1 Introduction

1.1 Overview
This application note aims to build a USB-CAN adapter where the USB data retransmit to CAN-bus and vice versa. NXP LPC55S16 has a high-speed USB port and CANFD controllers. HSUSB can reach up to 480 Mbit/s transmission speed, which is enough for transmitting CANFD frame at highest CAN baud rate.

To make the system easy to use and compatible with other devices, we use USB CDC virtual COM port as communication interface with PC and a simple ASCII protocol inherited from open-source project USBtinViewer.

2 Implementation

2.1 Overview
As shown in Figure 1, USB CDC uses two USB physical buck endpoints to transfer data between PC and MCU. Each endpoint is responsible for uni-directional data transfer.

SDK already provides MCAN driver and USB Stack. In software, add two buffers for each pipe, one for USB -> CAN bus and the other for CAN bus -> USB. To ensure the best performance, the two pipes are independent.

![System block diagram](image)

Figure 1. System block diagram

2.2 Related SDK example
Before continuing the task, we need the background knowledge of USB CDC and CAN usage. Fortunately, SDK provides two examples.

- MCAN loopback example
MCAN example is a simple CAN loopback example which demonstrates usage of LPC54608’s CAN module. This example enables the internal loopback of CAN module and send a CAN frame. The CAN frame loops back into CAN receiver and MCU displays any received CAN frames on UART terminal. To get familiar with this example, read the readme documentation and run the example.

Example location: `boards\lpcxpresso55s16\driver_examples\mcan\loopback`

- `usb_device_cdc_vcom` example

This example is USB CDC class example to enumerate USB as a communication device class. When USB enumeration completes, a COM port pops out on the device. Any character sent through this COM port is loop back to display. See the readme documentation for this example for how to install device driver and run the demo.

Example location: `boards\lpcxpresso55s16\usb_examples\usb_device_cdc_vcom\bm`

Be familiar with above two examples before continue reading. Those two examples are building blocks for USB-CAN adapter design.

### 2.3 Hardware

Table 1 describe GPIO pins used in USB-CAN adapter.

<table>
<thead>
<tr>
<th>GPIO</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIO1_2</td>
<td>CAN0_TX</td>
<td>CAN bus signal</td>
</tr>
<tr>
<td>PIO1_3</td>
<td>CAN0_RX</td>
<td>CAN bus signal</td>
</tr>
<tr>
<td>USB1_DM</td>
<td>USB1_DM</td>
<td>HSUSB DM</td>
</tr>
<tr>
<td>USB1_DP</td>
<td>USB1_DP</td>
<td>HSUSB DP</td>
</tr>
<tr>
<td>PIO0_29</td>
<td>UART_RXD</td>
<td>Debug UART RXD</td>
</tr>
<tr>
<td>PIO0_30</td>
<td>UART_TXD</td>
<td>Debug UART TXD</td>
</tr>
</tbody>
</table>

For full schematic, see Schematic of USB-CAN-adapter.

### 2.4 Serial communication protocol

USB-CAN adapter registers as a virtual serial port on the host computer. With simple ASCII commands, CAN bus configuration can be controlled over this serial port. You can send/receive commands from any serial terminal program or from your own program.

Table 2. ASCII protocol commands list:

<table>
<thead>
<tr>
<th>ASCII commands</th>
<th>Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O[CR]</td>
<td>[CR]</td>
<td>Open CAN channel</td>
</tr>
<tr>
<td>C[CR]</td>
<td>[CR]</td>
<td>Close CAN channel</td>
</tr>
<tr>
<td>tiiiidd.[CR]</td>
<td>Transmit standard (11 bit) frame.</td>
<td>iii: Identifier in hexadecimal format (000-7FF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l: Data length (0-8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dd: Data byte value in hexadecimal format (00-FF)</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
### Table 2. ASCII protocol commands list: (continued)

