AN12842

Libuavcan S32K1 Driver over CAN-FD

For the UAVCAN Communication Protocol

Rev. 0 — June, 2020 Application Note

by: NXP Semiconductors

Contents

1 Introduction

Libuavcan is a lightweight C++ library for implementing the UAVCAN communication protocol in embedded systems. This application note covers the driver for the transport layer of the protocol over CAN-FD, which utilizes the FlexCAN peripheral available in the S32K1 family of microcontrollers, running at 1 Mbit/s and 4 Mbit/s in nominal and data phases, respectively.

The library is completely statically defined, all the parameters of an application are determined at compile time, avoiding dynamic memory allocation and reducing possible points of failure, and making verification of the system easier.

The last top layers of the protocol implementation for the latest specification, UAVCAN V1.0, are under development at the time this document was written.

The goal of this document is to demonstrate a programming example of the FlexCAN peripheral in particular implementation of the UAVCAN protocol. This for serving as a code reference for custom applications that wish to integrate the CAN-FD capabilities of the module.

2 Overview of UAVCAN

The acronym originally stood as a reference for CAN for Unmanned Aerial Vehicles, but due to the diverse possible applications of the protocol. It later became an acronym for Uncomplicated Application-level Vehicular Communication And Networking. It is an open communication protocol used in avionics, aerospace, robotics and rovers. It is the de-facto protocol in the widely used PX4 autopilot firmware for communications over CAN.

Figure 1. Logo of the communication protocol from which Libuavcan is implemented from

It offers reliable, deterministic and real-time capabilities over the already robust CAN protocol and other transports like UDP, also, no licensing nor approval of any kind is necessary for its implementation. The specification is freely available at the UAVCAN web page, and the source code for many reference implementations are accessible under the MIT license. For links to the specification and code repositories, refer to References.

2.1 Features of the protocol

The protocol's principal abstraction is based on the publisher/subscriber software design pattern. For example, in a robotic system, the sensors would be abstracted as publishers of data and at a predetermined rate, and actuators become subscribers of that information. This pattern is also found in additional robotics software such as ROS.

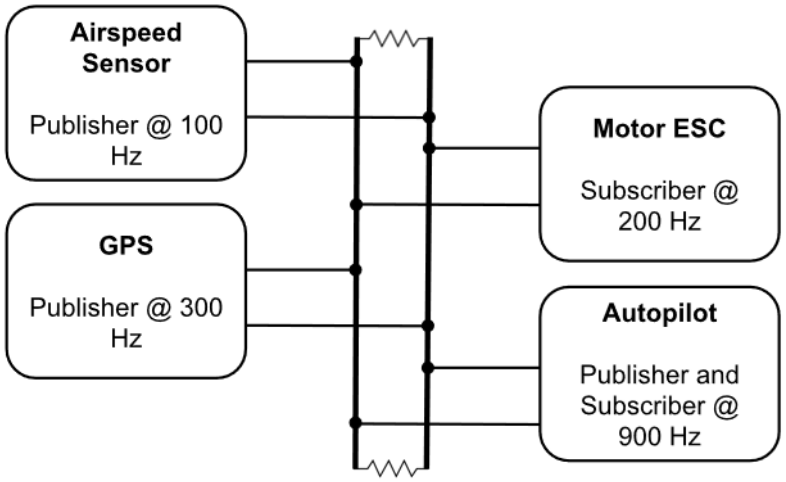

Figure 2. Example of UAVCAN nodes connected to a CAN bus

An intuitive analogy of the previously mentioned pattern is the traditional magazine subscription, in which the consumer decides to receive in a monthly basis the publication from a certain editorial. This keeps the subscriber from wasting time looking everyday into the mailbox for the magazine. In the opposite case, from getting outdated versions if the mailbox is revised in a longer than a monthly basis.

Returning to UAVCAN, this pattern offers many benefits, such as a superior scalability to the bus, increased performance and the ability to dynamically add nodes. It is the term used in the specification for the entities connected to the bus, e.g. A microcontroller and CAN transceiver pair. The request-response pattern is also available in the specification, for implementing client-server models of communication.

