AN12625
Multi-TTY Expansion

Rev. 0 — October 2019 Application Note

1 Introduction

In industry applications, it is a general requirement to expand massive serial ports and GPIOs. Usually, the specific solution is expensive and less flexible. In this document, we introduce one proof of concept to expand slow speed interfaces through general MCU. Only multi-TTY expansion is addressed in this document. The hardware and software architecture are suitable for virtual GPIOs and CAN ports as well.

The LPC540xx/LPC54S0xx is a family of Arm Cortex®-M4-based microcontrollers for embedded applications featuring a rich peripheral set with very low power consumption and enhanced debug features.

The LPC54018 includes 360 KB of on-chip SRAM, a quad SPI Flash Interface (SPIFI) for expanding program memory, one high-speed and one full-speed USB host and device controller, CAN FD, each Flexcomm interface of 10 serial peripherals can be selected by software to be a USART, SPI, or I2C interface.

2 Hardware Blocks

In Figure 1, we show a common multi-TTY expansion concept. The low-cost MCU LCP54018 device is connected to SoC MPU or general CPU through USB 2.0 interface.

Contents

1 Introduction.....1

2 Hardware Blocks.....1

3 Software Modules.....3
3.1 Multi-TTY expansion specification.....3
3.2 Linux device driver at host side.....5
3.3 USB and TTY bridge at MCU side..... 8

4 Functions and Features..... 10
4.1 USB device enumeration..... 10
4.2 MCU configurations and managements.....10
4.3 Features of expansion TTY Ports.....11

5 Performance Benchmark..... 13

6 Reference.....13

7 Revision history..... 13

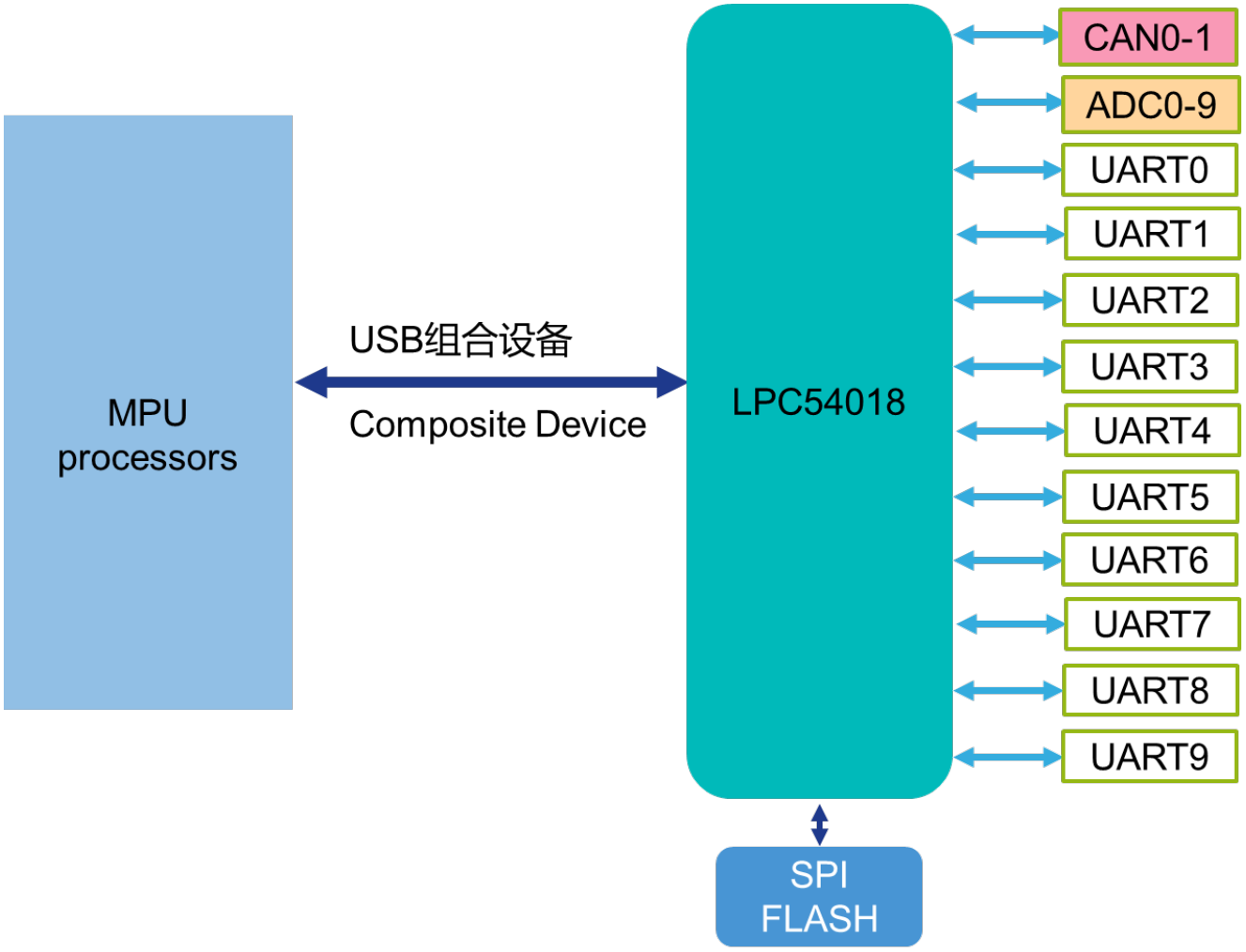


Figure 1. Hardware Connection of Multi-TTY Expansion

The Figure 2 is a proposal to convert USB interface to multi-TTY with the MCU LPC54018 device.

