Application Note

Running DroneCAN Applications on TI AM13E2x

Mithra Kandasamy and Shaunak Deshpande

ABSTRACT

DroneCAN is a lightweight, open communication protocol widely used in UAV and robotics systems for reliable data exchange between nodes over a CAN bus. This application note provides a practical guide to implementing DroneCAN on the TI AM13E230x microcontroller, using the on-chip MCAN peripheral and the Libcanard protocol stack. This document explains the hardware setup, software integration, a working example, and common issues to help developers build DroneCAN-enabled nodes on AM13x-family devices. This application note applies to all the TI MCUs that have the same MCAN peripheral.

Table of Contents

Trademarks

1 Introduction

1.1 DroneCAN: Protocol Overview and Key Features

DroneCAN is a lightweight, open-source communication protocol built on top of CAN bus, originally developed for UAV systems, and now widely adopted in robotics, industrial, and aerospace applications. DroneCAN defines a standardized message format, transfer segmentation for payloads exceeding 8 bytes, node ID management, and data type signatures for interoperability between nodes from different vendors. Libcanard is the official C implementation of the DroneCAN stack, designed specifically for resource-constrained embedded systems with minimal RAM and no dynamic memory allocator dependency beyond a user-supplied memory pool. Written entirely in C, Libcanard integrates cleanly into existing embedded projects and TI MCU SDK examples without requiring any OS or middleware layer. The stack exposes simple APIs that cover the full lifecycle of DroneCAN transfer encoding, transmission, reception, and decoding. This design is easy to adopt for developers already familiar with CAN-based communication.

DroneCAN ID Format: Figure 1-1 shows the 29-bit ID structure for a DroneCAN broadcast message.

Priority					Data Type ID												Service/ Message Flag	Source Node ID						
28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 1-1. CAN ID Format

A normal CAN frame id is typically 11 bits, but DroneCAN uses extended frame ID (EFF). Here, the frames are self-describing, and the receiving tool can decode any frame on the wire without any other prior knowledge of the application which makes integration of DroneCAN much easier.

Figure 1-2 shows an example of a Drone System, using DroneCAN for communication.

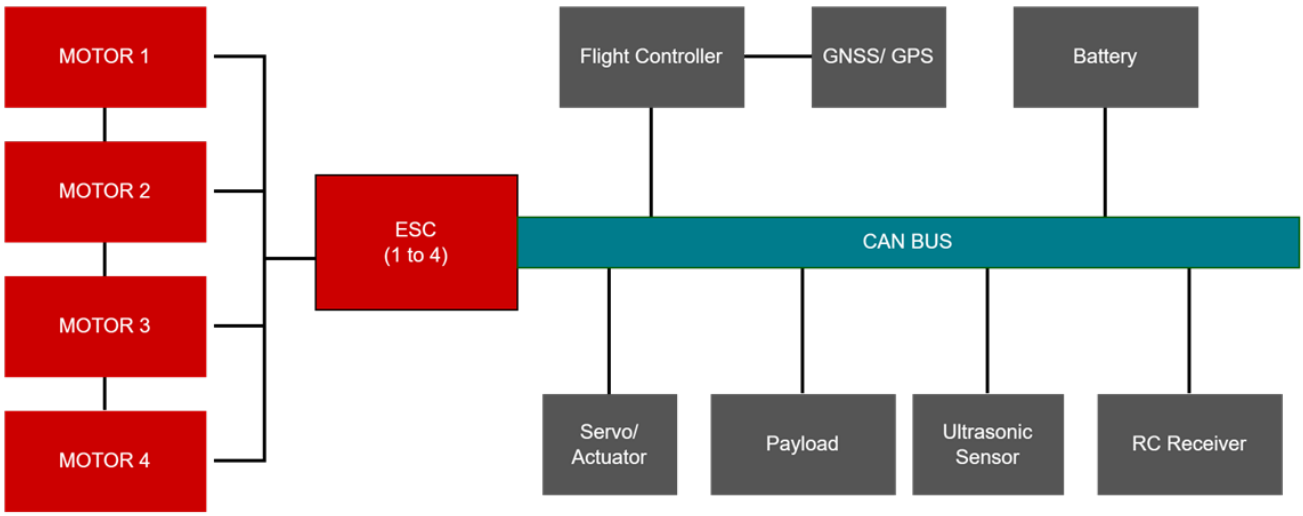

Figure 1-2. Drone System with DroneCAN

Here, DroneCAN is primarily used for the communication between the flight controller (brain) and electronic speed controller (ESC) which further controls the speed and rotation of the motors. A single CAN bus can handle all communication between different nodes without any overlapping issues. CAN only needs two wires to connect every node on the bus, and this bus handles all fault detection, retransmission of frames and other kinds of errors completely in hardware without any software involvement.

1.2 AM13E230x Overview:

The AM13E230x is a 32-bit Arm Cortex-M33 based microcontroller, designed for cost-sensitive industrial and automotive applications requiring reliable field communication. The device integrates an MCAN peripheral and can operate up to 200Mhz, supporting both Classical CAN and CAN FD at configurable baud rates. The MCAN peripheral features a flexible message RAM architecture supporting dedicated TX buffers, RX FIFOs, and hardware acceptance filters, enabling efficient multi-frame transfer handling with minimal CPU overhead. SysConfig support for AM13E230x allows complete peripheral configuration including bit timing, message RAM configurations, GPIO pin assignment, interrupt routing, and CAN mode config; through a graphical interface, significantly reducing bring-up time for new users. The combination of integrated CAN hardware, SysConfig based configuration, and DriverLib API support makes AM13E230x a practical and capable platform for DroneCAN integration.