Another great feature of the protocol is the ability to describe in a language-independent way the data structures being transmitted and the services between the nodes, this with the use of a tool for automatic implementation generation, the DSDL (Data Structure Description Language). It consists of a command-line compiler that aids in saving development time and effort. It also reduces the likelihood of errors when implementing the protocol while increasing the portability across different platforms of the protocol.

3 Structure of the library's driver

3.1 Structure of the namespaces and folders

In libuavcan, the structure of the files reassembles the same hierarchy and nesting of namespaces that in the code, this gives a one to one relation between the folders in the project and the namespace scoping in the code.

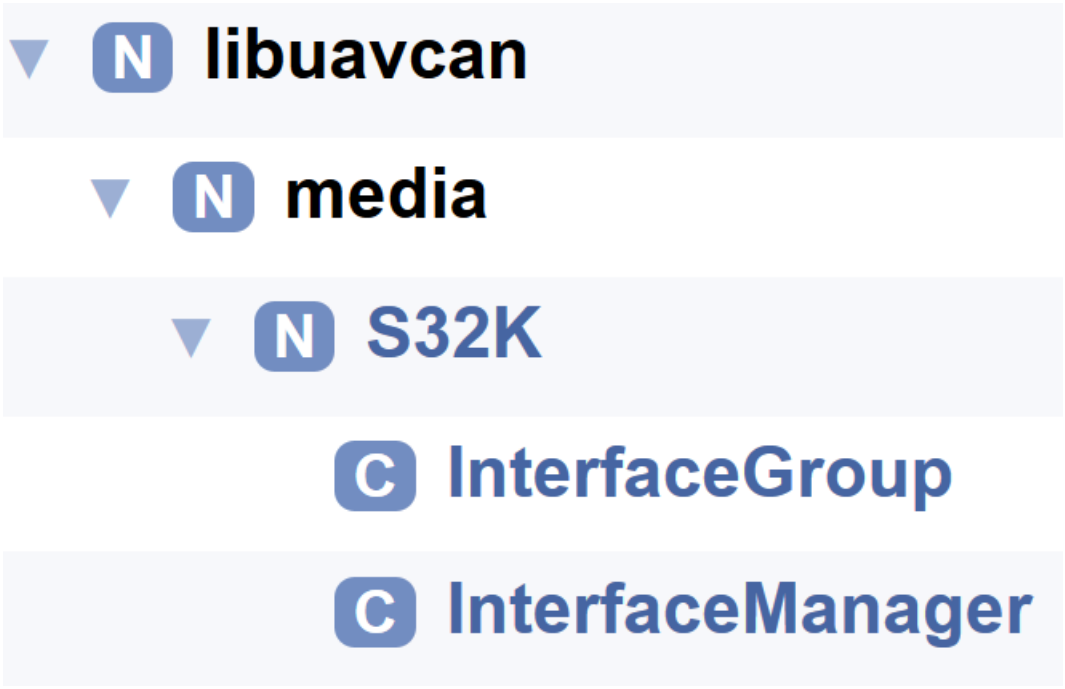

Figure 3. Namespace and classes structure of the library

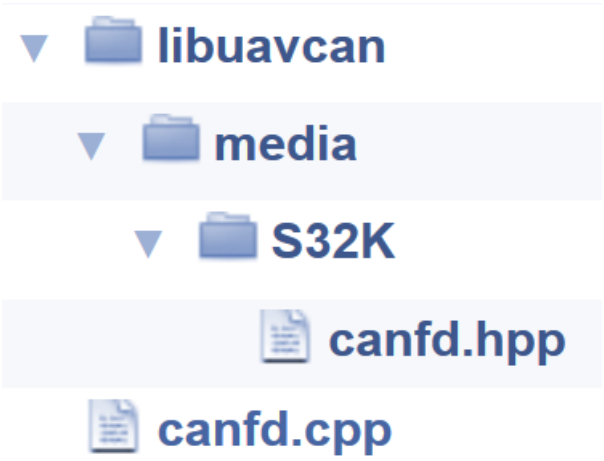

Figure 4. Folder structure

The template design pattern is used extensively in Libuavcan for defining the interface classes from which the driver's implementation classes inherit from. These classes organize in an object-oriented style the required details from the CAN specification, forming a robust and portable software architecture. The details of the template parameters are described in the next section.

