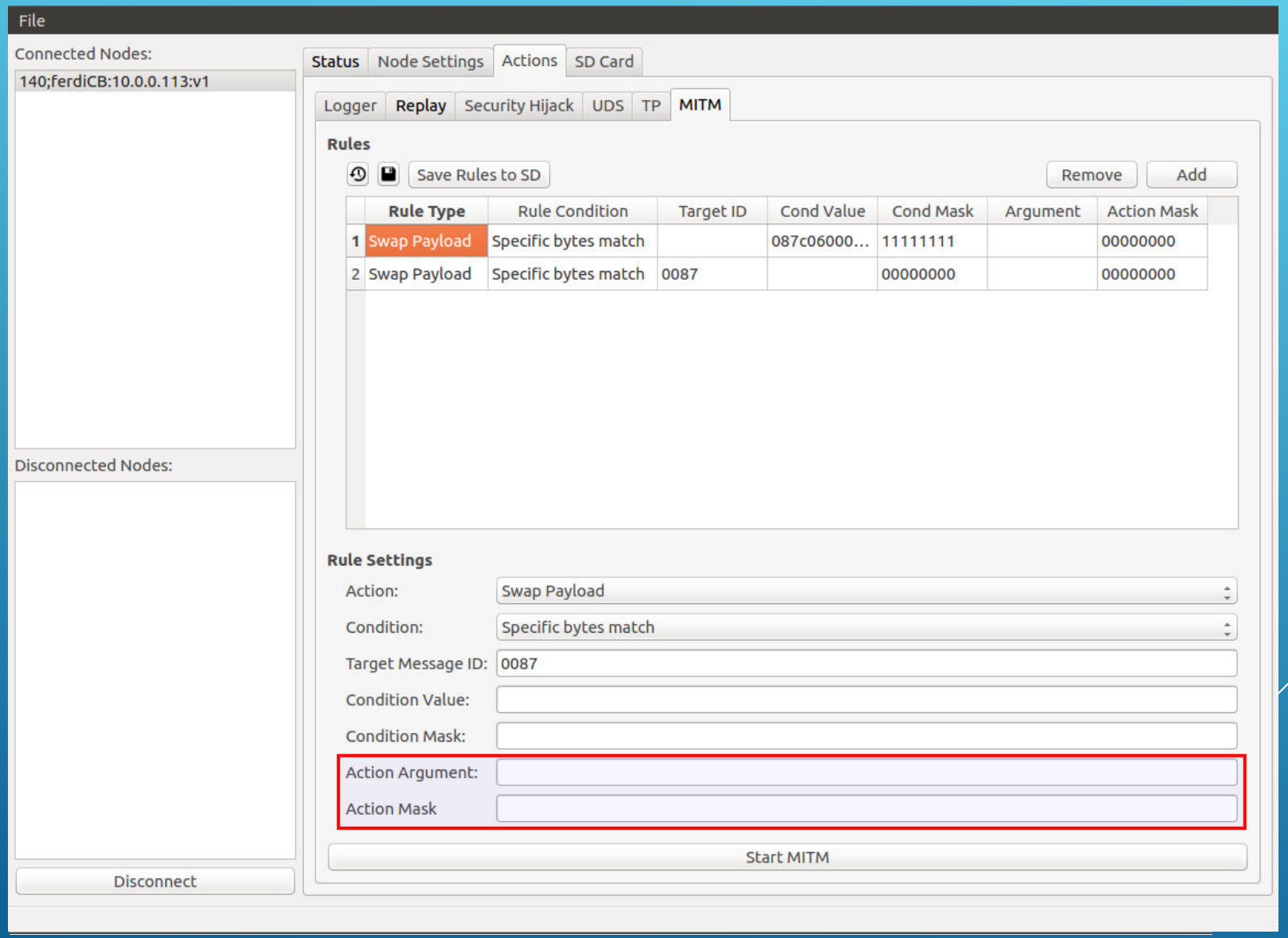
Purpose:

- Make using the CanBadger even easier
- Use multiple CanBadgers in parallel
- Provide programmable access to automate stuff

