Application Note

High-Speed (12Mb) UART Communication Based on TI's Programmable Real-Time Unit (PRU)

李彪, 唐超伦 *China Central FAE*

ABSTRACT

This document was translated from a simplified Chinese source. (ZHCAGC6)

The UART (Universal Asynchronous Receiver/Transmitter) interface remains widely used in various industrial and medical applications today due to its simple structure, strong noise immunity, and low cost. Moreover, some application scenarios require support for up to 12Mbps or even 16Mbps. In high-baud-rate UART communication based on a general-purpose MPU with a hardware UART, the thread scheduling latency of the operating system (e.g., Linux), uncertain interrupt-response latency, and multi-task concurrency can easily lead to data buffer overflows and high packet loss rates, making it difficult to meet the requirements of industrial applications that demand high reliability.

To address these bottlenecks, this document proposes and implements a high-speed UART communication scheme based on the programmable real-time unit (PRU) of TI Sitara processors. Achieving high-speed UART communication at up to 12Mbps, this scheme leverages the PRU's fully independent hardware architecture featuring a single-cycle instruction set with deterministic execution cycles to directly control PRU's I/Os and simulate the serial communication process. By eliminating dependence on the general-purpose processor's interrupt mechanism and establishing a shared-memory data exchange channel between the PRU and the ARM core, this scheme removes the real-time jitter introduced by the operating system. It achieves efficient, loss-free UART data transmission at 12Mbps while significantly reducing the CPU load on the main ARM core. This design utilizes the PRU core internal to TI processors at no additional cost, providing customers with a low-cost, high-reliability solution for high-speed UART communication in real-world industrial applications.

Table of Contents

1 Introduction 3
1.1 Introduction to TI PRU 3
1.2 Application Background and Industrial Requirements for High-Speed UART 4
1.3 Technical Challenges of 12Mbps High-Speed UARTs 4
2 Design and Implementation of PRU-Based High-Speed UART Communication System 6
2.1 UART Low-Level Physical-Layer Design 6
2.2 Implementation of Multi-Channel High-Speed UART by PRU Software Using the PEFIF Interface 7
2.3 Design of Application-Layer Driver for UARTs on ARM Linux 9
3 Experimental Verification of High-Speed UARTs 10
3.1 Bare-Metal Experiment of High-Speed PRU UART 10
3.2 High-Speed PRU UART Application Based on Linux 10
4 Conclusion 13
5 References 13

List of Figures

Figure 1-1. AM62x PRU-ICSS Functional Block Diagram 3
Figure 2-1. PRU UART Data-Link Layer Hardware Implementation Design 8
Figure 3-1. Hardware Environment Setup Based on AM62 SK EVM 10
Figure 3-2. Software Environment Setup Based on AM62 SK EVM 10
Figure 3-3. Random UART Data Waveform At Linux Side 12

Copyright © 2026 Texas Instruments Incorporated

List of Tables

Table 1-1. Number of UARTs Supported by Sitara AM6x Series....4

Trademarks

All trademarks are the property of their respective owners.

Copyright © 2026 Texas Instruments Incorporated

1 Introduction

1.1 Introduction to TI PRU

The block diagram of the PRU (Programmable Real-Time Unit) on TI SoCs is shown in Figure 1-1. The PRU is a RISC core specifically designed for hard real-time, low-latency control. It guarantees deterministic timing at the physical boundary through its pipeline-free single-cycle execution mechanism and direct pin-mapping architecture.

