1. Basic tutorial

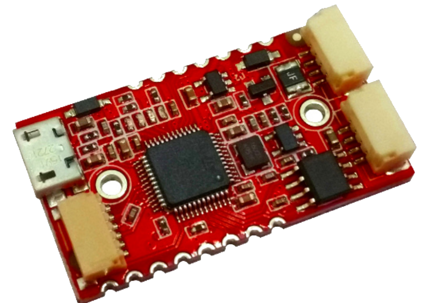
Introduction

In this article we will show how to make a basic UAVCAN node and explore it in the UAVCAN GUI Tool using the Zubax Babel hardware.

From the software point of view, we will keep things as simple as possible. We will use only the libcanard library, which is a very lightweight UAVCAN implementation, and the stm32 standard peripheral library for the stm32f37x family of microcontrollers. Usage of different HALs and any type of RTOS is avoided on purpose, as this is a very basic example.

> **OBSOLETE**
> THE MATERIALS PRESENTED HERE ARE BASED ON **UAVCAN v0** – AN EXPERIMENTAL VERSION OF THE PROTOCOL THAT IS NOW OBSOLETE. TO LEARN ABOUT THE LONG-TERM STABLE REVISION OF THE PROTOCOL, PROCEED TO UAVCAN.ORG.

Goal



So our goal is to make Babel act as a UAVCAN node, then check if it can be seen in the UAVCAN GUI Tool, explore the raw CAN data using the Bus monitor, and basically get acquainted with UAVCAN in general and libcanard in particular.

A couple of words about UAVCAN. A UAVCAN bus is obviously a CAN bus where at least two nodes should present. In our case one node is a Zubax Babel and the other node is a UAVCAN GUI Tool. We will need the second Zubax Babel or any other SLCAN/LAWICEL compatible converter to connect the UAVCAN GUI Tool to our CAN bus.

Important note. CAN bus in general needs termination resistors connected on both sides of the line. If the line is very short (10-20 cm), only one resistor may be sufficient. onveniently, Zubax Babel has a software-programmable termination resistor. If the CAN bus seems to be not working for some reason, please check if the termination resistor is activated (it can be activated using the UAVCAN GUI Tool itself).

Implementation

The general hardware initialization must be performed in the first order, as the STM32 driver shipped with libcanard configures the CAN peripheral only and does not configure clocking and gpio. The `hwInit()` function is trivial and is left out of this text.

Some global definitions must be presented in the project in some way. Here they are:

Definitions

```
USE_STDPERIPH_DRIVER
STM32F37X
HSE_VALUE=16000000
HSE_STARTUP_TIMEOUT=0x1000
```

Lets begin with initialization of libcanard.

Libcanard initialization

```
static CanardInstance g_canard;               // The library instance
static uint8_t g_canard_memory_pool[1024];  // Arena for memory
allocation, used by the library

static void swInit(void)
{
    CanardSTM32CANTimings timings;
    int result =
canardSTM32ComputeCANTimings(RCC_Clocks.PCLK1_Frequency, 1000000,
&timings);
    if (result)
    {
        __ASM volatile("BKPT #01");
    }
    result = canardSTM32Init(&timings, CanardSTM32IfaceModeNormal);
    if (result)
    {
        __ASM volatile("BKPT #01");
    }

    canardInit(&g_canard,                         // Uninitialized
library instance
               g_canard_memory_pool,              // Raw memory chunk
used for dynamic allocation
               sizeof(g_canard_memory_pool),      // Size of the above,
in bytes
               onTransferReceived,                // Callback, see
CanardOnTransferReception
               shouldAcceptTransfer,              // Callback, see
CanardShouldAcceptTransfer
               NULL);

    canardSetLocalNodeID(&g_canard, 100);
}
```

`canardSTM32Init` and `canardSTM32ComputeCANTimings` are stm32-specific driver functions shipped with libcanard intended to simplify the CAN peripheral setup procedure.

Important note: the libcanard's STM32 CAN driver does not use interrupts or DMA. Its up to user to decide if CAN interrupts are needed and to implement them if necessary.

Libcanard is a static library and does not use heap, so it needs some memory pool for operation which the user must give it manually. That is `uint8_t g_canard_memory_pool[1024]`. Libcanard also needs two functions that must be implemented by the user:

- `shouldAcceptTransfer` - this callback is called every time a transfer is received to determine if it should be passed further to the library or ignored. Here we should filter out all messages that are not needed for our particular task.
- `onTransferReceived` - this callback is called every time a transfer is received and accepted in `shouldAcceptTransfer`. It is a good idea to put incoming data handlers here.