1.2.1 Why AM13E230x for DroneCAN

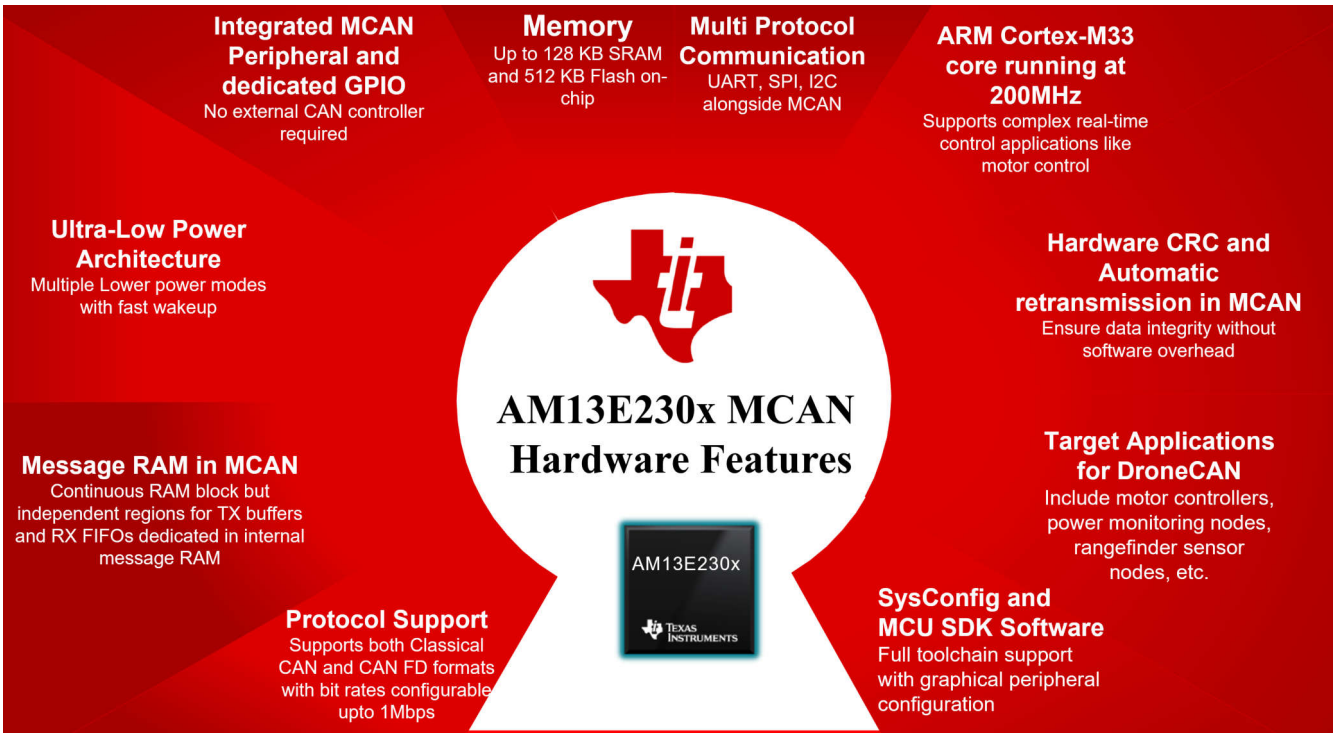

Figure 1-3. MCAN Hardware Features

- The AM13E230x MCAN Peripheral is on-chip with dedicated GPIO, so there is no external CAN controller needed.
- The dedicated CAN Core (Controller) makes sure not much of CPU bandwidth is consumed for CAN communication, so the device offers very efficient motor control even when running DroneCAN.
- DroneCAN relies on every frame being delivered properly, and the MCAN peripheral handles automatically retransmitting frames that hit some error without any software involvement.
- MCAN has an internal message RAM with separate regions for TX Buffers, RX FIFOs, and hardware acceptance filters. So, all incoming frames are stored and filtered before being processed, which is very efficient when the bus is busy with multiply node types transmitting simultaneously.
- The register space connects the Tx and Rx transfer handlers with the actual hardware FIFOs and the message RAM. All the register access is abstracted through the TI Driverlib.
- Both CAN and CAN FD are supported, so higher throughput applications can be supported in the future by the same hardware without any hardware board changes.
- This design is an ultra-low power architecture. The multiple low power modes with fast wakeup enable quick response from nodes that are mostly idle.

- The entire MCAN peripheral, bit timing, message RAM configurations, GPIO pin assignment, interrupt routing, time, and CAN mode config are configured through Syscfg, reducing the overhead on the end developers.
- Alongside MCAN, UART, SPI and I2C communication protocols are available which enable the device to talk to other components simultaneously.

2 System Architecture

Libcanard handles everything involving the DroneCAN protocol but is completely unaware of the TI MCAN peripheral. Libcanard manages encoding, decoding, and the handling of Canard frames, but cannot physically transfer these frames across the bus. Meanwhile, the TI MCAN peripheral can handle transmission and reception of CAN frames but has no knowledge of the Canard ID structure or the attributes. To bridge this gap, a **dedicated interface layer** is required, which enables the developer to build a DroneCAN node on an AM13x device without writing peripheral interaction code directly in the application.

2.1 Software Stack Overview

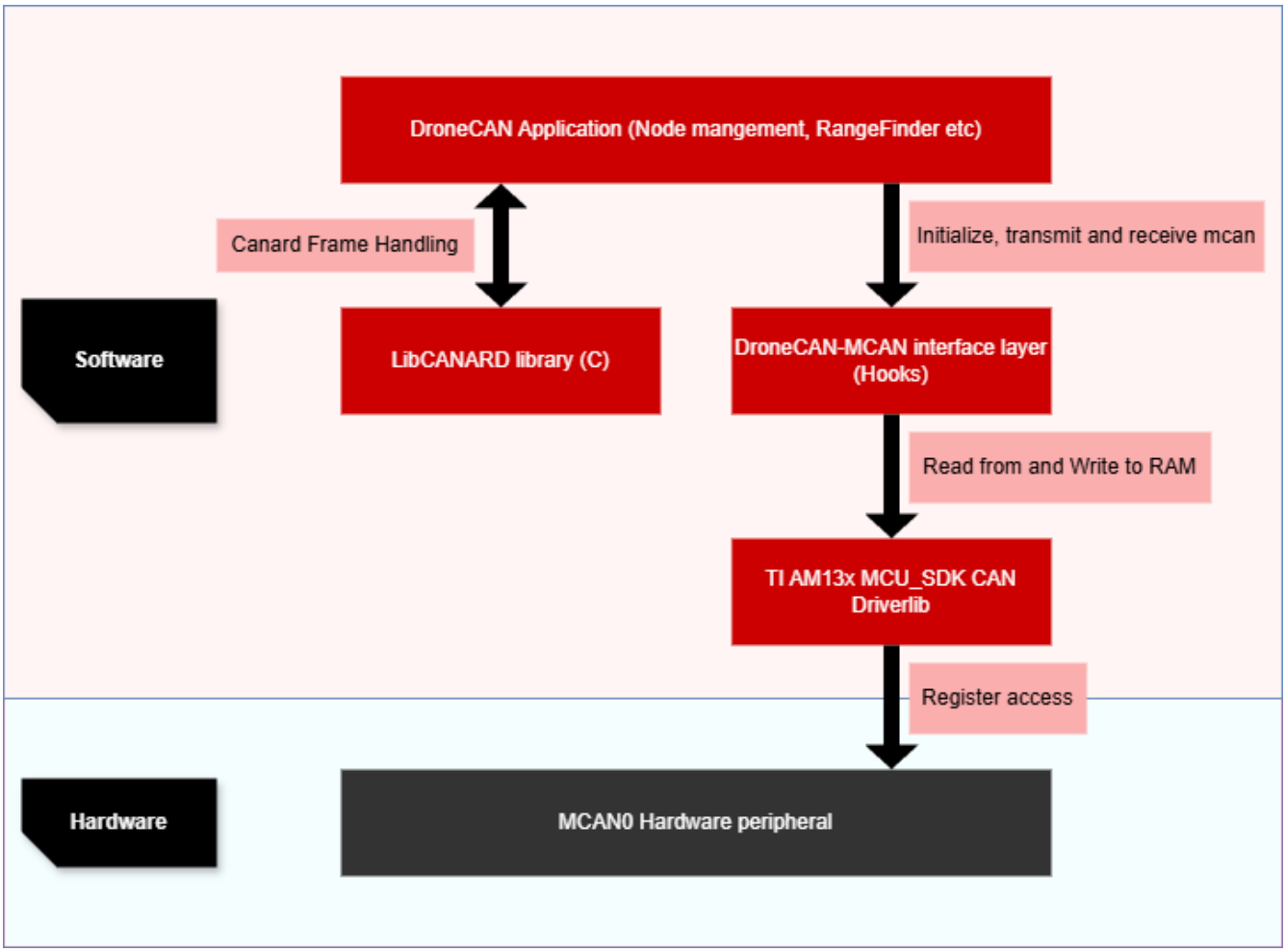

Figure 2-1. Software Application Structure

- **DroneCAN Application** sits at the top and drives the entire system. The application initializes the stack, encodes and decodes transfers, manages the node, and handles periodic maintenance.
- **Libcanard** is the protocol library that the application interacts with directly. This protocol library handles all DroneCAN protocol responsibilities including frame encoding, decoding, transfer reassembly, CRC, and memory management with no hardware knowledge.
- **DroneCAN-MCAN Interface Layer** is the hardware abstraction that connects the application to the TI peripheral.
  - The application translates Libcanard frame structures into MCAN message RAM format and handles transceiver control, filter programming, and mode transitions.
  - All hardware-specific complexity is contained within this layer, keeping both the application and Libcanard completely free of any peripheral-specific code.
- **MCAN0 Hardware Peripheral** is the physical CAN controller on the AM13E230x that drives the CAN bus.

2.2 The Libcanard-MCAN Interface Layer: TX and RX Flow

The interface layer primarily exposes APIs that include conversion of frame format from MCAN to canard and vice-versa, initialization, filter configuration, transmission, and reception. This is directly reusable across any DroneCAN project on a device with similar MCAN peripherals. Here is the rough flow of how packets and transmitted and received in applications that are integrated with Libcanard.

2.2.1 TX Flow

Figure 2-2 shows the typical flow for a frame transfer.

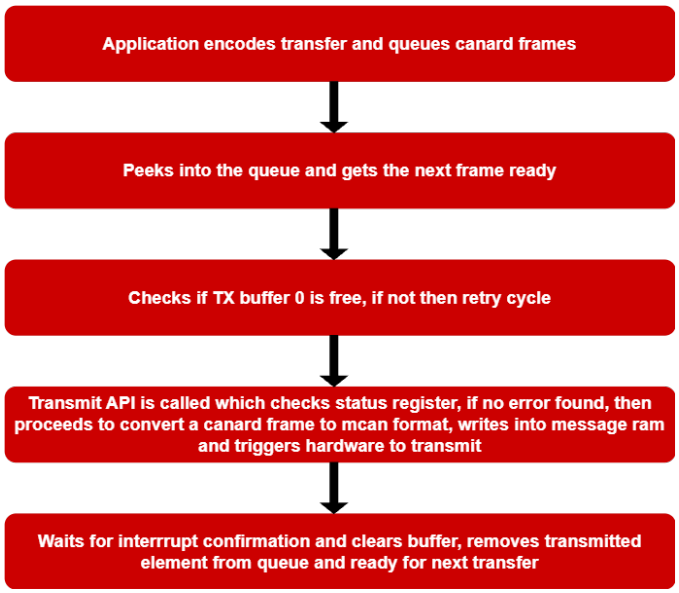

Figure 2-2. Frame Transfer Flow

2.2.2 RX Flow

Figure 2-3 shows the typical flow for frame reception.

