AN-1435
APPLICATION NOTE

One Technology Way • P.O. Box 9106 • Norwood, MA 02062-9106, U.S.A. • Tel: 781.329.4700 • Fax: 781.461.3113 • www.analog.com

Implementing UART Using the ADuCM3027/ADuCM3029 Serial Ports

by Sachin Dwivedi

INTRODUCTION

Using the synchronous serial peripheral ports (SPORTs) on the ADuCM3027/ADuCM3029 processor, it is possible to implement a full duplex, asynchronous port to communicate with the universal asynchronous receiver/transmitters (UARTs) with minimal software overhead. This application note describes how to implement a full UART interface for multiple standard baud rates.

SPORT OVERVIEW

The SPORT interface supports a variety of serial data communication protocols. In addition, the SPORTs provide a glueless hardware interface to many industry-standard data converters, codecs, and other processors, including digital signal processors (DSPs).

Key features and capabilities of the SPORT interface include

- A continuously running clock.
- Serial data words from 3 bits to 32 bits in lengths, either MSB or LSB first.
- Two synchronous transmit and two synchronous receive data signals.
- Buffers to double the total supported data stream.
- Configurable frame synchronization signals.

Refer to https://wiki.analog.com/resources/eval/sdp/sdp-b/peripherals/sport for further information about the SPORT interface.

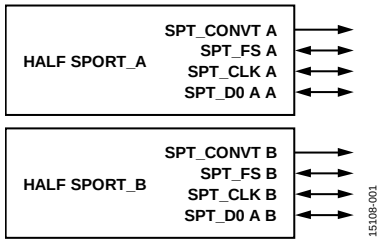

Figure 1. SPORT Signals

TABLE OF CONTENTS

REVISION HISTORY

4/2017—Revision 0: Initial Version

ASYNCHRONOUS COMMUNICATION

The difference between synchronous and asynchronous serial communication is the presence of a clock signal and frame synchronization signal. A synchronous serial port has a clock signal and an optional frame sync signal. An asynchronous port does not have clock and frame sync signals. In the absence of a clock signal, the asynchronous ports must communicate at a predetermined data rate (bit rate). In the absence of a frame sync, the word framing information is embedded in the data stream. A start bit marks the beginning of a transmission. A stop bit marks the completion of a transmission. The word length is predetermined between the receiver and transmitter.

ASYNCHRONOUS SPORT TRANSMITTER

The transmit side of the serial port must be configured for internal clock generation with a clock rate equal to the desired bit rate of the UART. This configuration is performed by setting the CLKDIV bit in the clock divider register (SPORT_DIV_A) for the SPORT_A block.

$$CLKDIV\ bit\ in\ the\ SPORT\_DIV\_A\ register = \frac{PCLK}{2 \times Baud\ Rate} - 1$$

where *PCLK* is the peripheral clock signal.

The SPORT_A clock is only used to synchronize the serial port to the desired bit rate. The actual clock signal (SPORT_ACLK) does not connect to anything. Configure the frame sync signal (SPORT_AFS) to be internally generated and leave the signal floating. The SPORT_A block must always transmit LSB first to emulate UART transmission. Program the number of bits to be transmitted by the SPORT_A block in the SLEN field of the SPORT_CTL_A register. Program the total number of words to be transferred in the SPORT_NUMTRAN_A register, with each word size determined by the SLEN field.

In the case of a SPORT transmission where the SPORT_A block transmits to a UART device, the UART always receives the first transfer as 0x00, which can be discarded, followed by a correct sequence of data transmitted by the SPORT_A block. This sequence occurs because, at the start of transfer (after configuration), the UART Rx line is idle high (Logic 1) and the SPORT data line is idle low (Logic 0). The UART interprets this Logic 0 as a start bit and receives an entire frame of Logic 0 at the beginning of the transmission.

ASYNCHRONOUS SPORT RECEIVER

The serial port must determine where a new transmission begins without the presence of an internal frame synchronization signal. The transmit pin of the UART device connects to the data line pin (SPORT_BD0) and the frame synchronization pin (SPORT_BFS) on the SPORT_B block of the ADuCM3029/ADuCM3027. The SPORT_B block is configured for internal clock generation and an active low external frame sync signal.

