Application Note

Intra Drive Communication Using 8b-10b Line Code With Programmable Real Time Unit

Chen Gao, Thomas Leyer

ABSTRACT

In industrial motor drive systems, multiple devices or chips are often required to communicate with each other in a quick speed, low-jitter, low-latency, and synchronized manner. The typical example application is intra drive communications with either standard interface (like serial peripheral interface) or custom protocols in physical layer (like 8b-10b line code).

Programmable Real-time Unit (PRU) is unique because it can execute single-cycle functions with 960-bit-wide data bus which results in no jitter for the user in real-time execution of communication and control applications. TI Sitara™ processors offer two types of PRU subsystems: PRU-ICSS and PRU_ICSSG. PRU-ICSS is available for the AM335x, AM437x and AM57x series. PRU_ICSSG is available on the AM243x, AM65x and AM64x.

This application note describes how to use PRU to implement 8b-10b line coding for intra drive communication with 100Mbps data rate. This document also introduces Low-voltage Differential Signaling (LVDS) and Multipoint Low Voltage Differential Signaling (M-LVDS) interfaces for high-speed signaling.

Table of Contents

List of Figures

List of Tables

Trademarks

Sitara™ and LaunchPad™ are trademarks of Texas Instruments.
Arm® is a registered trademark of Arm Limited.
All trademarks are the property of their respective owners.

1 Introduction to 8b-10b Line Coding

As the scheme name suggests, 8b-10b means that 8 bits of data are transmitted as 10-bit symbols to achieve DC balance and bounded disparity. The low five bits of data are encoded into a 6-bit group (the 5b and 6b section) and the top three bits are encoded into a 4-bit group (the 3b and 4b section). These code groups are concatenated together to form the 10-bit symbol that is transmitted on the wire and received, then decoded in reverse. This algorithm is invented and patented by IBM Corporation.

To balance the DC signals during high-frequency data receiving, the two inserted control bits help to make the difference between the counts of 0 and 1 in a string of at least 20 bits, is no more than two, and there are not more than five 1s or 0s in a row. This scheme helps to balance the DC signals during high frequency data receiving and also reduce the demand for the lower bandwidth limit of the channel necessary to transfer the signal. However, using the 8b-10b encoding algorithm adds 25% overhead to each character. This character overhead is not the only over-head, however it is the most significant factor. The example for 8b-10b encoding is shown in Figure 1-1.

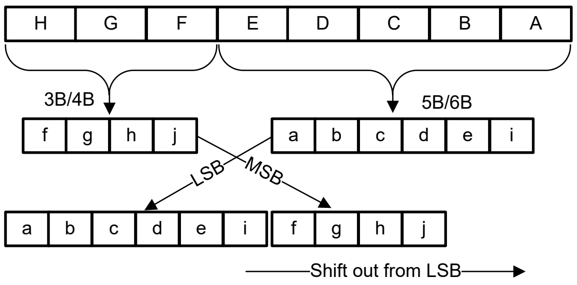


Figure 1-1. 8b-10b encoding

2 PRU Implementation for Data Transmitting and Receiving

2.1 Encoding and Decoding Data

Typically, there are four tables written into the memory in advance, including an encoding 5b-6b table, encoding 3b-4b table, decoding 6b-5b table, and decoding 4b-3b table. Data has to be encoded using a look-up table (LUT) prior to transmission and decoded with the LUT after receiving. To demonstrate encoding and decoding data, the Sitara™ AM243x LaunchPad™ development kit is selected as the device for verification. The HW_WR_REG8 function can be used to put the four tables into PRU Dynamic Random Access Memory (DRAM) with different offset addresses using the C code of the Arm® core project, see also the following code.

```
uint32_t enc_5b6b = CSL_PRU_ICSSG0_DRAM1_SLV_RAM_BASE + ENC_5B6B_OFFS;
uint32_t enc_3b4b = CSL_PRU_ICSSG0_DRAM1_SLV_RAM_BASE + ENC_3B4B_OFFS;
uint32_t dec_5b6b = CSL_PRU_ICSSG0_DRAM0_SLV_RAM_BASE + DEC_5B6B_OFFS;
uint32_t dec_3b4b = CSL_PRU_ICSSG0_DRAM0_SLV_RAM_BASE + DEC_3B4B_OFFS;
    //Encoding LUTs (input MSB first, output LSB first)
    //LUT 5b/6b encoding
HW_WR_REG8(enc_5b6b + 0x00, 0x18);
...
    //LUT 3b/4b encoding
HW_WR_REG8(enc_3b4b + 0x00, 0x04);
...
    //Decoding LUTs (input LSB first, output MSB first)
    //LUT 6b/5b decoding
HW_WR_REG8(dec_5b6b + 0x00, INVAL);
...
    //LUT 4b/3b decoding
HW_WR_REG8(dec_3b4b + 0x00, INVAL);
...
```

Data can be encoded by the LUT in a PRU firmware project using load byte burst (LBBO) instruction. The LBBO instruction is used to read a block of data from memory into the register file. The REG_TMP11 register stores the LUT header address and REG_ENC register stores the original data and encoded data. See the following code of encode 8-bit data (0x34):

```
Ldi REG_ENC.b0, 0x14                         ;raw data 5b LSB
ldi REG_TMP11, (PDMEM00+LUT_5b6b_ENC)        ; TEMP11 for 5b/6b LUT header
lbbo &REG_ENC.b3, REG_TMP11, REG_ENC.b0, 1 ; FNC.b0 for original 5 bit -data, ENC.b3 for encoded 6
bit
ldi REG_ENC.b1, 0x01                         ; raw data 3b MSB
ldi REG_TMP11, (PDMEM00+LUT_3b4b_ENC)        ; TEMP11 for 3b/4b LUT header
lbbo &REG_ENC.b2, REG_TMP11, REG_ENC.b1, 1 ; FNC.b1 for original 3 bit -data, ENC.b2 for encoded 4
bit
```

The lower five bits of 0x34 is 0x14 and these bits load immediately into byte 0 of REG_ENC. After the LUT, the six encoded bits are written into byte 3 of REG_ENC. The higher three bits of 0x34 is 0x01 and these bits load immediately into byte 1 of REG_ENC. After LUT, the four encoded bits are written into the byte 2 of REG_ENC.

Figure 2-1 shows the REG_ENC register distribution for the original data and encoded data.

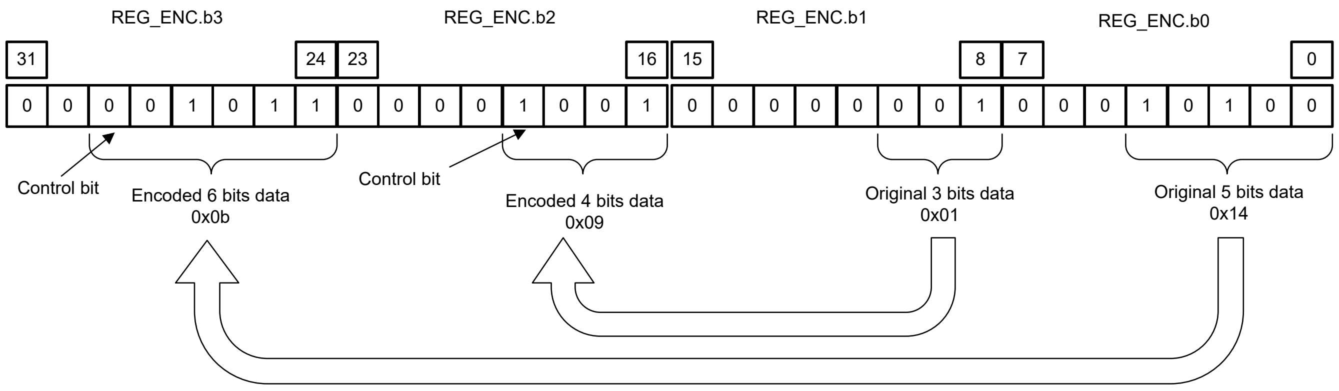


Figure 2-1. REG_ENC Encoding Register Distribution

The decoding process is nearly identical to the encoding process. The decoding process also uses LBBO instructions with the LUT for decoding after receiving the encoded data. The REG_TMP11 register stores the LUT header address and the REG_DEC register stores the encoded data and decoded data.

The six encoded bits are received and moved into byte 1 of REG_DEC. After the LUT, the five decoded bits are written into the byte 2 of REG_DEC. The higher four bits are received and moved into byte 0 of REG_DEC. After LUT, the three decoded bits are written into byte 3 of REG_DEC. To combine the 10 bits of data from two bytes of the register into eight bits of data, byte 3 of REG_DEC needs to be shift left by five bits and logically added with byte 2 of REG_DEC. The final decoded eight bits of data is stored in byte 0 of REG_DEC. The following code shows the decoding process.

```
ldi REG_TMP11, (PDMEM00+LUT_5b6b_DEC)        ; TEMP11 for 5b/6b LUT header
lbbo &REG_DEC.b2, REG_TMP11, REG_DEC.b1, 1 ; decode 6b
ldi REG_TMP11, (PDMEM00+LUT_3b4b_DEC)        ; TEMP11 for 3b/4b LUT header
lbbo &REG_DEC.b3, REG_TMP11, REG_DEC.b0, 1 ; decode 4b
lsl REG_DEC.b3, REG_DEC.b3, 5                ; shift left 5 bit
add REG_DEC, REG_DEC.b2, REG_DEC.b3          ; combine to 8 bit data
```

Figure 2-2 shows the REG_DEC register distribution for encoded data and decoded data.

