VORAGO VA108x0 UART programming application note

May 17, 2017 Version 1.0

VA10800/VA10820

Abstract

One of the oldest and still most popular serial interfaces is the UART (Universal Asynchronous Receive Transmit) interface. The VORAGO UART allows for high speed (up to 2 Mbps) communications with minimal CPU overhead. The flexible interrupt sources and the two 16-byte FIFOs reduce CPU intervention for most strings of characters. This application note provides guidance on using the UART control block. The VA108x0 evaluation board's BSP (board support package) comes with UART drivers included and lower level drivers accompany this document in the form of a compressed file, AN1209_SW.7z.

Table of Contents

1 Introduction to the UART interface

The first UART circuits were designed around 1970 by DEC for their PDP line of computers. It was intended to be a point to point communication protocol with a minimal number of wires. When personal computers first hit the scene around 1985, each had several UART based RS-232 interfaces to connect to peripherals. Since then, nearly all microcontrollers come equipped with UARTs to allow easy connection to either other boards or in some cases, to other ICs on the same board. Ironically, most PCs stopped including UART interfaces around 1990 in favor of USB with smaller cables and faster data rates.

The UART standard provides flexibility in five areas:

i) number of characters,
ii) bit rate,
iii) number of stop bits
iv) use of parity and
v) flow control (a.k.a. handshaking)

Flow control is not commonly used and is assumed not present unless expressly called out. One of the most common settings is 9600 baud (104.2 microseconds/bit), 8-bits of data, with no parity and a single stop bit. This is sometimes shortened to: 9600 8N1 or 9600,8,N,1.

While both Tx and Rx sides need fairly accurate clock references for reliable communications, the standard does allow for transceiver wave shaping and clock inaccuracy. See Figure 1. The receiving state-machine samples each bit a number of times, quite often 16, but only relies on samples in the middle of each bit to determine a "1" or "0" value. In the below example, the samples used are 7 – 9. If these three samples are not the same or if a stop bit is not detected, a framing error will be triggered.

Due to the STOP bit and START bit being opposite polarities, there will always be a signal transition at least every 9-bits regardless of the data value. Depending on the receiver sampling implementation, a transmitter to receiver timing mismatch of approximately 3% can be tolerated without an error occurring.

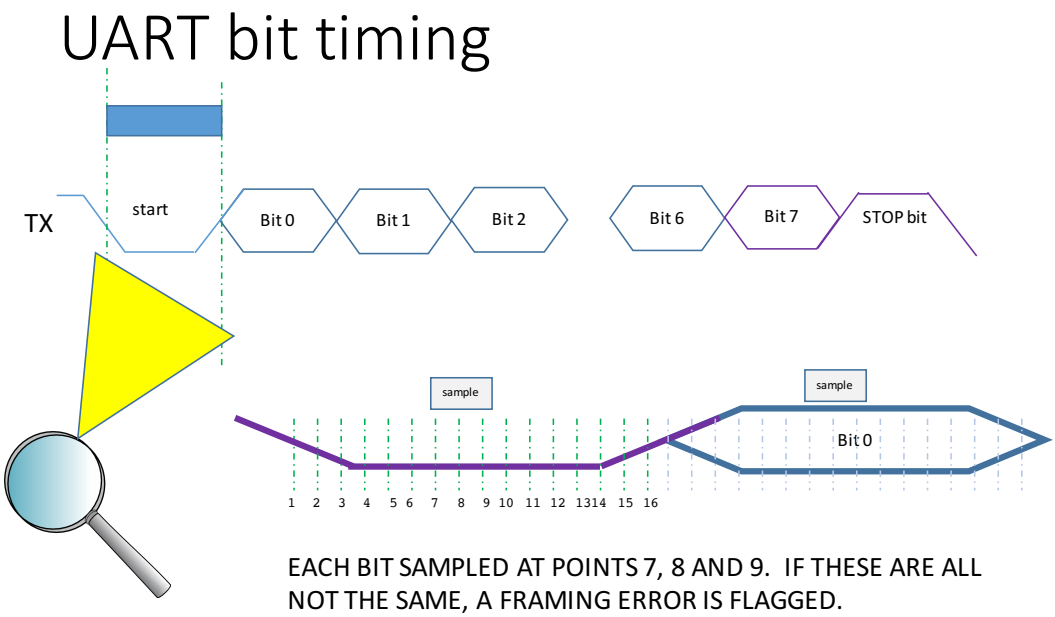


Figure 1 - UART bit sequence and individual bit timing diagram

Flow control can be handled with dedicated hardware signals RTS (ready to send) and CTS (clear to send) or by sending special ASCII flow control characters XOFF (pause transmission) and XON (resume transmission). See Figure 2 for implementation of flow control using a Null modem