Because the SPORT cannot guarantee any phase synchronization with the incoming bit stream, it is necessary to oversample the incoming asynchronous data stream. The receive clock on the SPORT must be set to three times the desired baud rate. For example, if the ADuCM3029/ADuCM3027 SPORT communicates with the UART device at 9600 bps, the receive clock on the SPORT must be set to 28,800 bps. Perform this setting by calculating the appropriate divisor and programming the CLKDIV bit in the clock divider register (SPORT_DIV_B) for the SPORT_B block.

$$CLKDIV\ Bit\ in\ the\ SPORT\_DIV\_B\ Register = \frac{PCLK}{2 \times 3 \times Baud\ Rate} - 1$$

The active low frame synchronization signal (SPORT_BFS) is polled on the active edge of the internally generated clock (SPORT_BCLK). When the SPORT_BFS signal is asserted due to the low level start bits of the UART packet, the SPORT_B block starts receiving the word transmitted from the UART device and does not check the SPORT_BFS line until all N bits of the packet are received (N is programmed by the SLEN field in the SPORT_CTL_B register). The SPORT uses the oversampled start bit as a frame synchronization to begin the reception of the incoming asynchronous data stream.

HARDWARE AND SOFTWARE OVERVIEW

HARDWARE OVERVIEW

Figure 2 shows the connection between the ADuCM3029/ ADuCM3027 SPORTs and the transmit (Tx) and receive (Rx) pins of a basic UART port on another device.

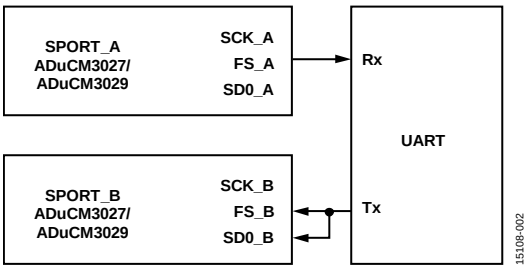

Figure 2. ADuCM3029/ADuCM3027 Microcontroller Unit (MCU) to UART Interface

SOFTWARE OVERVIEW

The software required to manage the asynchronous data moving in and out of the SPORT is minimal. The C functions for the SPORT transmission and reception are described in the Code for the SPORT_UART_Emulator section. The code is tested for multiple baud rates and multiple number of transfers between the UART host and the ADuCM3029/ADuCM3027 SPORTs for both transfer directions.

Asynchronous SPORT Transmitter (SPORT_A Block)

On the transmit side, the N bit data to be transmitted must be formatted into a UART transmission packet. A start bit and stop bit must be added to the word for correct reception by the UART device.

An example of the data format is as follows:

- For an 8-N-1 transmission format (8-bit data + 0 parity bit + 1 stop bit), data = 0xAA (b#1010 1010).
- Modified data = b# 1 10101010 0 1 (1 stop-bit + 8-bit data + 1 start bit + 1 stop bit).

A stop bit must be appended at the beginning of the transmission because the SPORT_AD0 line retains the value of the LSB (if the LSB is transmitted first) when a complete word is sent. The UART Rx line must be set to idle high to avoid glitches in the generated signal between consecutive bytes, leading to corrupt data.

Asynchronous SPORT Receiver (SPORT_B Block)

The receive side is more complex than the transmit side because the SPORT_B block receives oversampled data. For an 8-N-1 transfer format, because the data is oversampled by a factor of 3, the serial port must be programmed to receive 27 bits, thereby discarding the three sampled start bits, which are accounted for in the frame synchronization (SPORT_BFS). The 27 bits received represent the packet transmitted by the UART device, 8 bits of data, and 1 stop bit (oversampled by a factor of 3).

Then, the actual data is extracted from the oversampled data by bit manipulation operations. The middle bit, which is the correct value, is extracted from the 3-bit sequence in the received data for every transmitted bit from the UART device. The extracted bits are assembled to form a byte of data.

DATA FORMAT: UART	START BIT	DATA BYTE = 8 BITS								STOP BIT
		LSB	1	2	3	4	5	6	MSB	
EQUIVALENT BIT PATTERN FOR SPORT0	000	xxx	yyy	xxx	yyy	xxx	yyy	xxx	yyy	111
	3 ZEROES	BYTE REPRESENTED BY 24 BITS								3 ONES

15108-003

Figure 3. Expected Data Formats for UART Frame and SPORT Receive Frame

DRIVER FUNCTION PROTOTYPES

The following functions work with 8-bit asynchronous data, but can easily be changed to support other data widths. The C functions for the use case are detailed in the Code for the SPORT_UART_Emulator section.