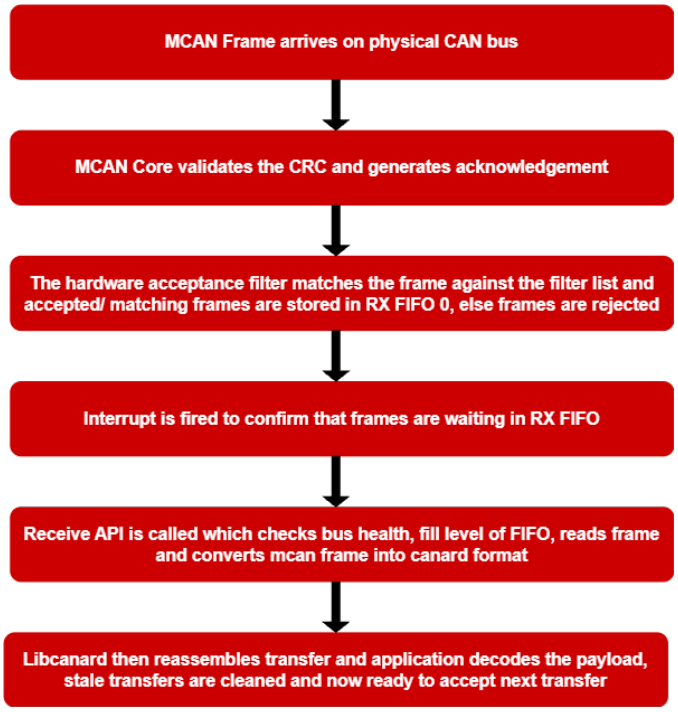

Figure 2-3. Frame Reception Flow

3 Software Application Overview

Sample Libcanard application implemented on the AM13E230x Launchpad is available on request. Please reach out on the E2E™ forum or through your TI sales or field contact to get access to the same.

3.1 Project Structure

A sample CCS DroneCAN project structure is discussed below:

- The example.syscfg file at the root is the single point of hardware configuration. OPen the file in the CCS SysConfig editor to modify any peripheral settings.
- The Libcanard directory contains the unmodified libcanard source files and interface files canard_am13x (inside drivers). These files import into the project without any changes.
- Application constants including node ID, memory pool size, data type IDs, timer instance, and tick rate are centralized in app_cfg.h. Modify these details as required.
- The application-level DroneCAN frame encode and decode logic lives in app_mcan.c, and main.c serves purely as the entry point.

```
example_root_folder
│
├── example.syscfg              ← Hardware configuration. Open this
│                                 in CCS SysConfig editor to change
│                                 MCAN settings, GPIO, timers.
│
├── am13e230x_lp/
│   └── app_cfg.h               ← Application constants: node ID,
│                                 memory pool size, DTID values,
│                                 timer instance, tick rate, etc.
│
├── libcanard/                  ← Migrate the entire libcanard
│   ├── canard.c                   directory into the project root.
│   ├── canard.h                   Do not modify these files.
│   ├── canard_internals.h
│   └── drivers/
│       ├── canard_am13x.c      ← TI driver layer implementation.
│       └── canard_am13x.h      ← TI driver layer header.
│
├── app_mcan.c                  ← Application layer. Your DroneCAN
│                                 encode/decode logic lives here.
│
├── main.c ← Entry point.
│
└── targetConfigs/              ← CCS debug configuration.

WARNING: Not Thread-Safe

This driver does not use IRQ or critical sections. It is safe to call the API functions from IRQ
context **provided the application verifies that no concurrent calls occur from thread and IRQ
context simultaneously**. The application is responsible for all guarding.
```

3.2 SysConfig Peripheral Configuration

All peripheral configuration is managed through SysConfig. The following settings are required for correct DroneCAN operation:

MCAN General Settings

- Uncheck "Messages Will Only Be Sent Once" so that frames can be retransmitted automatically, so that erroneous frames are not lost forever.

Message RAM Layout

- Standard filter list: 2 elements, start address 0
- Extended filter list: 2 elements, start address 8
- TX buffer: start address 24, 1 element
- TX event FIFO: start address 96
- RX FIFO 0: configured with element count, watermark, and start address following the TX regions with no overlap

Acceptance Filters

- Standard ID filter: classic bitmask, routes matching frames to RX FIFO 0
- Extended ID filter: range filter, EFID2 set to 536870911 to cover full 29-bit range
- Accept non-matching extended frames: set to Reject

Interrupts

- Interrupt line 1 enabled
- RX FIFO 0 new message, TX complete, bus-off, error passive, and warning status interrupt sources are all routed to line 1

Timer

- A timer peripheral instance is required for periodic interrupt generation. The application uses a timer peripheral to maintain the timestamp passed to handle the received frame and clean the stale transfers.