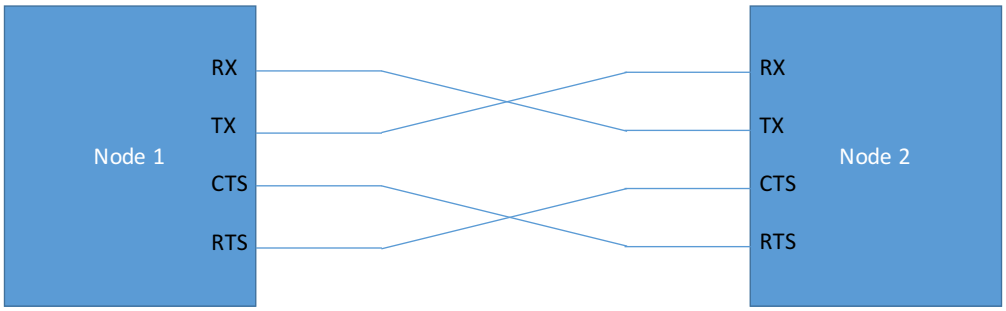


Figure 2 - Null modem connection between two UART nodes. CTS and RTS are only needed when using hardware handshaking.

ASCII (American Standard Code for Information Interchange) coded information is required for XOFF/XON flow control to be used. ASCII assigns an 8-bit number for standard keyboard entries (A-Z, a-z, 0-9, carriage return, punctuation marks and special characters). Many UART protocols are based on ASCII characters which allows the use of the Null character (0x00) to separate different messages or character strings.

The UART block on an MCU only supports the digital encoding and decoding of messages. Electrical signaling levels such as transceivers for RS-232, LIN or RS-485 are handled by a separate driver circuit external to the UART controller. The LIN (local-area-network) bus protocol is popular in automotive applications and is based on UART. LIN allows one master node and several slave nodes on the same bus. LIN requires a 13-bit break character. If the transmit line is held in the logic low condition for longer than a character time of 10-bits, this is a break condition that can be detected by the UART.

A related peripheral, USART (universal synchronous/asynchronous receiver/transmitter) supports both asynchronous and synchronous operation. The VA108x0 only supports asynchronous communication with no shared clock between devices.

2 VORAGO UART block overview

The full description of the block can be found in the VA10800/VA10820 Programmers Guide. This section provides supplemental information on how to use the block in a final application. A block diagram of the UART is shown in Figure 3. There are separate receive and transmit engines which allow full duplex communications.