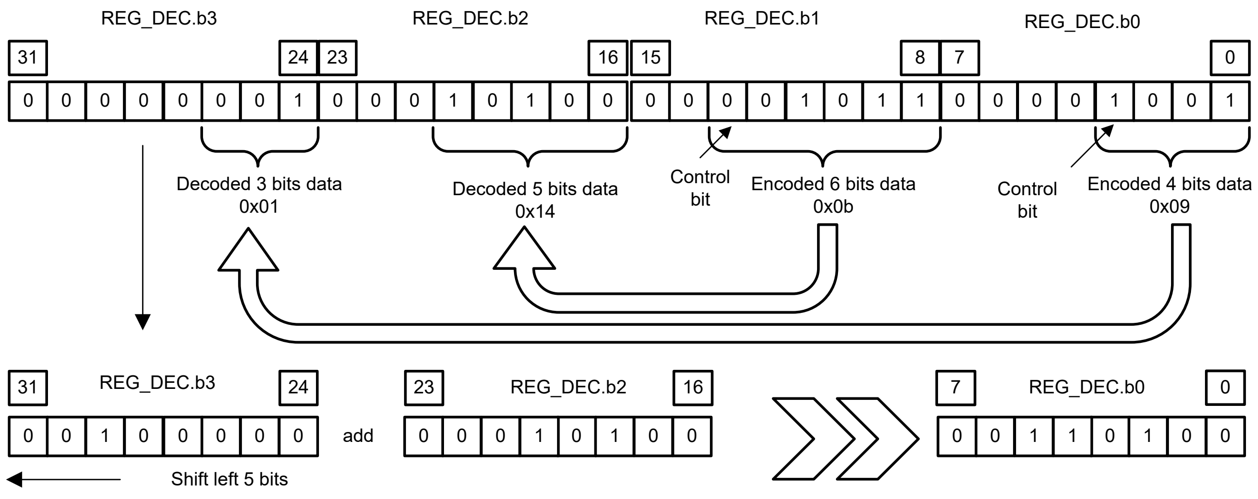


Figure 2-2. REG_DEC Decoding Register Distribution

2.2 PRU Module Interface and GPIO Mode

The PRU module interface consists of the PRU internal registers 30 and 31 (R30 and R31). Figure 2-3 shows the PRU module interface and the functionality of R30 and R31. Register R31 serves as an interface between the dedicated PRU general purpose input (GPI) pins and the PRU Interrupt controller (INTC). Reading R31 returns status information from the GPI pins and PRU INTC using the PRU Real Time Status Interface. Writing to R31 generates PRU system events through the PRU Event Interface. Register R30 serves as an interface with the dedicated PRU general purpose output (GPO) pins.

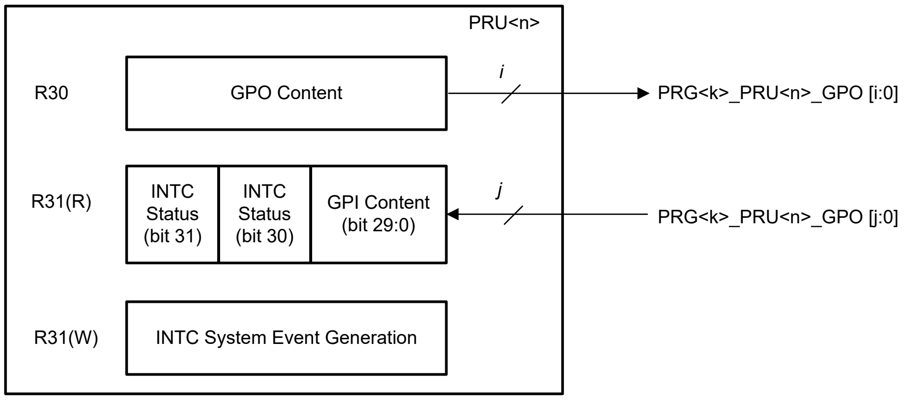


Figure 2-3. PRU Module Interface

The PRU Event Interface directly sends pulsed event information out of the internal Arithmetic and Logic Unit (ALU) of the PRU. These events are exported out of the PRU and need to be connected to the system interrupt controller at the System on Chip (SoC) level. The event interface can be used by the firmware to create software interrupts from the PRU to the Arm® core (host processor). For example, the event can be generated when the communication frame package completes to signal the Arm® for an interrupt.

The PRU implements an enhanced general-purpose input or output (GPIO) module that supports the following general-purpose input modes: direct input, 16-bit parallel capture, 28-bit serial shift in, and MII_RT (Ethernet MAC Interface). Register R31 serves as an interface with the general-purpose inputs. R31 also supports two general-purpose output modes: direct output and shift out. Register R30 serves as an interface with the general-purpose outputs.

2.3 PRU GPIO Shift-out and Shift-in Mode for Communication

The PRU Enhanced General Purpose Output (EGPO) can be configured in shift-out mode for data transmitting, data is shifted out of PRU GPO0 (DATAOUT) on every rising edge of PRU GPO1 (CLOCKOUT). The shift rate is controlled by the effective divisor of two cascaded dividers applied to the PRU core clock. To achieve a 100Mbit data rate with 8b-10b coding, a 125-MHz clock has to be used with a 250-MHz core clock divided by two. Shift-out mode supports two clocking sub-modes: Free Running Clock Mode (default) and Fixed Clock Count Mode. For fixed mode, packet lengths can be configured up to maximum 255 bits and there is an additional PRU_GPO_SHIFT_CLK_DONE flag set in bit 17 of the ICSSG_GPECFG<n>_REG register.

Two 16-bit shadow registers (GPO_SH0 and GPO_SH1) are used to support ping-pong buffers. Each shadow register has independent load controls programmable through bit 29 and 30 of the PRU R30 register. Note that GP_SH0 register needs to load 0x8000 as the start frame since the shift-in GPI detects the start bit as the first 1 or 0, which is not included in the data frame. After loading the start frame, the updated data can be loaded into GP_SH0 without any constrain. Figure 2-4 shows the block diagram of PRU shift-out mode.

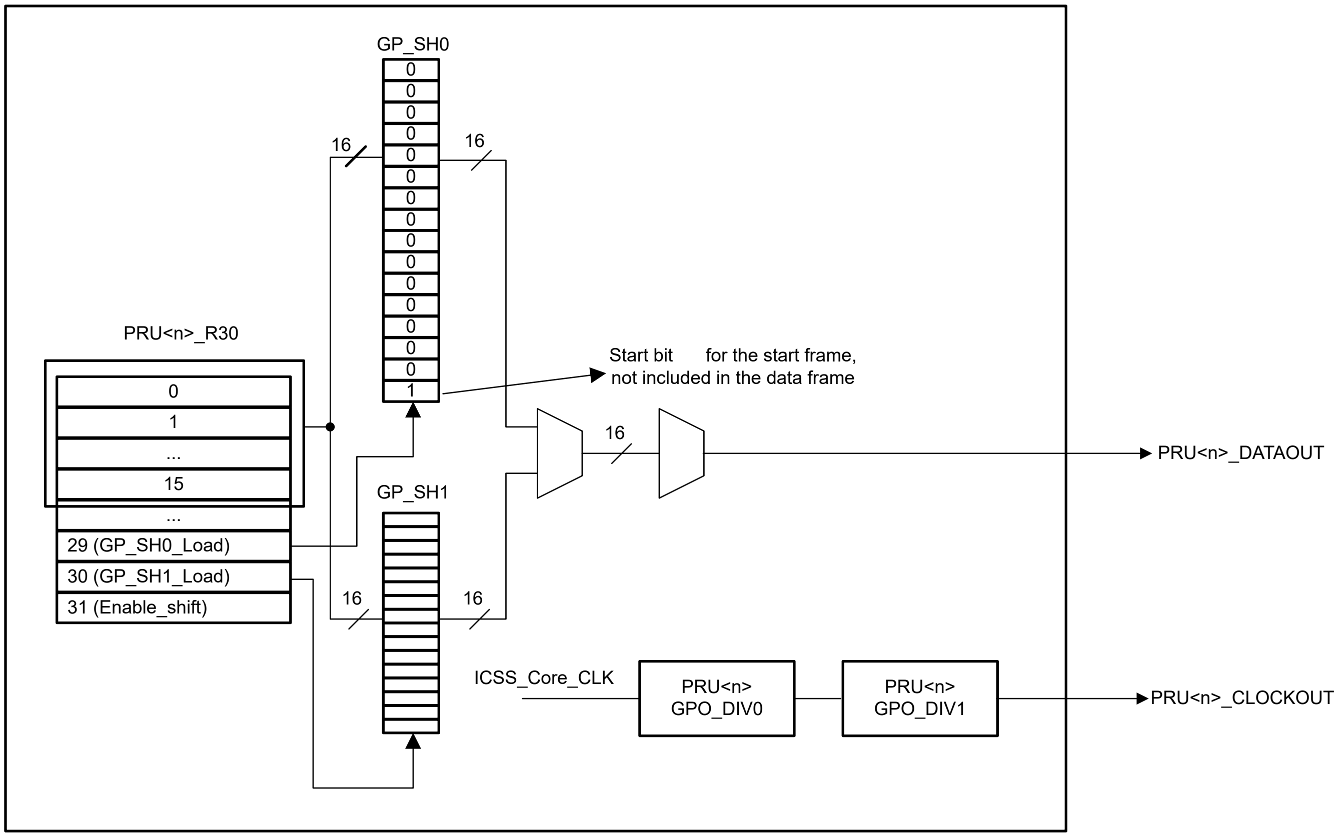


Figure 2-4. PRU Shift-out Mode Block Diagram

The programming model of PRU GPO Shift out mode is shown in Figure 2-5.