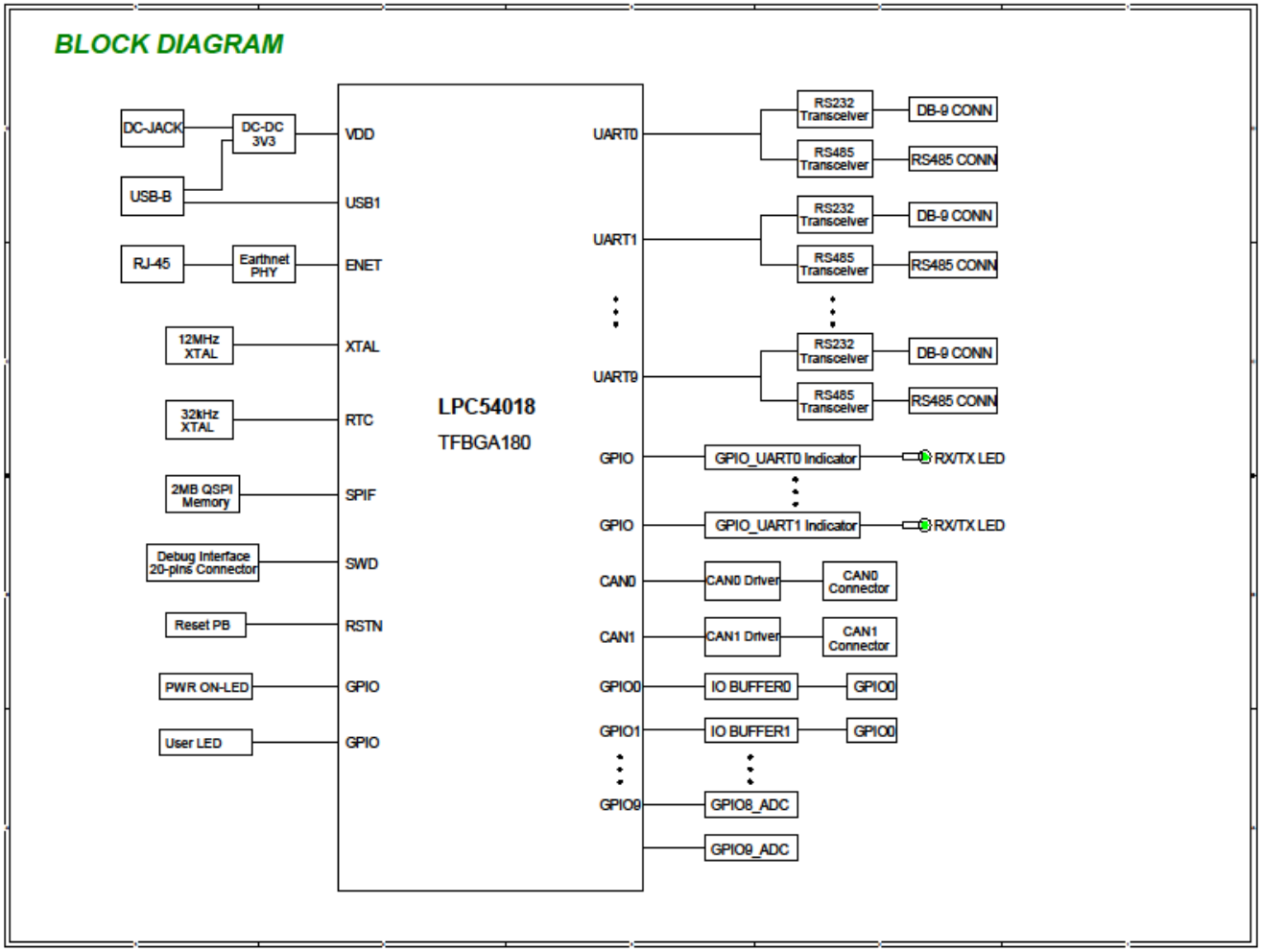


Figure 2. Block Diagram of Multi-TTY Expansion

3 Software Modules

3.1 Multi-TTY expansion specification

A USB virtual COM port is a software interface that enables applications to access a USB device as if it were a built-in serial port. Instead of a vendor-specific driver, the PC uses the USB communication devices class (CDC) driver included with Windows and Linux. During enumeration, the USB device responds to requests for descriptors that identify the device CDC function.

The SET_LINE_CODING and GET_LINE_CODING requests set and request the bit rate, number of stop bits, parity, and number of data bits. In the SET_CONTROL_LINE_STATE request, the host tells the device how to set the RS-232 control signals RTS and DTR. The SEND_BREAK request requests the device to send an RS-232 break signal (a positive RS-232 voltage on the TX line) for a specified number of milliseconds. The SERIAL_STATE notification provides a way for a device to send the states of the RS-232 status signals RI, DSR, and CD, the Break state, and error states for buffer overrun, parity error, and framing error. The notification consists of an 8-byte header and two notification bytes. The interrupt IN endpoint returns a notification or NAK in response to receive IN token packets.

The Figure 3 shows a USB Communication Device Class (CDC) descriptor. This type of USB device is supported by Linux and Windows built-in USB Communication Device Class (CDC) Abstract Control Model (ACM) driver.