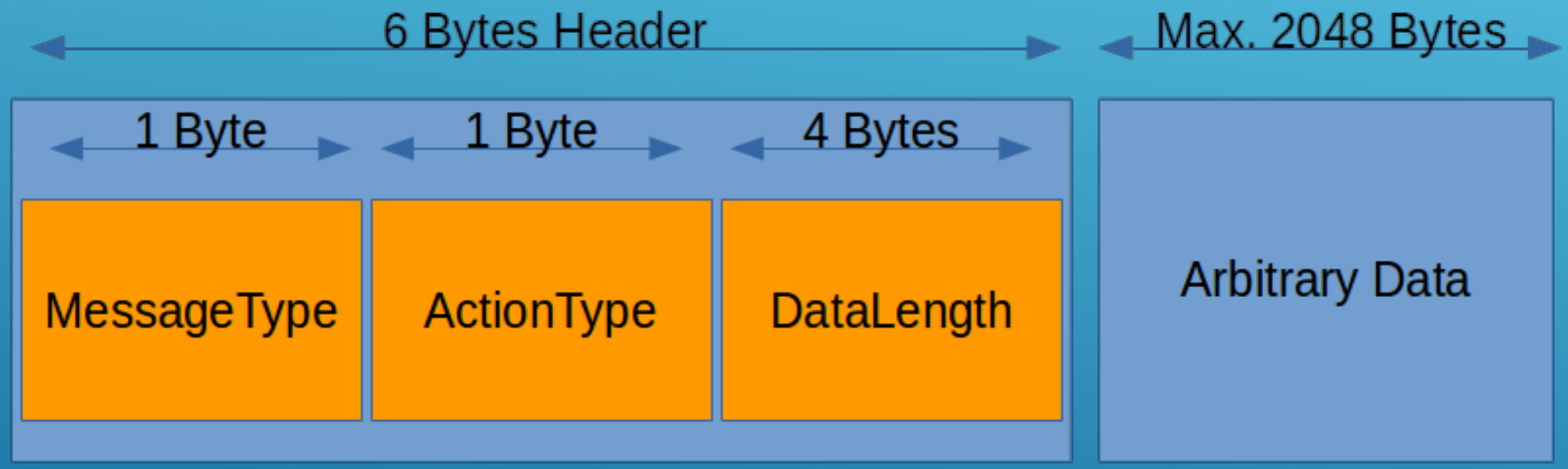
CANBadger Ethernet Mode - GUI

Feature	Available in UART?	Available in GUI?
CAN Logger	☺	☺
TP	☺	☹
UDS	☺	☹
Security Hijack	☺	☹
MITM	☺	☺
Create MITM Rules	☹	☺
Emulator	☺	☹
Read SD Files	☹	☺
Modify CB Settings	☹	☺