3.2 Hierarchy of the library's classes

The classes of the library have the next arrangement, the implementation of the template abstract classes is inside the S32K namespace.

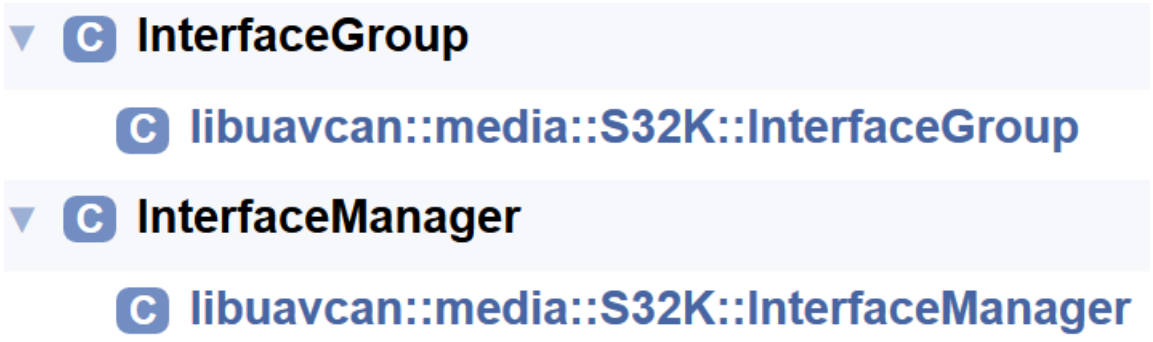

Figure 5. Overall hierarchy of the classes

The *InterfaceManager* class has the factory method *startInterfaceGroup* which outputs a ready-to-use *InterfaceGroup* pointer if the method exits successfully, this last class has the methods required for performing the services of the transport layer.

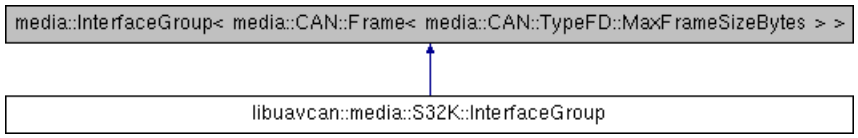

Figure 6. Inheritance diagram for InterfaceGroup class

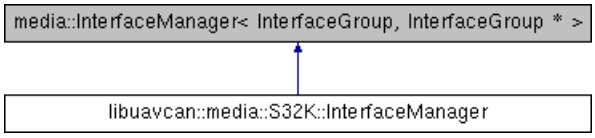

Figure 7. Inheritance diagram for the InterfaceManager class

The Libuavcan template abstract class *media::InterfaceGroup* has as a parameter the type of frame which is used for transmission and reception. In this case a CAN-FD frame, and the *media::InterfaceManager* class, has parameters the type of interface that is going to manage and a pointer to that same type, in this case the *S32K::InterfaceGroup* mentioned class. All the implementation of the driver is nested inside the *Libuavcan::media::S32K* namespace.