Here is the SysConfig file of the mentioned TX and RX example for reference:

```
/**
 * These arguments were used when this file was generated. They will be automatically applied on
subsequent loads
 * through the GUI or CLI. Run CLI with '--help' for additional information on how to override
these arguments.
 * @cliArgs --device "AM13E230x" --part "AM13E230x" --package "LQFP64_G" --product "TI MCU
SDK@26.01.00"
 * @v2CliArgs --device "AM13E23019" --package "LQFP64_G" --product "TI MCU SDK@26.01.00"
 * @versions {"tool":"1.27.0+4565"}
 */

/**
 * Import the modules used in this configuration.
 */
const config = scripting.addModule("/drivers/config");
const gpio   = scripting.addModule("/drivers/gpio");
const gpio1  = gpio.addInstance();
const mcan   = scripting.addModule("/drivers/mcan", {}, false);
const mcan1  = mcan.addInstance();
const sysctl = scripting.addModule("/drivers/sysctl");
const timer  = scripting.addModule("/drivers/timer", {}, false);
const timer1 = timer.addInstance();
const uart   = scripting.addModule("/drivers/uart", {}, false);
const uart1  = uart.addInstance();

/**
 * Write custom configuration values to the imported modules.
 */
gpio1.$name                          = "GPIO_GRP_0";
gpio1.port                           = "PORTA";
gpio1.associatedPins[0].$name        = "TCAN_STB";
gpio1.associatedPins[0].pin.$assign = "PA24";

mcan1.$name                             = "MCAN";
mcan1.wkupReqEnable                     = true;
mcan1.emulationEnable                   = true;
mcan1.autoWkupEnable                    = true;
mcan1.tdcEnable                         = true;
mcan1.additionalCoreConfig              = true;
mcan1.rrfe                              = true;
mcan1.rrfs                              = true;
mcan1.txEventFIFOWaterMark              = 0;
mcan1.txEventFIFOSize                   = 0;
mcan1.txBufNum                          = 1;
mcan1.txBufElemSize                     = "DL_MCAN_ELEM_SIZE_8BYTES";
mcan1.nomTimeSeg1_manual                = 126;
mcan1.nomTimeSeg2_manual                = 31;
mcan1.nomSynchJumpWidth_manual          = 31;
mcan1.dataTimeSeg1_manual               = 14;
mcan1.dataTimeSeg2_manual               = 3;
mcan1.dataSynchJumpWidth_manual         = 3;
mcan1.spPercent                         = 80;
mcan1.desiredNomRate                    = 250;
mcan1.rxFIFO1startAddr                  = 0;
mcan1.rxFIFO1size                       = 0;
mcan1.rxFIFO1waterMark                  = 0;
```

```
mcan1.m0interrupts                    = ["DL_MCAN_INTERRUPT_LINE1"];
mcan1.registerInterrupt               = true;
mcan1.enableInterrupt                 = true;
mcan1.interruptLine                   = ["DL_MCAN_INTR_LINE_NUM_1"];
mcan1.interruptLine1Flag              =
["DL_MCAN_INTR_SRC_BUS_OFF_STATUS","DL_MCAN_INTR_SRC_ERR_PASSIVE","DL_MCAN_INTR_SRC_PROTOCOL_ERR_ARB
","DL_MCAN_INTR_SRC_PROTOCOL_ERR_DATA","DL_MCAN_INTR_SRC_RX_FIFO0_FULL","DL_MCAN_INTR_SRC_RX_FIFO0_N
EW_MSG","DL_MCAN_INTR_SRC_RX_FIFO0_WATERMARK","DL_MCAN_INTR_SRC_TRANS_COMPLETE","DL_MCAN_INTR_SRC_WA
RNING_STATUS"];
mcan1.interruptFlags                  =
["DL_MCAN_INTR_SRC_BUS_OFF_STATUS","DL_MCAN_INTR_SRC_ERR_PASSIVE","DL_MCAN_INTR_SRC_PROTOCOL_ERR_ARB
","DL_MCAN_INTR_SRC_PROTOCOL_ERR_DATA","DL_MCAN_INTR_SRC_RX_FIFO0_FULL","DL_MCAN_INTR_SRC_RX_FIFO0_N
EW_MSG","DL_MCAN_INTR_SRC_RX_FIFO0_WATERMARK","DL_MCAN_INTR_SRC_TRANS_COMPLETE","DL_MCAN_INTR_SRC_WA
RNING_STATUS"];
mcan1.lss                             = 2;
mcan1.lse                             = 2;
mcan1.flesa                           = 8;
mcan1.txStartAddr                     = 24;
mcan1.txEventFIFOStartAddr            = 96;
mcan1.rxBufStartAddr                  = 96;
mcan1.rxFIFO0startAddr                = 112;
mcan1.rxFIFO0size                     = 10;
mcan1.rxFIFO0waterMark                = 8;
mcan1.stdFiltType                     = "10";
mcan1.stdFiltElem                     = "001";
mcan1.extFiltElem                     = "001";
mcan1.extFiltType                     = "00";
mcan1.extFiltID2                      = 536870911;
mcan1.stdFiltID1                      = 3;
mcan1.stdFiltID2                      = 4;
mcan1.peripheral.$assign              = "MCAN0";
mcan1.peripheral.txPin.$assign        = "PA12";
mcan1.peripheral.rxPin.$assign        = "PA11";
mcan1.txPinConfig.direction           = scripting.forceWrite("OUTPUT");
mcan1.txPinConfig.hideOutputInversion  = scripting.forceWrite(false);
mcan1.txPinConfig.onlyInternalResistor = scripting.forceWrite(false);
mcan1.txPinConfig.passedPeripheralType = scripting.forceWrite("Digital");
mcan1.txPinConfig.$name               = "ti_driverlib_gpio_GPIOPinGeneric0";
mcan1.rxPinConfig.hideOutputInversion  = scripting.forceWrite(false);
mcan1.rxPinConfig.onlyInternalResistor = scripting.forceWrite(false);
mcan1.rxPinConfig.passedPeripheralType = scripting.forceWrite("Digital");
mcan1.rxPinConfig.$name               = "ti_driverlib_gpio_GPIOPinGeneric1";

sysctl.useHFCLK     = true;
sysctl.SYSPLLSource = "DL_SYSCTL_SYSPLL_REF_HFCLK";
sysctl.SYSPLL_Qdiv  = 32;

timer1.$name             = "APP_TIMER_0";
timer1.registerInterrupt = true;
timer1.timerClkDiv       = "DL_TIMER_CLOCK_DIVIDE_8";
timer1.timerClkPrescale  = 256;
timer1.timerMode         = "DL_TIMER_TIMER_MODE_PERIODIC";
timer1.interrupts        = ["DL_TIMER_INTERRUPT_ZERO_EVENT"];
timer1.timerPeriod       = "1000 ms";

uart1.$name                           = "APP_MCANTX_LOGUART_0";
uart1.targetBaudRate                  = 115200;
uart1.peripheral.$assign              = "UC4";
uart1.peripheral.rxPin.$assign        = "PA1";
uart1.peripheral.txPin.$assign        = "PA0";
uart1.txPinConfig.direction           = scripting.forceWrite("OUTPUT");
uart1.txPinConfig.hideOutputInversion  = scripting.forceWrite(false);
uart1.txPinConfig.onlyInternalResistor = scripting.forceWrite(false);
uart1.txPinConfig.passedPeripheralType = scripting.forceWrite("Digital");
uart1.txPinConfig.$name               = "drivers_gpio_v0_gpioPinConfig0";
uart1.rxPinConfig.hideOutputInversion  = scripting.forceWrite(false);
uart1.rxPinConfig.onlyInternalResistor = scripting.forceWrite(false);
uart1.rxPinConfig.passedPeripheralType = scripting.forceWrite("Digital");
uart1.rxPinConfig.$name               = "drivers_gpio_v0_gpioPinConfig1";

/**
 * Pinmux option for unlocked pins/peripherals. This means that minor changes to the automatic
solver in a future
 * version of the tool will not impact the pinmux you originally saw.  These lines can be
completely deleted in order to
 * re-solve from scratch.
 */
sysctl.peripheral.$suggestSolution       = "SYSCTL";
```

```
sysctl.peripheral.x1Pin.$suggestSolution = "PC16_X1";
sysctl.peripheral.x2Pin.$suggestSolution = "PC17_X2";
timer1.peripheral.$suggestSolution       = "TIMG4_0";
```

3.3 Memory Pool Allocation

The pool can be defined and configured through the application. The following block has been included in the linker command file for this purpose:

```
SECTIONS
{
    .canard_pool : ALIGN(8),
                   RUN_START(__CANARD_POOL_START),
                   RUN_END(__CANARD_POOL_END)
                   > RAM_S
}
```

Memory Footprint:

The sample application implemented takes up approximately 37KB of memory in debug build and approximately 35KB of memory in release build, with further scope of optimization. This means, enough memory if preserved for running other applications on the M33 core or run AI models on the TinyEngine™.

4 Hardware Setup

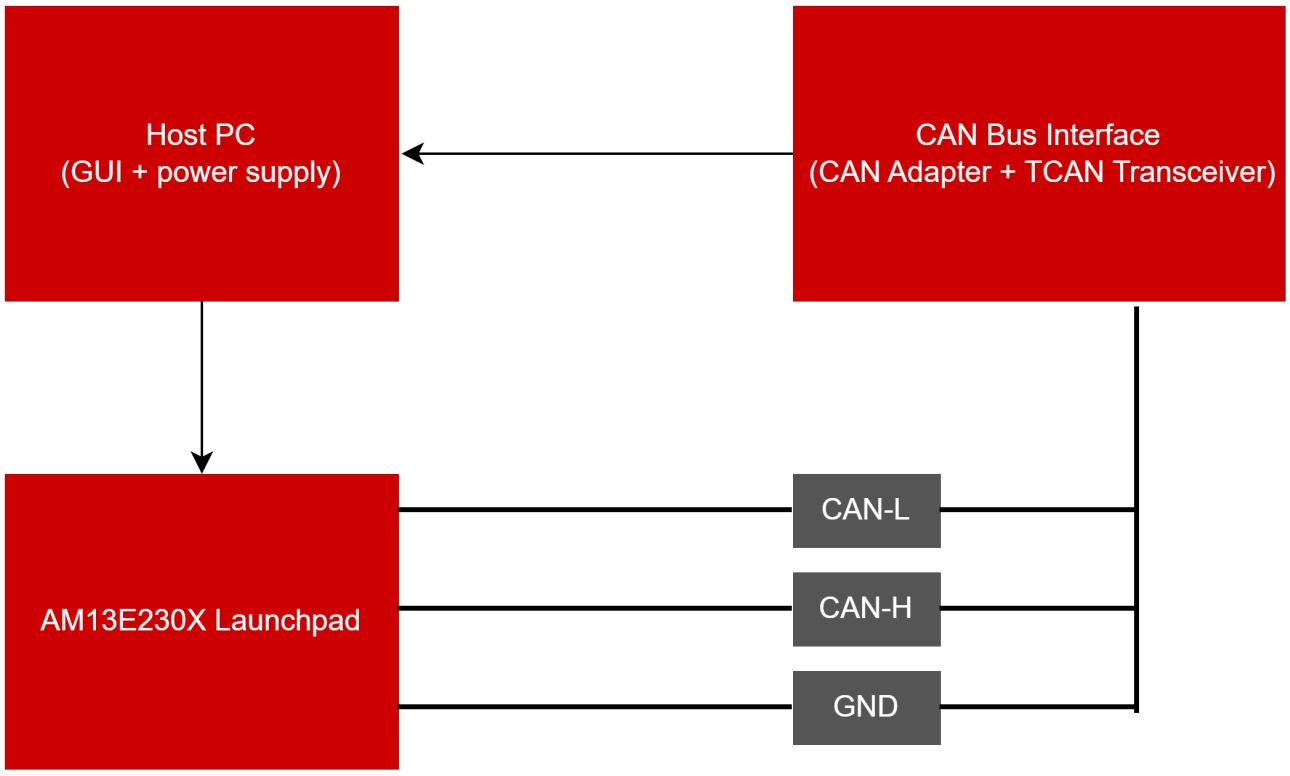

Figure 4-1. Hardware Setup

Hardware Connections:

- **Host Workstation to AM13E230X**: Provides power to the AM13E230X chip and serves as the primary control interface.
- **AM13E230X to CAN Bus Interface (or CAN Bus Interface to AM13E230X)** Three-wire CAN connection: **CAN-H**, **CAN-L**, and **GND** linking the chip to the CAN Bus Interface module.
- **CAN Bus Interface to GUI Tool or Host Workstation** the CAN Bus Interface connects (through the USB) to whichever machine is running the GUI tool. This connection can be the Host Workstation or a separate machine running only the GUI.