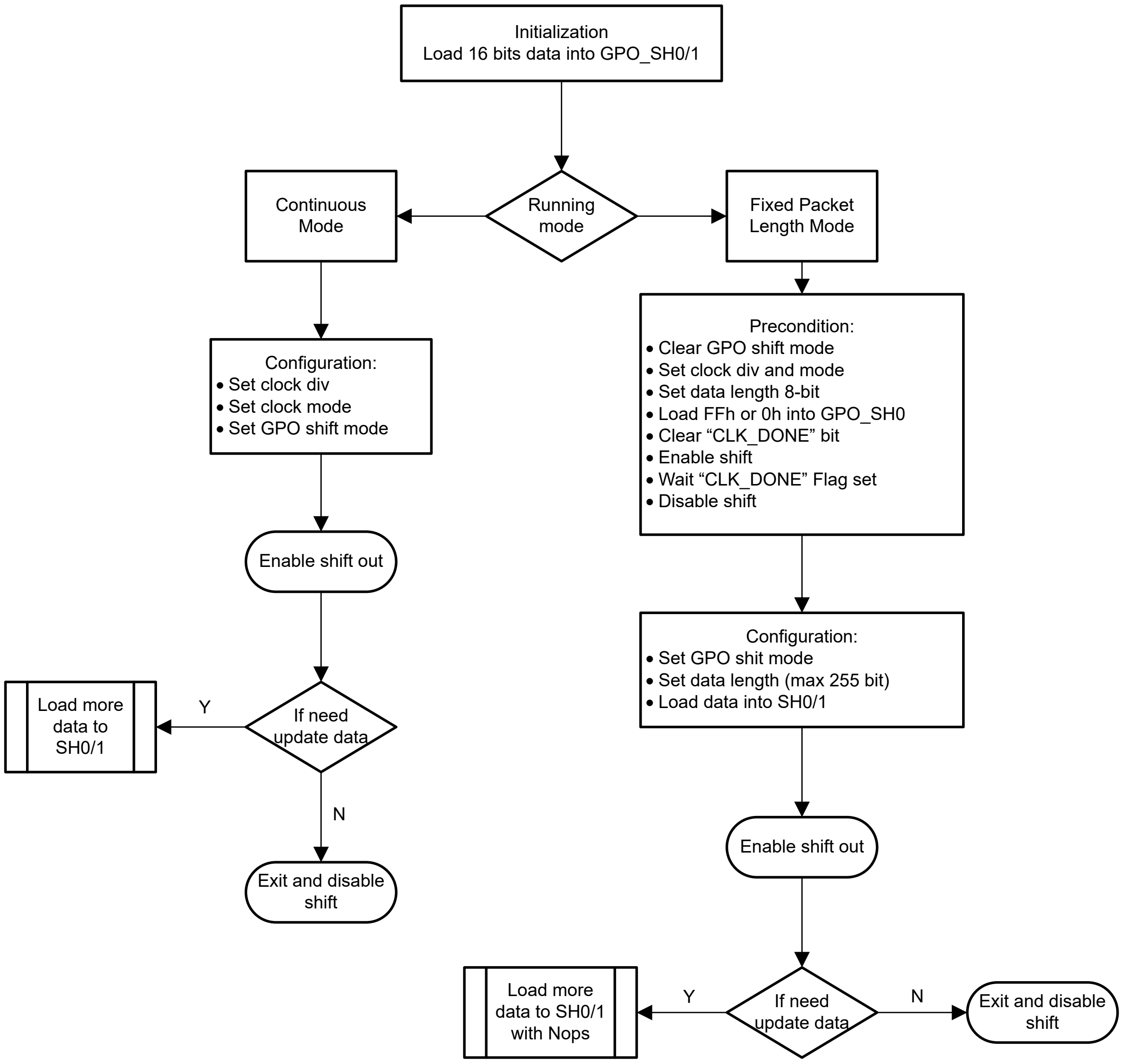


Figure 2-5. Programming Model of PRU Shift-out Mode

The GPI shift-in mode can be configured for data receiving. In 28-bit shift-in mode, general-purpose input pin PRU<n>_DATAIN is sampled and shifted into a 28-bit shift register on an internal clock pulse. The register fills in least-significant bit (LSB) order (from bit 0 to 27) and then overflows into a bit bucket. The 28-bit register is mapped to pru<n>_r31_status [0:27]. The shift rate is controlled by the effective divisor of two cascaded dividers applied to the PRU core clock. Similarly to shift-out mode, a 125-MHz sampling clock with 250-MHz core clock is selected. Due to the shift bit counter, CNT_16 sets and self-clears every 16 shifted clock samples after the start bit has been received, then only a 16-bit size out of the 28-bit shift register can be utilized as the receiving buffer.

Figure 2-6 shows the block diagram of the PRU GPI shift-in mode and Figure 2-7 shows the programming model.

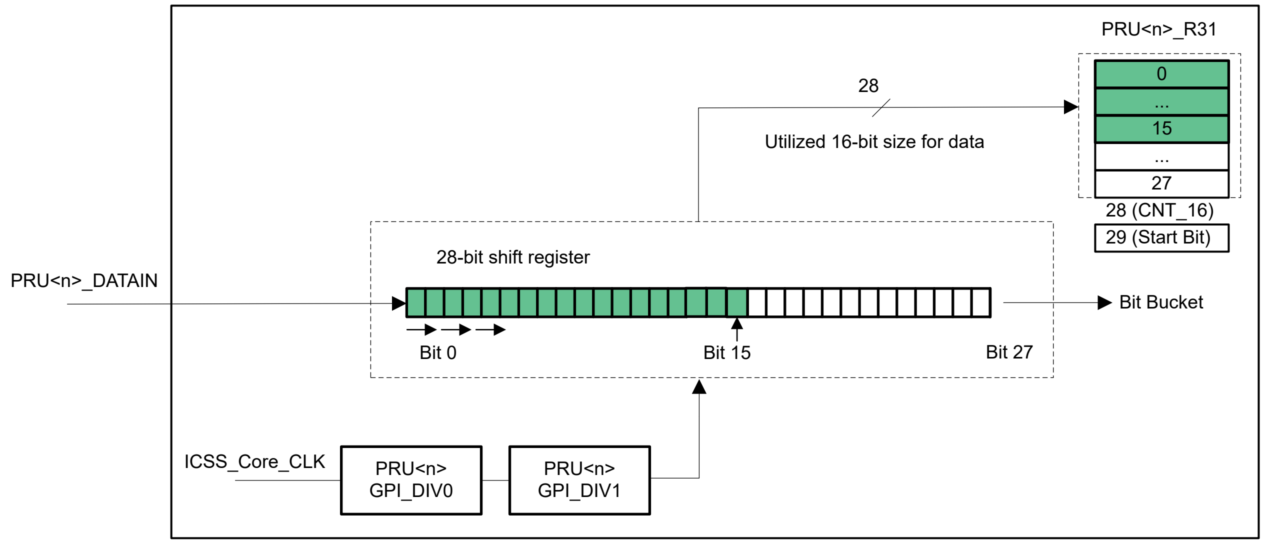


Figure 2-6. PRU GPI Shift-in Mode Block Diagram

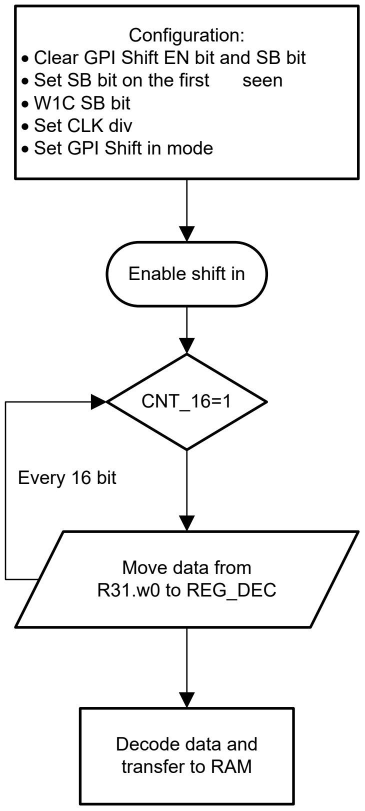
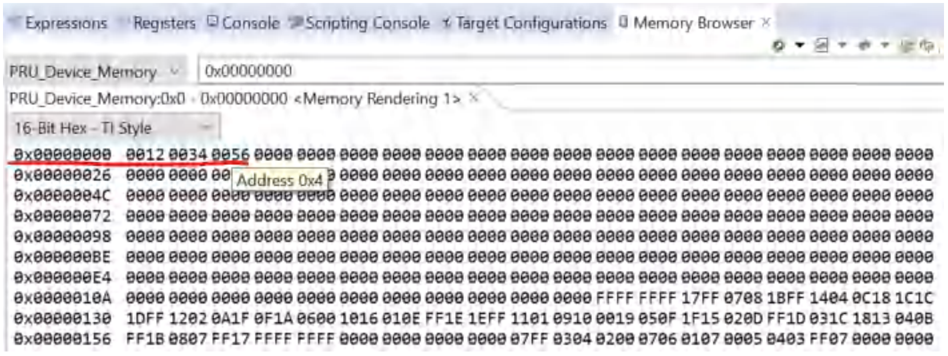