<table>
<thead>
<tr>
<th>ASCII commands</th>
<th>Response</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sx[CR]</td>
<td></td>
<td>Set baud rate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x: Bitrate id (0-8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S0 = 10 kBaud</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S1 = 20 kBaud</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S2 = 50 kBaud</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S3 = 100 kBaud</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S4 = 125 kBaud</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S5 = 250 kBaud</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S6 = 500 kBaud</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S7 = 800 kBaud</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S8 = 1 MBaud</td>
</tr>
<tr>
<td>Tiiiiiiidd.[CR]</td>
<td></td>
<td>Transmit extended (29 bit) frame.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iiiiiii: Identifier in hexadecimal format (0000000-1FFFFFFF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l: Data length (0-8) dd:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data byte value in hexadecimal format (00-FF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iii: Identifier in hexadecimal format (000-7FF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l: Data length (0-8)</td>
</tr>
<tr>
<td>Riiiiiiii[CR]</td>
<td></td>
<td>Transmit extended RTR (29 bit) frame.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>iiiiiii: Identifier in hexadecimal format (0000000-1FFFFFFF)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l: Data length (0-8)</td>
</tr>
<tr>
<td>mxxxxxxxx[CR]</td>
<td></td>
<td>Set acceptance filter mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SJA1000 format (AM0..AM3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only first 11bit are relevant.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxxxxxxxx: Acceptance filter mask</td>
</tr>
<tr>
<td>Mxxxxxxxx[CR]</td>
<td></td>
<td>Set acceptance filter code.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SJA1000 format (AC0..AC3).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only first 11bit are relevant.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxxxxxxxx: Acceptance filter code</td>
</tr>
</tbody>
</table>

**Example:**
Set 10 kBaud, open CAN channel, send CAN message (id = 001 h, dlc = 4, data = 11 22 33 44), and close CAN.

### Table 3. CAN message

<table>
<thead>
<tr>
<th>Command</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0[CR]</td>
<td>[CR]</td>
</tr>
<tr>
<td>O[CR]</td>
<td>[CR]</td>
</tr>
</tbody>
</table>

*Table continues on the next page...*
Table 3. CAN message (continued)

<table>
<thead>
<tr>
<th>Command</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>t001411223344[CR]</td>
<td>z[CR]</td>
</tr>
<tr>
<td>C[CR]</td>
<td>[CR]</td>
</tr>
</tbody>
</table>

With a state machine, the software accepts serial stream from CDC port, parses the ASCII, and applies the command. Below lists some of the important code snippet used in the software. For full source code, see AN13515SW.

- To send a CAN frame,

  ```c
  txFrame.xtd  = kMCAN_FrameIDStandard;
  txFrame.rtr  = kMCAN_FrameTypeData;
  txFrame.fdf  = 0;
  txFrame.brs  = 0;
  txFrame.dlc  = len;
  txFrame.id   = id << STDID_OFFSET;
  txFrame.data = buf;
  txFrame.size = CAN_DATASIZE;
  txXfer.frame = &txFrame;
  txXfer.bufferIdx = 0;
  MCAN_TransferSendNonBlocking(EXAMPLE_MCAN, &mcanHandle, &txXfer);
  ```

- To receive a CAN frame,

  ```c
  static void mcan_callback(CAN_Type *base, mcan_handle_t *handle, status_t status, uint32_t result, void *userData)
  {
    switch (status)
    {
      case kStatus_MCAN_RxFifo0Idle:
      {
        memcpy(rx_data, rxFrame.data, rxFrame.size);
        MCAN_TransferReceiveFifoNonBlocking(EXAMPLE_MCAN, 0, &mcanHandle, &rxXfer);
        can_rx_cb(rxFrame.id >> STDID_OFFSET, rx_data, rxFrame.dlc);
      }
      break;

      case kStatus_MCAN_TxIdle:
      {
      }
      break;

      default:
      {
        break;
      }
    }
  }
  ```