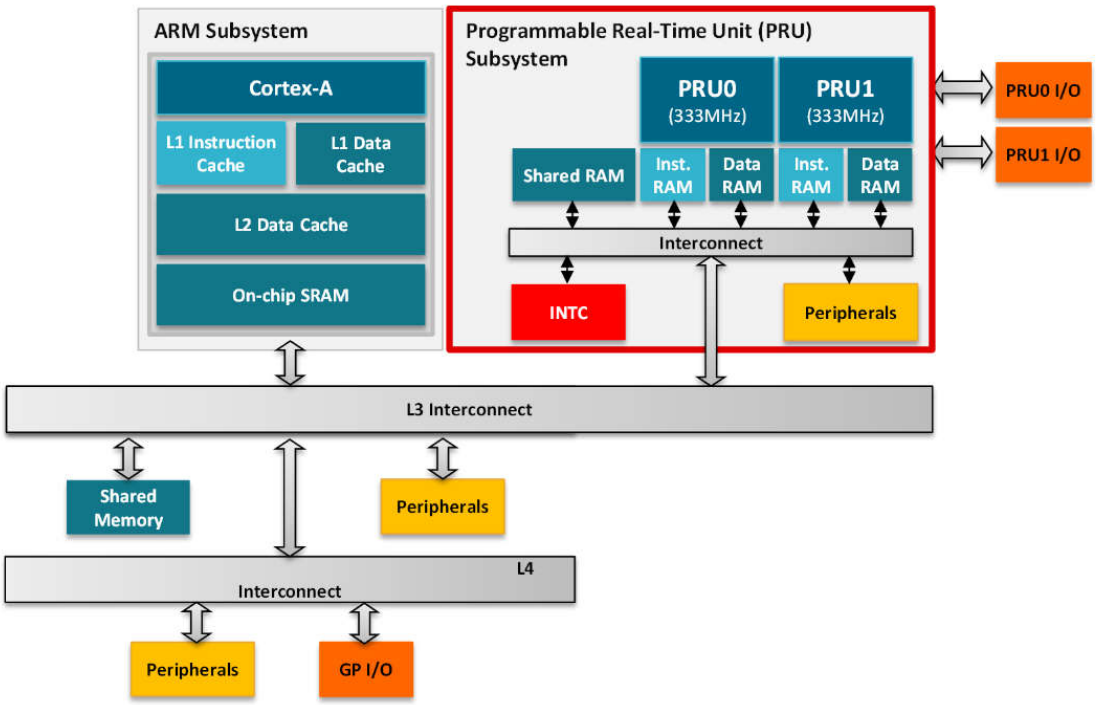

Figure 1-1. AM62x PRU-ICSS Functional Block Diagram

In actual industrial applications, systems often need to interface with external devices that do not use standard protocols. For interfaces with special timing requirements that the SoC may not be able to connect to directly, the traditional approach is to use an FPGA to achieve high-bandwidth communication between the SoC and the special-timing peripheral. However, because Sitara processors include a PRU (programmable real-time unit) core internally, there is no need to add an FPGA, ASIC, CPLD, or similar device in such cases. Another approach is to use an MCU's GPIO pins for simulation; however, using an MCU core to toggle GPIO pins has fundamental drawbacks. Compared to an FPGA, a standard MCU exhibits higher latency and lower determinism when interacting with GPIOs. When software is used to control GPIO toggling, the achievable frequency and precision are severely limited by internal bus latency, making it impossible to meet the requirements for duty-cycle control and precise timing control.

The PRU system contains 32 32-bit registers, two of which are dedicated to GPIO functions. One register is directly connected to the General-Purpose Output (GPO) signals; writing to this register toggles the GPO signals. The other register is directly connected to the General-Purpose Input (GPI) signals; reading this register reads the GPI signals. With this design, there is a direct connection from the PRU to the actual GPIO pins, and the performance of both input and output signals is improved. The PRU has instructions that pause program execution until a GPI event occurs, enabling optimal-latency output response upon detection of an input trigger. The PRU can thus achieve fully deterministic timing relationships between a single input and output, as well as among multiple input or multiple output signals. This highly deterministic physical-layer communication is decoupled from the complex, non-real-time upper-layer operating system. At a 333MHz core frequency, the single-cycle GPIO execution precision is controlled to 3ns, and the interrupt-response cycle can be controlled to 6ns. In addition, the PRU core integrates an innovative Broadside Interface, which breaks the traditional

Copyright © 2026 Texas Instruments Incorporated

word-by-word read bottleneck of general-purpose CPUs. This interface allows the PRU to achieve parallel bulk-data throughput of up to 124 bytes with local hardware accelerators in a single clock cycle. Furthermore, each PRU core is equipped with 2kB of dedicated Broadside RAM. This memory is mounted to the same dedicated Broadside bus, providing access speeds equivalent to shared memory while also guaranteeing zero bus contention during multi-core concurrency through physical isolation.

The above description applies to all Sitara processors with a PRU subsystem, including the AM243x, AM263x, AM335x, AM437x, AM57x, AM62x, AM64x, and AM65x. TI processors include various PRU subsystems such as PRU-ICSS, PRU-ICSSG, and PRUSS. This document focuses primarily on the PRUSS for the AM625 series. For specific differences among each PRU subsystem, please refer to the following document: PRU-ICSS Feature Comparison (Rev. H). The high-speed UART implementation described in this document is based on PRUSS and can, in theory, be directly ported to other PRU subsystems.

1.2 Application Background and Industrial Requirements for High-Speed UART

As modern industry demands higher production precision and monitoring frequency, UART communication is evolving toward higher frequencies, greater throughput, and megabit-per-second (Mbps) data rates. This trend often requires continuous data streaming to the main control unit at rates of several megabits per second. Such high-speed UART communication places higher demands on the real-time response capability and throughput performance of the underlying control system. In existing industrial embedded designs, the mainstream approach typically uses a general-purpose MPU (e.g., ARM Cortex-A series) running a general-purpose OS such as Linux, with the chip's built-in hardware UART controller handling UART data. This traditional "ARM + Linux + hardware UART" architecture performs well at low baud rates; however, in high-speed (Mbps-level) scenarios, its task scheduler, virtual memory management, and complex network protocol stack introduce unpredictable latency jitter. When a high-speed UART generates high-frequency interrupts, the thread-scheduling latency of the Linux OS typically ranges from tens to hundreds of microseconds. At Mbps-level baud rates, the transmission time for a single byte is only a few microseconds or a few hundred nanoseconds. Although hardware UART chips incorporate 16-byte or 64-byte FIFOs (First-In, First-Out buffers), under high-speed communication, these FIFOs can easily fill up within a few microseconds. If not serviced in time, this readily leads to overflow errors and packet loss. To prevent packet loss, the system is forced to expend a significant amount of computational power on serial UART processing.