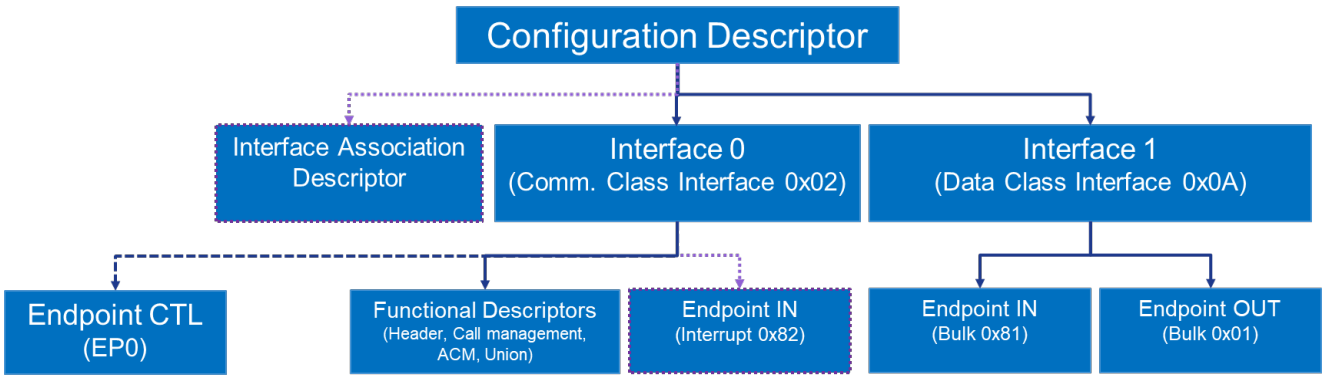


Figure 3. Descriptor for USB Communication Device Class (CDC)

Typically, this driver module needs at least 3 endpoints to emulate one TTY device. The number of endpoints in USB device limit the maximum number of virtual TTY device. For example, there are 10 physical (5 logical) endpoints in USB0 and 12 physical (6 logical) endpoints in USB1 including control endpoints separately in LPC540xx device.

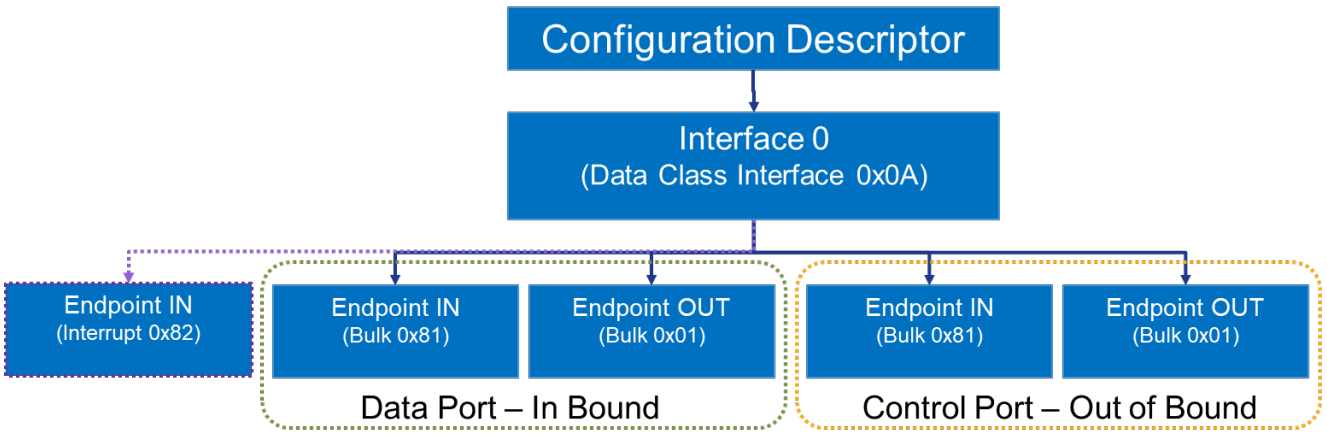


Figure 4. Descriptor for Multi-TTY Expansion

To reduce the dependency on the number of endpoints, the multi-TTY emulation is designed to share two groups of endpoints: one group is used as a control channel for out of bound communication, and another group is used for in bound data communication. If needed, the interrupt endpoint could be added into the communication interface as well like Figure 4 above.

To support common TTY device APIs, the frame format between USB host and agent is defined as below in Figure 5 to show the concepts.

