SHOWCASE

Showcase

CDE100 device can translate ETH/CAN messages via two different protocols. It is a set of values that are send/received via TCP socket on port 3333.
Data can be translated via BINARY or ASCI protocol, for detailed information please seek official documentation

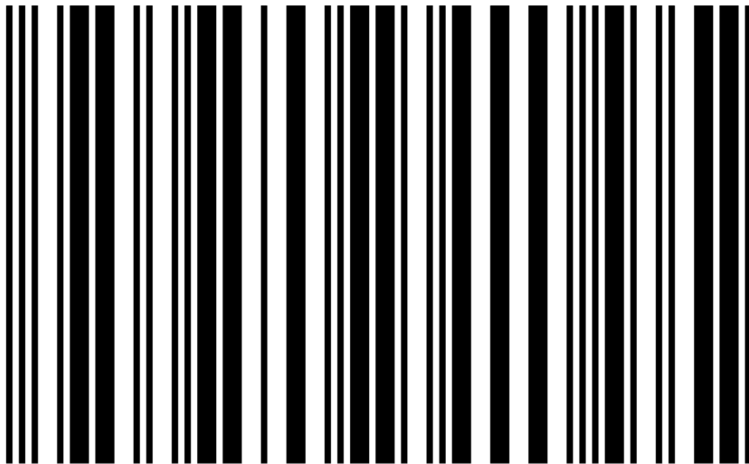
Barcodes

Application developed supports only barocdes with numbers, which means that we can user **EAN**, **UPC** or **ITF** type barcodes

Below you can find barcode numbers that application will demonstrate.

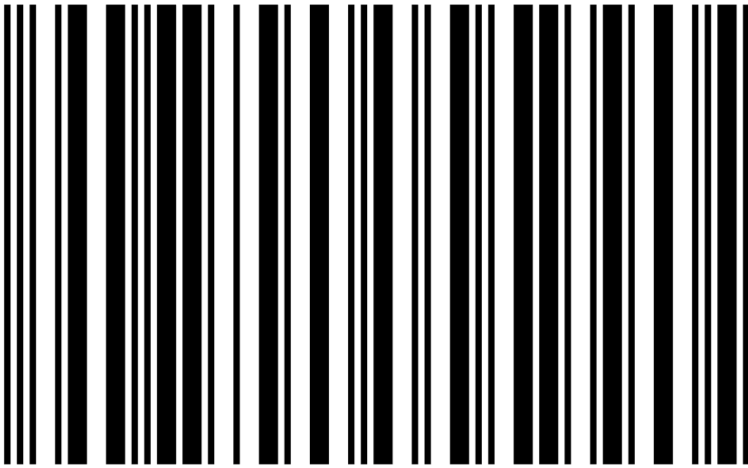
ITF

87123123319



19245672347

56756856856



EAN - 8

69244063

ASCII protocol

In this section we will demonstrate how the app works with ascii protocol.

Sending code to CDE

```
Barcode scanned: 87123123319
Barcode sent to CDE100: 32,S,8,0000001448F15C77

Barcode scanned: 19245672347
Barcode sent to CDE100: 32,S,8,000000047B21A79B

Barcode scanned: 56756856856
```

```
Barcode sent to CDE100: 32,S,8,0000000D36F8E818

Barcode scanned: 69244063
Barcode sent to CDE100: 32,S,8,000000000420949F
```

> ***NOTE:*** CDE100 then receives and converts the ethernet frame to CAN frame and sends it to TDC-E

Explaining the AppSpace application

Code example can be found inide this project.
`sendingQRcode_ASCII` appspace app is used for simulating the Barcode and sending it to another app called `recQRcode_ASCII` which is used to format the Barcode as Ethernet frame.

Protocol Description

Protocol is divided into four (in some cases three) segments, separated with a comma character (ASCII: 0x2C) and terminated with new-line character (ASCII: 0x0A)
ETH/CAN message segments are:

- CAN ID - String-encoded, decimal value
- Range: 0 -> 2^29
- CAN ID type - Single-character
- Range: S→ Standard type, E → Extended type
- CAN DLC - String-encoded, decimal value
- Range: 0 → 8 (If 0, data part must be omitted)
- Data - (Optional) String-encoded, hexadecimal value(s)
- Range: 00 → FF (per byte)

Formating Barcode to Ethernet Frame

This section will explain the most important part of creating a proper ethernet frame. Only snippet of sample code will be explaind