The SPORT_UART_Tx_Initialise function configures and sets up the SPORT_A block on the ADuCM3029/ADuCM3027 processor for UART transmission emulation. The SPORT_A internal clock is derived from the PCLK, which is configured to 6.5 MHz (default). The desired baud rate is set for the transmission, along with configuration using the SPORT_CTL_A register. An interrupt to signal that the transmit data buffer is empty is configured using the SPORT_IEN_A register. The number of words to be transferred is programmed using the SPORT_NUMTRAN_A register, before enabling the SPORT_A block.

The SPORT_UART_Rx_Initialise function configures and sets up the SPORT_B block on the ADuCM3029/ADuCM3027 processor for UART reception emulation. The SPORT_B block is configured to oversample the incoming data stream by a factor of 3. The frame synchronization is configured for an external, low, active state. An interrupt to signal that the receive data buffer is full is configured using the SPORT_IEN_B register. Also, the SLEN field of the SPORT_CTL_B register is configured to the following before enabling the SPORT_B block:

$$3 \times (Word\ Size + Number\ of\ Stop\ Bits) - 1$$

The SPORT_UART_Tx_Transfer function creates the UART transmission data format by modifying the data in location pointed to by the buffer. The modified data is then output on the SPORT_A_TX register for transmission. The function uses bit masking and shifting operations.

The SPORT_UART_Rx_Transfer function receives the over-sampled data from the SPORT_B_RX register. A bit manipulation operation extracts the middle bits of every 3-bit sequence of the SPORT_B_RX data (3 bits received for every 1 bit transmitted by the UART device). The extracted bits are assembled into a byte sized data. The function returns the assembled byte, representing the actual received data. The following code example shows how to extract the data from the 27-bit SPORT registers into 8-bit UART data:

```
/* Receive data into the RX buffer */
temp = *pREG_SPORT0_RX_B;
  /* Extract the 8 bits from the 27 bits
  received */
  value = 0;
  value += ((temp >> 23) & (1 << 0));
  value += ((temp >> 19) & (1 << 1));
  value += ((temp >> 15) & (1 << 2));
  value += ((temp >> 11) & (1 << 3));
  value += ((temp >> 7) & (1 << 4));
  value += ((temp >> 3) & (1 << 5));
  value += ((temp >> 1) & (1 << 6));
  value += ((temp >> 5) & (1 << 7));
  /* Return the assembled byte */
  return value;
```

SOFTWARE FLOWCHARTS

SPORT_A BLOCK TRANSMISSION

The SPORT_A block emulates the UART Tx port. To use the SPORT_A block to emulate the UART Tx port, the SPORT block must be initialized for transmission and must be enabled. When enabled, check the SPORT_STAT register if there is pending data to be transmitted. If there is data to be transmitted, create the UART data packet from the pending data, then write the data packet to the SPORT transmit register.

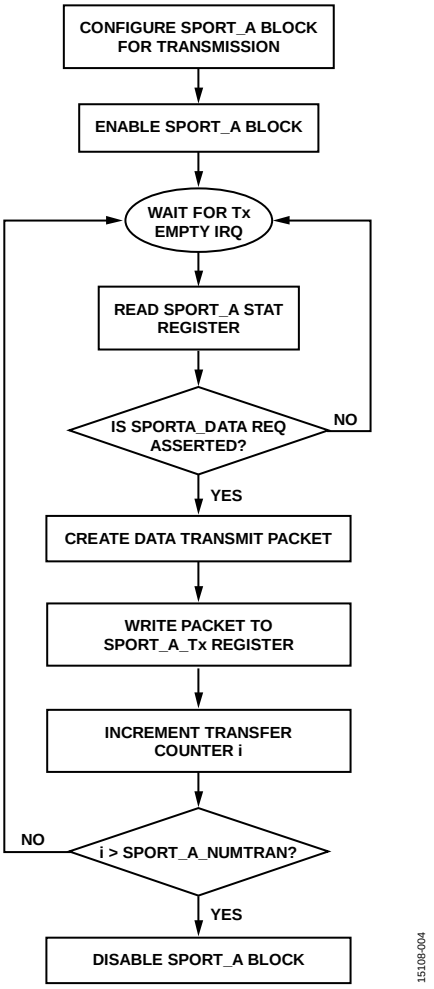

Figure 4. SPORT_A Block Transmission Flowchart

SPORT_B BLOCK RECEPTION

The SPORT_B block emulates the UART Rx port. To use the SPORT_B as the UART Rx port, the SPORT_B block must first be initialized for reception and must be enabled. When the SPORT is enabled, check the SPORT_STAT register for pending data in the SPORT_B data register. If there is a data, retrieve the data and extract the 8-bit UART data.