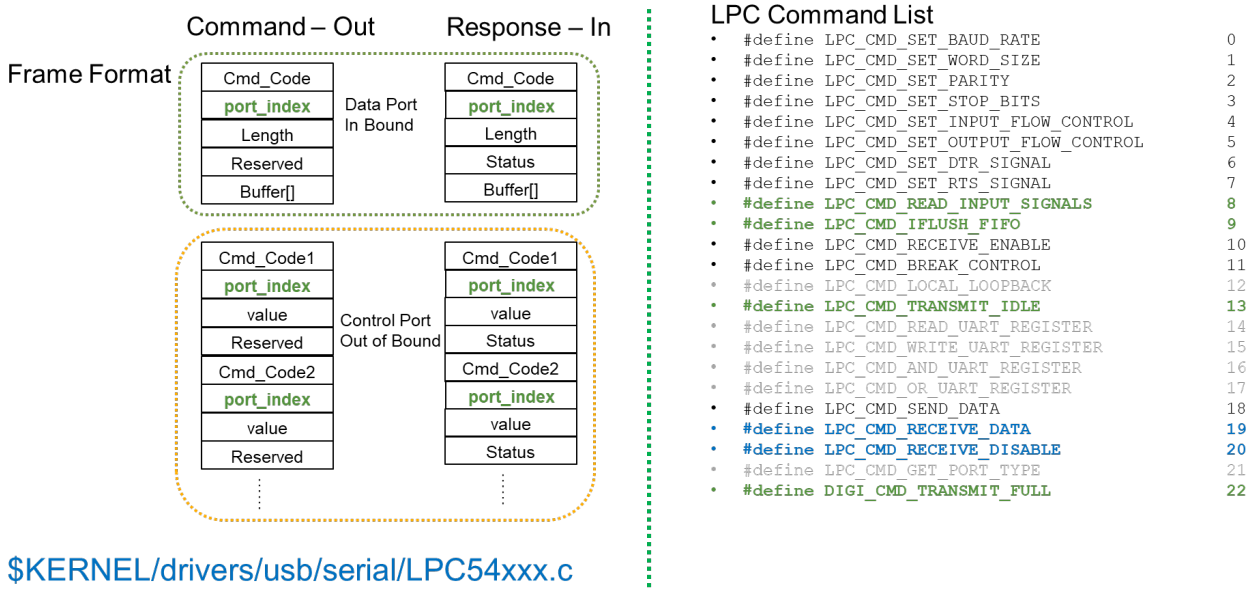


Figure 5. Frame Format between USB Host and Agent

3.2 Linux device driver at host side

Similar with the USB Communication Device Class (CDC) Abstract Control Model (ACM) driver, a type of vendor-dependent multi-TTY device is defined as a virtual serial port device.

