CAN Data Logger for ESP32-C6 Pico

Complete Technical Documentation

Version: 1.0

Date: 2024

Author: CAN Data Logger Development Team

Table of Contents

Executive Summary

The CAN Data Logger for ESP32-C6 Pico is a comprehensive data logging system designed for capturing, encrypting, and storing CAN bus messages. The system features:

- **Real-time CAN Bus Monitoring**: Logs data from at least 10 CAN IDs simultaneously
- **Encrypted Storage**: AES-128-CBC encryption ensures data security

- **Accurate Timestamps**: DS3231 RTC module with NTP synchronization
- **Web-Based Interface**: Access logged data via WiFi Access Point
- **Professional Decoder**: Python-based GUI application with DBC support and multiple export formats
- **Robust Design**: Auto-recovery, error handling, and status indicators

This documentation provides complete technical details, installation instructions, and usage guidelines for the system.

Introduction

Purpose

The CAN Data Logger is designed for automotive and industrial applications requiring secure, timestamped logging of CAN bus communications. It provides a complete solution from data capture to analysis.

Key Features

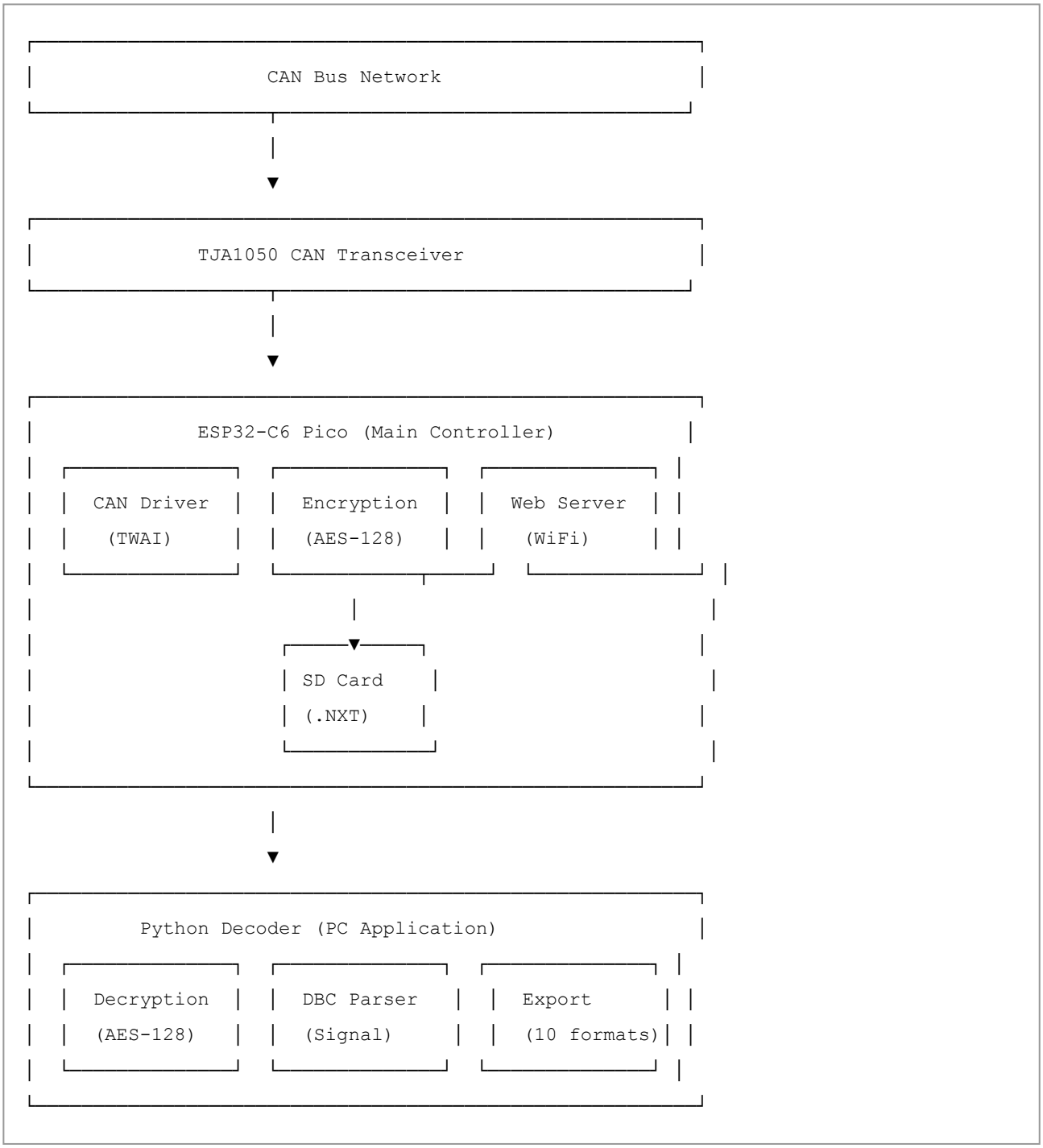
- **Multi-ID Tracking**: Simultaneously logs messages from 10+ CAN IDs
- **Secure Storage**: AES-128-CBC encryption with unique IV per frame
- **Time Accuracy**: RTC with NTP synchronization for precise timestamps
- **Remote Access**: WiFi Access Point for wireless file access
- **Data Analysis**: Professional decoder with DBC support and visualization
- **Export Formats**: 10+ export formats for data analysis