4 Methods of the transport layer from the library

- getInterfaceCount() : **libuavcan::media::S32K::InterfaceGroup**
- getMaxFrameFilters() : **libuavcan::media::S32K::InterfaceManager**
- read() : **libuavcan::media::S32K::InterfaceGroup**
- reconfigureFilters() : **libuavcan::media::S32K::InterfaceGroup**
- select() : **libuavcan::media::S32K::InterfaceGroup**
- startInterfaceGroup() : **libuavcan::media::S32K::InterfaceManager**
- stopInterfaceGroup() : **libuavcan::media::S32K::InterfaceManager**
- write() : **libuavcan::media::S32K::InterfaceGroup**

Figure 8. Complete outline of the functions provided by the library

For the complete list of parameters, refer to the library header file or source code at the GitHub repository link at References

4.1 Function's implementation details

The library uses a dequeue from the C++ Standard Template Library with a custom static allocator as a frame's RX FIFO. The reception mechanism is done through an interrupt triggered by the FlexCAN peripheral when a frame is received, this due to the lack of a dedicated RX FIFO by hardware nor a DMA triggering source to a DMA transfer inside the peripheral.

Figure 9. FIFO mechanism for frame reception implemented with a C++ STL dequeue

A frame is pushed into the FIFO in the interrupt and popped when the read() method is used, the frame capacity can be tuned with the *Frame_Capacity* constant located at the top of the source file in order to satisfy memory constrains. Each frame increase in the capacity adds 80 bytes to the .bss section. Any frame received when the dequeue is in its maximum capacity is discarded.

The driver supports redundant interfaces for increased reliability, this means that when multiple CAN-FD capable FlexCAN instances are available. These cannot be managed separately, but an individual frame can be sent or read via a single instance.

The received frames are time-stamped with a microsecond resolution 64-bit value that virtually never overflows, making the time stamp unique while the system is active. FlexCAN has a built-in timer that is only 16-bit in resolution, and this value is automatically saved into a frame in reception or transmission by hardware.

The following diagram shows the mechanism of obtaining an absolute 64-bit timestamp making use of the 16-bit FlexCAN timer and leveraging from a chained pair of 32-bit LPIT channels.

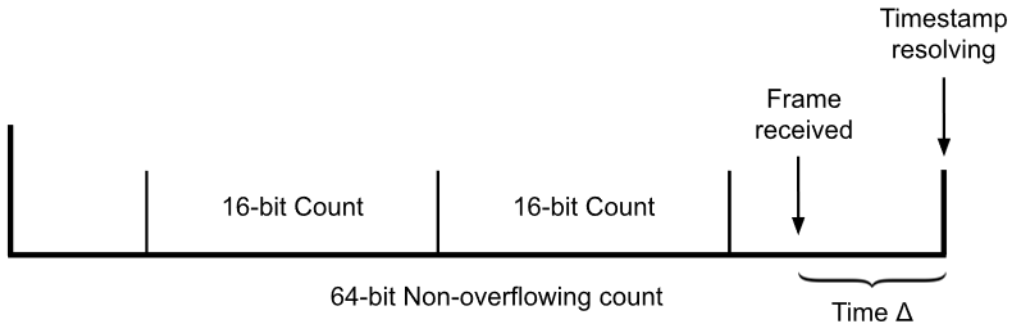

Figure 10. Sequence diagram of timestamp resolving

When a frame is received, the FlexCAN's Interrupt Service Routine (ISR) handler is executed, it performs the recollection of the received frames information such as the paylod and ID from the message buffers. Some time latency to be accounted for between the moment that the actual frame is received by the hardware, and a read from the 64-bit timestamping timer is performed inside the ISR to subsequently instantiate a frame object. This is accounted and solved by the next equation in order to reconstruct an absolute 64-bit timestamp from the overflowing 16-bit timer value that is saved in the message buffer.

$$Resolved\ timestamp = LPIT\ absolute\ time\ -\ FlexCAN\ \Delta$$

$$FlexCAN\ \Delta = FlexCAN\ timer\ -\ Message\ Buffer\ hardware\ timestamp$$

Figure 11. Equation 1

The LPIT absolute time and FlexCAN timer are read in sequence and assumed to be done at the same time. In the case that the read from the FlexCAN timer is a lower value than the timestamp read in the message buffer that received a frame, the order of subtraction is swapped. In this case a timer overflow has occurred between the frame reception and timestamp resolving procedure, thus is assumed that no more than a full 16-bit timer count have passed between the two time events for the computation to have a valid result.