Configuration:
• Clear GPI Shift EN bit and SB bit
• Set SB bit on the first seen
• W1C SB bit
• Set CLK div
• Set GPI Shift in mode
Enable shift in
CNT_16=1
Every 16 bit
Move data from R31.w0 to REG_DEC
Decode data and transfer to RAM

Figure 2-7. Programming Model of PRU Shift-in Mode

To verify the accuracy of data transmitting with PRU_GPIO shift mode, 64-bit widths encoded data (0x8000 0012 0034 0056) are sent by the shift-out GPO of the PRU1 core with the first 16-bit 0x8000 (data frame is following the first 1) and received by shift-in GPI of the PRU0 core. The data is then moved from the PRU0 register file to PRU0 RAM from starting address 0x00000000. Both the shift-out and shift-in clock are 125 MHz. Figure 2-8 shows that the PRU1 core received the correct data without errors.



```
0x00000000  0012 0034 0056 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x00000026  0000 0000 00               0 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000004C  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x00000072  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x00000098  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x000000BE  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x000000E4  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0x0000010A  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 FFFF FFFF 17FF 0708 1BFF 1404 0C18 1C1C
0x00000130  1DFF 1202 0A1F 0F1A 0600 1016 010E FF1E 1EFF 1101 0910 0019 050F 1F15 020D FF1D 031C 1813 040B
0x00000156  FF1B 0807 FF17 FFFF FFFF 0000 0000 0000 0000 07FF 0304 0200 0706 0107 0005 0403 FF07 0000 0000
```

Figure 2-8. Verification of Data Transmitting With PRU GPIO Shift Mode

2.4 Three-channel Peripheral Interface for Communication

The 3-channel peripheral interface can also support data transmission that is similar to the GPIO shift mode. Since this application note is focused on the GPIO shift mode, the 3-channel peripheral interface is only briefly introduced in this section. The detailed introduction of 3-channel peripheral mode can be found in the AM62x Processors Silicon Revision 1.0 Texas Instruments Families of Products technical reference manual.

The PRU uses the R30 and R31 registers to interface with the peripheral interface (I/F). The transmit (TX) First in First out (FIFO) buffer is 32-bits and can operate with continuous mode while receiving a (RX) FIFO buffer size of 4 bits with maximum eight times oversampling. Both the TX and RX clock can be sourced either by ICSS_UART_CLK or ICSS_CORE_CLK. There are two independent clock dividers for TX and RX clock and each clock divider is configurable by two cascading dividers. Figure 2-9 shows the block diagram of a single-channel peripheral I/F where the other 2 channels are the same.

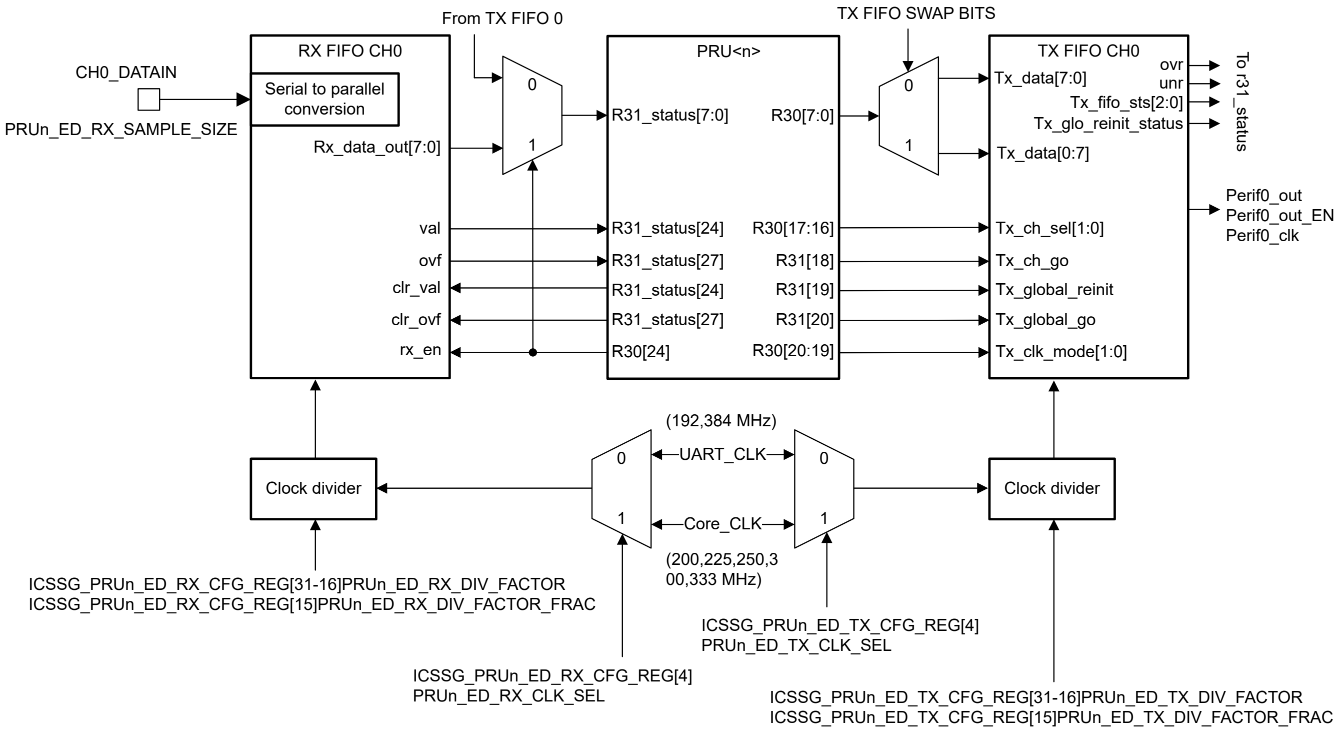


Figure 2-9. Single-channel Peripheral I/F Block Diagram

The basic programming model for Three Peripheral Mode can be found in the AM62x Processors Silicon Revision 1.0 Texas Instruments Families of Products technical reference manual.

2.5 LVDS and M-LVDS Interface

Low-voltage Differential Signaling (LVDS) devices generally comply with the American National Standards Institute (ANSI) standard TIA/EIA-644. And the Multipoint Low Voltage Differential Signaling (M-LVDS) comply with ANSI TIA/EIA-899. Table 2-1 highlights some of the important specifications for drivers and receivers between LVDS and M-LVDS.

Table 2-1. Comparison of LVDS Specification

Parameter	TIA/EIA-644-A (LVDS REV A)	TIA/EIA-899 (M-LVDS)	Unit
Driver characteristics			
Offset voltage: Vos (maximum)	1375	2100	mV
Offset voltage: Vos (minimum)	1125	300	mV
Differential output voltage: Vod (maximum)	454 (100 Ω)	650 (50 Ω)	mV
Differential output voltage: Vod (minimum)	247 (100 Ω)	480 (50 Ω)	mV
Offset voltage variation: Vospp	150	150	mV
Short circuit current: Ios	12/24	43	mA
Differential voltage change: ΔVod	50	50	mV
Offset voltage change: ΔVos	50	50	mV
Transition time: tr/tf (minimum)	260	1000	ps
Receiver characteristics			
Ground potential difference: Vgpd	±1	±1	V
Input leakage current: Iin	20	20	µA
Differential input leakage current: Iid	6	4	µA
Input voltage range: Vin	0 to 2.4	-1.4 to 3.8	V
Input threshold: Vith	100	50	mV

Intra drive communication designs require high speed, low power and electromagnetic compatibility. Hence, LVDS devices provide a wide range of solutions for intra drive communication designs from point-to-point to multidrop data transmission. The data transfer rate can range from up to 1500Mbps with a 100-meter cable length and only required 1.2 mW power consumption. Differential signals also help by having high noise immunity. The DS90x series for LVDS and SN65MLVDSx series for M-LVDS from Texas Instruments incorporates a wide variety of single and multichannel devices for designers. Moreover, TI's ISO7821LLx series provides reinforced isolation with LVDS interface when required in intra drive designs, especially for cold-hot side control applications. The intra drive communication block diagram with LVDS interface and PRU is shown as Figure 2-10.