Target Applications

- Automotive diagnostics and testing
- Motor performance monitoring
- CAN bus network analysis
- Data logging for research and development
- Quality assurance and validation

System Overview

Architecture



Data Flow

1. **Capture**: CAN messages received via TJA1050 transceiver
2. **Filter**: Only tracked CAN IDs are processed
3. **Timestamp**: RTC provides accurate timestamps
4. **Encrypt**: AES-128-CBC encryption with unique IV
5. **Store**: Encrypted data written to SD card as .NXT files
6. **Access**: Files downloadable via web interface
7. **Decode**: Python decoder decrypts and analyzes data

Hardware Components

ESP32-C6 Pico

Image Placeholder: [ESP32-C6 Pico Board Image]

The ESP32-C6 Pico is the main controller, featuring:

- **Processor**: RISC-V 32-bit single-core CPU at 160 MHz
- **WiFi**: 802.11 b/g/n (2.4 GHz)
- **Memory**: 320 KB SRAM, 512 KB ROM
- **GPIO**: 30 programmable pins
- **USB**: Native USB support with CDC

Pinout Diagram Placeholder: [ESP32-C6 Pico Pinout Diagram]

Pin Configuration

Component	Pin	Function
CAN TX	5	CAN Transmit
CAN RX	4	CAN Receive
SD CS	20	SD Card Chip Select
SD MOSI	19	SD Card SPI Data Out
SD MISO	18	SD Card SPI Data In
SD SCK	21	SD Card SPI Clock
RTC SDA	6	RTC I2C Data
RTC SCL	7	RTC I2C Clock
Neopixel	8	LED Status Indicator

TJA1050 CAN Transceiver

Image Placeholder: [TJA1050 CAN Transceiver Module Image]

- **Function**: Converts CAN bus differential signals to/from digital
- **Speed**: Supports up to 1 Mbps
- **Voltage**: 5V operation
- **Features**: Silent mode, high-speed operation

Connection:

- TXD → ESP32 Pin 5
- RXD → ESP32 Pin 4
- VCC → 5V
- GND → GND
- CANH/CANL → CAN Bus

DS3231 RTC Module

Image Placeholder: [DS3231 RTC Module Image]

- **Function**: Real-time clock with battery backup
- **Interface**: I2C (400 kHz)
- **Accuracy**: ±2 ppm from -40°C to +85°C
- **Battery**: CR2032 coin cell backup
- **Features**: Temperature compensation, alarm functions

Connection:

- SDA → ESP32 Pin 6
- SCL → ESP32 Pin 7
- VCC → 3.3V or 5V
- GND → GND

SD Card Module

Image Placeholder: [SD Card Module Image]

- **Interface**: SPI
- **Format**: FAT32
- **Capacity**: 2GB - 32GB recommended
- **Speed**: 4 MHz SPI clock
- **Features**: Auto-detection, write protection support

Connection:

- CS → ESP32 Pin 20
- MOSI → ESP32 Pin 19
- MISO → ESP32 Pin 18
- SCK → ESP32 Pin 21
- VCC → 5V
- GND → GND

Neopixel LED

Image Placeholder: [Neopixel LED Image]

- **Type**: WS2812B RGB LED
- **Function**: Visual status indication
- **Connection**: Pin 8, 5V, GND

Software Architecture

Firmware Structure

The firmware (`CAN_Data_Logger_Only.ino`) is organized into functional modules:

1. Initialization Module

- Serial communication setup
- LED initialization
- RTC initialization and verification
- WiFi AP/STA configuration
- SD card detection and formatting
- CAN bus driver initialization