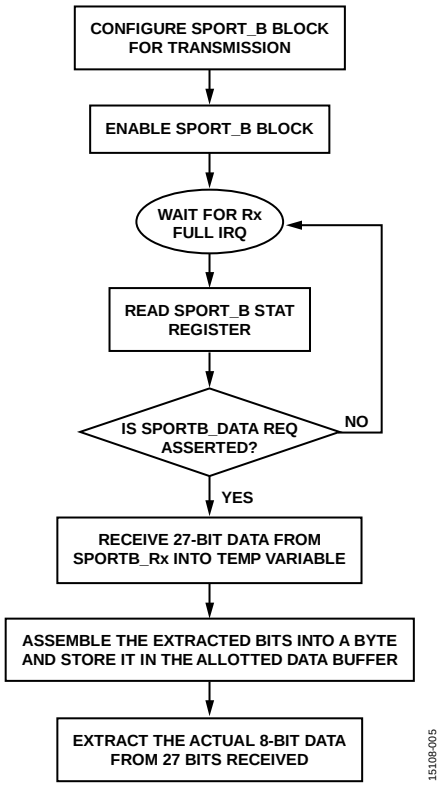

Figure 5. SPORT_B Block Reception Flowchart

WAVEFORMS

Figure 6 shows the waveforms for transmission from the SPORT_A block and reception on the UART device at 9600 bps, for a single frame of 8-bit data (0x96), and additional formatting bits required for UART transmission emulation.

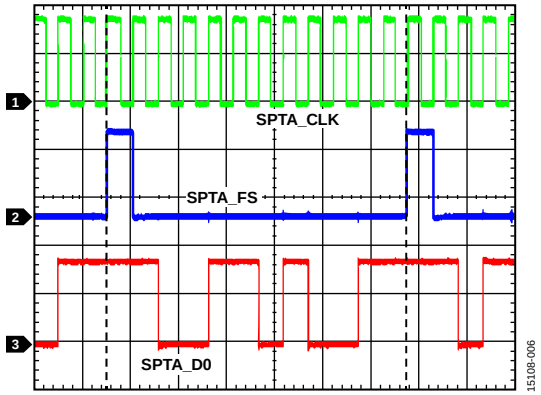

Figure 6. SPORT_A Block Transmission and UART Device Reception for a Single Frame

Figure 7 shows the waveforms for transmission from the UART device and reception on the SPORT_B block at 9600 bps, for a single frame of 8-bit data (0x96) (with a start bit and a stop bit), sampled at 3 times the transmission baud rate for proper emulation of UART reception.

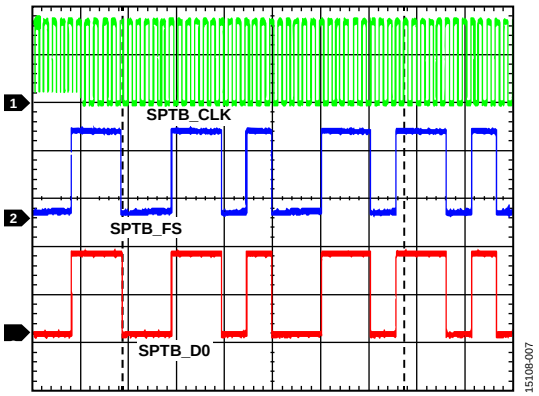

Figure 7. UART Device Transmission and SPORT_B Block Reception for a Single Frame

The red traces in the figures indicate a single frame of transfer.

CODE FOR THE SPORT_UART_EMULATOR

The code shown in this section provides an example for the following use cases:

- Transmission from the SPORT_A block and reception by the UART.
- Transmission from the UART and reception by the SPORT_B block.

The data format used for a single frame of transfer is 8-N-1 (1 start bit, 8 bits of data, 0 parity bits, and 1 stop bit). These cases are tested at PCLK = 26 MHz and multiple baud rates.