5 Running the Example and Test Results

1. Once you have access to the AM13E230x DroneCAN example, Import the example into CCS 21.0.0 or higher, and make sure SysConfig version is 1.28.0 or higher.
2. Download the DroneCAN GUI Tool. See the Setup webpage for detailed instructions.
3. For now, DroneCAN support along with PCAN has been tested only on Linux systems.
4. Download the PEAK-PCAN from PCAN-View webpage.
5. The mcan_message_libcanard example contains both TX and RX functionalities and the mode can be changes using the macros 'APP_MODE' defined in app_cfg.h file.
6. TX Testing:
   a. PCAN-View:
      i. Install PCAN Basic Driver and open PCAN View.
      ii. Click on connect and set Nominal Bit Rate as 250kbps and Data Bit Rate as 1000kbps.
      iii. In the example, verify that APP_MODE is set to APP_MODE_TX. Rebuild and debug project.
      iv. After running the example, the transmitted frames are visible in the PCAN receive space.

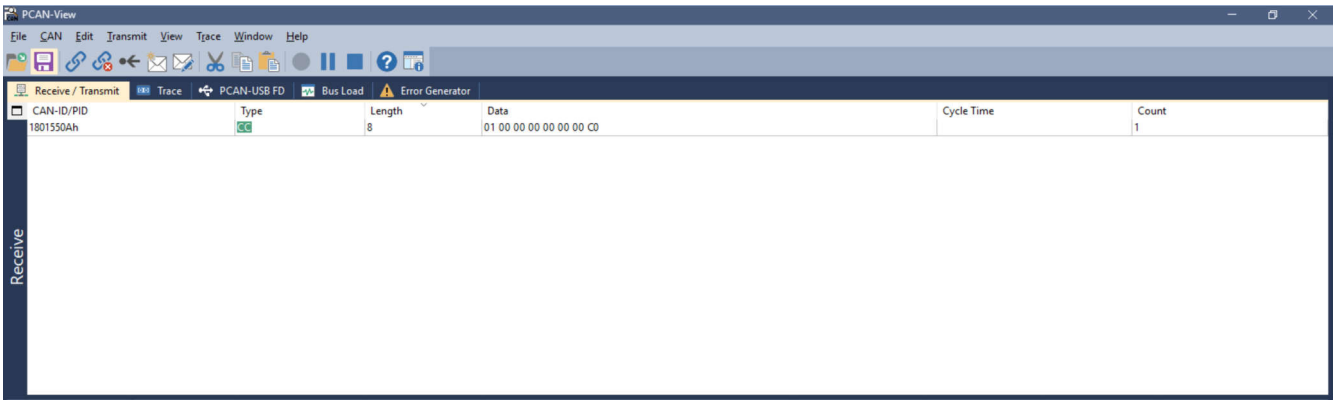

Figure 5-1. PCAN Frame

   v. Figure 5-2 shows the expected output for this example.

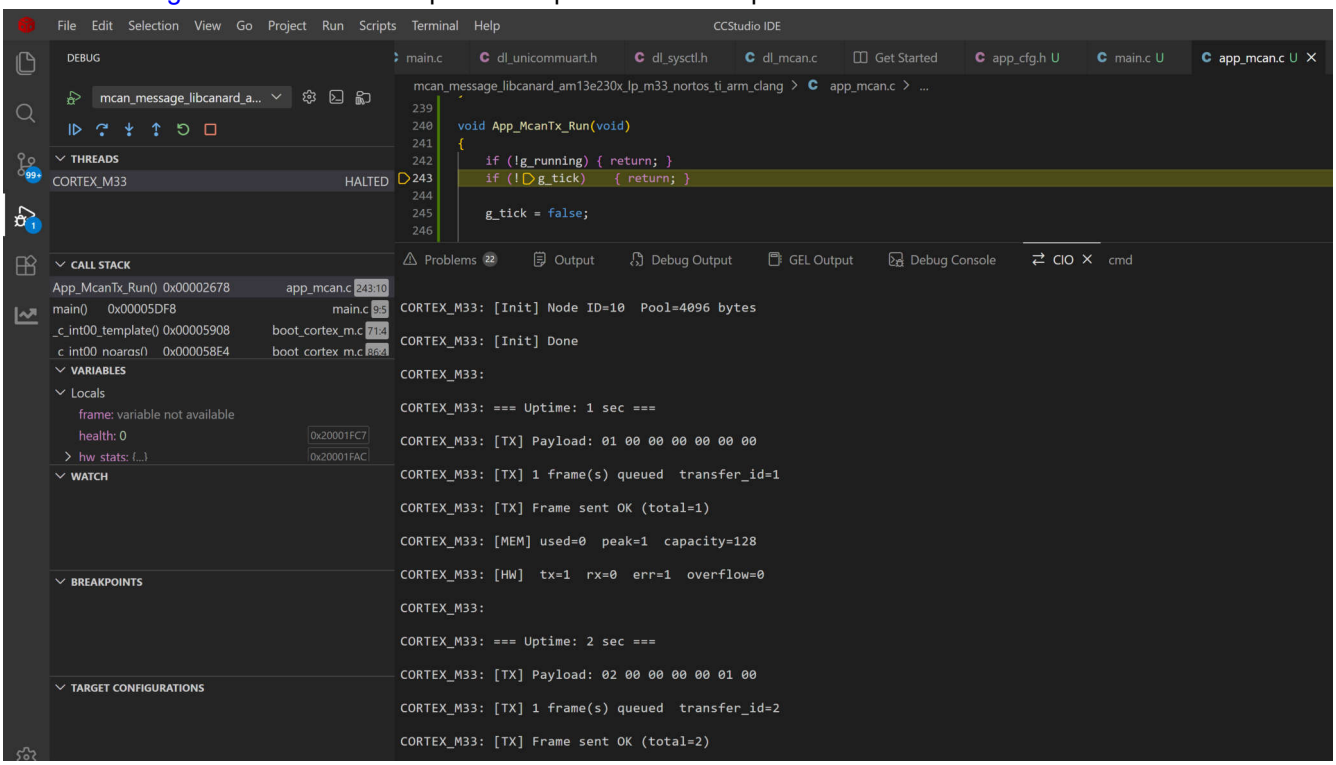

Figure 5-2. TX Console Log Output

b. DroneCAN GUI Tool:
   i. Before launching the DroneCAN GUI Tool, set up the CAN bus on the system.
   ii. Use the following commands:

```
sudo ip link set can0 up

sudo ip link set can0 type can bitrate 250000
```

   iii. To know the exact bitrate, open syscfg in text mode and look up the value set for desiredNomRate and use that. Here NomRate is set as 250, so use 250000 while setting up CAN.
   iv. Now launch the DroneCAN GUI Tool, set this bitrate again on the GUI and set a local node ID, and select the checkbox.
   v. Now open Panels, navigate to Bus Monitor and select the video icon to start receiving frames.
   vi. After debugging and running the TX example, the frames on DroneCAN are visible, along with a few other heartbeat frames.