2. CAN Processing Module

- Message reception via TWAI driver
- CAN ID filtering (tracked IDs)
- Frame structure creation
- Timestamp assignment

3. Encryption Module

- AES-128-CBC encryption using mbedtls
- Random IV generation per frame
- PKCS7 padding
- Frame encryption before storage

4. Storage Module

- File creation with timestamp naming
- Header writing (magic, version, start time)
- Encrypted frame writing
- File size management (10 MB max)
- Auto-rollover to new files

5. Web Server Module

- HTTP server on port 80
- Root page with status dashboard
- File listing endpoint
- File download endpoint
- Live data streaming endpoint

6. Time Management Module

- RTC time reading
- NTP synchronization (when WiFi STA connected)

- Timestamp generation
- Time formatting for display

Key Functions

CAN Initialization:

```
bool initCAN()
```

- Configures TWAI driver
- Sets CAN speed (500 kbps default)
- Applies filter (accept all, filter in software)
- Starts CAN driver

Encryption:

```
bool encryptFrame(const uint8_t* plaintext, size_t plaintextLen,
                  uint8_t* iv, uint8_t* ciphertext)
```

- Generates random 16-byte IV
- Pads plaintext to 16-byte boundary
- Encrypts using AES-128-CBC
- Returns IV and ciphertext

File Writing:

```
bool writeCANMessage(const CanFrame& frame, time_t timestamp, uint32_t micros)
```

- Prepares 20-byte frame structure
- Encrypts frame
- Writes IV (16 bytes) + Length (1 byte) + Ciphertext (32 bytes)
- Manages file size and rollover

File Format Specification (.NXT)

Overview

The .NXT file format is a binary format designed for encrypted CAN data storage. It uses the CAND (CAN Data) binary format internally but is saved with the .NXT extension for compatibility with the decoder.

File Structure

```
[Header: 9 bytes][Frame 1: 49 bytes][Frame 2: 49 bytes]...[Frame N: 49 bytes]
```

Header Format (9 bytes)

Offset	Size	Field	Description	Value
0x00	4 bytes	Magic	File identifier	0x43414E44 ("CAND")
0x04	1 byte	Version	Format version	0x01
0x05	4 bytes	Start Time	Unix timestamp (little-endian)	uint32_t

Example Header:

```
Magic:      43 41 4E 44  (CAND in ASCII)
Version:    01
Start Time: E4 07 01 00  (Unix timestamp, little-endian)
```

Frame Format (49 bytes per frame)

Offset	Size	Field	Description
0x00	16 bytes	IV	Initialization Vector for AES encryption
0x10	1 byte	Length	Decrypted data length (always 20)
0x11	32 bytes	Ciphertext	Encrypted CAN frame data

Decrypted Frame Data (20 bytes)

After decryption, each frame contains:

Offset	Size	Field	Description	Format
0x00	4 bytes	CAN ID	Message identifier	Little-endian uint32_t
0x04	1 byte	DLC	Data Length Code (0-8)	uint8_t
0x05	8 bytes	Data	CAN message data	uint8_t array
0x0D	4 bytes	Timestamp	Unix timestamp	Little-endian uint32_t
0x11	2 bytes	Sequence	Frame sequence number	Little-endian uint16_t
0x13	1 byte	Padding	Reserved/padding	uint8_t (always 0)

File Naming Convention

Files are named using the format:

```
CAN_LOG_YYYYMMDD_HHMMSS.NXT
```

Example:

```
CAN_LOG_20240115_143022.NXT
```

Represents a file created on January 15, 2024 at 14:30:22.

File Size Limits

- **Maximum File Size**: 10 MB (10,485,760 bytes)
- **Auto-Rollover**: New file created when limit reached
- **Frame Size**: 49 bytes per frame
- **Maximum Frames per File**: ~214,000 frames (theoretical)

Encryption System

Algorithm

- **Cipher**: AES-128-CBC (Advanced Encryption Standard)
- **Key Size**: 128 bits (16 bytes)
- **Block Size**: 128 bits (16 bytes)
- **IV**: Random 16 bytes per frame
- **Padding**: PKCS7

Encryption Key

The encryption key is **16 bytes** and must match exactly in both firmware and decoder:

```
const uint8_t ENCRYPTION_KEY[16] = {
    0x23, 0xF8, 0xF4, 0x96, 0xBC, 0x67, 0x49, 0x0F,
    0xBD, 0x9D, 0xCE, 0x01, 0xB1, 0x6E, 0xE0, 0x6F
};
```

⚠ **IMPORTANT:** This key must match exactly in both firmware and decoder!