SPORT_UART_EMULATOR.H

```
/* SPORT Based UART Emulator Application */
/* SPORT_A emulates Transmission Side while SPORT_B emulates Reception Side */
/* Two Use Cases
       (a) Transmission from SPORT_A and Reception by UART
(b) Transmission from UART and Reception by SPORT_B */
/* Tested with PCLK = 26 MHz and Baud Rates - 9600bps, 19200bps, 38400bps, 57600bps,  115200bps, 230400bps */
/* Define the word_size and baud_rate for UART before proceeding */

#include "system.h"
#include "startup.h"
#include "stdint.h"

/* Definitions used for supporting both use cases */

#define SLEN_TX     (word_size + stopbits + paritybit + 1)
#define SLEN_RX     (3 * (word_size + stopbits + paritybit) - 1)
#define FSDIV_TX    (word_size + stopbits + paritybit + 2)
#define SYS_PCLK    26000000
#define TRAN_SIZE 3
#define baud_rate 9600
#define word_size 8
#define stopbits    1

/* Global Variables used for both use cases */

uint32_t temp;
uint8_t flag = 0;
uint8_t tbuf[TRAN_SIZE];
uint8_t rbuf[TRAN_SIZE];

int i=0; /* Transfer Loop Counter */
uint16_t res;

/* Definitions for Functions used for both use cases */
void Change_CLKDIV(int pVal, int hVal);
void SPORT_UART_Tx_Initialise();
void SPORT_UART_Rx_Initialise();
```

```
void SPORT_UART_Tx_Transfer(uint8_t *buf);
uint8_t SPORT_UART_Rx_Transfer();

/* Description: Function to change the PCLKDIV and HCLKDIV
   Input Parameters:       int pVal - Value of PCLK Divisor
                    int hVal - Value of HCLK Divisor
   Return:          void
*/
void Change_CLKDIV(int pVal, int hVal)
{
  uint32_t uiTemp;
  // Change PCLKDIVCNT
  uiTemp = *pREG_CLKG0_CLK_CTL1;
  uiTemp &= ~(BITM_CLKG_CLK_CTL1_PCLKDIVCNT);
  uiTemp |= (pVal << BITP_CLKG_CLK_CTL1_PCLKDIVCNT);
  *pREG_CLKG0_CLK_CTL1 = uiTemp;

  // Change HCLKDIVCNT
  uiTemp = *pREG_CLKG0_CLK_CTL1;
  uiTemp &= ~(BITM_CLKG_CLK_CTL1_HCLKDIVCNT);
  uiTemp |= (hVal << BITP_CLKG_CLK_CTL1_HCLKDIVCNT);
  *pREG_CLKG0_CLK_CTL1 = uiTemp;
}

/* Description: Function to initialize and configure the SPORT_A for UART Transmission Emulation
   Input Parameters:       void
   Return:          void
*/
void SPORT_UART_Tx_Initialise()
{
  float value;
  value = ((SYS_PCLK / (2 * baud_rate)) - 1);

  /* Configure the GPIO pins as alternate functions for SPORT_A_Tx */
  *pREG_GPIO2_CFG |= (1 << BITP_GPIO_CFG_PIN00) | (1 << BITP_GPIO_CFG_PIN02);
  *pREG_GPIO1_CFG |= (1<< BITP_GPIO_CFG_PIN15);
  *pREG_GPIO0_CFG |= (1<< BITP_GPIO_CFG_PIN12);
  *pREG_GPIO0_PE |= (1 << 12);

  /* Disable the SPORT_A_Tx before the configuration*/
  *pREG_SPORT0_CTL_A &= ~(1 << BITP_SPORT_CTL_A_SPEN);

  /* Configure CLk Divider */
  *pREG_SPORT0_DIV_A |=    ((uint16_t) value << BITP_SPORT_DIV_A_CLKDIV) |
  ((FSDIV_TX) << BITP_SPORT_DIV_A_FSDIV);

  /* Configure the Data interrupts and the Transfer Complete interrupts */
```

```
  *pREG_SPORT0_IEN_A |= (1<< BITP_SPORT_IEN_A_TF) | (1<< BITP_SPORT_IEN_A_DATA);

  /* Program Number of Transfers */
  *pREG_SPORT0_NUMTRAN_A = TRAN_SIZE;

  /* Write the CTL register */
  *pREG_SPORT0_CTL_A | = ((SLEN_TX) << BITP_SPORT_CTL_A_SLEN)
    | (1 << BITP_SPORT_CTL_A_ICLK)
    | (1 << BITP_SPORT_CTL_A_IFS)
    | (1<< BITP_SPORT_CTL_A_FSR)
    | (1 << BITP_SPORT_CTL_A_SPTRAN)
    | (1 << BITP_SPORT_CTL_A_LSBF);

  /* Enable SPORT_A */
  *pREG_SPORT0_CTL_A |= (1<< BITP_SPORT_CTL_A_SPEN);
}

/* Description: Function to initialize and configure the SPORT_B for UART Reception Emulation
Input Parmaeters:   void
Return:                   void
*/
void SPORT_UART_Rx_Initialise()
{
  float value;
  value = ((SYS_PCLK /(2 * 3 * baud_rate)) - 1);

  /* Configure the GPIO pins as alternate functions for SPORT_B_Rx */
  *pREG_GPIO0_CFG |= (2 << BITP_GPIO_CFG_PIN00)
    | (2 << BITP_GPIO_CFG_PIN01)
    | (2 << BITP_GPIO_CFG_PIN02)
    | (2 << BITP_GPIO_CFG_PIN03);

  /* Configure Clk Divider */
  *pREG_SPORT0_DIV_B |= ((uint16_t) value << BITP_SPORT_DIV_B_CLKDIV);

  /* Use external FS */
  /* Configure Data interrupts and Transfer Complete Interrupt */