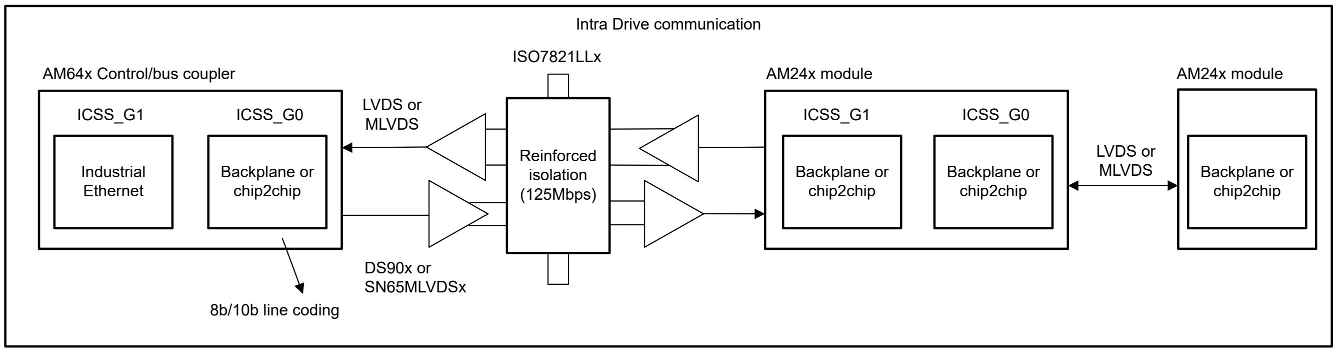
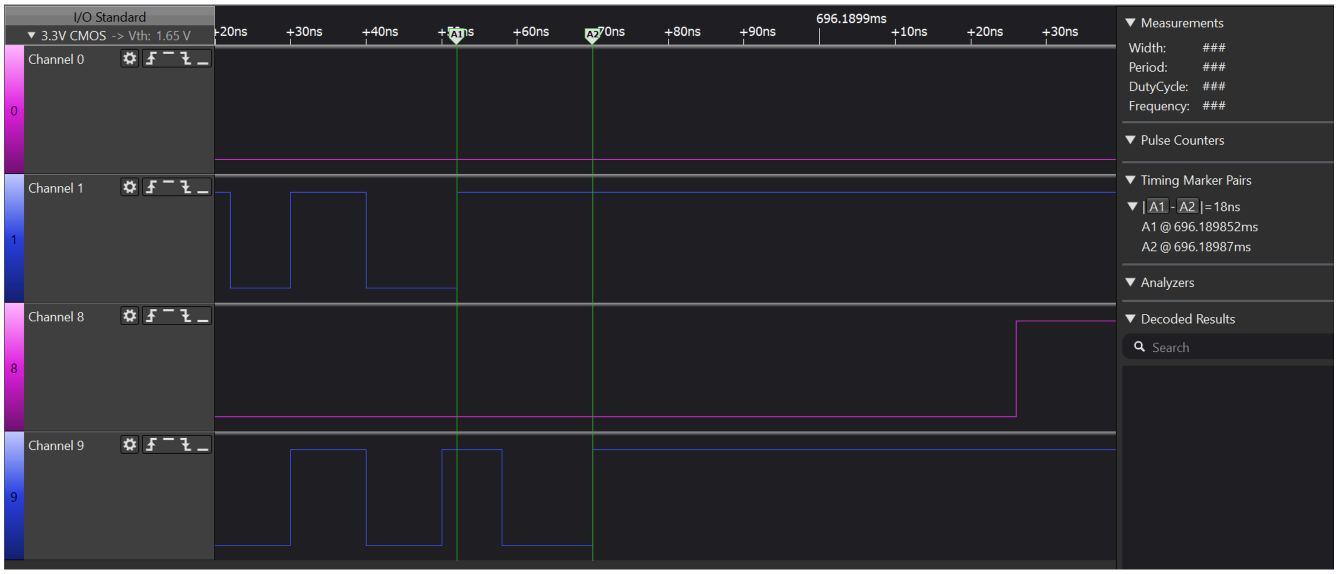


Figure 2-10. Intra Drive Communication Block Diagram With LVDS Interface and PRU

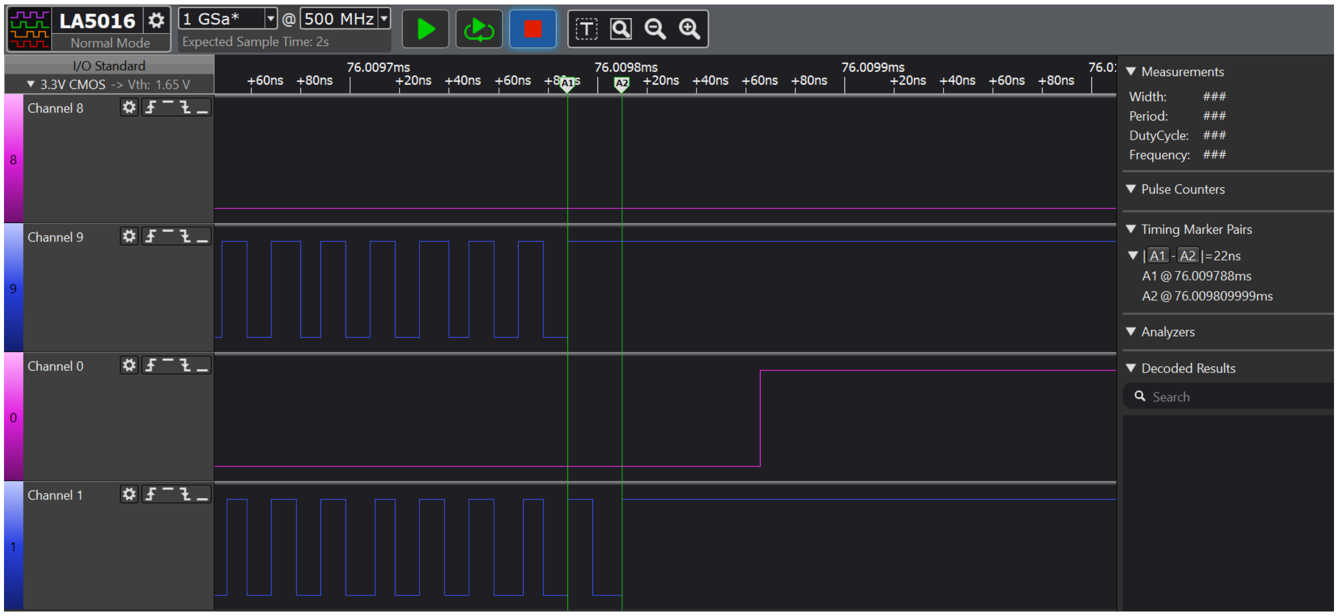
To verify the latency of LVDS transceivers and isolators, a 100 MHz clock signal is sent by PRU_GPO using the DS90LV049+ISO7821LL for LVDS interface and SM65MLVDS203+ISO7840 for M-LVDS interface. Figure 2-11 and Figure 2-12 show the latency for LVDS interface is 18ns and for M-LVDS interface is 22ns.



Width: ###
Period: ###
DutyCycle: ###
Frequency: ###

|A1 - A2|=18ns
A1 @ 696.189852ms
A2 @ 696.18987ms

Figure 2-11. 18ns Latency for DS90LV049 and ISO7821LL LVDS Interfaces



1 GSa* @ 500 MHz
Expected Sample Time: 2s

Width: ###
Period: ###
DutyCycle: ###
Frequency: ###

|A1 - A2|=22ns
A1 @ 76.009788ms
A2 @ 76.009809999ms

Figure 2-12. 22ns Latency for SN65MLLVDS203 and ISO7840 M-LVDS Interfaces

3 System Solution With CRC Module and Over-head Optimization

3.1 PRU CRC16/32 Module

Cyclic redundancy check (CRC) is required in the communication frame package to ensure data accuracy during data transmission. Each PRU core has a designated peripheral CRC16/32 module to provide error detection capability to communication systems. CRC16/32 modules support features for CRC32, CRC16, and CRC16-CCITT with different polynomials. The module also connects with the PRU internal register R25 to R29 through use of the PRU broadside interface. R29 is mapping to the CRC_DATA register, supporting a maximum of 32-bit data widths and reads back checked data either as 16-bits or 32-bits, depending on the configuration. The 16-bit CRC data with 16-bit data words can be generated and put into each frame. Figure 3-1 shows the block diagram of the PRU CRC16/32 module and the following code demonstrates the CRC16 function for 16-bit data:

```
zero &r25, 4              ; config CRC type
xout CRC_XID, &25, 4      ; enable CRC module, CRC_XID = 0x1
mov  r29, r20             ; load CRC data, ASCII equivalent for "21"
xout CRC_XID, &29.w0, 2   ; push CRC data to CRC16 module
nop
xin  CRC_XID, &r28, 4     ; load the accumulated CRC result into PRU
```

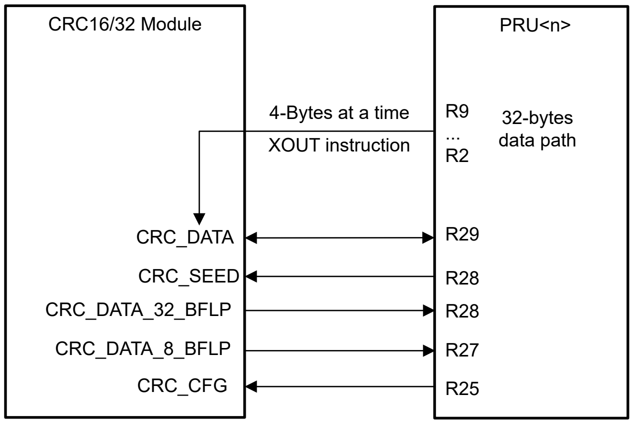
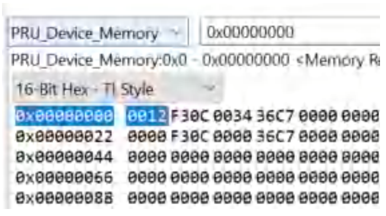


Figure 3-1. PRU CRC16/32 Module Block Diagram

The CRC input data must be swapped bytes, since the PRU is based on Least Significant Byte (LSByte)-first architecture. As an example, 0x3412 has to be given as the input data to perform CRC16 for 0x1234. Figure 3-2 shows the verification results for the CRC16/32 module. PRU0 receives data from thePRU1 GPO every 16 bits and stores in PRU DRAM from address 0x00000000.