Encryption Process

1. **Prepare Frame Data**: Create 20-byte frame structure (CAN ID + DLC + Data + Timestamp + Sequence + Padding)
2. **Pad Data**: Pad to 16-byte boundary using PKCS7 (results in 32 bytes)
3. **Generate IV**: Create random 16-byte IV using ESP32's `esp_random()`
4. **Encrypt**: Use AES-128-CBC with key and IV
5. **Write to File**: Write IV (16 bytes) + Length (1 byte) + Ciphertext (32 bytes)

Decryption Process

1. **Read IV**: Read 16 bytes from file
2. **Read Length**: Read 1 byte (should be 20)
3. **Read Ciphertext**: Read 32 bytes from file
4. **Decrypt**: Use AES-128-CBC with key and IV
5. **Remove Padding**: Remove PKCS7 padding
6. **Parse Data**: Extract CAN ID, DLC, data, timestamp, sequence

Security Features

- **Unique IV per Frame**: Each frame uses a random IV, preventing pattern analysis
- **AES-128-CBC**: Industry-standard encryption algorithm
- **Key Protection**: Key is embedded in firmware (consider additional security for production)
- **No Plaintext Storage**: All data is encrypted before writing to SD card

---

CAN Bus Configuration

CAN Speed

Default speed: **500 kbps**

The CAN speed is configured in the `initCAN()` function:

```
twai_timing_config_t t_config = TWAI_TIMING_CONFIG_500KBITS();
```

Available Speeds:

- 25 kbps: `TWAI_TIMING_CONFIG_25KBITS()`
- 50 kbps: `TWAI_TIMING_CONFIG_50KBITS()`
- 100 kbps: `TWAI_TIMING_CONFIG_100KBITS()`
- 125 kbps: `TWAI_TIMING_CONFIG_125KBITS()`
- 250 kbps: `TWAI_TIMING_CONFIG_250KBITS()`
- 500 kbps: `TWAI_TIMING_CONFIG_500KBITS()` (default)
- 800 kbps: `TWAI_TIMING_CONFIG_800KBITS()`
- 1 Mbps: `TWAI_TIMING_CONFIG_1MBITS()`

Tracked CAN IDs

Default tracked CAN IDs (10 IDs):

```
const uint32_t trackedCANIDs[] = {
    0x100, 0x101, 0x102, 0x103, 0x104,
    0x105, 0x106, 0x107, 0x108, 0x109
};
```

To Modify:

1. Edit the array in the firmware
2. The system automatically adjusts to the number of IDs
3. Minimum: 1 ID, Maximum: Limited by memory

Example - Add More IDs:

```
const uint32_t trackedCANIDs[] = {
    0x100, 0x101, 0x102, 0x103, 0x104,
    0x105, 0x106, 0x107, 0x108, 0x109,
    0x10A, 0x10B, 0x10C  // Added 3 more IDs
};
```

CAN Frame Types

The system supports:

- **Standard Frames**: 11-bit CAN ID (0x000 to 0x7FF)
- **Extended Frames**: 29-bit CAN ID (0x000 to 0x1FFFFFFF)
- **Data Frames**: Normal CAN messages with data
- **RTR Frames**: Remote Transmission Request frames

Filtering

- **Hardware Filter**: Accepts all CAN messages (TWAI_FILTER_CONFIG_ACCEPT_ALL)
- **Software Filter**: Only messages with tracked CAN IDs are processed and logged
- **Performance**: Software filtering allows dynamic ID changes without reconfiguring hardware

---

DBC File Support

What is DBC?

DBC (Database CAN) files are text-based files that define CAN message structures, signal definitions, and decoding information. They are widely used in automotive and industrial applications.

DBC File Format

A DBC file contains:

1. **Message Definitions**: `BO_ <ID> <Name>: <DLC> <Node>`
    - Example: `BO_ 256 EngineData: 8 Engine`
2. **Signal Definitions**: `SG_ <Name> : <StartBit>|<Length>@<ByteOrder><ValueType> (<Min>, <Max>) [<Default>|<Factor>,<Offset>] "<Unit>" <Receivers>`
    - Example: `SG_ EngineSpeed : 0|16@1+ (0,8000) [0|0.25,0] "rpm" Vector`

Decoder DBC Support

The Python decoder (`OnlyCAN_Data_decoder.py`) includes a DBC parser that:

1. **Loads DBC Files**: Parses standard DBC file format
2. **Decodes Signals**: Extracts physical values from raw CAN data
3. **Displays Messages**: Shows DBC message definitions
4. **Exports Decoded Data**: Includes decoded signals in export formats

Using DBC Files

1. **Load DBC File**: Click "Browse" next to "DBC File" in decoder
2. **View Messages**: Click "Show DBC Messages" to see all defined messages
3. **Auto-Decode**: Decoder automatically decodes frames when DBC is loaded
4. **Export**: Decoded signals included in exported data (if option enabled)

DBC File Example

```
VERSION ""

BO_ 256 EngineData: 8 Engine
 SG_ EngineSpeed : 0|16@1+ (0,8000) [0|0.25,0] "rpm" Vector
 SG_ CoolantTemp : 16|8@1+ (-40,215) [0|1,0] "C" Vector
 SG_ ThrottlePos : 24|8@1+ (0,100) [0|0.392,0] "%" Vector

BO_ 257 VehicleData: 8 Body
 SG_ VehicleSpeed : 0|16@1+ (0,300) [0|0.01,0] "km/h" Vector
 SG_ Odometer : 16|32@1+ (0,999999) [0|1,0] "km" Vector
```

NTP Time Synchronization

Overview

The system uses NTP (Network Time Protocol) to synchronize the RTC with internet time servers when WiFi Station mode is connected.

Configuration

```
#define NTP_SERVER      "pool.ntp.org"
#define GMT_OFFSET_SEC  19800  // IST = +5:30 = 19800 seconds
#define DST_OFFSET_SEC  0
```

Time Zone Offsets

Common time zone offsets (in seconds):

Time Zone	Offset (seconds)	Example
IST (India)	+19800	+5:30
EST (US)	-18000	-5:00
PST (US)	-28800	-8:00
GMT (UK)	0	0:00
CET (Europe)	+3600	+1:00

Synchronization Process

1. **WiFi STA Connection**: System connects to configured WiFi network
2. **NTP Request**: Sends time request to NTP servers
3. **Time Update**: Updates RTC with synchronized time
4. **Background Sync**: NTP sync happens in background (non-blocking)

NTP Servers

Default servers (in order of priority):

- `pool.ntp.org` (Primary)
- `time.nist.gov` (Secondary)
- `time.google.com` (Tertiary)

Fallback Behavior

- **No WiFi STA**: Uses RTC time (if available) or compile time
- **NTP Failure**: Falls back to RTC time or compile time
- **RTC Not Connected**: Uses compile time from firmware build

---

WiFi Access Point Network

Access Point Configuration

The ESP32 creates its own WiFi network for direct connection:

```
#define WIFI_AP_SSID    "CAN_Data_Logger"
#define WIFI_AP_PASS    "CANDataLogger123"
#define WIFI_AP_IP      IPAddress(192, 168, 10, 1)
#define WIFI_AP_GATEWAY IPAddress(192, 168, 10, 1)
#define WIFI_AP_SUBNET  IPAddress(255, 255, 255, 0)
```

Network Details

- **SSID**: `CAN_Data_Logger`
- **Password**: `CANDataLogger123`
- **IP Address**: `192.168.10.1`
- **Subnet Mask**: `255.255.255.0`
- **Gateway**: `192.168.10.1`
- **Channel**: 6 (default)
- **Mode**: 802.11 b/g/n (2.4 GHz only)

Station Mode (Optional)

The system can also connect to an existing WiFi network:

```
#define WIFI_STA_SSID   "YourWiFiNetwork"
#define WIFI_STA_PASS   "YourPassword"
```

Dual Mode Operation:

- **AP Mode**: Always active for direct connection
- **STA Mode**: Connects to router (if configured)
- **Benefits**: Internet access for NTP sync while maintaining direct access

Connection Process

1. **Power On**: ESP32 starts Access Point
2. **Network Visible**: "CAN_Data_Logger" appears in WiFi list
3. **Connect**: Enter password "CANDataLogger123"
4. **Access**: Open browser to `http://192.168.10.1`

Security Considerations

- **Default Password**: Change password in production
- **Network Isolation**: AP network is isolated from internet (unless STA connected)
- **Encryption**: WPA2 encryption enabled
- **Access Control**: Only devices with password can connect

Web Server Interface

Overview

The web server provides a user-friendly interface for monitoring system status and accessing logged data files.