  *pREG_SPORT0_IEN_B = (1<< BITP_SPORT_IEN_B_TF) | (1<< BITP_SPORT_IEN_B_DATA);

  /* Program Number of Transfers */
  *pREG_SPORT0_NUMTRAN_B = 2;

  /* Write to CTL register */
  *pREG_SPORT0_CTL_B | = ((SLEN_RX) << BITP_SPORT_CTL_B_SLEN)
    | (1 << BITP_SPORT_CTL_B_ICLK)
    | (1 << BITP_SPORT_CTL_B_FSR)
    | (1 << BITP_SPORT_CTL_B_LFS);
```

```
  /* Enable SPORT_B to receive data */
  *pREG_SPORT0_CTL_B |= (1<< BITP_SPORT_CTL_B_SPEN);
}

/* Decription: Function to transmit data from SPORT_A_TX register to UART Device after
formatting
   Input Parameters:       uint8_t *buf - Value of the data to be transmitted
   Return :         void
*/
void SPORT_UART_Tx_Transfer(uint8_t *buf)
{
  uint16_t res;

  /* Place a start and a stop bit */
  uint16_t tx_mask, tx_startbits, tx_stopbits;

  /* Create Masks for transmitting the word
  Example: if word_size = 8
  tx_mask = b'11111111
  tx_startbits = b'01111111100
  tx_stopbits = b'10000000001
  */

  tx_mask = (1 << word_size) - 1;
  tx_startbits = tx_mask << 2;
  tx_stopbits = ((0x0C) << (word_size + paritybit)) | 1;

  /* Remove all the bits that won't be transmitted */
  (*buf) &= tx_mask;
  res = (*buf) << 2;       /* Make space for the start bit and previous stop bit */
  res &= tx_startbits;     /* Add the start bit */
  res |= tx_stopbits;      /* Add the stop bits */

  /* Put this value into the SPORTA_TX register */
  *pREG_SPORT0_TX_A = res;
}

/* Description: Function to receive data into SPORT_B_RX register from UART Device,
                extract the sampled bits and return the assembled data for storage.
   Input Parameters:       void
   Return:          uint8_t value - Assembled Received Data for storage
*/
uint8_t SPORT_UART_Rx_Transfer()
{
  /* Oversample by 3 and extract the middle bit of every transmittted bit */
  uint32_t value;
```

```
/* Get the received middle stop bit */
uint8_t rxd_stop;

/* Receive data into Rx Buffer */
temp = *pREG_SPORT0_RX_B;

/* Extract the 8 bits from the 27 bits received */
value = 0;

switch (word_size)
{
  case 8: value += ((temp >> 23) & (1 << 0));      // bit 0
    value += ((temp >> 19) & (1 << 1));      // bit 1
    value += ((temp >> 15) & (1 << 2));      // bit 2
    value += ((temp >> 11) & (1 << 3));      // bit 3
    value += ((temp >> 7) & (1 << 4));       // bit 4
    value += ((temp >> 3) & (1 << 5));       // bit 5
    value += ((temp << 1) & (1 << 6));       // bit 6
    value += ((temp << 5) & (1 << 7));       // bit 7
    break;
  case 7: value += ((temp >> 20) & (1 << 0));
    value += ((temp >> 16) & (1 << 1));
    value += ((temp >> 12) & (1 << 2));
    value += ((temp >> 8) & (1 << 3));
    value += ((temp >> 4) & (1 << 4));
    value += ((temp >> 0) & (1 << 5));
    value += ((temp << 4) & (1 << 6));
    break;
  case 6: value += ((temp >> 17) & (1 << 0));
    value += ((temp >> 13) & (1 << 1));
    value += ((temp >> 9) & (1 << 2));
    value += ((temp >> 5) & (1 << 3));
    value += ((temp >> 1) & (1 << 4));
    value += ((temp << 3) & (1 << 5));
    break;
  case 5: value += ((temp >> 14) & (1 << 0));
    value += ((temp >> 10) & (1 << 1));
    value += ((temp >> 6) & (1 << 2));
    value += ((temp >> 2) & (1 << 3));
    value += ((temp << 2) & (1 << 4));
}

*pREG_SPORT0_CTL_B &= ~(1<< BITP_SPORT_CTL_B_SPEN);
*pREG_SPORT0_NUMTRAN_B = 2;
*pREG_SPORT0_CTL_B |= (1<< BITP_SPORT_CTL_B_SPEN);
```

```
  return value;
}

/* Interrupt Handler Routine for SPORT_A_TX */
void SPORT0A_Int_Handler()
{
  if ((i < (TRAN_SIZE)) && (*pREG_SPORT0_STAT_A & BITM_SPORT_STAT_A_DATA))
  {
    SPORT_UART_Tx_Transfer(&tbuf[i++]);
  }
  if(i >= TRAN_SIZE)
  {
    *pREG_SPORT0_CTL_A &= ~(1<< BITP_SPORT_CTL_A_SPEN);
  }
}

/* Interrupt Handler Routine for SPORT_B_RX */
void SPORT0B_Int_Handler()
{
  if((i < TRAN_SIZE) && (*pREG_SPORT0_STAT_B & BITM_SPORT_STAT_B_DATA))
  {
    rbuf[i++] = SPORT_UART_Rx_Transfer();
  }
  if(i >= TRAN_SIZE)
  {
    *pREG_SPORT0_CTL_B &= ~(1<< BITP_SPORT_CTL_B_SPEN);
  }
}
```