Here are these functions:

shouldAcceptTransfer

```
static bool shouldAcceptTransfer(const CanardInstance* ins,
                                 uint64_t* out_data_type_signature,
                                 uint16_t data_type_id,
                                 CanardTransferType transfer_type,
                                 uint8_t source_node_id)
{
    if ((transfer_type == CanardTransferTypeRequest) &&
        (data_type_id == UAVCAN_GET_NODE_INFO_DATA_TYPE_ID))
    {
        *out_data_type_signature =
UAVCAN_GET_NODE_INFO_DATA_TYPE_SIGNATURE;
        return true;
    }

    return false;
}
```

onTransferReceived

```
static void onTransferReceived(CanardInstance* ins, CanardRxTransfer*
transfer)
{
    if ((transfer->transfer_type == CanardTransferTypeRequest) &&
    (transfer->data_type_id == UAVCAN_GET_NODE_INFO_DATA_TYPE_ID))
    {
        canardGetNodeInfoHandle(transfer);
    }
}
```

As should be obvious from `shouldAcceptTransfer`, our node will accept only one type of transfers:

- `UAVCAN_GET_NODE_INFO_DATA_TYPE_ID` - this is a request that the UAVCAN GUI Tool sends to all nodes that it discovers to get some data like name, software version, hardware version and so on from them. In fact, this is optional, but supporting this type of service is a good idea.

Besides receiving UAVCAN messages, each node must also broadcast at least one type of messages periodically - NodeStatus (once in every 100-1000 ms should be fine). So let's make a function for that.

canard_service

```
#define CANARD_SPIN_PERIOD    1000

static void spinCanard(void)
{
    static uint32_t spin_time = 0;
    if (getUptime() < spin_time + CANARD_SPIN_PERIOD) { return; }  //
rate limiting
    spin_time = getUptime();
    gpioToggle(GPIOE, GPIO_Pin_8);                           // some
indication

    uint8_t buffer[UAVCAN_NODE_STATUS_MESSAGE_SIZE];
    static uint8_t transfer_id = 0;                          // This
variable MUST BE STATIC; refer to the libcanard documentation for the
background

    makeNodeStatusMessage(buffer);

    canardBroadcast(&canard,
                    UAVCAN_NODE_STATUS_DATA_TYPE_SIGNATURE,
        UAVCAN_NODE_STATUS_DATA_TYPE_ID,
           &transfer_id,
           CANARD_TRANSFER_PRIORITY_LOW,
        buffer,
        UAVCAN_NODE_STATUS_MESSAGE_SIZE);
}
```

To make a node status message we will have to compose it manually. For that we will need three values:

- Uptime in seconds.
- Node health. Our node will always be 100% healthy.
- Node mode. Our node will always be in the operational mode.

These values have to be encoded according to `NodeStatus` message description:

makeNodeStatusMessage

```
static void makeNodeStatusMessage(uint8_t
buffer[UAVCAN_NODE_STATUS_MESSAGE_SIZE])
{
 const uint8_t node_health = UAVCAN_NODE_HEALTH_OK;
 const uint8_t node_mode   = UAVCAN_NODE_MODE_OPERATIONAL;
    memset(buffer, 0, UAVCAN_NODE_STATUS_MESSAGE_SIZE);
    const uint32_t uptime_sec = get_uptime() / 1000;
    canardEncodeScalar(buffer,  0, 32, &uptime_sec);
    canardEncodeScalar(buffer, 32,  2, &node_health);
    canardEncodeScalar(buffer, 34,  3, &node_mode);
}
```