Image Placeholder: [Web Server Interface Screenshot]

Features

1. **Status Dashboard**: Real-time system status cards
2. **File Management**: List and download .NXT files
3. **Live Data Stream**: Real-time CAN message display
4. **Responsive Design**: Works on desktop and mobile devices

Status Cards

The dashboard displays six status cards:

1. **CAN Bus**: Connection status and message reception
2. **Storage**: SD card status and availability
3. **RTC Clock**: Time synchronization status
4. **Messages Logged**: Total message count
5. **WiFi Status**: AP/STA connection status
6. **Web Server**: Server online status

File Management

- **File List**: Displays all .NXT files on SD card
- **File Information**: Shows file name and size
- **Download**: One-click download of files
- **Auto-Refresh**: File list updates automatically

Live Data Stream

- **Real-time Updates**: Updates every 2 seconds

- **Frame Display**: Shows CAN ID, timestamp, and data
- **Manual Refresh**: Button to manually refresh data
- **Buffer Size**: Last 200 frames stored in memory

Web Server Endpoints

Endpoint	Method	Description
`/`	GET	Main dashboard page
`/files`	GET	JSON list of files
`/download?file=<name>`	GET	Download .NXT file
`/live?limit=<n>`	GET	JSON live data stream

Design Features

- **Modern UI**: Gradient headers, card-based layout
- **Color Coding**: Green (OK), Red (Error), Yellow (Warning)
- **Animations**: Smooth transitions and hover effects
- **Responsive**: Adapts to different screen sizes

Python Decoder Application

Overview

The Python decoder (`OnlyCAN_Data_decoder.py`) is a professional GUI application for decrypting and analyzing .NXT files.

Image Placeholder: [Decoder GUI Screenshot]

Features

1. **File Loading**: Load and decrypt .NXT files
2. **DBC Support**: Load DBC files for signal decoding
3. **Data Display**: Tabular view of decoded CAN frames
4. **Statistics**: Comprehensive data analysis
5. **Visualization**: Charts and graphs
6. **Export**: 10+ file formats

GUI Layout

Tab 1: Decoded Data

- **File Selection**: Browse and load .NXT files
- **DBC File**: Optional DBC file for signal decoding

- **Data Table**: Scrollable table with all decoded frames
- **Filter**: Filter by CAN ID
- **Columns**: Frame, Timestamp, CAN ID, DLC, Data, Sequence, DBC Decoded

Tab 2: Statistics

- **Overview**: Total frames, unique IDs, errors, success rate
- **CAN ID Distribution**: Count and percentage per ID
- **DLC Distribution**: Data length code statistics
- **Time Analysis**: Time span and message rate
- **Visual Bars**: ASCII bar charts for distributions

Tab 3: Visualization

- **Plot Types**: Message Count, CAN ID Distribution, Data Rate
- **Interactive Charts**: Matplotlib-based visualizations
- **Export Plots**: Save charts as images

Tab 4: DBC Messages

- **Message List**: All DBC-defined messages
- **Signal Information**: Signal names and definitions
- **CAN ID Mapping**: Message ID to name mapping

Tab 5: Export

- **Format Selection**: Dropdown with 10 formats
- **Options**: Include DBC data, include statistics
- **Export Button**: Save data in selected format

Export Formats

1. **CSV**: Excel-compatible, universal format
2. **XLSX**: Native Excel with multiple sheets
3. **MAT**: MATLAB format for signal processing
4. **JSON**: Web/API format for dashboards
5. **TXT**: Plain text for simple logging
6. **HDF5**: Large datasets for archiving
7. **PARQUET**: Fast analytics format
8. **SQLITE**: Database format for queries
9. **XML**: Structured data for integration
10. **BINARY**: Compact format for performance

DBC Decoding

When a DBC file is loaded:

1. **Auto-Decode**: All frames automatically decoded
2. **Signal Extraction**: Physical values extracted from raw data
3. **Display**: Decoded signals shown in "DBC Decoded" column
4. **Export**: Decoded signals included in exports (if enabled)

Statistics Calculation

The decoder calculates:

- **Total Frames**: Number of decoded frames
- **Unique CAN IDs**: Number of different CAN IDs
- **Errors**: Decryption or parsing errors
- **Success Rate**: Percentage of successfully decoded frames
- **Time Span**: Duration of logged data
- **Message Rate**: Average messages per second
- **ID Distribution**: Count and percentage per CAN ID
- **DLC Distribution**: Distribution of data lengths

Launch Methods

1. **Batch File**: Double-click `Launch_Decoder.bat`
2. **Command Line**: `python OnlyCAN_Data_decoder.py [file.NXT]`
3. **Drag & Drop**: Drag .NXT file onto batch file