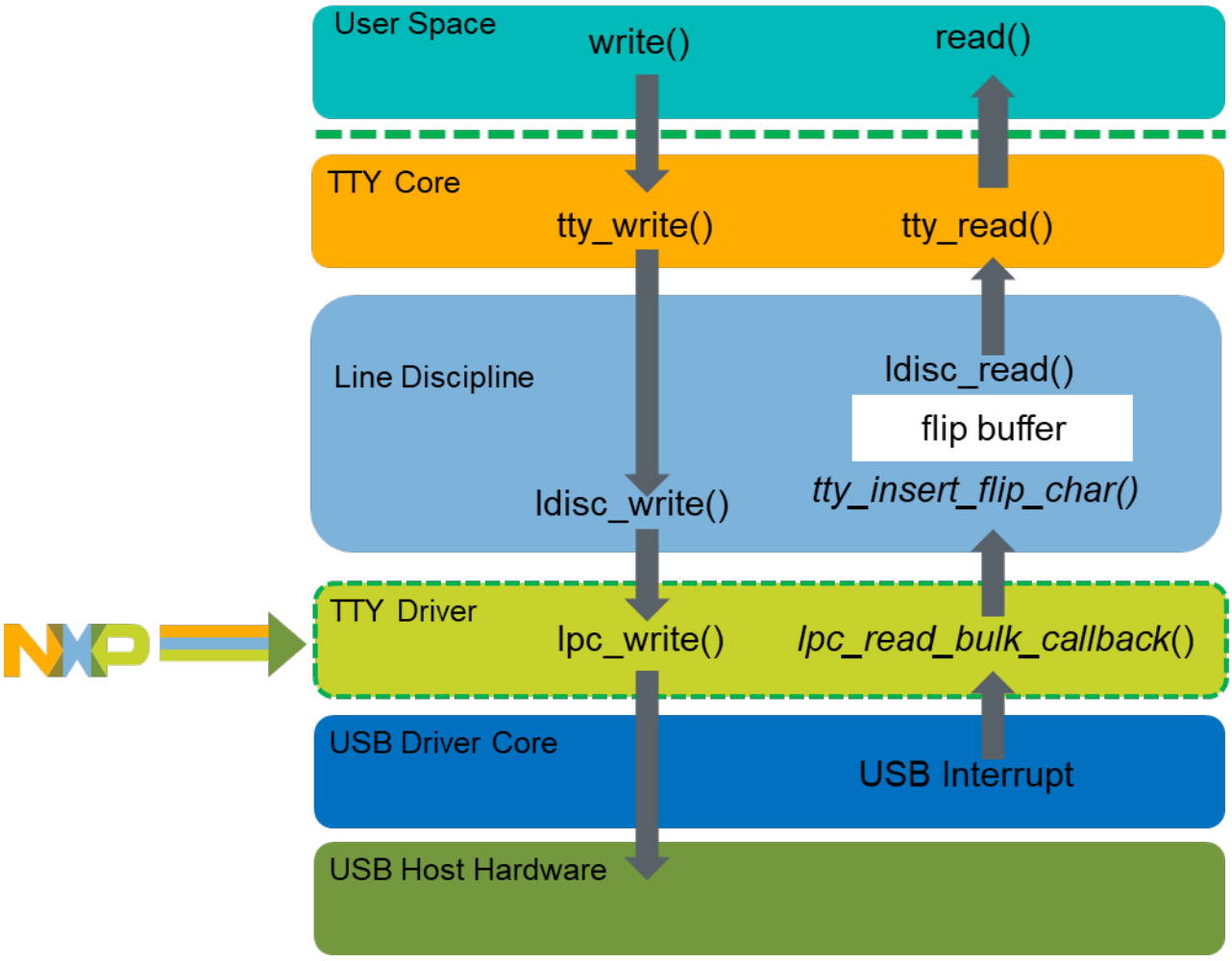


Figure 6. Host Device Driver Block Diagram for Multi-TTY Expansion

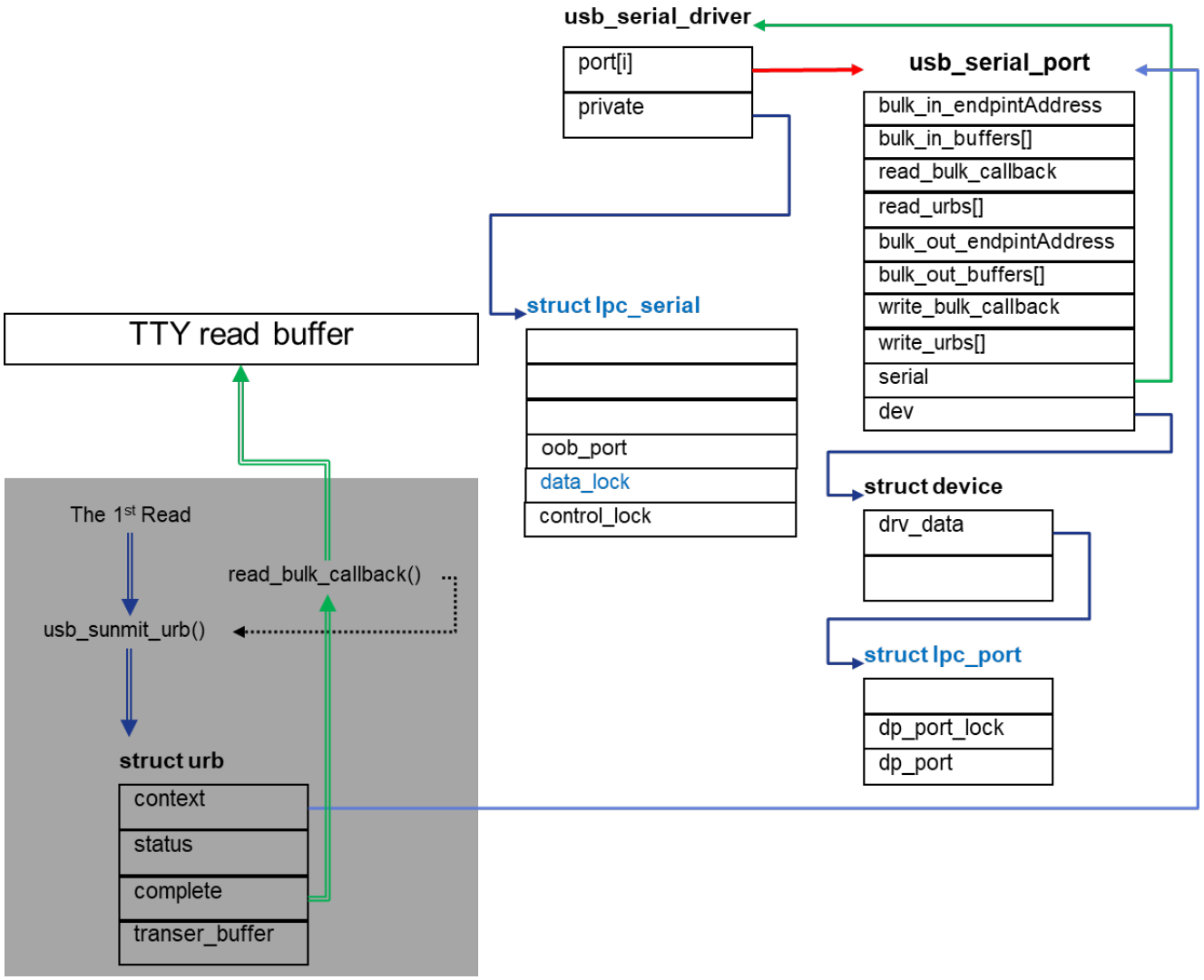


Figure 7. Host Device Driver Data Structure for Multi-TTY Expansion

One USB serial port device could support virtual multi-TTY ports per hardware specification, for example ten virtual TTY ports on LPC54018 device. Each port could be operated independently. The virtual TTY port could support the TTY operations as below:

```
static const struct tty_operations serial_ops = {
        .open =                 serial_open,
        .close =                serial_close,
        .write =                serial_write,
        .hangup =               serial_hangup,
        .write_room =           serial_write_room,
        .ioctl =                serial_ioctl,
        .set_termios =          serial_set_termios,
        .throttle =             serial_throttle,
        .unthrottle =           serial_unthrottle,
        .break_ctl =            serial_break,
        .chars_in_buffer =      serial_chars_in_buffer,
        .wait_until_sent =      serial_wait_until_sent,
        .tiocmget =             serial_tiocmget,
        .tiocmset =             serial_tiocmset,
        .get_icount =           serial_get_icount,
        .cleanup =              serial_cleanup,
        .install =              serial_install,
        .proc_fops =            &serial_proc_fops,
};
```

Figure 8. TTY Operations for Multi-TTY Expansion

Especially, we could provide extension APIs for easy debugging, like MCU register operation.

3.3 USB and TTY bridge at MCU side

At USB agent side, the software is straight forward.

For transmission, the USB interrupt handler sends semaphore to USB Tx thread, this thread copies the data from USB FIFO to thread FIFO in first. Then it copies the data to the UART Tx FIFO which is ready for serial port hardware to transmit. Once the serial port transmission DMA is finished, the interrupt handler sends a semaphore to the transmission thread for serial port, which set up DMA to send the further data in UART Tx FIFO.

For receiving, the receive thread for serial port checks the status of DMA for receiving and adjust the UART Rx FIFO pointer in time. The USB Rx thread checks the status of UART Rx FIFO, copy and encapsulate the frame per Figure 5 into USB Rx FIFO, and make it ready for USB hardware to move per USB host read request.