Inside `main.lua` script of `recQRcode_ASCII` application you can find the example code.

```lua
local function createCANMessage(code)
-- random ID
local canID = 0x20
-- Standard CAN ID type
local canIDType = 'S'
-- number of data, max size is 8 bytes
local canDLC = 8

-- Base message structure
```

```
local message = tostring(canID) .. ',' .. canIDType .. ',' ..
tostring(canDLC) .. ','

-- Formatting code as hex with padding
local code_str = string.format('%02X', tostring(code))
local padding_length = 16 - #code_str
local padded_msg = string.rep('0', padding_length) .. code_str

-- Complete message
return message .. padded_msg .. '\n'
```

This Lua function createCANMessage(code) generates a CAN message string. It starts by setting a fixed CAN ID (0x20), a message type ('S' for standard), and a data length of 8 bytes. The function then formats the code parameter as a 2-digit hexadecimal string, pads it with leading zeros if necessary, and appends it to the message. The output is a message string that includes the CAN ID, type, data length, and the padded hexadecimal code, followed by a newline character.

Receiving code on TDC-E

To show received CAN messages use `candump can1` or `candump can0` depending on which CAN interface you use on TDC-E.

Candump output

```
<0x020> [8] 00 00 00 14 48 f1 5c 77
<0x020> [8] 00 00 00 04 7b 21 a7 9b
<0x020> [8] 00 00 00 0d 36 f8 e8 18
<0x020> [8] 00 00 00 00 04 20 94 9f
```

Candump is used here so we really confirm that the CAN message really came to TDC-E device.

Another way to confirm that messages are recieved on CDE100 is through WEB user interface on ip address of CDE 100. As you can see on image below.

As long as frames are beeing received and transmitted, you are good to go. If frames are not received

or they are going to `Frames Transmit Errors`, then you should have to check how ethernet frame is formated. Ethernet frame formating will be showcased down below.

If we want to manipulate the CAN frame, to recreate Barcode from it, this can be done by simple Go application.

Inside the folder `go_CDE100_Ascii` you can find all `main.go` and `initial-setup.sh`, make sure shell script has executable mod.

Go application output

```
Received CAN message: 0000001448f15c77
Scanned QR number: 87123123319

Received CAN message: 000000047b21a79b
Scanned QR number: 19245672347

Received CAN message: 0000000d36f8e818
Scanned QR number: 56756856856

Received CAN message: 000000000420949f
Scanned QR number: 69244063
```

Explaining the Go application

Code example can be found inide this project.
`go_CDE100_Ascii` golang app is used for formatting CAN message into Barcode.

Formatting received CAN message

This section will explain the most important part of formatting a CAN message into real Barcode.

Inside `main.go` script of `go_CDE100_Ascii` application you can find the example code.

```go
func convertBytesToInt(frame []byte) uint64 {
	hexString := fmt.Sprintf("%x", frame)
	fmt.Println("Received CAN message:", hexString)

	parsedInt, err := hex.DecodeString(hexString)
	if err != nil {
		log.Fatalf("Error parsing hexadecimal string: %v", err)
	}
	var number uint64
	for i := 0; i < len(parsedInt); i++ {
		number = number*256 + uint64(parsedInt[i])
	}
```

```
    return number
}
```

This Go function convertBytesToInt takes a byte slice frame as input and converts it into a 64-bit unsigned integer. First, it converts the byte slice into a hexadecimal string using fmt.Sprintf("%x", frame) and prints it. Then, it decodes the hexadecimal string back into bytes using hex.DecodeString(). After that, it processes the byte array by iterating through it and constructing the corresponding integer value, multiplying the current result by 256 and adding each byte's value. Finally, it returns the computed 64-bit integer, which is Barcode in our case.

Binary protocol

In this section we will demonstrate how the app works with binary protocol.

Sending code to CDE

```
Barcode scanned: 87123123319
Barcode sent to CDE100: \x01\x00\x00\x00\x14\x00\x08w\\xF1H����

Barcode scanned: 19245672347
Barcode sent to CDE100: ����������!{����

Barcode scanned: 56756856856
Barcode sent to CDE100: \x01\x00\x00\x00\x14\x00\x08\x18\xE8\xF86���

Barcode scanned: 69244063
Barcode sent to CDE100: ���������� ������
```

> ***NOTE:*** Signs like:��, are representing binary message.

> ***NOTE:*** CDE100 then receives and converts the ethernet frame to CAN frame and sends it to TDC-E

Explaining the AppSpace application

Code example can be found inide this project.
`sendingQRcode_BINARY` appspace app is used for simulating the Barcode code and sending it to another app called `recQRcode_BINARY` which is used to format the Barcode as Ethernet frame.