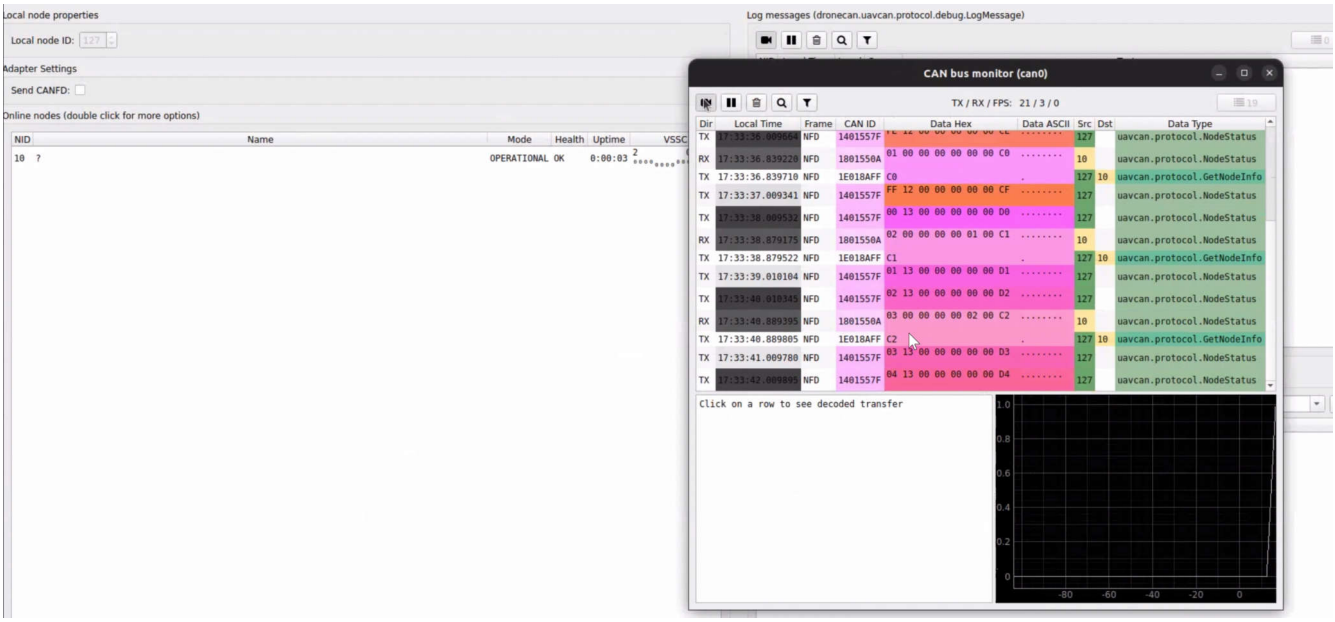

Figure 5-3. DroneCAN Frame

   vii. Three types of frames are visible here
      1. The pink rows represent the Node Status broadcast from Node 10.
      2. The orange rows represent heartbeat messages from Node 127.
      3. The green rows represent service requests from the GUI Tool to get node info.

7. RX Testing:
   a. PCAN-View:
      i. To send frames from PCAN, go to Transmit and navigate to New Message, then check on CAN FD if needed.

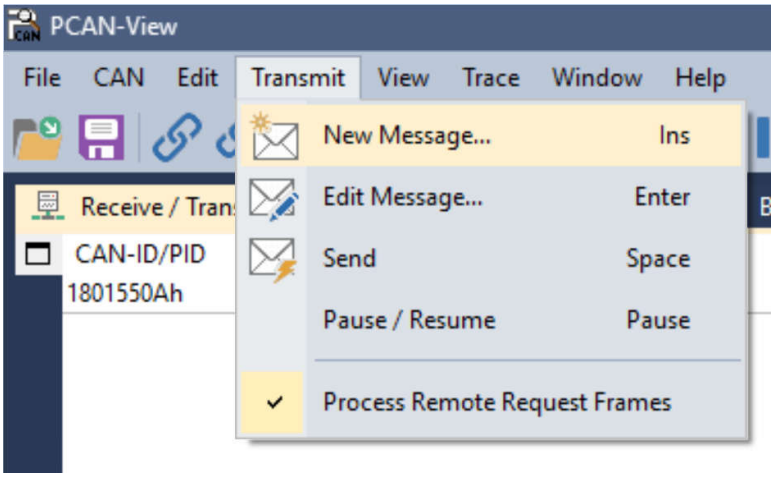

Figure 5-4. Creating New PCAN Frame

      ii. Create your own customized DroneCAN frame following proper ID structure (check DroneCAN Frame ID structure in the AppNote). Check on extended frame and bit rate switch.

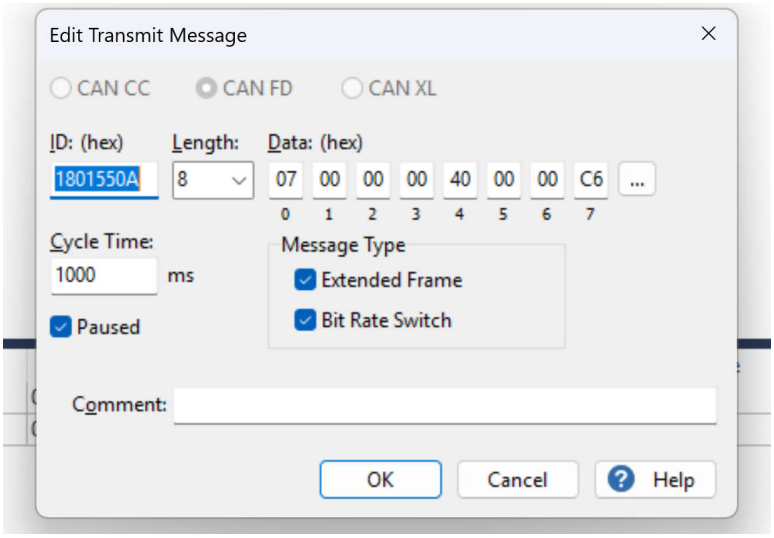

Figure 5-5. Attributes of a PCAN Frame

      iii. To transmit frames continuously, set a cycle time to constantly send frames at a particular interval.
      iv. Now before starting to send frames from PCAN, to view these frames in the console of the example, open a serial port COM8 with a baud rate (as defined in SysConfig, check for this variable named targetBaudRate and use that value).
      v. The serial console shows the decoded frames.

b. DroneCAN GUI Tool:
   i. To send frames from DroneCAN, write a python script.
   ii. Navigate to Panels then select Interactive console to open a jupyter console.
   iii. The following shows a sample python script to generate a customized frame corresponding to Node 10.

```
import time

# Clean state tracker for the incrementing byte
tx_state = {'counter': 0}

def send_classical_can_frame():
    try:
        # 1. Build your exact 8-byte payload structure
        payload_bytes = bytearray([tx_state['counter'] & 0xFF, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xC6])

        # 2. Modify the ID flags:
        # (1 << 31) forces Extended 29-bit ID representation (REQUIRED for 0x1801550A)
        # REMOVED: (1 << 30) - This completely disables the CAN FD BRS high-speed bit
rate switch
        can_id_flags = 0x1801550A | (1 << 31)

        # 3. Inject directly into the positional driver hook
        can_send(can_id_flags, payload_bytes)

        # Print confirmation logs locally inside your Linux window
        hex_string = " ".join(f"{b:02X}" for b in payload_bytes)
        print(f"[CLASSICAL CAN TX] ID: 0x1801550A | Data: {hex_string}")

        # Increment counter stepping up sequentially
        tx_state['counter'] = (tx_state['counter'] + 1) % 256

    except Exception as err:
        print(f"[TX ERROR]: {err}")

# Wipe all old background timers completely
stop()

# Force kill any lingering heartbeat ghosts on the driver instance
try:
    __get_node__().heartbeat_publisher.stop()
except:
    pass

# Start the new Classical CAN loop executing at a 1-second interval
classical_task = periodic(1.0, send_classical_can_frame)
print("\n>>> ACTIVE: Classical CAN Extended frame loop running at 1Hz! <<<")


#NOTE: The above example sends a CAN frame whose source ID corresponds to node 10(0A),
to change this, change the last 2 bytes to the respective source node
```

iv. The serial console shows two types of decoded transfers: one is the decoded Node Status and the other is the Heartbeat of the Host Node.