TI processors support multiple UART interfaces for communication. Table 1-1 lists the number of UARTs supported by various TI processors and their maximum speeds. Taking AM62x as an example, the chip natively supports up to 9 UARTs. However, due to the limitations of traditional Linux and hardware UARTs, the maximum software-supported UART speed on TI Sitara processors is 3Mbps. Even though UARTs of TI processor support DMA mode and other optimization measures to reduce CPU usage during reception and transmission, actual tests with the traditional approach still cannot achieve more than 3Mbps throughput without adversely affecting other system tasks (i.e., without significantly increasing CPU load). Therefore, this document proposes using the PRU to simulate a UART in order to achieve a higher speed (12Mbps) without substantially increasing CPU load.

Table 1-1. Number of UARTs Supported by Sitara AM6x Series

Domain	AM62x	AM62L	AM62P	AM64x	AM62A	Max Speed	UART IP
WKUP	1	1	1	0	1	3Mbps	16C750
MCU	1	0	1	2	1	3Mbps	16C750
Main	7	7	7	7	7	3Mbps	16C750
Total	9	8	9	9	9	3Mbps	16C750

1.3 Technical Challenges of 12Mbps High-Speed UARTs

When using an ARM chip to achieve 12Mbps ultra-high-baud-rate UART communication, the CPU processing capability of traditional solutions faces enormous challenges. At 12Mbps, the transmission time for each data bit is approximately 83.3ns. Since standard UART communication typically uses 1 start bit, 8 data bits, and 1 stop bit (10 bits total) per byte, a complete byte is transmitted approximately every 0.83µs (i.e., 833ns). This means that with a traditional receive-interrupt mechanism, the CPU would need to handle up to 1.2 million interrupts per second (an interrupt frequency of 1.2MHz), which is completely unacceptable and impractical in embedded CPU

Copyright © 2026 Texas Instruments Incorporated

applications. Each time an interrupt is triggered, the CPU must perform a series of operations including context saving, jumping to the interrupt service routine, reading registers, and context restoration.

To allow the CPU to survive and operate correctly under such high data throughput, designers must completely eliminate CPU intervention in low-level data transfers and rely instead on the DMA (Direct Memory Access) controller. In DMA mode, the UART peripheral moves data directly to/from memory via the DMA bus during reception or transmission, without CPU involvement throughout the entire process. However, the design of the DMA transfer length is also a challenge. Setting it too large causes significant data latency, while setting it too small fails to substantially reduce the number of interrupts. Currently, in the TI SDK, the UART DMA mode is configured for single-byte transfers to ensure real-time performance of data; as a result, the problem of excessive interrupt generation has not been effectively solved.

On the hardware side, at 12Mbps, the data bit lasts only about 83ns. The TTL or CMOS level pins used for standard UART are severely affected by parasitic capacitance and inductance, causing slower signal rise/fall times and waveform distortion. Therefore, 12Mbps UART is suitable only for intra-PCB communication over short traces; for board-to-board communication, signal-integrity effects must be taken into account.

Given the inherent shortcomings of general-purpose processors in high-speed real-time communication and their inability to support high-speed UART, the PRU in Sitara processors offers a viable solution. The PRU guarantees absolute timing determinism at the physical boundary through its pipeline-free single-cycle execution mechanism and direct pin-mapping architecture. This highly deterministic physical-layer communication can be decoupled from the upper-layer complex non-real-time OS. The PRU can precisely simulate UART timing logic by counting clock cycles using deterministic firmware written in assembly or C language.

TX implementation: When data is detected in the transmit queue, the firmware pulls the TX pin low (start bit) and then calculates the required number of clock cycles per bit based on the configured baud rate. The firmware implements precise delays via loops or decrementing counters, and sequentially writes the data bits (Bit 0–7), parity bit, and stop bit to register R30.

RX implementation: The firmware captures the start bit by polling R31 or by triggering on a falling-edge interrupt on the pin. Once the start bit is captured, the PRU delays by 1.5 bit periods (skipping the start bit and centering on the middle of the first data bit) to read the data at the most stable sampling point, and then samples every subsequent bit period. This ensures an extremely low bit-error rate under high throughput.

Copyright © 2026 Texas Instruments Incorporated

2 Design and Implementation of PRU-Based High-Speed UART Communication System

2.1 UART Low-Level Physical-Layer Design

The PRU UART peripheral is based on the industry-standard TL16C550 asynchronous communication protocol, which itself is a functional upgrade of TL16C450. It supports FIFOs and can also support DMA mode. Both the receive and transmit FIFOs can store up to 16 bytes of data. The primary tasks of the PRU UART are to perform serial-to-parallel conversion on data received from the peripheral and parallel-to-serial conversion on data received from the CPU. For the PRU subsystem, a maximum 16-byte FIFO is entirely sufficient due to the PRU's deterministic timing characteristics, and it can be flexibly adjusted for actual application scenarios. Therefore, the 16C550 is chosen as the reference for implementing the PRU UART, minimizing resource consumption. The PRU high-speed UART is specifically designed for industrial real-time communication to support protocols based on serial interfaces, such as Modbus and Profibus. For example, Profibus-DP is a high-speed bus (up to 12Mbps) that uses a master-slave polling and token-passing mechanism. Because the line is half-duplex (RS-485), the slave must switch the physical direction from "receive" to "transmit" and send back data within a very short, fixed time (Tsl, slot time) after receiving a command from the master. If residual data remains in the FIFO, the driver cannot easily determine when the last bit has actually left the chip's transmitter. A small FIFO design allows the PRU to disable the RS-485 DE (Driver Enable) and enable RE (Receiver Enable) within microseconds at the very moment the last byte has finished transmitting. If the switching speed is slightly slow, the system will miss the next bus high level from the master, resulting in communication failure.

