AN12842

S32K1 CAN-FD Libuavcan驱动程序

用于UAVCAN通信协议

Rev. 0 — June, 2020 应用笔记

作者：恩智浦半导体

目录

1 简介

LIBUAVCAN是一个用于在嵌入式系统中实现UAVCAN通信协议的轻量级C++库。本应用笔记涵盖CAN-FD协议传输层的驱动程序，该协议利用S32K1系列微控制器中可用的FlexCAN外围设备，分别在标称(nominal)段和数据(data)段以1 Mbit/s和4 Mbit/s的速度运行。

该库是完全静态定义的，应用程序的所有参数都是在编译时确定的，避免了动态内存分配，减少了可能的故障点，使系统的验证更容易。

在编写本文档时，最新UAVCAN V1.0协议顶层的实现正在开发中。

本文档的目标是演示FlexCAN外设的编程示例，特别是UAVCAN协议的实现。作为希望在模块应用程序中集成CAN-FD功能的代码参考。

2 UAVCAN概览

UAVCAN最初指应用于无人机的CAN，但由于该协议可能有多种应用。它后来成为简单应用级车辆通信和网络的首字母缩略词，是一种用于航空电子、航空航天、机器人和漫游车的开放式通信协议。它是广泛使用的PX4自动驾驶仪固件中通过CAN进行通信的事实协议。

图 1. 实现Libuavcan的通信协议的徽标

UAVCAN在已经强大的CAN协议和其他传输协议（如UDP）上提供可靠、确定和实时的能力，而且它的实施不需要任何形式的许可或批准。该规范在UAVCAN网页上免费提供，许多参考实现的源代码可以在MIT许可下访问。有关规范和代码存储库的链接，请参考 References.

2.1 协议特征

协议的主要抽象基于发布者/订阅者软件设计模式。例如，在机器人系统中，传感器将被抽象为数据发布者，并以预定的速率发布，而执行器将成为该信息的用户。这种模式也可以在其他机器人软件（如ROS）中找到。

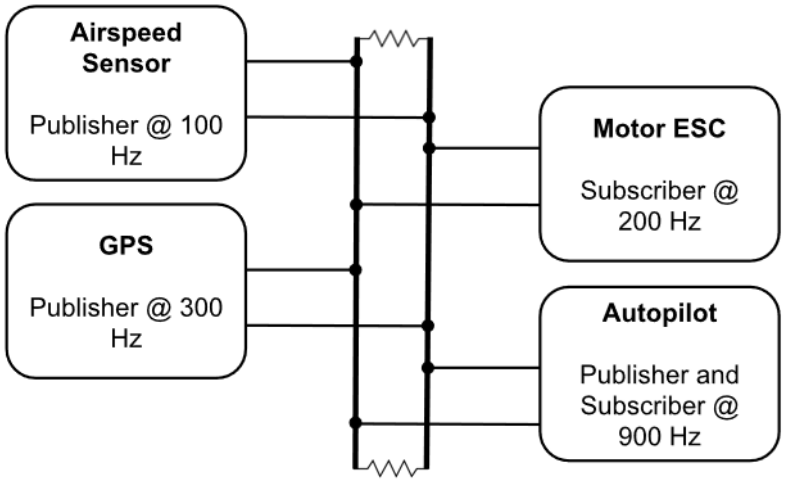

图 2. 连接到CAN总线的UAVCAN节点示例

上述模式的一个直观类比是传统的杂志订阅，消费者决定每月从某处收到出版物。这样可以避免订户浪费时间每天查看杂志邮箱。在相反的情况下，如果邮箱的修改时间超过每月一次，则可能会收到过时的版本。

回到UAVCAN，这种模式提供了许多好处，例如总线的优异可扩展性、提高的性能和动态添加节点的能力。它是本规范中用于连接到总线的实体的术语，例如微控制器和CAN收发器对。规范中还提供了请求-响应模式，用于实现客户机-服务器通信模型。