Installation & Configuration

Hardware Setup

1. **Connect CAN Transceiver**:
    - TX → Pin 5
    - RX → Pin 4
    - VCC → 5V
    - GND → GND
2. **Connect SD Card Module**:
    - CS → Pin 20
    - MOSI → Pin 19
    - MISO → Pin 18
    - SCK → Pin 21
    - VCC → 5V
    - GND → GND

3. **Connect RTC Module**:
    - SDA → Pin 6
    - SCL → Pin 7
    - VCC → 3.3V or 5V
    - GND → GND
4. **Connect Neopixel LED**:
    - Data → Pin 8
    - VCC → 5V
    - GND → GND

Software Setup

1. **Install Arduino IDE**: Version 1.8.19 or later
2. **Install ESP32 Board Support**: Add ESP32 board package
3. **Install Libraries**:
    - RTClib (Adafruit)
    - Adafruit NeoPixel
    - SD Library (built-in)
    - WiFi Library (built-in)
    - WebServer Library (built-in)
    - mbedtls (built-in)
4. **Configure Code**:
    - Open `CAN_Data_Logger_Only/CAN_Data_Logger_Only.ino`
    - Modify WiFi credentials if needed
    - Adjust CAN IDs if needed
    - Set time zone offset
5. **Upload Code**:
    - Select board: ESP32C6 Dev Module
    - Select port
    - Enable "CDC On Boot" (important for Serial Monitor)
    - Upload

Python Decoder Setup

1. **Install Python**: Version 3.x
2. **Install Dependencies**:

```
pip install -r requirements.txt
```

3. **Required Packages**:
    - pycryptodome
    - matplotlib
    - pandas
    - scipy
    - openpyxl
    - h5py
    - pyarrow

Usage Guide

Basic Operation

1. **Power On**: Connect ESP32 to USB power
2. **Wait for Initialization**: LED will show status colors
3. **Connect to WiFi**: Connect to "CAN_Data_Logger" network
4. **Access Web Interface**: Open `http://192.168.10.1`
5. **Monitor Status**: Check status cards for system health
6. **View Live Data**: Scroll to "Live Data Stream" section
7. **Download Files**: Click "Download" next to .NXT files

Logging CAN Data

1. **Connect CAN Bus**: Connect CAN transceiver to CAN network
2. **Verify CAN IDs**: Ensure tracked IDs match your network
3. **Check CAN Speed**: Verify speed matches network (500 kbps default)
4. **Monitor Serial**: Watch Serial Monitor for received messages
5. **Check LED**: MAGENTA = logging active, ORANGE = waiting
6. **Download Files**: Files created automatically on SD card

Using the Decoder

1. **Launch Decoder**: Double-click `Launch_Decoder.bat`
2. **Load .NXT File**: Click "Browse" and select file
3. **Click "Load & Decrypt"**: Decoder processes file
4. **View Data**: Switch to "Decoded Data" tab
5. **Load DBC (Optional)**: Browse and load DBC file
6. **View Statistics**: Check "Statistics" tab
7. **Export Data**: Go to "Export" tab and select format

LED Status Guide

Color	Meaning	Description
RED	Error	CAN init failed, SD error, WiFi error
BLUE	Initializing	System boot sequence
YELLOW	CAN Initializing	CAN bus setup in progress
CYAN	SD Ready / AP Active	SD card ready or AP visible
MAGENTA	Data Transfer Active	Actively logging CAN messages
ORANGE	Data Transfer Stopped	Waiting for CAN messages
GREEN	WiFi Connected	STA + AP both connected
WHITE	System Ready	All systems initialized

Technical Specifications

System Specifications

Parameter	Value
Microcontroller	ESP32-C6 Pico (RISC-V, 160 MHz)
CAN Interface	TJA1050 Transceiver
CAN Speed	500 kbps (configurable)
Tracked CAN IDs	10 (configurable)
Storage	SD Card (FAT32, 2-32 GB)
File Format	.NXT (encrypted binary)
Encryption	AES-128-CBC
RTC	DS3231 with battery backup
WiFi	802.11 b/g/n (2.4 GHz)
Web Server	HTTP on port 80
LED Indicator	WS2812B Neopixel

Performance Specifications

Parameter	Value
Max File Size	10 MB
Max Frames per File	~214,000
Frame Size	49 bytes (encrypted)
Live Buffer Size	200 frames
Message Processing	Up to 10 frames per loop
Flush Interval	Every 10 frames
Web Server Response	< 100 ms