The maximum UART communication rate is primarily limited by signal quality and software architecture. Using the PRU can break through this limitation to achieve 12Mbps high-speed transmission. In the transmit protocol section, the design includes a Transmit Holding Register (THR) and a Transmit Shift Register (TSR). The THR is memory-mapped, specifically corresponding to the TBR_DATA bit field of the UART_RBR_TBR[17-8] register, while the TSR is not mapped to memory space. When the PRU UART is in FIFO mode, the THR functions as a 16-byte FIFO buffer. The control functions for the transmitter section are implemented by the PRU UART's Line Control Register, UART_LCTR. Based on the settings selected in this register, the PRU UART transmitter sends the following format to the receiving device:

1 START bit + data bits (5, 6, 7, 8) + 1 PARITY bit (optional) + STOP bit (1, 1.5, 2)

The THR receives data from the internal data bus. When the TSR is ready, the PRU UART transfers data from the THR into the TSR. The PRU UART then serializes the data in the TSR and transmits it via the UART*_TXD pin.

In non-FIFO mode, if the THR is empty and the Transmit Holding Register Empty (THRE) interrupt is enabled in the Interrupt Enable Register (UART_INT_EN[1] ETBEI), an interrupt is generated. This interrupt is cleared when a character is loaded into the THR or when the IIR_INTID bit field of the Interrupt Identification Register (UART_INT_FIFO[3-1]) is read. In FIFO mode, an interrupt is generated when the transmit FIFO is empty; it is cleared when at least one byte is loaded into the FIFO or when the UART_INT_FIFO[3-1] IIR_INTID bit field is read.

In the receive protocol section, the design includes a Receiver Shift Register (RSR) and a Receiver Buffer Register (RBR). The RSR is not mapped to memory space, while the RBR is memory-mapped, corresponding to the RBR_DATA bit field of the UART_RBR_TBR[7-0] register. When the PRU UART is in FIFO mode, the RBR functions as a 16-byte FIFO buffer. The control functions for the receiver section are implemented by the PRU UART's Line Control Register, UART_LCTR. Based on the settings selected in this register, the PRU UART receiver accepts the following format from the transmitting device:

1 START bit + data bits (5, 6, 7, 8) + 1 PARITY bit (optional) + 1 STOP bit (Any additional stop bits transmitted by the sender along with the above data will not be detected).

The RSR is responsible for receiving data bits from the UART*_RXD pin. Subsequently, the RSR assembles these data bits (serial-to-parallel conversion) and transfers the combined value into the RBR (or receive FIFO). This value can be read by accessing the RBR_DATA bit field of the RBR_TBR[7-0] register. In addition, the PRU UART stores 3 bits of error-status information alongside each received character, indicating whether a parity error, framing error, or break condition occurred.

In non-FIFO mode, when a character is placed into the RBR and the "Receive Data Available" interrupt is enabled in the Interrupt Enable Register (UART_INT_EN[0] ERBI), an interrupt is generated. This interrupt is

Copyright © 2026 Texas Instruments Incorporated

cleared when the character is read from the RBR. In FIFO mode, an interrupt is generated when the FIFO is filled to the trigger level selected by the FIFO control MSB field of the UART_INT_FIFO register; this interrupt is cleared when the FIFO contents (byte count) fall below this trigger level.

The PRU UART design must support baud-rate changes and interaction with the ARM application layer. The system can be customized for actual use cases to minimize software management of the communication link. It must include a programmable baud-rate generator that can divide the PRU UART input clock by a divisor (division factor) from 1 to 65535, generating a 16× or 13× reference clock (oversampling clock) for the internal transmit and receive logic.

2.2 Implementation of Multi-Channel High-Speed UART by PRU Software Using the PEFIF Interface

Since the PRU internally integrates only one hardware high-speed UART channel, when multiple high-speed UARTs are required, we can use the PRU I/O's PEFIF (Peripheral Interface) mode for hardware-assisted reception and transmission of UART data. The three-channel peripheral interface present on each PRU is typically used to support serial-based motor encoder protocols such as EnDat 2.2 and BiSS. Each transmit PERIF channel has a 32-bit FIFO for directly storing transmit data, and each receive PERIF channel has a 4-bit FIFO for receiving data. We can use this interface to send and receive data via the UART protocol.

The architecture of the low-latency, multi-channel high-speed UART transmission system based on the PRU, i.e., the hardware implementation of the data-link layer, is shown in Figure 3-1. By hardware-accelerating traditional low-speed/high-latency tasks and using the PRU real-time core for data scheduling, it achieves extremely high throughput and microsecond-level ultra-low deterministic latency.