CANBadger Ethernet Mode - GUI

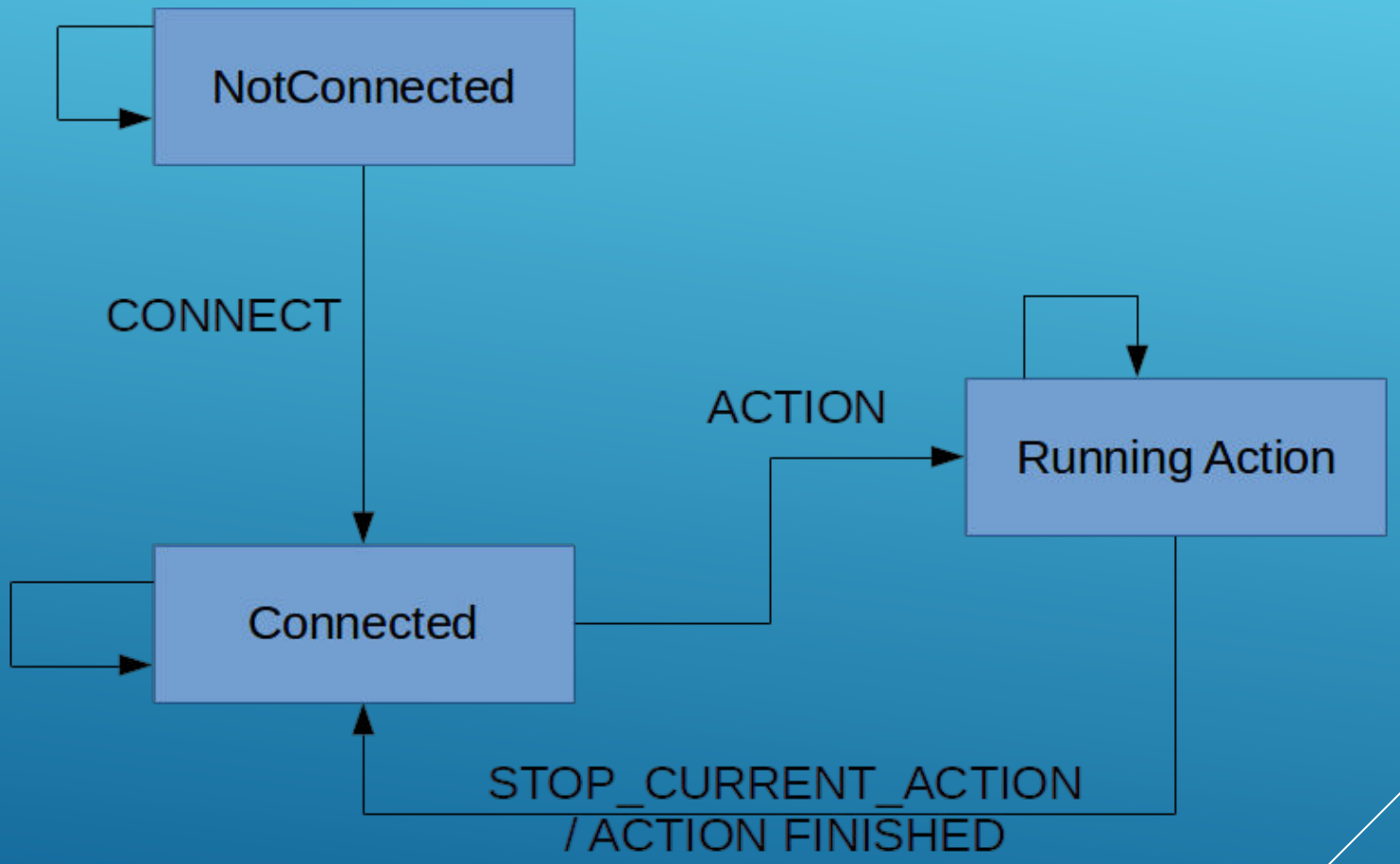


CANBadger Ethernet Mode - Protocol



CANBadger Ethernet Mode - Protocol

States:



CANBadger Ethernet Mode – Rule File Format

B01,380,12,FF,00,F3,00,00,00,00,00,11,22,33,44,55,66,77,88

- /sd/MITM/rules.txt
- ASCII-Hex encoded bytes or double-bytes
- Separated by commas

CANBadger Ethernet Mode – Rule File Format

B01,380,12,FF,00,F3,00,00,00,00,00,11,22,33,44,55,66,77,88

- First byte: Condition mask 10110000 = 0xB0
- Second byte: Condition type = 1

Condition Type	Entire PL Matches	Specific PL Bytes match	Specific PL bytes are greater	Specific PL bytes are less
Value	0x00	0x01	0x02	0x03

CANBadger Ethernet Mode – Rule File Format

B01,380,12,FF,00,F3,00,00,00,00,00,11,22,33,44,55,66,77,88

- Target msg ID is mandatory
  - Rules are stored in the XRAM in a key-value matter
  - The ID is the lookup key

CANBadger Ethernet Mode – Rule File Format

B01,380,12,FF,00,F3,00,00,00,00,00,11,22,33,44,55,66,77,88

- Condition Payload Data
  - Mandatory, as condition should match payload too
  - Can be all 0, for example when mask is 0

CANBadger Ethernet Mode – Rule File Format

B01,380,12,FF,00,F3,00,00,00,00,00,11,22,33,44,55,66,77,88

- First Byte: Action mask
- Second Byte: Action type

Action Type	Swap PL	Swap specific bytes	Add fixed Value to specific bytes	Substract fixed value to specific bytes	Multiply specific bytes	Divide specific bytes	Increase specific bytes by %	Decrease by %	Drop
Value	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08

CANBadger Ethernet Mode – Rule File Format

B01,380,12,FF,00,F3,00,00,00,00,00,11,22,33,44,55,66,77,88

- Action Payload
  - Examples:
    - The payload to swap
    - A percentage

    (in that case the following bytes are 0)

CANBadger Ethernet Mode - Code

List available Nodes:

```
import socket

UDP_IP = "0.0.0.0"  # bind on all interfaces
UDP_PORT = 13370

sock = socket.socket(socket.AF_INET,# Internet
                     socket.SOCK_DGRAM)# UDP
sock.bind((UDP_IP, UDP_PORT))

nodes = []
while True:
    data, addr = sock.recvfrom(256)
    data_split = data.split("|")
    id = data_split[1]
    version = data_split[2]
    node = [addr, id, version]
    if node not in nodes:
        nodes.append(node)
        print "discovered new node with ip=%s, id=%s, version=%s" % (addr[0], id, version)
```