The base address stores the raw data (0x12) sent by the PRU1 GPO. The received data is sent to the PRU0 CRC module and loads the accumulated CRC results into both the PRU0 R28 register and the memory address with an offset of 0x24. The memory address with offset 0x02 stores the CRC16 results (0xf30c) for data 0x12 from PRU1. Address offset 0x04 stores the raw data (0x34) sent by PRU1 GPO. The received data is sent to the PRU0 CRC module and loads the accumulated CRC results into both the PRU0 R28 register and memory address with offset 0x26. The memory address with offset 0x06 stores the CRC16 results (0x36c7) for data 0x34 from PRU1. Figure 3-2shows that from the memory values, both the raw data and CRC data are correct.



```
0x00000000  0012 F30C 0034 36C7 0000 0000
0x00000022  0000 F30C 0000 36C7 0000 0000
0x00000044  0000 0000 0000 0000 0000 0000
0x00000066  0000 0000 0000 0000 0000 0000
0x00000088  0000 0000 0000 0000 0000 0000
```

Figure 3-2. PRU CRC16 Module Verification

3.2 Encode and Decode Over-head Optimization

As the 8b-10b encoding method described in preceding sections describes, the 8-bit data generates 10-bit encoded data and is stored in 2 bytes of the REG_ENC register which leaves 6 bits unused in byte 2. The transmitting buffer is a 16-bit width so that the data can be combined and transmitted continuously to reduce the over-head. Figure 3-3 shows the optimized approach for transmitting encoded data:

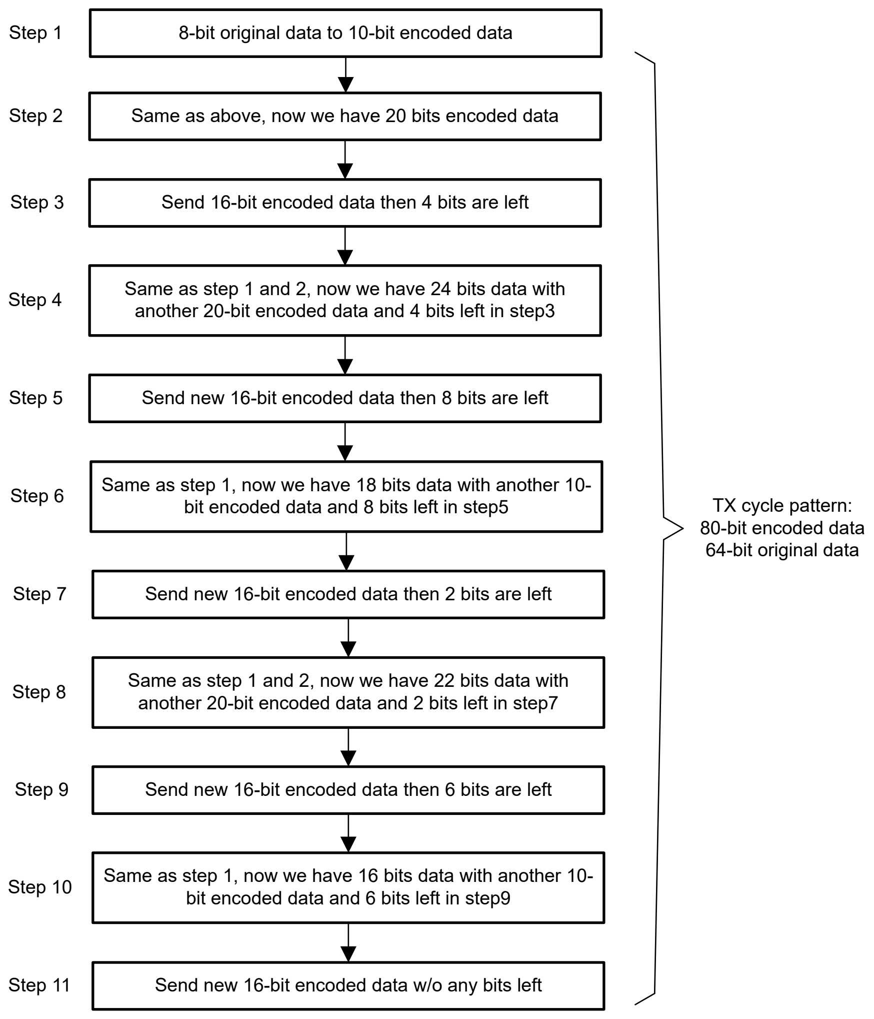

Step 1
8-bit original data to 10-bit encoded data
Step 2
Same as above, now we have 20 bits encoded data
Step 3
Send 16-bit encoded data then 4 bits are left
Step 4
Same as step 1 and 2, now we have 24 bits data with another 20-bit encoded data and 4 bits left in step3
Step 5
Send new 16-bit encoded data then 8 bits are left
Step 6
Same as step 1, now we have 18 bits data with another 10-bit encoded data and 8 bits left in step5
Step 7
Send new 16-bit encoded data then 2 bits are left
Step 8
Same as step 1 and 2, now we have 22 bits data with another 20-bit encoded data and 2 bits left in step7
Step 9
Send new 16-bit encoded data then 6 bits are left
Step 10
Same as step 1, now we have 16 bits data with another 10-bit encoded data and 6 bits left in step9
Step 11
Send new 16-bit encoded data w/o any bits left
TX cycle pattern:
80-bit encoded data
64-bit original data

Figure 3-3. Optimized Approach for Transmitting Encoded Data

One transmitting cycle pattern is 80-bits encoded data width (64-bits original data width) with 8 times encoding process.

Similar to the approach for transmitting encoded data, Figure 3-4 shows the optimized process for receiving decoded data:

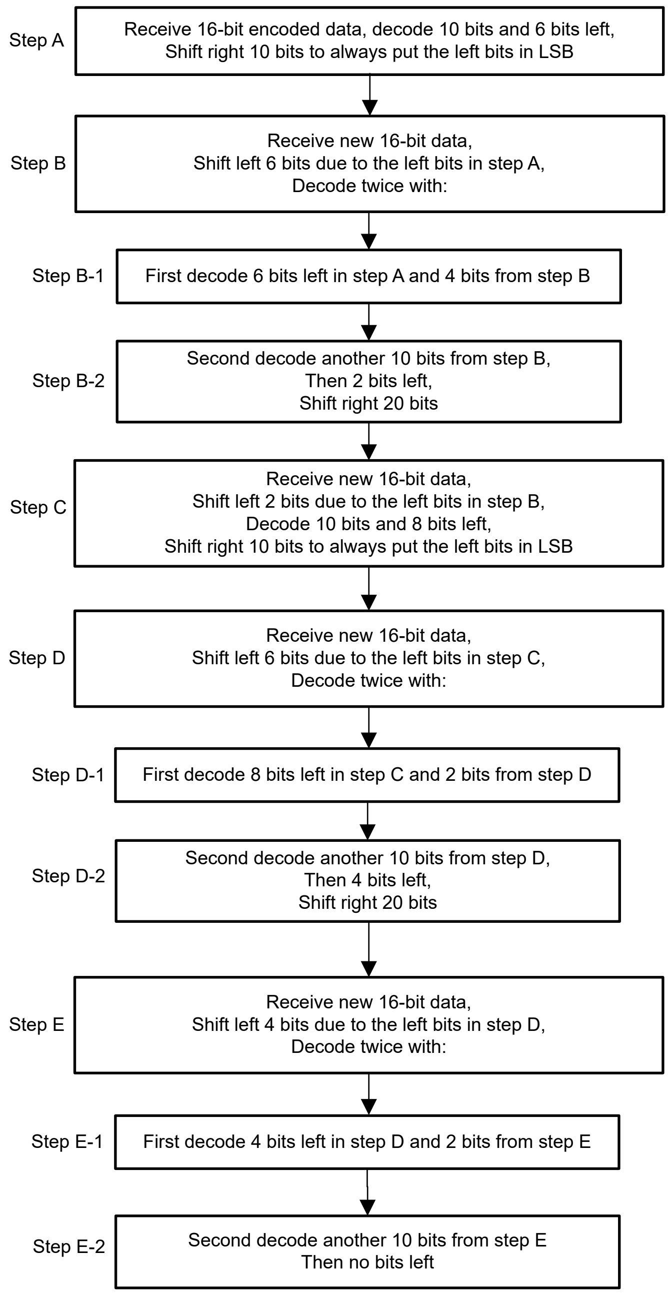

Step A
Receive 16-bit encoded data, decode 10 bits and 6 bits left, Shift right 10 bits to always put the left bits in LSB
Step B
Receive new 16-bit data, Shift left 6 bits due to the left bits in step A, Decode twice with:
Step B-1
First decode 6 bits left in step A and 4 bits from step B
Step B-2
Second decode another 10 bits from step B, Then 2 bits left, Shift right 20 bits
Step C
Receive new 16-bit data, Shift left 2 bits due to the left bits in step B, Decode 10 bits and 8 bits left, Shift right 10 bits to always put the left bits in LSB
Step D
Receive new 16-bit data, Shift left 6 bits due to the left bits in step C, Decode twice with:
Step D-1
First decode 8 bits left in step C and 2 bits from step D
Step D-2
Second decode another 10 bits from step D, Then 4 bits left, Shift right 20 bits
Step E
Receive new 16-bit data, Shift left 4 bits due to the left bits in step D, Decode twice with:
Step E-1
First decode 4 bits left in step D and 2 bits from step E
Step E-2
Second decode another 10 bits from step E Then no bits left

Figure 3-4. Optimized Approach for Receiving Decoded Data

One receiving cycle pattern is 80-bits decoded data width with 8 times decoding process.