The entire architecture is physically and logically divided into two core latency domains to isolate the uncertainty caused by system scheduling on the ARM side. The high-latency domain primarily includes the ARM core (responsible for high-level application logic), DMASS (DMA system), general-purpose timers, standard UARTs (with 64-byte TX-FIFO), MSRAM, and DDR memory. Processing in the above components involves multiple layers of bus arbitration and general-purpose OS scheduling, resulting in an overall access latency of approximately 250ns that varies with system load, with jitter potentially reaching several microseconds. The low-latency domain mainly consists of the ICSS internal interconnect bus (32-bit, 250MHz), PRU 0/PRU 1 dual cores, dedicated DMA (XFR2VBUS RD DMA), SPAD shared memory, and peripheral PERIF TX/RX hardware. Based on the 250MHz ICSS bus, it achieves a read latency of 9ns and a write latency of 6ns. Furthermore, to achieve smooth and efficient data transfers from the high-latency storage area to the real-time transmitter, the system employs a dedicated channel architecture with Ping-Pong buffer channels. Four independent logical channels (Channel 1–4) are deployed in the low-latency domain, each equipped with a double-buffer structure (e.g., Channel 1 includes Tx1a and Tx1b for Ping-Pong operation), with each individual buffer being 128 bytes in size. This design allows the DMA to transfer the current data block while the PRU simultaneously reads and transmits the other data block without contention. The ARM interacts with the PRU core via the INTC (Interrupt Controller). The ARM transfers and refreshes the data for these four channels via the Main CBASS. Refreshing 88 bytes of data each time takes approximately 70µs. Transmitting 128 bytes (1280 bits) at 12Mbps takes about 106.6µs. Therefore, the 70µs window is sufficient for UART transmit and receive requirements. The core computation and control of UART data are performed cooperatively by the two PRU cores running at 333MHz, with PRU 0 dedicated to UART data transmission for all four channels. The transmit data is directly fetched from the channel buffers on the PRU-ICSS interconnect bus through the dedicated XFR2VBUS RD DMA, without requiring intervention by the ARM core. PRU 1 is dedicated to UART data reception for the four channels, feeding received data back to the bus via the lower XFR2VBUS RD DMA.

Copyright © 2026 Texas Instruments Incorporated

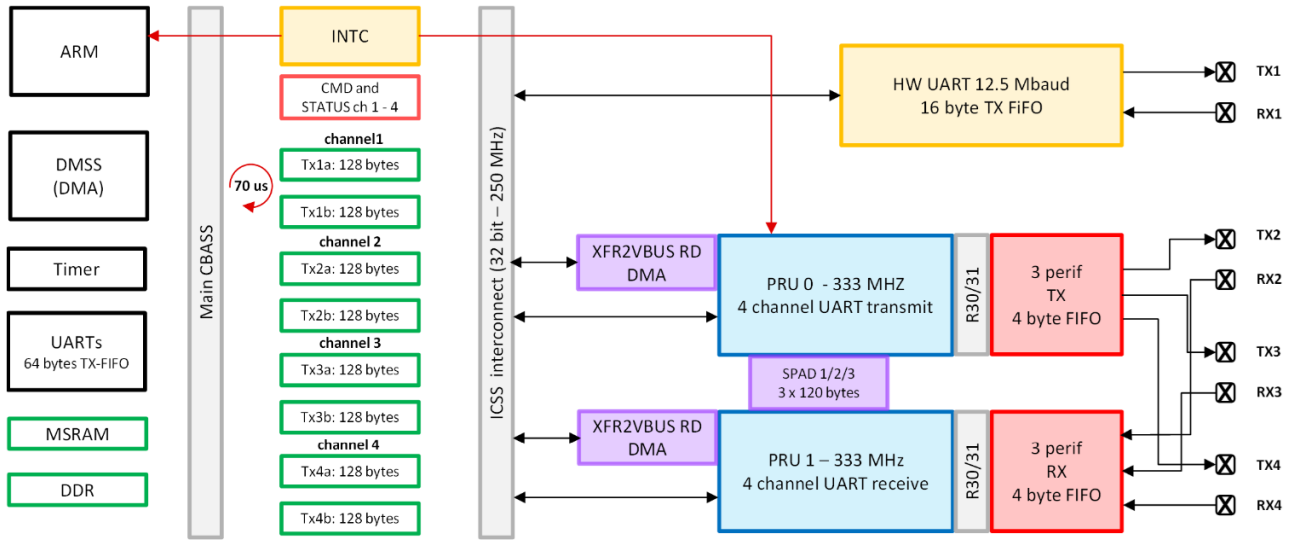

Figure 2-1. PRU UART Data-Link Layer Hardware Implementation Design

Transmit data is ultimately serialized and output through hardware peripherals and pins, exhibiting extremely high determinism: the high-speed hardware UART is directly integrated in the low-latency domain, supports rates up to 12 Mbps, and includes a built-in 16-byte TX FIFO. The other three channels utilize the PRU-specific peripheral transmit and receive interfaces (3 PERIF TX/RX). The PRU core drives these PERIF TX and RX interfaces directly via the dedicated R30/R31 register interface with extremely low jitter, outputting to the physical pins. Even under high throughput, there is no long-delay idle between serial frames. In this design, full-speed 12Mbps data transmission is achieved. Through the SPAD (Scratchpad shared memory), PRU0 and PRU1 can exchange data directly at high speed with zero latency, which can be used directly for inter-core synchronization of PRU cores.