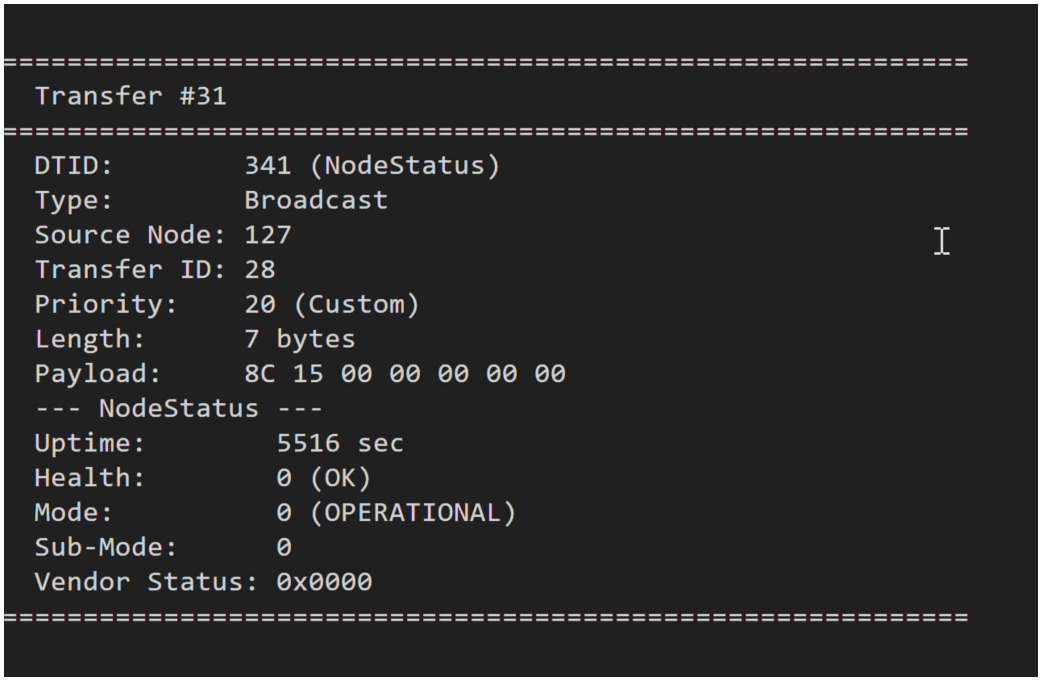

Figure 5-6. Decoded Node Status on RX Serial Port

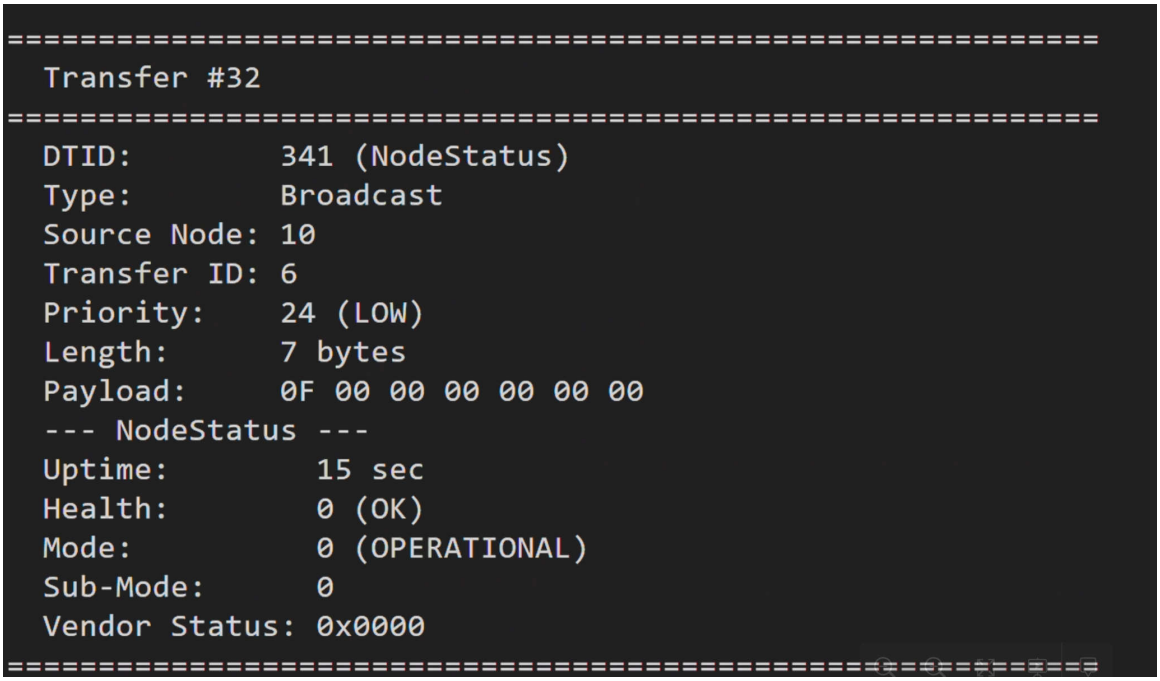

Figure 5-7. Heartbeat Node Status from Host

6 Key Pitfalls

- **DroneCAN GUI Tool not opening?** While setting up the CAN bus on the system, set the bus with the correct baud rate at the top. Verify the location using the command *sudo ip link show*. Find the correct baud rate in the syscfg file, open syscfg in text mode, and see the value of the variable targetBaudRate.
- **Is the PCAN device connected but no frames are visible?** If the PEAK-PCAN device shows green light, there is no issue in the transceiver connections, and the issue is with the application. If there is a constant red light, the CAN hardware connections are wrong. Check the PCAN configurations and connect accordingly.

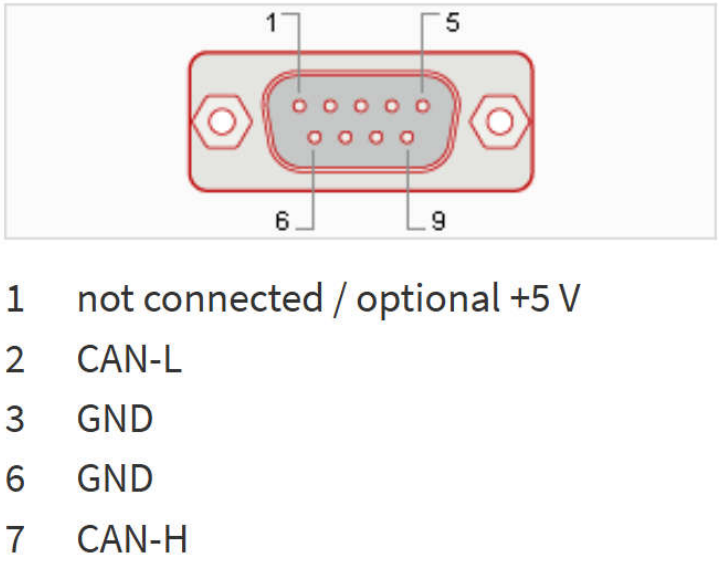

Figure 6-1. PCAN Connections

- **Frames are being sent from PCAN or the GUI Tool but are not getting received?** Verify that non-matching frames are also set to accept in Rx FIFO 0 in syscfg.

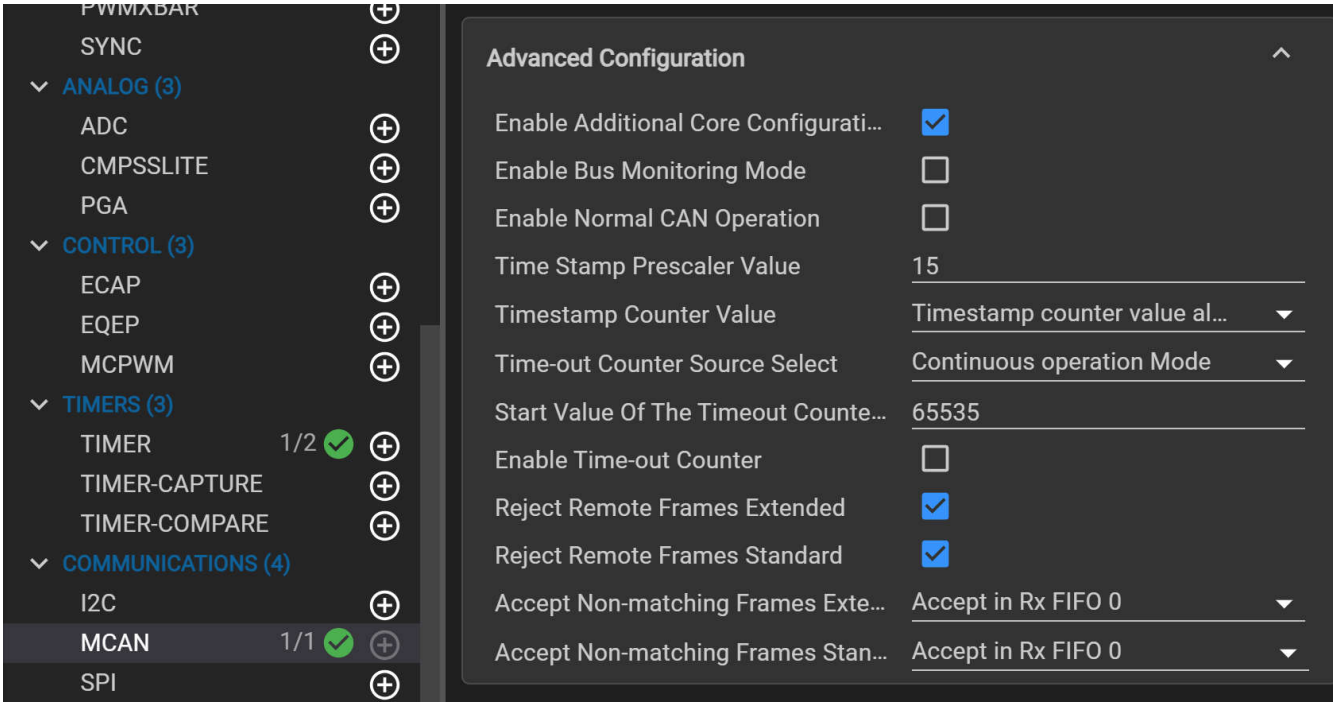

Figure 6-2. SysConfig View

- **Serial logs are not visible during RX Testing?** To view the RX console output, open a serial terminal on COM8 at 115200 baud before launching the application. The baud rate can be confirmed in the Syscfg UART configuration.