4.1.1 getInterfaceCount()

This method returns the number of CAN-FD capable FlexCAN instances in the current S32K1.

Table 1. Number of CAN-FD interfaces available in S32K14x

Target MCU	CAN-FD instances
S32K142	1
S32K144	1
S32K146	2
S32K148	3

4.1.2 getMaxFrameFilters()

Returns the maximum number of ID-Mask pairs that are supported by the FlexCAN module, which is 5 since the 7 available CAN-FD message buffers are configured as follows:

Message buffers structure

Table 2. Configured function of available message buffer

MB	TX/RX
MB0	TX
MB1	TX
MB2-MB6	RX

4.1.3 read()

This method is used to pop a frame from the back of the RX FIFO dequeue, e.g. the oldest received frame, this method could be called from within a periodic interrupt with at a matched frequency as the planned rate that the node will be receiving incoming frames from other node in the bus.

4.1.4 select()

This function offers a POSIX service, it will block with a given timeout in microseconds until a frame is received, and optionally a message buffer becomes available for transmission, as shown in the next flow diagram.

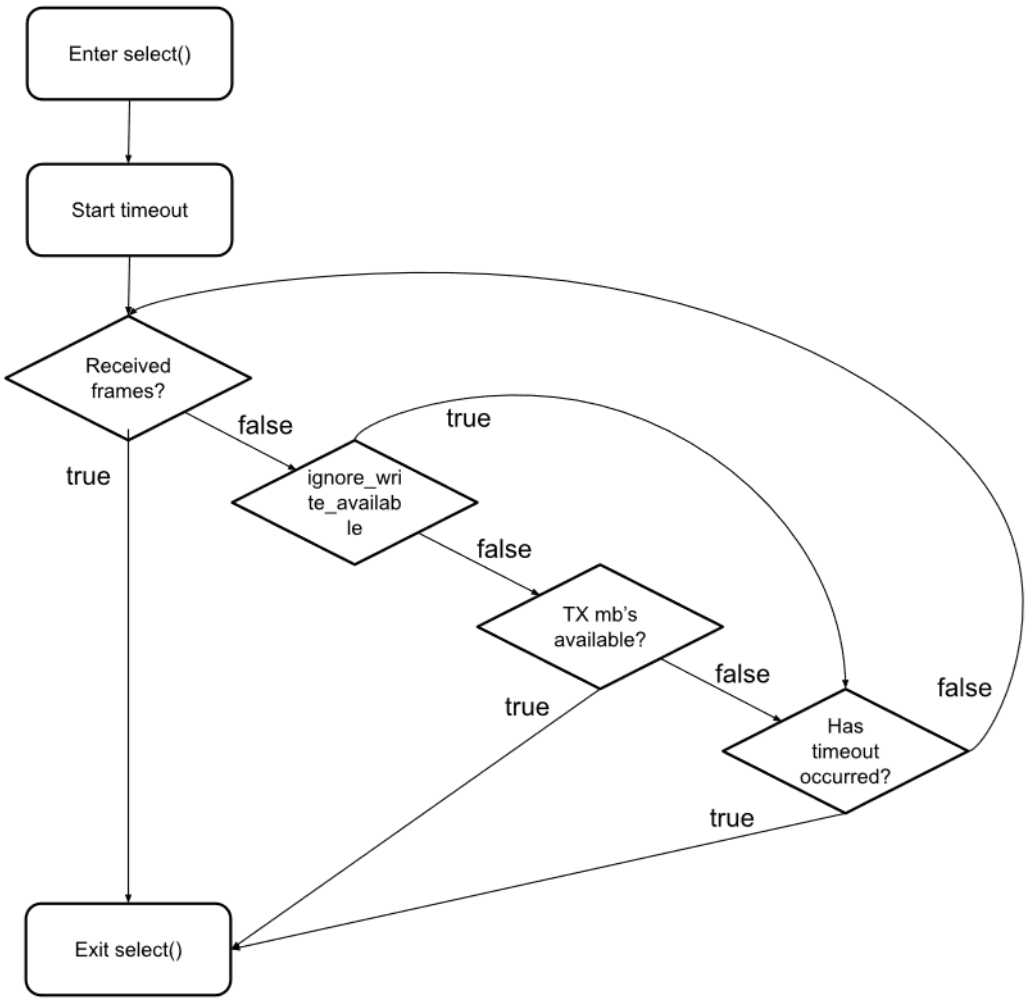

Figure 12. Select() service flow diagram

4.1.5 reconfigureFilters()

This function can be utilized for dynamically reconfiguring the ID-Mask pairs of the node, thus start receiving frames that match the new configuration

4.1.6 startInterfaceGroup()

This factory method performs the initial system initialization and returns as an output parameter a ready to use object InterfaceGroup object if a successful startup is performed. It also installs a list of the ID's of the node and the initial filtering in order to start receiving frames that pass throughout it. The next tables summarize the multiple configurations performed.

Table 3. Synchronous clock sources setup