该协议的另一个重要特性是能够以独立于语言的方式描述节点之间传输的数据结构和服务，这是通过使用自动实现生成工具DSDL（数据结构描述语言）实现的。它由一个命令行编译器组成，有助于节省开发时间和精力。它还降低了实现协议时出错的可能性，同时提高了协议在不同平台之间的可移植性。

3 库驱动程序的结构

3.1 名称空间和文件夹的结构

在libuavcan中，文件的结构重组了与代码中相同的层次结构和名称空间嵌套，这使项目中的文件夹和代码中的名称空间范围之间形成了一对一的关系。

图 3. 库的名称空间和类结构

图 4. 文件夹结构

模板设计模式在Libuavcan中广泛使用，用于定义驱动程序实现类从中继承的接口类。这些类以面向对象的方式组织来自CAN规范的所需细节，形成鲁棒的、可移植的软件架构。模板参数的详细信息将在下一节中介绍。

3.2 库的类的层次结构

库的类有下一个安排，模板抽象类的实现在S32K的命名空间内。

图5. 类的总体层次结构

*InterfaceManager*类具有factory方法*startInterfaceGroup*，该方法输出一个随时可用的*InterfaceGroup*指针。如果该方法成功退出，则最后一个类具有执行传输层服务所需的方法。

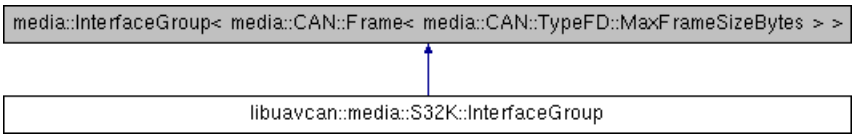

图6. InterfaceGroup类的继承图

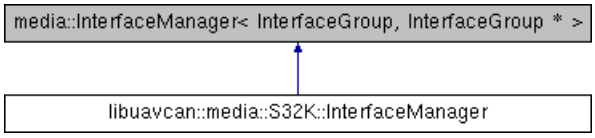

图7. InterfaceManager类的继承图

Libuavcan模板抽象类*media::InterfaceGroup*将用于传输和接收的帧类型作为参数。在本例中，CAN-FD帧和*media::InterfaceManager*类具有要管理的接口类型的参数和指向同一类型的指针，在本例中为*S32K::InterfaceGroup*类。驱动程序的所有实现都嵌套在*Libuavcan::media::S32K*命名空间中。

4 库中传输层的方法

- 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

图 8.库提供的功能的完整概述

有关参数的完整列表，请参考References中GitHub存储库链接的库头文件或源代码

4.1 功能的实施细节

库使用C++标准模板库的队列，并将一个自定义静态分配器作为一个数据帧的RX FIFO。接收机制是通过FlexCAN外设在接收帧时触发的中断完成的，这是由于硬件没有专用的RX FIFO，外设内部也没有DMA传输的DMA触发源。

图 9. 用C++ STL DEFALE实现帧接收的FIFO机制

一个帧在中断中被推入FIFO并在使用read( )时弹出，帧容量可以通过位于源文件顶部的frame_capacity常量进行调整，以满足内存限制。容量每增加一帧，就向.bss部分添加80个字节。当队列处于其最大容量时接收到的任何帧都将被丢弃。

驱动程序支持冗余接口以提高可靠性，这意味着当需要多个CAN-FD时，FlexCAN接口是可用的。这些接口不能单独管理，但单个帧可以通过单个接口发送或读取。

接收到的帧带有微秒分辨率64位值的时间戳，该值几乎不会溢出，从而在系统运行时时间戳是唯一的。FlexCAN有一个分辨率16位的内置计时器，硬件在接收或传输时自动保存计时器的值到帧中。