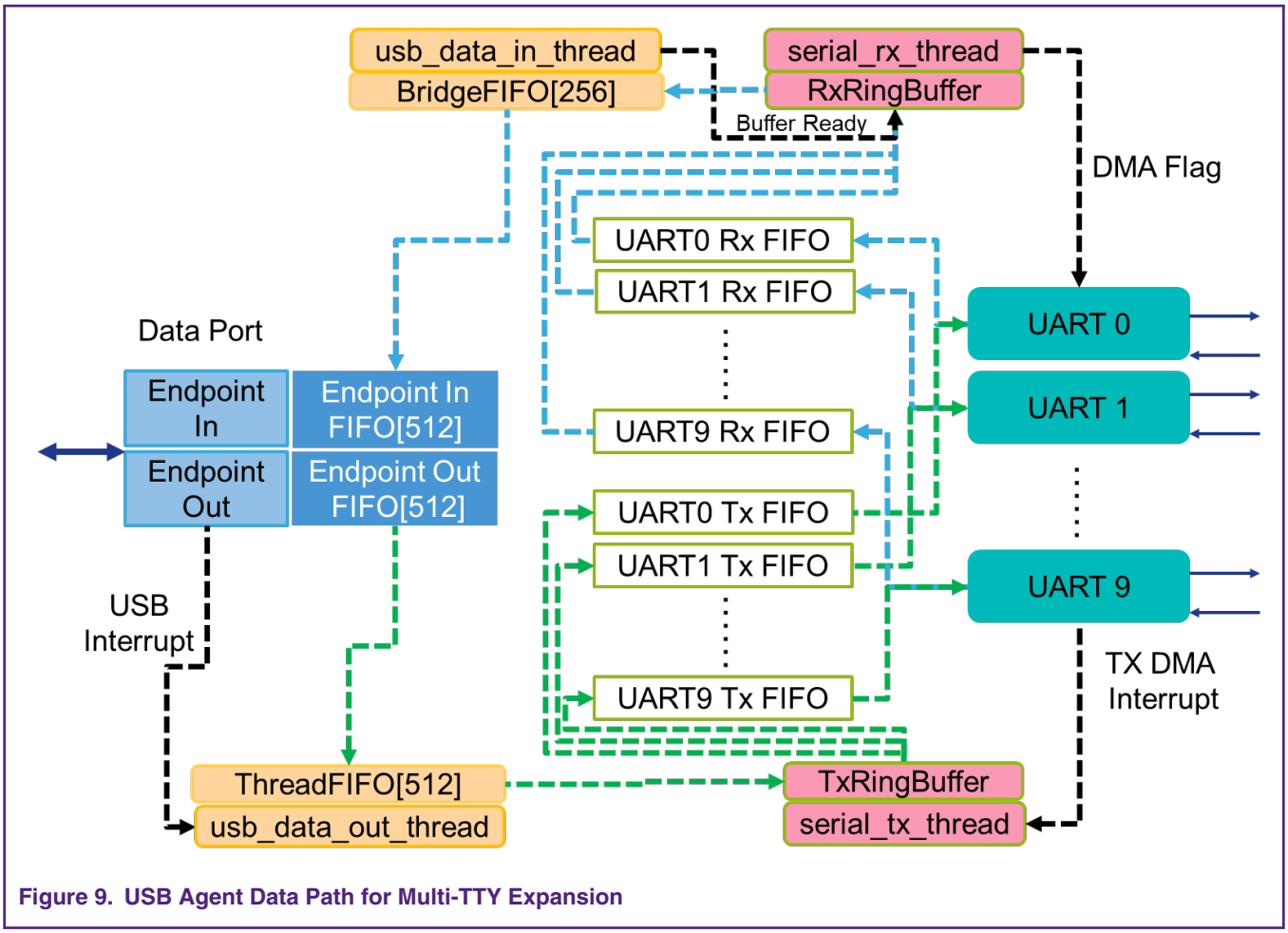


Figure 9. USB Agent Data Path for Multi-TTY Expansion

For the control path, the control thread needs to execute the command from USB host side to set or get the real serial port registers, then response the result back to the USB host through control in endpoint.

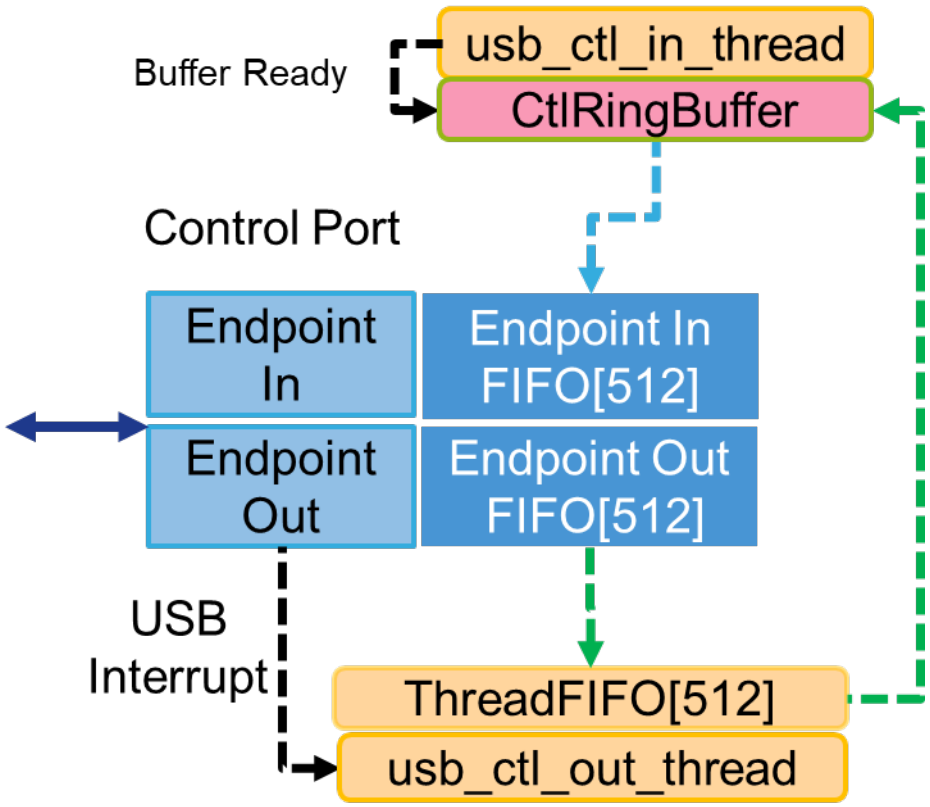


Figure 10. USB Agent Control Path for Multi-TTY Expansion

4 Functions and Features

4.1 USB device enumeration

As this hardware is attached to the host USB port, as a USB CDC device, it is probed and registered by the vendor provided device driver as multi-TTY port device. Like Figure 4 above, the USB agent needs to program to provide the correct USB descriptor information, including the endpoints for communication and the data channel separately and the endpoint for interrupt if needed (optional). The host device driver will finish the initiation of multi virtual serial ports at device driver level and will be ready to be used by the host application.