Copyright © 2026 Texas Instruments Incorporated

2.3 Design of Application-Layer Driver for UARTs on ARM Linux

To lower the integration barrier for end users, the driver layer must be encapsulated using the Linux character-device or TTY core framework (tty_driver), abstracting a standard POSIX UART API to user space. The ultimate goal is that, for end customers, using the PRU UART is indistinguishable from using a regular UART.

The Linux kernel manages PRU startup and firmware through the standard remoteproc (remote processor control) framework. During the firmware loading phase, at Linux boot, the Linux-side remoteproc driver loads the corresponding PRU binary from the path "/lib/firmware/", parses it, and loads it into the PRU's dedicated instruction memory (IRAM). A resource table is embedded in the PRU firmware. By reading this table, Linux can determine which system resources the PRU has requested (such as virtual interrupt numbers, RPMsg channels, and reserved memory segments). The inter-processor communication (IPC) mechanism uses RPMsg and the hardware Mailbox to implement low-frequency, non-real-time control commands such as baud-rate changes, parity configuration, and flow-control start/stop. The system directly utilizes the PRU shared memory inside the AM62, which is adjacent to the PRU cores and accessible by both the PRU and ARM. Using virtual address mapping (ioremap): In the Linux kernel driver, the shared physical memory is mapped to the Linux kernel's virtual address space via the ioremap() function.

The PRU firmware only needs to handle UART waveform generation and memory data parsing, while the ARM provides the data to be transmitted to the PRU. This reduces the number of interrupts and Linux scheduling overhead, transforming the operation into periodic data handling occurring approximately every 70µs. Upper-layer applications can use mmap() or even directly map user-space addresses to PRU shared memory, eliminating memory copying for data from the physical pins to the user-space program. The interface between the ARM and the PRU firmware is a simple Ping-Pong buffer with command and status registers. The current design uses PRU Data RAM 0, with a global physical address mapped on the ARM side (A53 core) of 0x30040000. The data buffer is located at offset 0; the command and status registers are located at offset 0x400 (i.e., 1024 bytes in decimal).

From the driver side, the specific programming/execution sequence is as follows:

- Open and initialize the driver: (This step downloads and starts the PRU firmware; pin muxing and clock settings are currently configured inside the PRU).
- Set baud rate: (This step does not involve the PRU firmware; it is programmed directly by the ARM. The PRU is set to 12Mbps by default at startup. The ARM can modify the baud rate before loading the first frame and trigger transmission via the command register).
- Check the status of transmit (TX) channel buffer A: ensure TXA_CH0_STATUS! = STATUS_ACTIVE (i.e., confirm that buffer A is currently inactive/idle).
- Load the data frame into transmit frame buffer A (i.e., TX_CH0_A at the corresponding physical address).
- Set the command register TXA_CH0_CMD = CMD_FULL: (The PRU firmware will poll and detect the CMD_FULL command, then start transmitting the frame at the configured baud rate).
- Simultaneously, the ARM can process transmit buffer B (i.e., TX_CH0_B) in parallel.
- When the PRU finishes transmitting the frame, it updates the status register to TXA_CH0_STATUS = STATUS_DONE.
- Once the ARM detects this status, it can read/write transmit buffer A (TX_CH0_A) again.
- The above steps are repeated for all channels.

The driver will create /dev/ttyPRU0/1/2/3 nodes in Linux user space. User space can use these four nodes to transmit and receive UART data on the corresponding UART channels.

Copyright © 2026 Texas Instruments Incorporated

3 Experimental Verification of High-Speed UARTs

After completing the above theoretical software design, the functionality of the implemented high-speed UARTs for the PRU can be tested using a TI EVM.

3.1 Bare-Metal Experiment of High-Speed PRU UART

The corresponding pins for the PRU IO can be tested on the SK-AM62B-K1 Board with the following hardware pin distribution:

3 channel RX: J10.14 – PRU1_GPI9 -- N23; J10.15 – PRU1_GPI10 – N24; J10.16 – PRU1_GPI11 – N25

3 channel TX: AA3, SOC_MMC0_DAT2 rework via R376. PRU0_GPO1 – AA3; J10.17 – PRU0_GPO4 – P24; J10.20 – PRU0_GPO7 – R23