When the UAVCAN GUI Tool receives this message for the first time, it will attempt to get more info about the new node, so we also have to implement a handler that will form a `GetNodeInfo` response and send it back to the requesting node (client):

canard_get_node_info_handle

```
#define APP_VERSION_MAJOR        99
#define APP_VERSION_MINOR        99
#define APP_NODE_NAME         "com.zubax.babel.demo"
#define GIT_HASH        0xBADC0FFE            // Normally this should be
queried from the VCS when building the firmware
#define UAVCAN_GET_NODE_INFO_RESPONSE_MAX_SIZE  ((3015 + 7) / 8)

static uint16_t makeNodeInfoMessage(uint8_t
buffer[UAVCAN_GET_NODE_INFO_RESPONSE_MAX_SIZE])
{
    memset(buffer, 0, UAVCAN_GET_NODE_INFO_RESPONSE_MAX_SIZE);
    makeNodeStatusMessage(buffer);

    buffer[7] = APP_VERSION_MAJOR;
    buffer[8] = APP_VERSION_MINOR;
    buffer[9] = 1;  // Optional field flags, VCS commit is set
    const uint32_t git_hash = GIT_HASH;
    canardEncodeScalar(buffer, 80, 32, &git_hash);

    readUniqueID(&buffer[24]);
    const size_t name_len = strlen(APP_NODE_NAME);
    memcpy(&buffer[41], APP_NODE_NAME, name_len);
    return 41 + name_len ;
}

static void getNodeInfoHandleCanard(CanardRxTransfer* transfer)
{
    uint8_t buffer[UAVCAN_GET_NODE_INFO_RESPONSE_MAX_SIZE];
    memset(buffer, 0, UAVCAN_GET_NODE_INFO_RESPONSE_MAX_SIZE);
    const uint16_t len = makeNodeInfoMessage(buffer);
    int result = canardRequestOrRespond(&g_canard,
                                        transfer->source_node_id,

UAVCAN_GET_NODE_INFO_DATA_TYPE_SIGNATURE,

UAVCAN_GET_NODE_INFO_DATA_TYPE_ID,
                                        &transfer->transfer_id,
                                        transfer->priority,
                                        CanardResponse,
                                        &buffer[0],
                                        (uint16_t)len);
    if (result < 0)
    {
        // TODO: handle the error
    }
}
```

App architecture

As libcanard does not use any interrupts, and because of our intention to keep everything simple, the application will be organised as an ordinary cycle:

main

```
int main(void)
{
    /*!< At this stage the microcontroller's clock setting is already
configured,
        this is done through SystemInit() function which is called from
startup
        file (startup_stm32f37x.s) before branching to the application's
main().
        To reconfigure the default setting of SystemInit() function,
refer to the
        system_stm32f37x.c file */
    RCC_GetClocksFreq(&RCC_Clocks);         // To make sure RCC is
initialised properly
    hwInit();
    swInit();
    SysTick_Config(SystemCoreClock / 1000); // To make systick event
happen every 1 ms
    while(1)
    {
        sendCanard();
        receiveCanard();
        spinCanard();
    }
}
```

The only interrupt used in the application is the SysTick interrupt for uptime counter with 1 ms resolution.

systick_isr

```
void systickISR(void);
static uint32_t getUptime(void);
static uint32_t  g_uptime = 0;

static uint32_t getUptime(void)
{
    return g_uptime;     // Atomic read, locking is not needed
}

void systickISR(void)
{
    g_uptime++;
}
```

As libcanard does not use any interrupts, it is up to the user to decide when and how to receive and transmit UAVCAN messages. In this application we will constantly poll if any message was received by the MCU's CAN peripheral and process it. We will also poll if the library has any new messages to transmit and manually extract them from the library and pass them to the CAN transmitter.

canard_sender

```
static void sendCanard(void)
{
    const CanardCANFrame* txf = canardPeekTxQueue(&g_canard);
    while(txf)
    {
        const int tx_res = canardSTM32Transmit(txf);
        if (tx_res < 0)         // Failure - drop the frame and report
        {
            __ASM volatile("BKPT #01");   // TODO: handle the error
properly
        }
        if(tx_res > 0)
        {
            canardPopTxQueue(&g_canard);
        }
        txf = canardPeekTxQueue(&g_canard);
    }
}
```

canard_receive

```
static void receiveCanard(void)
{
    CanardCANFrame rx_frame;
    int res = canardSTM32Receive(&rx_frame);
    if(res)
    {
        canardHandleRxFrame(&canard, &rx_frame, get_uptime() * 1000);
    }
}
```

Now it's time to try the app. When your Zubax Babel is flashed with it, you should see a green LED blinking once a second. If you then connect it to the UAVCAN GUI Tool, you should see something like this:

It may be also useful to go to the bus monitor and check if messages are coming properly:

Messages from node with ID 100 are present in the picture above. And they keep appearing once a second. This means everything works as planned.