下图显示了利用16位FlexCAN定时器以及链式32位LPIT通道获得绝对64位时间戳的机制。

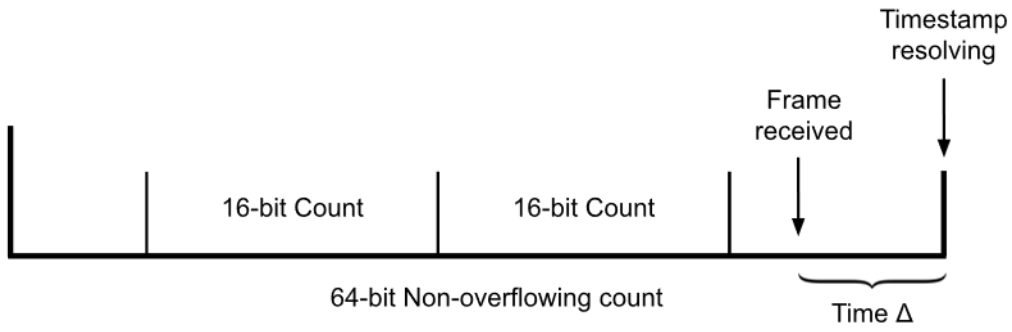

图 10. 时间戳解析的序列图

当接收到一个帧时，FlexCAN的中断服务例程（ISR）处理程序被执行，它从消息缓冲区中获取接收到的帧信息，如payload和ID。从硬件接收到实际帧到在ISR内执行从64位时间戳定时器读取数据以随后实例化一个帧对象之间需要考虑一些时间延迟。通过下一个等式对此进行计算和求解，以便从保存在消息缓冲区中的溢出16位计时器值中重建一个绝对64位时间戳.

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

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

图 11. 方程式 1

LPIT绝对时间和FlexCAN定时器按顺序读取，并假定同时执行。如果从FlexCAN定时器读取的值低于接收帧的消息缓冲区中读取的时间戳，减法顺序将被交换。在这种情况下，在帧接收和时间戳解析过程之间发生了计时器溢出，因此假设在两个时间事件之间传递的计时器计数不超过完整的16位，以使计算具有一个有效的结果。

4.1.1 getInterfaceCount()

此方法返回当前S32K1中支持CAN-FD的FlexCAN实例数。.

Target MCU	CAN-FDinstances
S32K142	1
S32K144	1
S32K146	2
S32K148	3

4.1.2 getMaxFrameFilters()

返回FlexCAN模块支持的最大ID-Mask对数，即5个，因为7个可用CAN-FD消息缓冲区配置如下：

消息缓冲区结构

表 2. 可用消息缓冲区的配置功能

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

4.1.3 read()

此方法用于从RX FIFO出列的后面弹出一个帧，例如最早接收的帧。此方法可在一个周期中断内被调用，频率与节点从总线中其他节点接收传入帧的计划速率一致。

4.1.4 select()