- To send data via USB CDC,

  ```c
  void usbd_cdc_send(uint8_t *buf, uint32_t len)
  {
    USB_DeviceCdcAcmSend(s_cdcVcom.cdcAcmHandle, USB_CDC_VCOM_BULK_IN_ENDPOINT, buf, len);
  }
  ```
To receive data from USB CDC,

```c
usb_status_t USB_DeviceCdcVcomCallback(class_handle_t handle, uint32_t event, void *param)
{
    switch (event)
    {
        case kUSB_DeviceCdcEventRecvResponse:
        {
            if ((1 == s_cdcVcom.attach) && (1 == s_cdcVcom.startTransactions))
            {
                uint8_t rx_size;
                rx_size = epCbParam->length;
                
                error = USB_DeviceCdcAcmRecv(handle, USB_CDC_VCOM_BULK_OUT_ENDPOINT, cdc_rx_buf, g_UsbDeviceCdcVcomDicEndpoints[1].maxPacketSize);
            }
            
            cdc_rx_cb(cdc_rx_buf, rx_size);
        }
        break;
    }
}
```

3 Hands on with USBtinViewer

To verify the functionality USB-CAN adapter, in this section, we use open-source software USBtinViewer and a commercial USB-CAN adapter (PCAN-USB).

USBtinViewer can be downloaded from https://www.fischl.de/usbtin/#usbtinviewer.

We use PCAN-USB for commercial USB-CAN adapter and busmaster for software.

Busmaster can be downloaded from https://rbei-etas.github.io/busmaster/.

Figure 2 shows the hardware test environment.
3.1 Connecting hardware to USBtinViewer

1. Download USBtinViewer and connect the USB port of USB-CAN-adapter to the PC. A USB CDC COM port pops up. The COM port number varies from PC to PC.

NOTE

Open USBtinViewer, select COM port and CAN baud rate (500 K in this example). Click Connect, and the USBtinViewer returns the firmware information, as shown in Figure 4. The information means that connection succeeds.
2. Open busmaster and connect PCAN-USB. Select 500 K baud rate, as shown in Figure 5.

3. Send CAN data from USBtinViewer and received by busmaster.

   Connect USB-CAN adapter and PCAN-USB. In the CAN TX box, at the bottom of USBtinViewer, enter the CAN message ID, DLC, and data field. Click **Send**, and the USB-CAN adapter sends the CAN message.
4. Send CAN data from busmaster and verify by USBtinViewer.

In busmaster, open **Transmit Window**. Click the empty space under the **Message Name** column. Enter the new message name, DLC, and frame data field. Click **Send message**, and the busmaster sends the CAN message. On USBtinViewer, this CAN message can be monitored, as shown in Figure 8.
3rd-party and community resources

Many useful 3rd-party resources are available on USBtin web page. It includes libraries and tools supporting USBtin. It provides rich resources and supports various programming languages.

Third party and community projects - Software supporting USBtin

- pyUSBtin - Python implementation of the USBtin API
- CsharpUSBtinLib - C# USBtin library
- CANFlasherUTNL - Bootloader GUI for NXP LPC11C22/24 devices
- CANToolz - Framework for analysing CAN networks and devices
- CANopen API for Windows (C++, C#/.NET) by Datalink Engineering
- UsbTinQt - Qt application written in C++
- CANcool - Open Source CAN bus Analyser and Simulation Software (Pascal / Delphi 7)
5 Schematic of USB-CAN-adapter

![Schematic of USB-CAN-adapter](image)

Figure 10. Schematic of USB-CAN-adapter

6 Reference

1. [https://www.fischl.de/usbtin/#usbtinviewer](https://www.fischl.de/usbtin/#usbtinviewer)
2. [https://rbei-etas.github.io/busmaster/](https://rbei-etas.github.io/busmaster/)

7 Revision history

<table>
<thead>
<tr>
<th>Rev.</th>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18 January 2022</td>
<td>Initial release</td>
</tr>
</tbody>
</table>
Limited warranty and liability — Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer’s technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer’s applications and products. Customer’s responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer’s applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

© NXP B.V. 2022. All rights reserved.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com
Date of release: 18 January 2022
Document identifier: AN13515