Protocol Description

Protocol is a series of small CAN message structures lead by message count value

ETH/CAN message structure is:

- CAN ID - 4 byte binary value (Network byte order)
  - Range: 0 → 2^29

- CAN ID type - 1 byte binary value
  - Range: 0 → Standard, 1 → Extended 2 → Heartbeat
- CAN DLC - 1 byte binary value
  - Range: 0 → 8
- Data - 8x 1 byte binary value
  - Range: 0 → 255 (per byte)
  - Unused values must be padded with zeros.

Formating Barcode to Ethernet Frame

This section will explain the most important part of creating a proper ethernet frame. Only snippet of sample code will be explaind

Inside `main.lua` script of `recQRcode_BINARY` application you can find the example code.

```
local function parseCodeToData(code)
local data = {}
local i = 1
while i <= 8 do
  data[i] = code % 256
  code = math.floor(code / 256)
  i = i + 1
end
return data
end

local function parseCanID(canID)
local canIDBytes = {}
local j = 1
while j <= 4 do
  canIDBytes[j] = math.floor(canID / 256^(4 - j)) % 256
  j = j + 1
end
return canIDBytes
end
```

The parseCodeToData function converts a given number (Barcode) into an array of 8 bytes, where each byte is obtained by repeatedly dividing the number by 256 and extracting the remainder.

The parseCanID function converts a CAN ID (a 32-bit value) into an array of 4 bytes, extracting each byte by shifting and masking the value according to the CAN ID's byte positions. Both functions break down a larger numeric value into its byte-wise components for further processing.

Why are we dividing the number by 256 and extracting the remainder?

A byte consists of 8 bits, and 256 is 2^8, which is the number of possible values a byte can hold (ranging from 0 to 255).
By dividing the number by 256, we shift the digits and reduce the number, effectively isolating each byte. This allows us to extract the least significant byte by taking the remainder (% 256), and then reduce the number by dividing by 256 to extract the next byte.

For example, if the number is 1234, dividing by 256 helps separate it into two parts:

The least significant byte is found by 1234 % 256 = 210.
The next byte is obtained by dividing 1234 // 256 = 4.

By iteratively dividing the number by 256, we move through each byte in the number, starting with the least significant byte and proceeding to the more significant bytes.

Receiving code on TDC-E

To show received CAN messages use `candump can1` or `candump can0` depending on which CAN interface you use on TDC-E.

Candump output

```
<0x014> [8] 77 5c f1 48 14 00 00 00
<0x014> [8] 9b a7 21 7b 04 00 00 00
<0x014> [8] 18 e8 f8 36 0d 00 00 00
<0x014> [8] 9f 94 20 04 00 00 00 00
```

Candump is used here so we really confirm that the CAN message really came to TDC-E device.

Another way to confirm that messages are recieved on CDE100 is through WEB user interface on ip address of CDE 100. As you can see on image below.

As long as frames are beeing received and transmitted, you are good to go. If frames are not received or they are going to `Frames Transmit Errors`, then you should have to check how ethernet frame is formated. Ethernet frame formating will be showcased down below.

If we want to manipulate the CAN frame, to recreate Barcode from it, this can be done by simple Go application.

Inside the folder `go_CDE100_Binary` you can find all `main.go` and `initial-setup.sh`, make sure shell script has executable mod.

Go application output

```
Received CAN message: 119 92 241 72 20 0 0 0
Scanned QR number is: 87123123319

Received CAN message: 155 167 33 123 4 0 0 0
Scanned QR number is: 19245672347

Received CAN message: 24 232 248 54 13 0 0 0
Scanned QR number is: 56756856856

Received CAN message: 159 148 32 4 0 0 0 0
Scanned QR number is: 69244063
```

Explaining the Go application

Code example can be found inide this project.
`go_CDE100_Binary` golang app is used for formatting CAN message into Barcode.

Formatting received CAN message

This section will explain the most important part of formatting a CAN message into realBarcode.

Inside `main.go` script of `go_CDE100_Binary` application you can find the example code.

```
    for {
        frame, err := canBus.Recv()
        if err != nil {
            log.Fatalf("Can not recv frame: %+v", err)
        }

        var x uint64 = 0
        fmt.Print("Received CAN message: ")
        for i := 0; i < len(frame.Data); i++ {

            fmt.Print(frame.Data[i], " ")
            //right byte is most important
            x += uint64(frame.Data[i]) * uint64(math.Pow(256, float64(i)))

        }
```

```
        print("\n")
        fmt.Println("Scanned QR number is:", x)
        print("\n")
    }
```

This Go code snippet continuously receives CAN bus messages in a loop, extracting the data from each message. It then processes each byte of the message's data by multiplying it with powers of 256 (based on its position in the data array) and accumulating the result into a 64-bit unsigned integer x. After processing the data, it prints out the received CAN message and the calculated value of x, which represents a decoded Barcode.