Clocks in Normal RUN	Frequency set
CORE_CLK	80 MHz
SYS_CLK	80 MHz
BUS_CLK	40 MHz
FLASH_CLK	26.67 MHz

Table 4. Configuration of pins

PIN	Multiplexed function
PTE4	CAN0 RX
PTE5	CAN0 TX
PTA12	CAN1 RX
PTA13	CAN1 TX
PTB12	CAN2 RX
PTB13	CAN2 TX
PTE10	CAN0 STB[1] (GPIO)
PTE11	CAN1 STB[1] (GPIO)

1. This setup is only required when using the TJA1044 transceiver.

Table 5. Peripherals required

Module	Resource
FlexCAN	All message buffers from all CAN-FD capable instances available; the frame's reception interrupt handler is enabled, and its priority is set to default, could be modified to meet application-specific requirements
LPIT	Channels 0,1 and 2; the 3rd channel is available

Table 6. Asynchronous clocks configuration

Asynch peripheral clock source	Divider Value
SPLL2	1
1. The rest of the asynchronous clock sources are left available and left unset.	

4.1.7 stopInterfaceGroup()

This method performs the release and deinitialization of the peripherals needed by the driver, disables all the CAN-FD capable instances that were set by the previous described method; also resets and disables all the channels from the LPIT module.

It can be used for power saving scenarios where it is desired to disable and leave the resources previously mentioned in a default state before going to a sleep or a low-power mode.

Due to FlexCAN feeding from SYS_CLK clock source in the current implementation, it is dependent from changes in the clock configurations different from the Normal RUN setup and it won't function at the designed 4 Mbit/s baud rate in other presets.

4.1.8 write()

This method sends a frame object that has the ID and payload attributes after searching throughout an available message buffer in a specified instance, if there are no available MB's at the time, the function returns immediately a bufferFull status.