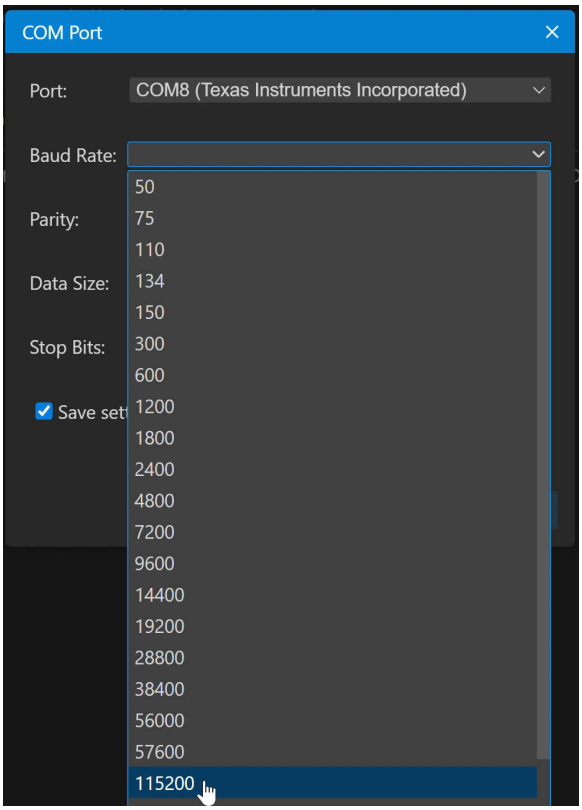

Figure 6-3. Serial Port and BaudRate Configuration

- **Transmitted frames not visible on PCAN?** Disconnect device, connect again, and check the bitrates. The arbitration bit rate is desiredNomRate. The data bit rate can be normally set as 1Mbps for CAN FD.
- **While using DroneCAN GUI Tool, are only heartbeat frames visible?** Typically while processing, the heartbeat frames get transmitted faster than a customized frame, and due to overlaps, the originally transmitted frame can get lost. If the originally transmitted frame is lost, include a segment in the python code to suppress the heartbeat frames in the Jupyter console of the DroneCAN GUI Tool and resend the packets.
- **DroneCAN GUI Tool is opening but bus monitor is not receiving any frames?** Make sure the device running DroneCAN has a node ID set. Enter a local node ID and check the box before testing the example.
- **Error canard_pool undefined error during build?** Check the linker command (.cmd) file whether the .canard_pool section has been defined correctly to start occupying memory in RAM.
- **The build is successful, but the application stops after one transmission and does not continue or give an error?** The application is waiting for acknowledgment of the successful transfer of the first frame. This issue is a connection issue, not an application issue. Check and tighten the transceiver connections again.
- **Application not running as expected after adding some more debug statements?** When the debug statements are written using the printf function, the statements interfere with the log statements running simultaneously and leads to an interrupt. Always use log statements for debugging any part of the code.
- **Application not running as expected after calling some more APIs from the libcanard library?** Make sure these are directly called from the application, do not edit the interface layer files. For example, canard_am13x.c and canard_am13x.h.
- **Project not building due to version conflict of dependencies?** Verify that the SysConfig version is 1.28.0 or higher and the CCS version is 21.0.0 or more.
- **Build failing with a file not found error ?**Make sure to include the libcanard source files under build and arm compiler to import from project root properly with the correct paths.

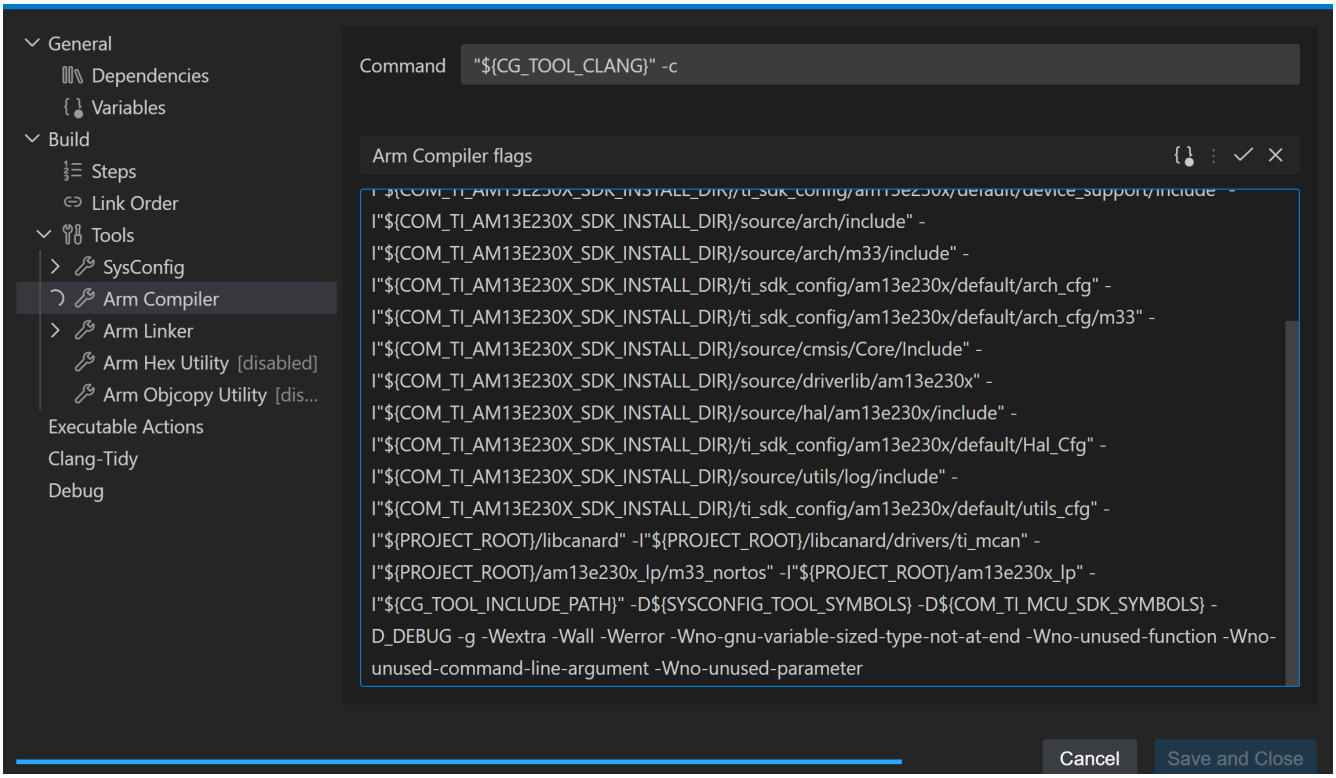

Figure 6-4. Compiler Include Paths Configuration

- **In case of encountering memory issues while transferring multiple frames?**Canard pool size must be defined in the application only; each multi-frame occupies 32 bytes of memory so set the memory variable accordingly.

7 Conclusion

This application note demonstrated the integration of the libcanard DroneCAN stack on the AM13E230x microcontroller using the on-chip MCAN peripheral. The canard_am13x interface layer provides a clean, reusable bridge between libcanard and TI DriverLib, enabling standards-compliant DroneCAN node development on AM13x-family devices with minimal integration effort. Combined with SysConfig-based peripheral configuration and the provided example applications, developers have a practical and ready-to-use foundation for building DroneCAN nodes in any AM13x-based design.

8 Tools and Software

Tools

E2E Forum	Texas Instruments support forums
Libcanard Github Repo	Support forum for TI AM13E230x

Software

About DroneCAN	Official DroneCAN website

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025