Power Requirements

Component	Voltage	Current
ESP32-C6	5V (USB)	~200 mA

Component	Voltage	Current
CAN Transceiver	5V	~70 mA
RTC Module	3.3V/5V	~1 mA
SD Card Module	5V	~100 mA
Neopixel LED	5V	~60 mA (max)
Total	**5V**	**~430 mA**

Troubleshooting

Serial Monitor Not Working

Symptoms: No output after code upload

Solutions:

1. Enable "CDC On Boot" in board settings
2. Check baud rate is 115200
3. Try different USB port
4. Press RESET button on ESP32
5. Restart Arduino IDE

WiFi Not Appearing

Symptoms: "CAN_Data_Logger" network not visible

Solutions:

1. Wait 10-20 seconds after power on
2. Check Serial Monitor for WiFi messages
3. Try resetting ESP32
4. Verify AP is enabled in code
5. Check WiFi channel (should be 6)

SD Card Not Detected

Symptoms: "SD Card authentication failed" or "No SD Card"

Solutions:

1. Check SD card is formatted as FAT32
2. Verify wiring connections
3. Try different SD card
4. Check card is not write-protected
5. Verify card compatibility (2-32 GB recommended)

CAN Messages Not Received

Symptoms: No messages in Serial Monitor or web interface

Solutions:

1. Verify CAN transceiver connections
2. Check CAN bus has data
3. Verify CAN IDs match tracked IDs
4. Check CAN speed matches network
5. Ensure termination resistors (120Ω) are present

Decoder Not Opening

Symptoms: Decoder GUI doesn't launch

Solutions:

1. Ensure Python 3.x is installed
2. Install dependencies: `pip install -r requirements.txt`
3. Check Tkinter is available: `python -m tkinter`
4. Try running from command line for error messages

Decryption Errors

Symptoms: "Failed to decode .NXT file" or "Encryption key mismatch"

Solutions:

1. Verify encryption key matches in firmware and decoder
2. Check file is not corrupted
3. Ensure file was created by ESP32 logger
4. Verify file format version matches

Appendices

Appendix A: Pin Reference

Complete pin mapping for ESP32-C6 Pico:

Pin	Function	Component
4	CAN RX	TJA1050
5	CAN TX	TJA1050
6	I2C SDA	DS3231 RTC

Pin	Function	Component
7	I2C SCL	DS3231 RTC
8	Data	Neopixel LED
18	SPI MISO	SD Card
19	SPI MOSI	SD Card
20	SPI CS	SD Card
21	SPI SCK	SD Card

Appendix B: File Format Examples

Example .NXT File Structure:

```
Offset    Content
------    -------
0x0000    Magic: 43 41 4E 44
0x0004    Version: 01
0x0005    Start Time: E4 07 01 00
0x0009    Frame 1 IV: [16 bytes]
0x0019    Frame 1 Length: 14
0x001A    Frame 1 Ciphertext: [32 bytes]
0x003A    Frame 2 IV: [16 bytes]
...
```

Appendix C: DBC File Format Reference

Message Definition:

```
BO_ <CAN_ID> <MessageName>: <DLC> <Node>
```

Signal Definition:

```
SG_ <SignalName> : <StartBit>|<Length>@<ByteOrder><ValueType>
    (<Min>,<Max>) [<Default>|<Factor>,<Offset>] "<Unit>" <Receivers>
```

Appendix D: Export Format Details

CSV Format:

- Comma-separated values
- Excel compatible
- Universal format

XLSX Format:

- Native Excel format
- Multiple sheets (Data + Statistics)
- Formatted cells

MAT Format:

- MATLAB compatible
- Structured data arrays
- For signal processing

Appendix E: Error Codes

Error Code	Description	Solution
ESP_ERR_INVALID_STATE	CAN driver not initialized	Check CAN initialization
ESP_ERR_INVALID_ARG	Invalid CAN configuration	Verify pin assignments
ESP_ERR_NOT_FOUND	RTC not detected	Check RTC wiring
ESP_ERR_NO_MEM	Out of memory	Reduce buffer sizes

Appendix F: Revision History

Version	Date	Changes
1.0	2024-01	Initial release

Conclusion

The CAN Data Logger for ESP32-C6 Pico provides a complete, professional solution for CAN bus data logging. With its encrypted storage, web interface, and comprehensive decoder, it meets the needs of automotive and industrial applications requiring secure, timestamped data capture and analysis.

For support, refer to the troubleshooting section or check the Serial Monitor output for detailed error messages.

Document End