此函数提供POSIX服务，它将以给定的超时时间（微秒为单位）进行阻塞，直到接收到一个帧，并且可以选择使用一个消息缓冲区进行传输，如下流程图所示。

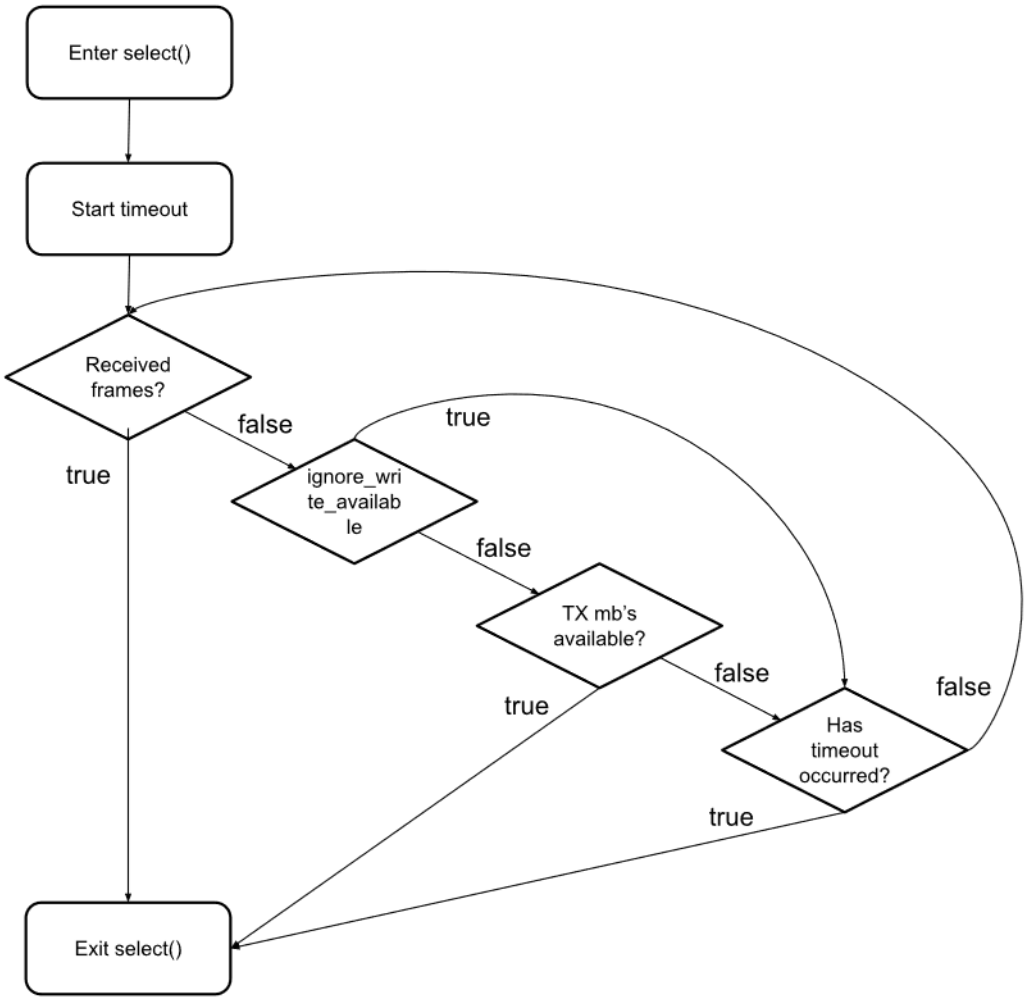

图 12. Select（）服务流程图

4.1.5 reconfigureFilters()

此功能可用于动态重新配置节点的ID-Mask对，从而开始接收匹配新配置的帧

4.1.6 startInterfaceGroup()

此方法执行系统初始化，如果执行成功，则返回一个可使用的 InterfaceGroup对象作为输出参数。它还安装节点ID列表和初始过滤功能，以便开始接收通过该节点的帧。下表总结了执行的多种配置。

表 3. 同步时钟源设置

正常运行的时钟	频率设置
CORE_CLK	80 MHz
SYS_CLK	80 MHz
BUS_CLK	40 MHz
FLASH_CLK	26.67 MHz

表 4. 引脚配置

引脚	多路复用功能
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. 仅当使用TJA1044收发器时才需要此设置。

表 5. 所需的外设

模块	资源
FlexCAN	所有CAN-FD可用实例的所有消息缓冲区可用；帧的接收中断处理程序已启用，其优先级设置为默认值，可以修改以满足特定于应用程序的要求
LPIT	通道0、1和2；第三通道可用

表6. 异步时钟配置

异步外围时钟源	除法器值
SPLL2	1
1. 其余的异步时钟源保持可用且未设置。	

4.1.7 stopInterfaceGroup()

此方法执行驱动程序所需外围设备的释放和去初始化，禁用前面描述的方法设置的所有支持CAN-FD的实例；还重置和禁用LPIT模块的所有通道。

它可以用于节能场景，既在进入睡眠或低功耗模式之前，需要禁用前面提到的资源并将其保持在默认状态。

由于在当前实施中FlexCAN时钟源来自SYS_CLK，它依赖于不同于正常RUN模式设置的时钟配置变化，在其他预设中它不会以设计的4 Mbit/s波特率工作。

4.1.8 write()

在某个特定实例中搜索所有可用的消息缓冲区后，发送具有ID和payload属性的帧对象。如果当时没有可用MB，函数将立即返回bufferFull状态。

5 使用示例

请参阅参考资料，以获取包含最新样例的GitHub存储库链接。样例可直接导入S32 Design Studio，以便烧写到UCANS32K146板中。

```
#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 参考文献

- 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