5 Usage example

Refer to References for the link to a GitHub repository with the latest full demo that is directly importable into S32 Design Studio for flashing into an UCANS32K146 board.

```
#if defined(NODE_A)

/* ID for the current UAVCAN node */
constexpr std:: uint32_t  Node_ID = 0xC0C0A;
/* ID of the frame to transmit */
constexpr std:: uint32_t  demo_FrameID = 0xC0FFE;

#elif defined(NODE_B)

/* ID and for the current UAVCAN node */
constexpr std::uint32_t Node_ID = 0xC0FFE;
/* ID of the frame to transmit */
constexpr std::uint32_t demo_FrameID = 0xC0C0A;

#endif

/* All care bits mask for frame filtering */
constexpr std:: uint32_t  Node_Mask = 0xFFFFF;
/* Number of ID's that the node will filter in */
constexpr std:: size_t  Node_Filters_Count = 1u;
/* Frames transmitted each time */
constexpr std:: size_t  Node_Frame_Count = 1u;
/* Interface instance used in this demo */
constexpr std:: size_t  First_Instance = 1u;
/* Size of the payload in bytes of the frame to be transmitted */
constexpr std:: uint16_t  payload_length = libuavcan::media::S32K:: InterfaceGroup ::
FrameType ::MTUBytes;

intmain()
```

```
{
/* Frame's Data Length Code in function of it's payload length in bytes */
libuavcan::media::CAN::FrameDLC demo_DLC =
libuavcan::media::S32K:: InterfaceGroup :: FrameType ::lengthToDlc (payload_length);

/* 64-byte payload that will be exchanged between the nodes */
std:: uint8_t  demo_payload[payload_length];

/* Initial value of the frame's payload */
std::fill(demo_payload,demo_payload+payload_length,0);

/* Instantiate factory object */
libuavcan::media::S32K:: InterfaceManager  demo_Manager;

/* Create pointer to Interface object */
libuavcan::media::S32K:: InterfaceGroup * demo_InterfacePtr;

/* Create a frame that will reach NODE_B ID */
libuavcan ::media::S32K:: InterfaceGroup :: FrameType
bouncing_frame_obj(demo_FrameID,demo_payload,demo_DLC);

/* Array of frames to transmit (current implementation supports 1) */
libuavcan ::media::S32K:: InterfaceGroup :: FrameType  bouncing_frame[Node_Frame_Count] =
{bouncing_frame_obj};

/* Instantiate the filter object that the current node will apply to receiving frames */
libuavcan ::media::S32K:: InterfaceGroup :: FrameType :: Filter  demo_Filter(Node_ID,Node_Mask);

std:: uint32_t  rx_msg_count = 0;

/* Status variable for sequence control */
libuavcan ::Result status;

/* Initialize the node with the previously defined filtering using factory method */
status = demo_Manager.startInterfaceGroup(&demo_Filter,Node_Filters_Count,demo_InterfacePtr);

/* Initialize an LED for toggling each time 1000 frames are received */
greenLED_init();

/* Node A kickstarts */
#ifdef NODE_A
    std:: size_t  frames_wrote = 0;
   if ( libuavcan::isSuccess(status) )
    {
        demo_InterfacePtr->write(First_Instance,bouncing_frame,Node_Frame_Count,frames_wrote);
    }
#endif

/* Super-loop for retransmission of the frame */
for(;;)
{
    std:: size_t  frames_read = 0;
    if ( libuavcan::isSuccess(status) )
    {
        status = demo_InterfacePtr->read(First_Instance, bouncing_frame, frames_read);
    }

    /* frames_read should be 1 from he previous read method if a frame was received */
    if(frames_read)
    {
```

```
        /* Increment receive msg  counter */
        rx_msg_count++;
        if ( rx_msg_count == 1000 )
        {
            PTD-> PTOR  |= 1<<16;    /* Toggle green LED*/
            rx_msg_count = 0;        /* Reset th  counter of received frames */
        }

        std:: size_t  frames_wrote;

        /* Swap the frame's ID for returning it back to the sender */
        bouncing_frame[0]. id  = demo_FrameID;

        /* The frame is sent back bt  with the payload   */
        payload_bounceADD(bouncing_frame[0]. data );

        /* Perform transmission */
        if ( libuavcan::isSuccess(status) )
        {
            status = demo_InterfacePtr->write(First_Instance,
bouncing_frame,
                                              Node_Frame_Count, frames_wrote);
        }
    }
}
}
```

A References

- Libuavcan library GitHub repository: https://github.com/UAVCAN/libuavcan
- S32K1 driver for Libuavcan V1: https://github.com/UAVCAN/platform_specific_components
- UAVCAN specification: https://uavcan.org/specification
- Demo of the library for S32 Design Studio: https://github.com/noxuz/libuavcan_demo
- König Hartmut. (2012). Protocol engineering. Berlin: Springer.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: June, 2020

Document identifier: AN12842

arm