SPORT_UART_EMULATOR_TRANSMIT.C

```
#include "SPORT_UART_Emulator.h"

/* Main Function for Use Case (a) Transmission from SPORT_A and Reception by UART */
int main()
{
  /* Change PCLK to 26 MHz */
  Change_CLKDIV(1, 1);

  /* Enable the NVIC IRQ ID for SPORT A handler */
  NVIC_EnableIRQ(SPORT_A_EVT_IRQn);

  /* Create Data pattern for transmit buffer */
  for (int i=0; i < TRAN_SIZE; i++)
  {
    tbuf[i] = 0x13 + (0x19 << (i % 5)) + (0x6D << (i % 3));
  }

  /* Configure the SPORT_A for use case */
```

```
  SPORT_UART_Tx_Initialise();

  while(1) {}
}
```

SPORT_UART_EMULATOR_RECEIVE.C

```
#include "SPORT_UART_Emulator.h"

/* Main Function for Use Case (b) Transmission from UART and Reception by SPORT_B */
int main()
{
  /* Change PCLK to 26 MHz */
  Change_CLKDIV(1, 1);

  /* Enable the NVIC IRQ ID for SPORTB_Rx handler */
  NVIC_EnableIRQ(SPORT_B_EVT_IRQn);

  /* Configure the SPORT_B for use case */
  SPORT_UART_Rx_Initialise();

  while(1) {}
      }
```

CONCLUSION

This application note describes how to use the SPORT communication protocol on the ADuCM3029/ADuCM3027 processor to emulate a full duplex UART communication, which can be then used to interface with any standard UART device.

The use case presented in this application note is tested in core and direct memory access (DMA) modes for all standard baud rates. Reliable results are observed for baud rates up to 115,200 bps on the SPORT transmission cycle and up to 57,600 bps on the SPORT reception cycle. Data sizes ranging from 5 bits to 8 bits for transfers in both directions are tested for proper operation.