4.2 MCU configurations and managements

4.2.1 Flash Upgrade

The vendor needs to provide a tool to program the MCU flash memory of the USB agent from the USB host side. This feature is out of the range of the USB TTY device driver. The detail are not discussed in this document..

4.2.2 Register operation for debug

To support users to extend and debug the real hardware of serial ports from the USB agent side without touching the code in the MCU, we provide APIs to operate the serial port registers from the USB host side directly, for example, to set internal loopback mode for whatever purpose.

4.2.3 Hardware reset

It is necessary to restart the USB agent fully. This feature needs the software and hardware support of the USB agent side.

4.3 Features of expansion TTY Ports

4.3.1 Parameter configuration and query

The application invokes the glibc API tcsetattr() to change the parameter of the serial port, and tcgetattr() to get the parameter separately. These glibc APIs invoke the system call as shown below:

The host USB core driver to set baud rate, parity, word size, stop bits, flow control, and so on.

The agent USB driver at MCU side executes the command from host side.

4.3.2 Open

The open system call of TTY device is as below:

The host USB core driver issue command to enable the serial port at USB agent side.

The agent USB driver at MCU side needs to enable the real serial port and make the FIFOs, DMA, threads to be ready.

4.3.3 Close

The close system call of TTY device is as below:

The host USB driver issue a command to disable the serial port from the USB agent side.

The agent USB driver from the MCU side needs to disable the real serial port and make the FIFOs, DMA, and threads to be an idle state.

4.3.4 Read data

The read system call of TTY device is as below. It gets the data from port buffer directly, without calling the TTY device driver.

The host USB core driver from the receiving side invokes the call-back routine in TTY device driver to put the data to the port buffer as Figure 6 above. We implemented this routine to decapsulate the frame format between USB host and agent and put the real data into the port buffer.

The agent USB driver from the MCU side needs to create the correct USB frame based on the data in Rx FIFO memory for serial port according to the frame format as Figure 5 above, which is received by the MCU serial port driver from real serial port.

4.3.5 Write data

The write system call of TTY device is as below. It directly invokes the write operation in TTY device driver.

We implemented this routine to decapsulate the frame format between USB host and agent and put the real data into the port buffer. Because the TTY device is a slow speed one, the write routine needs to check the availability of write buffer at USB agent side to avoid congestion.

The agent USB driver from the MCU side decapsulates the frame format between USB host and agent and put the real data into the Tx FIFO memory for serial port according to the frame format as Figure 5 . Then the serial port driver on MCU sends the data to real serial port.

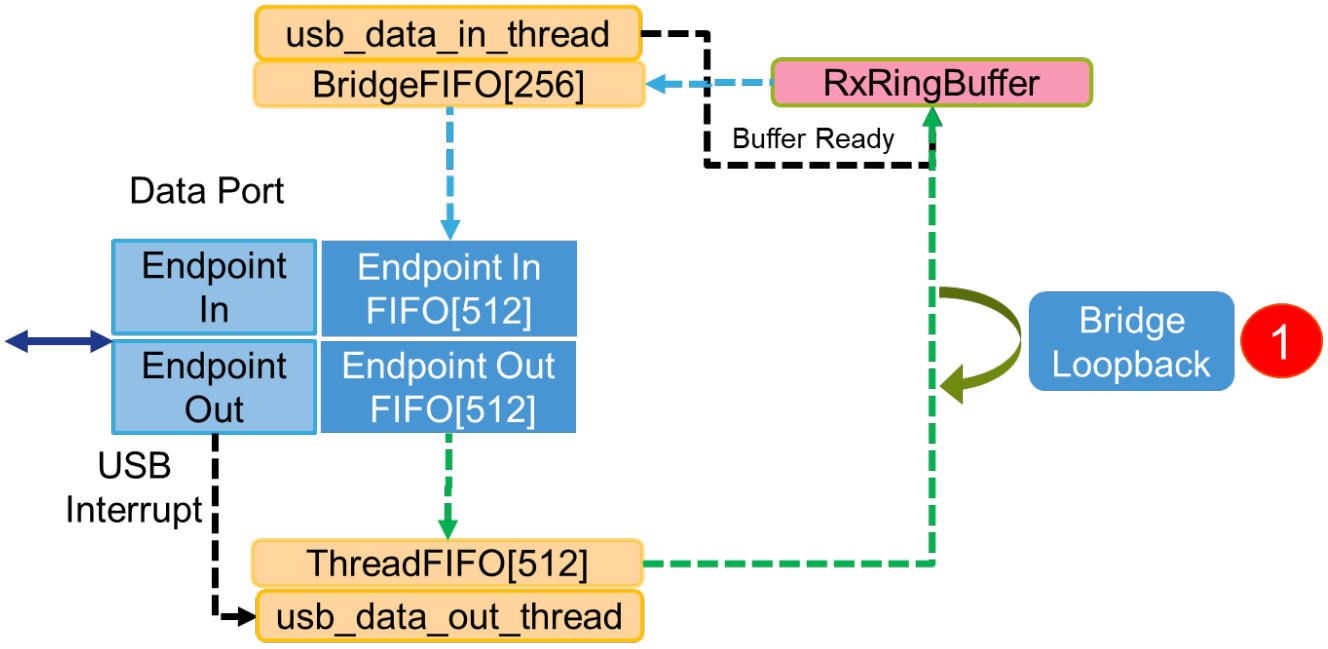
4.3.6 Extension control

4.3.6.1 Health status query

Provide the health status information of MCU, USB port, and real serial port.