CANBadger Ethernet Mode - Code

Replay a frame:

```
import socket
import struct
import time

from ethernet_message import *

NODE_IP = "10.0.0.113"
PORT = 13371
tport = 15555  # we will get msgs from cb after connect here
sock = socket.socket(socket.AF_INET,  # Internet
                     socket.SOCK_DGRAM) # UDP
sock.bind(('0.0.0.0', tport))
sock.sendto(b'\x04' + b'\x00' + struct.pack('<I', 4) + struct.pack('<I', tport) + b'\x00',
            (NODE_IP, PORT))
# receive ack
ack = sock.recvfrom(128)
# enable replay mode
sock.sendto(EthernetMessage("ACTION", "START_REPLAY", 0, '').serialize(), (NODE_IP, PORT))
time.sleep(1)  # sleep a second so we can see the led changing
# send a few frames
sock.sendto(EthernetMessage("ACTION", "REPLAY", 12, '00112233445566778800cafe').serialize(), (NODE_IP, PORT))
sock.sendto(EthernetMessage("ACTION", "REPLAY", 12, 'cafe1122334455667889900').serialize(), (NODE_IP, PORT))
# stop replay mode
time.sleep(0.1)
sock.sendto(EthernetMessage("ACTION", "STOP_CURRENT_ACTION", 0, '').serialize(), (NODE_IP, PORT))
```

Hacking Instructions

- Main IDE we used for firmware is LPCXpresso
  - Has code limit in free version
- For quick modifications you can also use the mbed online compiler
  - Export to Keil/IAR/CooCox is possible

Hacking Instructions

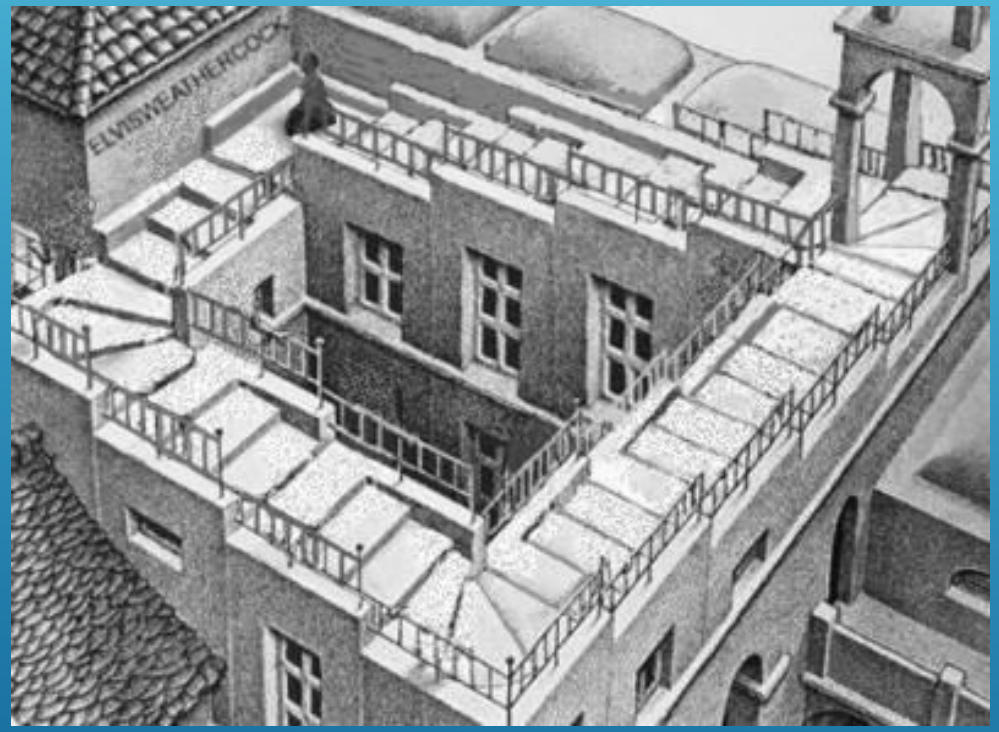
- Code will be published on Github shortly after DC
- You are encouraged to add Issues for:
  - Feature Requests
  - Bugs
  - Enhancements

Development aftermath

Trying to find the cause of memory corruptions:

Development aftermath

Merging code after a long time of implementing new features:



Development aftermath

- ~13000 LoCs Firmware, excluding libraries
- ~2200 LoCs GUI Code, excluding libraries and config files
- 190 commits during the last 100 days for firmware
- 95 commits during the last 100 days for GUI
- Most commits were made on Saturdays and Sundays

Development aftermath

- Firmware has size of 330k, unoptimized
- Flashing that takes long (~30 seconds)
- No automated tests lol
- Firmware needs refactoring now

Thanks!

- To all of you for being here today!
- To Code White for their support and trust in the project
- To our family and friends for supporting us even when we run out of coffee

Javier Vazquez
@fjvva
Javier.Vazquez.vidal@code-white.com

Ferdinand Nölscher
@s1ckcc
Ferdinand.noelscher@code-white.com