The original frame pattern can be a 64-bit width with 16-bit preamble, 32-bit data words, and 16-bit CRC data. To increase the efficiency, RTU_PRU0 and RTU_PRU1 auxiliary cores can be used for encoding, decoding, and CRC16 workloading in parallel. Figure 3-5 shows a block diagram with both PRU and RTU cores.

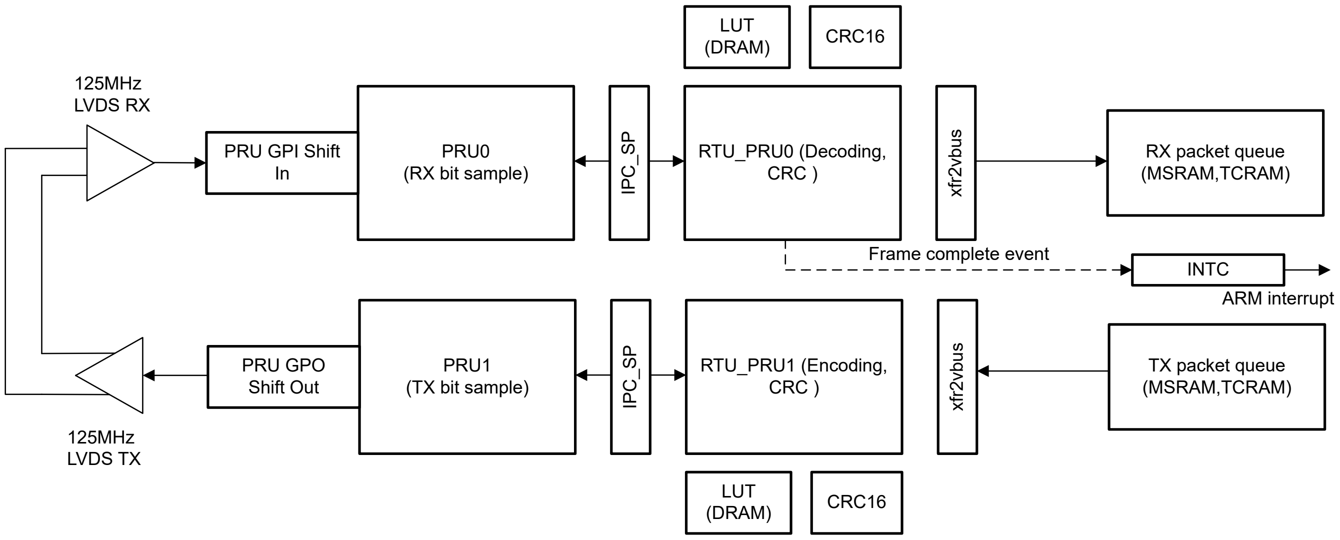


Figure 3-5. System Block Diagram With PRU and RTU Cores

4 Verification

Section 2.3 and Section 2.4 show how the data transmitting accuracy and LVDS interface latency is verified. This section shows how to test the jitter on the PRU GPIO side and the processing time for both transmitting and receiving. Figure 4-1 shows the transmitting data rate with 125 MHz. Figure 4-2 and Figure 4-3 shows that the jitter on the GPO side during rising and falling edge is only 60ps.

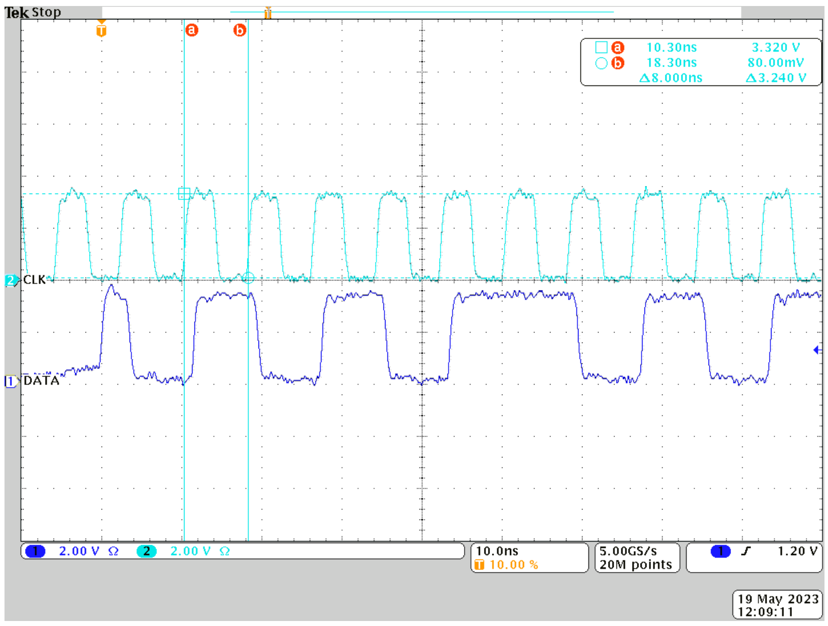


Figure 4-1. Transmitting Data Rates

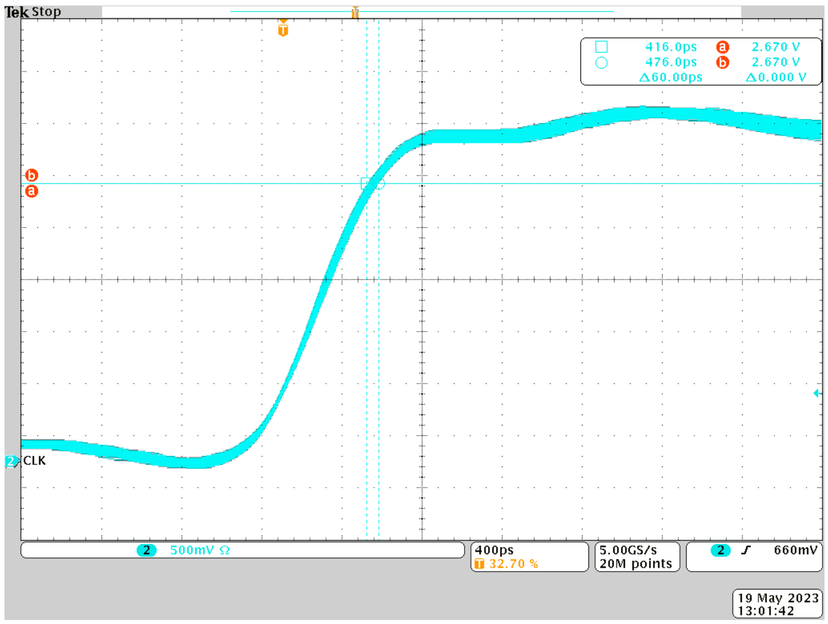


Figure 4-2. PRU GPO Jitter on Rising Edge

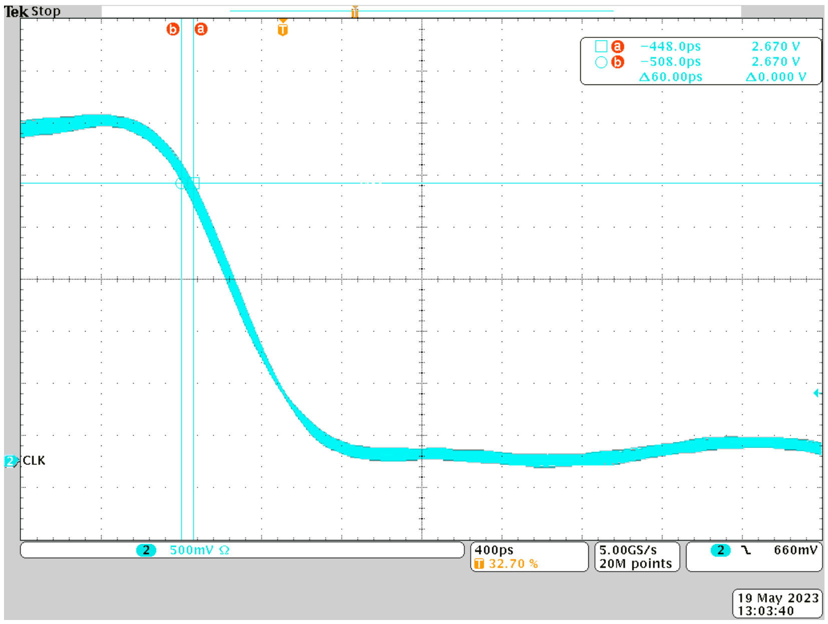


Figure 4-3. PRU GPO Jitter on Falling Edge

The processing time of transmitting data includes encode data, data CRC16 check, and IPC scratch pad memory sharing (as an option). The processing time takes 48ns for encoding 8-bit data and 20ns for a 32-bit CRC16 check under a maximum of 250 MHz PRU core clock. A 64-bit width transmitting cycle pattern requires 404ns with 384ns encoding time and 20ns CRC time. The processing time for receiving includes decode data, data CRC16 check, and IPC scratch pad memory sharing (as an option). Processing takes 60ns for decoding 10-bits of data and 20ns for a 32-bit CRC16 check under a maximum of 250 MHz PRU core clock. A 64-bit width receiving cycle pattern requires 500ns with 480ns encoding time and 20ns CRC time. Figure 4-4 through Figure 4-6 shows the test results of processing time using PRU_GPO toggling. Since the PRU_GPIO can only be configured for one mode at a time, the test condition is separated from the communication period and just tests the time for relative code instruction.