4.3.6.2 Loopback mode

Because the loopback feature is a usual function but there is no API defined in common TTY device driver, we added this API for the user. We can have the USB agent set the loopback mode for the virtual serial port in USB driver to skip the real serial port (bridge loopback in the Figure 11), inner or external of the real serial port as well.



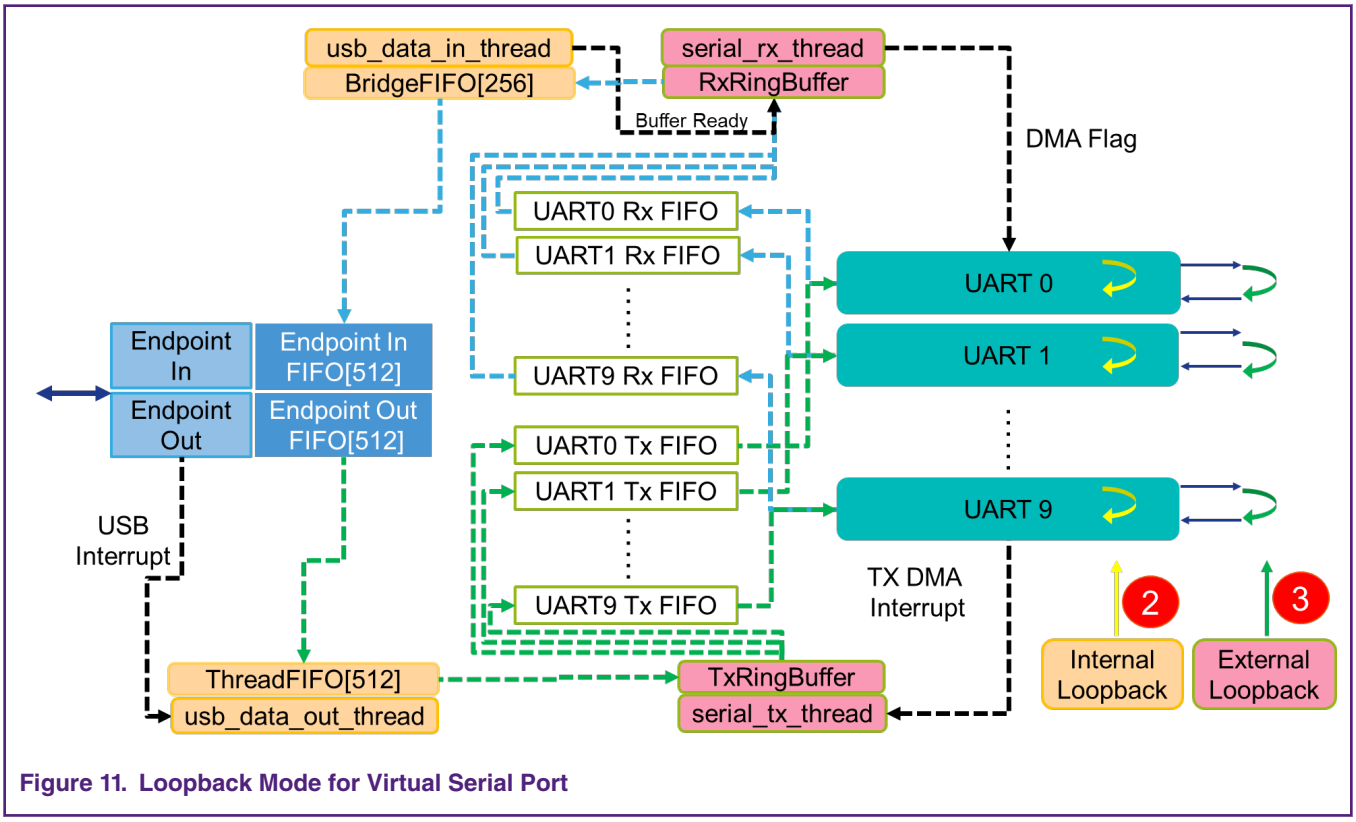


Figure 11. Loopback Mode for Virtual Serial Port

5 Performance Benchmark

Basically, 115200 bps baud rate could be supported on ten serial ports at same time while CPU utility is about 10 % (180 MHz LPC54018). Further benchmark needs to be conducted. For example, what is the throughput at bridge loopback mode in Figure 11. Then we can evaluate how much MCU resources can be used to expand the GPIOs and CAN ports in this solution in the next step.

6 Reference

1. LPC540xx/LPC54S0xx User manual, Rev. 1.3, 28 January 2019
2. Universal Serial Bus Class Definitions for Communications Devices Revision 1.2 (Errata 1), November 3, 2010
3. Universal Serial Bus Communications Class, Subclass Specification for PSTN Devices, Revision 1.2, February 9, 2007
4. Cortex-M4 Technical Reference Manual, Revision r0p0.

7 Revision history

Table 1. Revision history

Rev. Number	Date	Substantive Change(s)
0	11/2019	Initial release.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

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.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

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

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

Date of release: October 2019

Document identifier: AN12625

arm