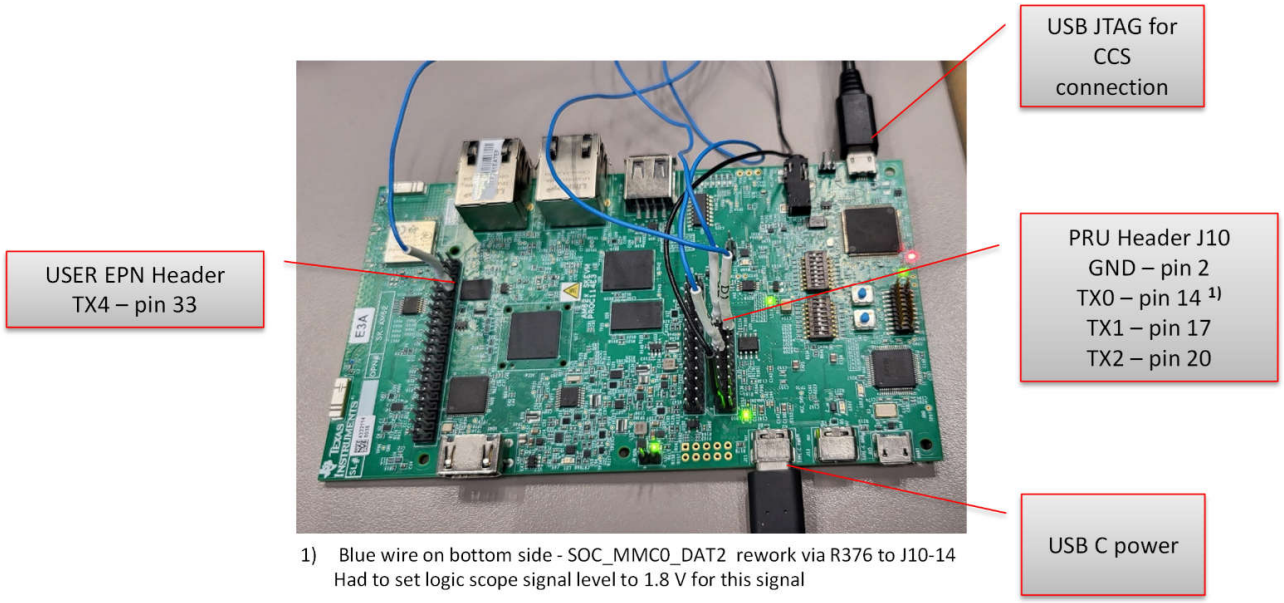

Figure 3-1. Hardware Environment Setup Based on AM62 SK EVM

The software environment primarily uses CCS via JTAG for debugging. The CCS version used for testing is 12.8.1. After connecting to the AM62 EVM, perform initialization as described in the referenced link. Then connect to the PRU core, load and run PRU_UART_12Mbps.out. In actual project applications, it can be loaded and run via Linux remoteproc. In CCS, memory operations can be easily performed to observe the UART waveforms output on the corresponding pins.

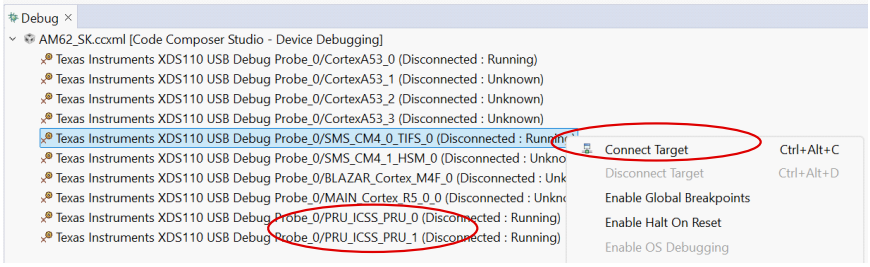

Figure 3-2. Software Environment Setup Based on AM62 SK EVM

3.2 High-Speed PRU UART Application Based on Linux

To apply and experimentally verify the PRU's high-speed UARTs based on SDK 11.1, the PRU UART driver must first be added to the native kernel. Apply the following commands and the corresponding patch.

Copyright © 2026 Texas Instruments Incorporated

1. Apply the kernel patch to the SDK 11.1 kernel, recompile the kernel, and install the new kernel image (Image), modules, and k3-am625-sk.dtb into the root file system (rootfs) on the SD card.

```
host# patch -p1 < pru-tty-driver-v6.12-20251204.diff
host# make linux
host# sudo DESTDIR=/media/tony/root make linux_install
```

2. Copy the PRU firmware to the /lib/firmware/ directory of the root file system, and then create a soft link named am62x-pru0-fw pointing to it.

```
# ln -sf PRU_UART_12Mb.out.v1.1 am62x-pru0-fw
```

3. Boot the AM62x SK EVM development board. The new pru-tty driver is built into the kernel image and creates 4 UART nodes:

```
# ls /dev/ttyPRU*
  /dev/ttyPRU0  /dev/ttyPRU1  /dev/ttyPRU2  /dev/ttyPRU3
```

4. Run the following command to start PRU0.

```
root@am62xx-evm:~# echo start > /sys/class/remoteproc/remoteproc2/state
[  703.729726] remoteproc remoteproc2: powering up 30074000.pru
[  703.737406] remoteproc remoteproc2: Booting fw image am62x-pru0-fw, size 25852
[  703.744705] remoteproc remoteproc2: remote processor 30074000.pru is now up
```

   Now you can run UART applications to write data to these UART nodes. Currently, simply using the open() and write() functions enables high-speed UART data transmission via the PRU. The following is reference code for transmitting data using the open() and write() functions.

```
write(fp0, data, data_len);
fp0 = open("/dev/ttyPRU0",  O_RDWR | O_NONBLOCK);
```

Figure 3-3 shows the waveform measured using a logic analyzer to verify the entire data path from the application layer to the underlying hardware by transmitting the data randomly generated on the Linux side.