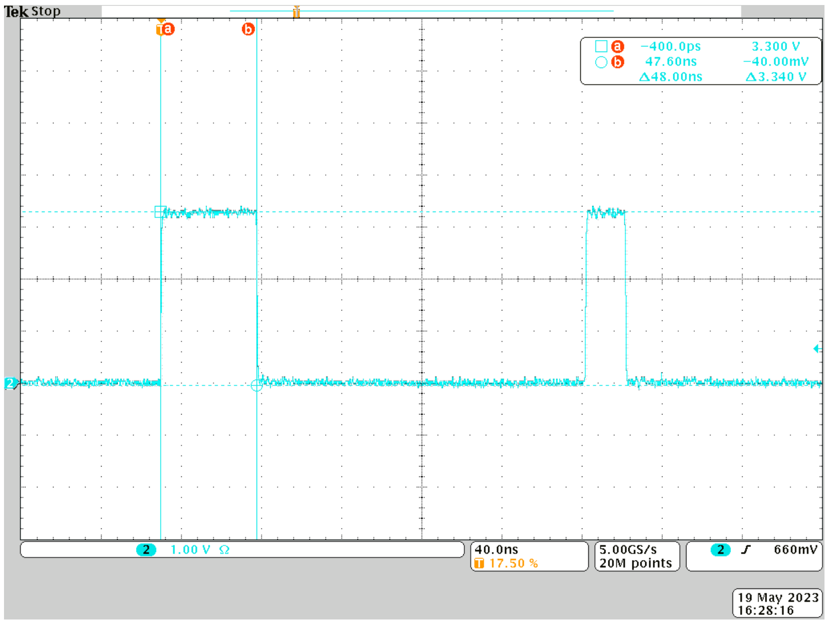


Figure 4-4. Processing Time for 8-bit Data Encoding

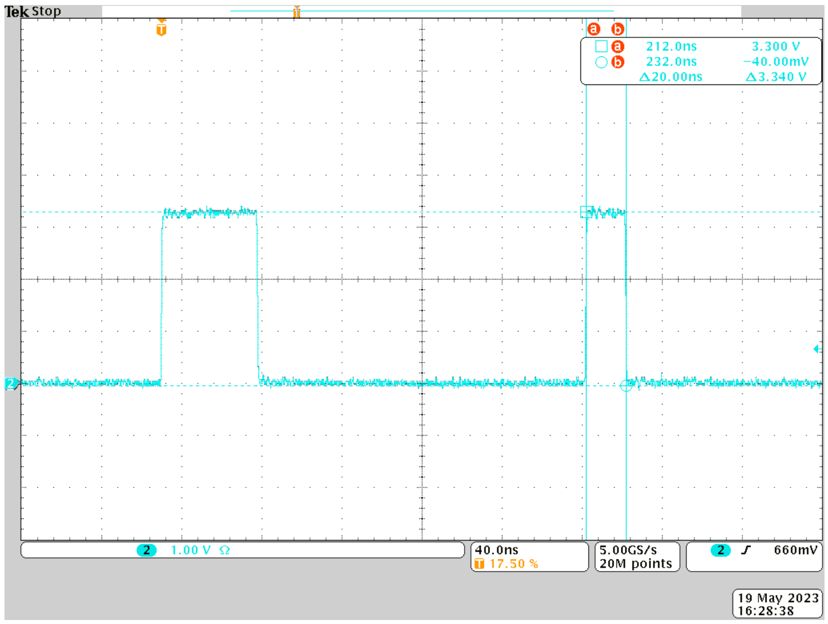


Figure 4-5. Processing Time for CRC16

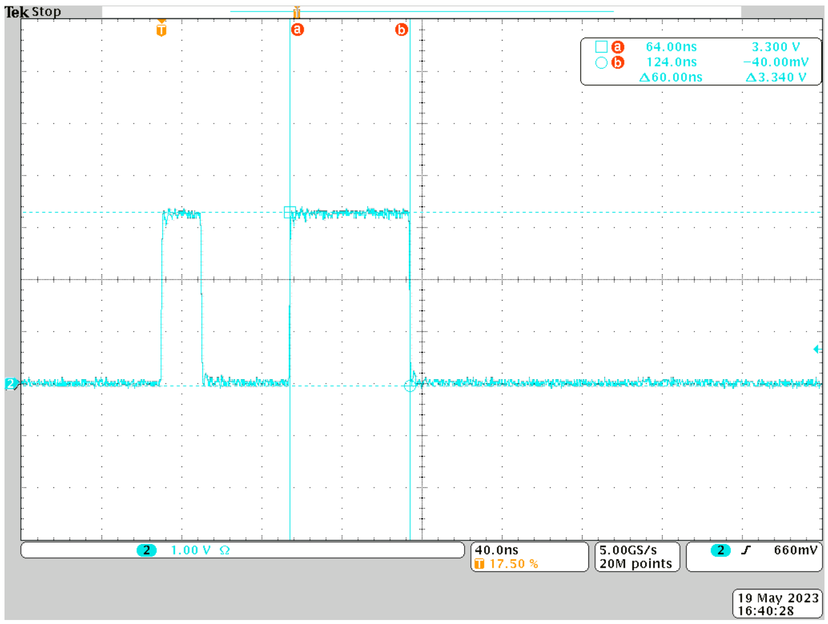


Figure 4-6. Processing Time for 10-bit Data Decoding

With the support from RTU_PRU0 and RTU_PRU1 auxiliary cores, data transfer and processing can be implemented in parallel. The IPC module can be utilized to exchange data between PRU and RTU with XFR instructions. The scratch pad (register R2:R9) is 32 bytes wide, connecting PRU and RTU_PRU cores together within a slice. XFR instructions define the start, size, direction of the operation, and device ID. Figure 4-7 shows the integration of the PRU and IPC Scratch Pad.

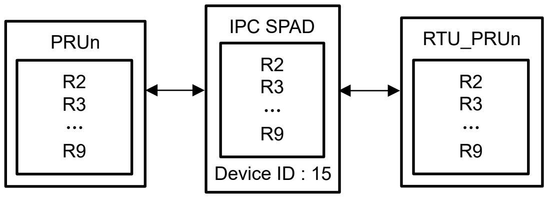


Figure 4-7. Integration of the PRU and IPC Scratch Pad

The XFR2VBUS hardware accelerator can support data movement to and from the Magneto-resistive random-access memory (MRAM) or Tightly Coupled random-access Memory (TCRAM). Both XFR2VBUS TX and RX buffers are 64-bytes deep and can transfer 64 bytes of data with a single register transfer in (XIN) or register transfer out (XOUT) instruction. The following code example shows the data movement to and from TCRAM using the XFR2VBUS widget.

```
; Read wait
wait_till_read_busy_0?:
  xin XFR2VBUSP_RD0_XID, &r18, 1
  qbbs wait_till_read_busy_0?, r18, 3    ; R18.3 :RD_MST_REQ Wait until Last Data has been latched

; TCM to XFR2VBUS RX buffer
  ldi   r18, 6                           ; 64 bytes
  ldi32 r19, CSL_R5FSS0_CORE0_ATCM_BASE  ; TCM Address
  ldi   r20, 0
  xout  XFR2VBUSP_RD0_XID, &r18, 10      ; transfer address
  xin   XFR2VBUSP_RD0_XID, &r2, 65       ; transfer data

; Write wait
wait_till_write_done_0?:
  xin   XFR2VBUSP_WR0_XID, &r20, 1
  qbbs  wait_till_write_done_0?, r20, 0  ; R20.0 : WR_BUSY Wait until Idle

; XFR2VBUS TX buffer to TCM
  ldi32 r18, CSL_R5FSS0_CORE0_ATCM_BASE  ; TCM Address
  ldi   r19, 0
  xout  XFR2VBUSP_RD0_XID, &r18, 10      ; transfer address
  xin   XFR2VBUSP_RD0_XID, &r2, 65       ; transfer data
```

PRUn can start transmitting data after the first 16-bit raw data is encoded by RTU_PRUn. Similarly, RTU_PRUn in the receiving path can start decoding data after the first 20-bit encoded data is received by PRUn. The transmitting processing time of 64-bit width cycle patterns can be reduced from 404ns to 96ns (16-bit data encoding) and the receiving processing time of 64-bit width cycle patterns can be reduced from 500ns to 120ns (20-bit data decoding). Table 4-1 summarizes the processing times with and without the support of the RTU_PRUn cores.

Table 4-1. Processing Time With and Without RTU_PRUn Cores

PRU cores	Sampling clock (MHz)	Data length (bit)	TX module data processing time (ns)	Data transfer time (ns)	RX module data processing time (ns)	Total time (ns)
PRU	125	64	404	640(1)	500	1544
PRU + RTU	125	64	96	640(1)	120	856

(1) 80-bit encoded data width for data transfer

5 Summary

This application note provides an approach for using PRU to implement 8b-10b line coding with 100Mbps data rate and transmitting data with an LVDS interface. The line coding in PRU with an LVDS interface is flexible for defining custom protocol for intra-drive communication to achieve high-speed data rates, low jitter, and low latency.

6 References

1. Texas Instruments, *AM243x Sitara™ Microcontrollers*, data sheet
2. Texas Instruments, *AM64x / AM243x Processors Silicon*, technical reference manual
3. Texas Instruments, *PRU Assembly Instruction User Guide*, user guide
4. Texas Instruments, *Introduction to M-LVDS (TIA/EIA-899)*, application note
5. IBM Research, *8B/10B Encoding and Decoding for High Speed Applications*, technical paper