Copyright © 2026 Texas Instruments Incorporated

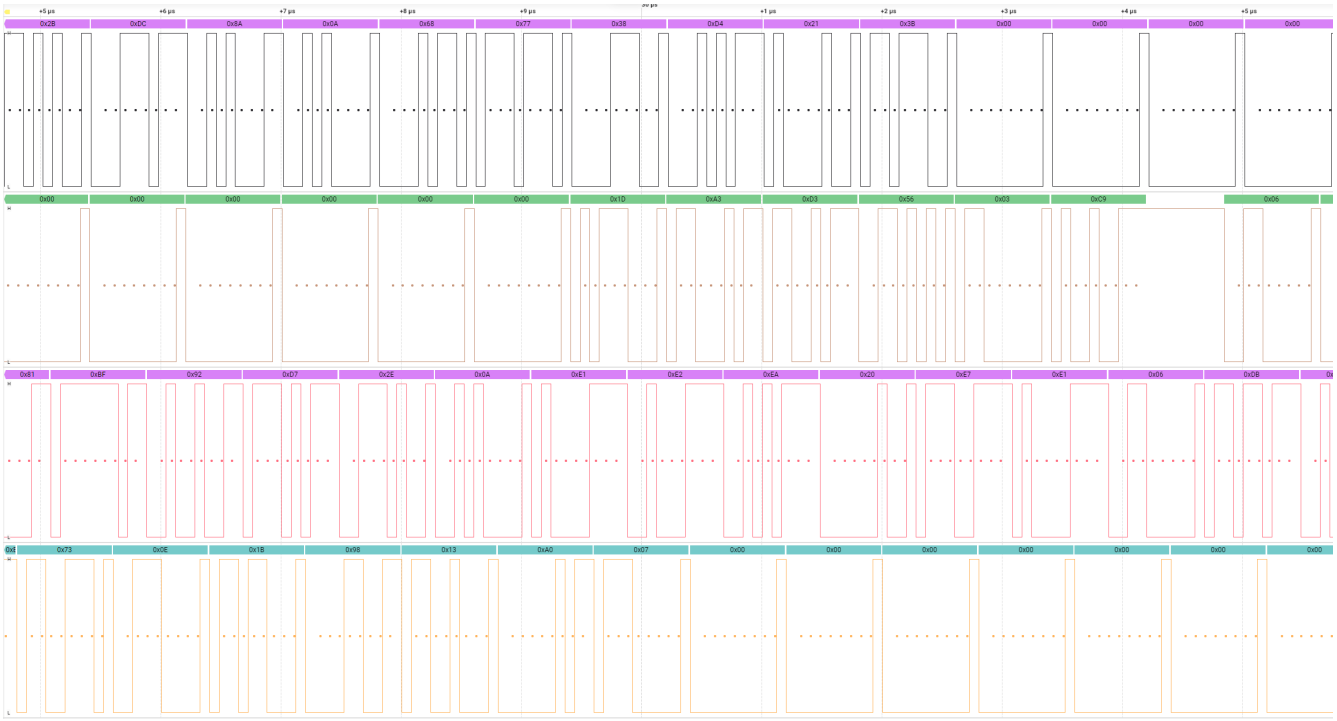

Figure 3-3. Random UART Data Waveform At Linux Side

Copyright © 2026 Texas Instruments Incorporated

4 Conclusion

This document introduces a high-speed (12Mb) UART communication scheme developed based on TI Sitara processors using the Programmable Real-Time Unit (PRU). By leveraging the PRU's deterministic architecture to simulate UART communication, the scheme achieves multi-channel high-speed (up to 12Mbps), loss-free UART communication without additional hardware cost. The design establishes an efficient data channel through the shared-memory mechanism between the PRU and ARM cores, significantly reducing CPU load. It provides a low-cost, high-reliability alternative solution for high-speed UART communication in industrial automation scenarios.

5 References

1. PRU-ICSS Feature Comparison (Rev. H)
2. PRU Getting Started Labs
3. AM62x Sitara™ Processors datasheet (Rev. C)
4. open-pru/examples/uart_12mb at eb29734cbc0913e28ff71c6e046d9b10a1c35e5d · TexasInstruments/open-pru · GitHub
5. PRU-ICSS Getting Starting Guide on Linux (Rev. A)
6. Programmable real-time unit and industrial communications subsystem | TI.com
7. [FAQ] What is a PRU core? Why are PRU GPIO signals different from regular GPIOs? - Processors forum - Processors - TI - E2E support forums
8. AM62x MCU+ SDK: CCS Launch, Load and Run
9. AM62x Sitara Processors Technical Reference Manual (Rev. C)

Copyright © 2026 Texas Instruments Incorporated

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Copyright © 2026, Texas Instruments Incorporated

Last updated 10/2025