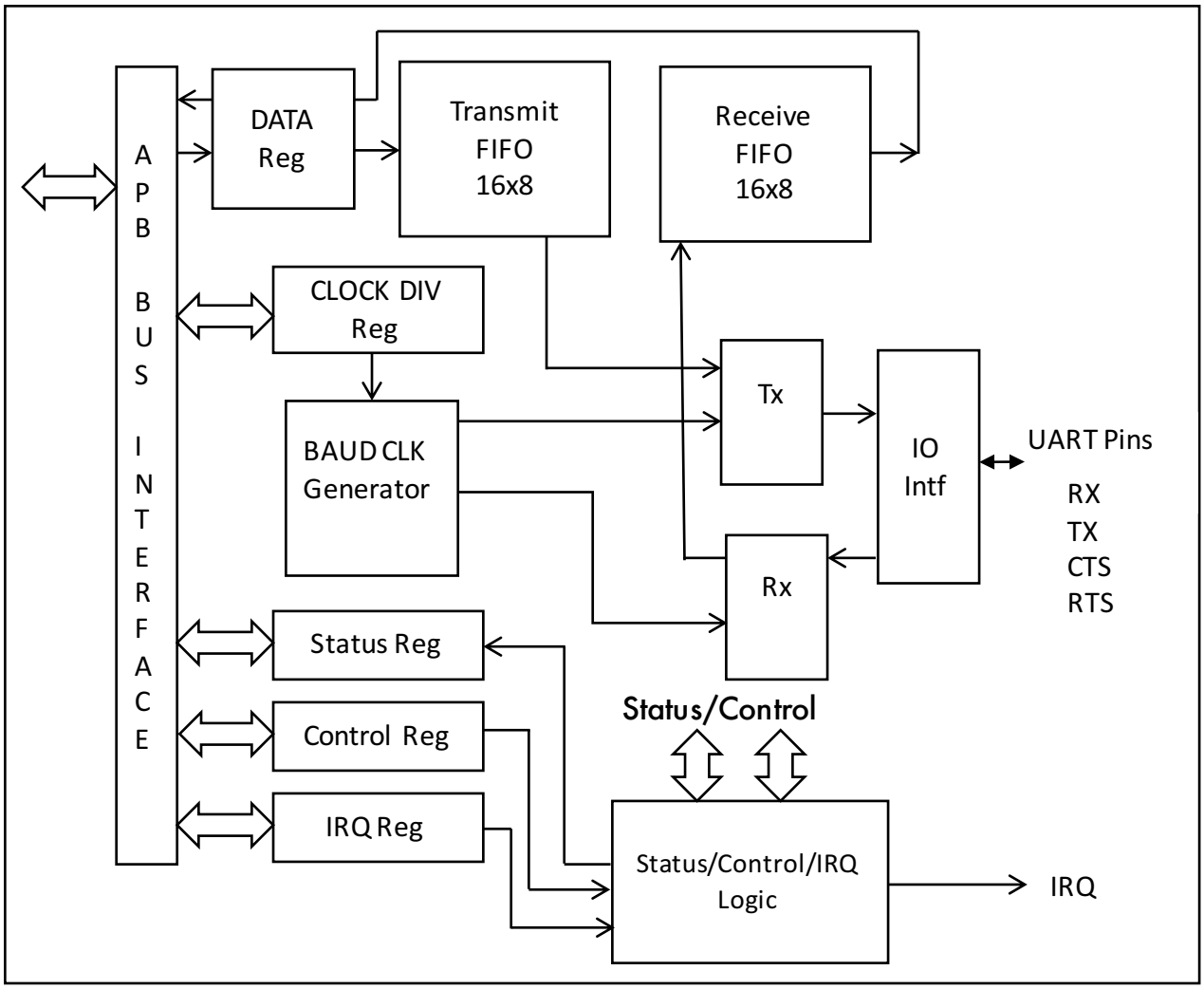


Figure 3 – VORAGO UART block diagram

There are two identical UART blocks on both the VA10800 and VA10820. The signals associated with these blocks are available on several different pins. The function select register (FUNSEL) in the IOCONFIG module determines which port pins have which functions assigned to them.

2.1 Register summary

Name	Description	Overview and Use
DATA	FIFO entry point for both Tx and Rx	This address is the CPU access point to the Transmit and Receive FIFOs.
ENABLE	Separate enable bits for Tx and Rx	Often the Rx enable is set and never cleared. This allows characters to be received at any time. Conversely, the Tx enable is normally set when data has been loaded in the FIFO and is ready for transmission.
CTRL	Configures the data format parameters and loopback mode	Parity, STOP bit length, word size and flow control are set.
CLKSCALE	Establishes baud rate	Fine granularity is provided with separate integer and fractional fields.
RXSTATUS	Contains 11 status bits on Rx data	Used to process information in FIFO and to handle overflow, framing and parity error conditions. Can be polled or read in ISR.
TXSTATUS	Contains 5 status bits on Tx data	Used to determine when to load more data into the Tx FIFO. Can be used as overflow error indicator.
FIFO_CLR	Clears status bits and FIFO	Used to clear error status flags and to empty Rx and Tx FIFO
TXBREAK	Sets the length of a break character	Some protocols such as LIN requires specific lengths to the break character
ADDR9	Enables 9-bit mode and contains match address	Used only for 9-bit mode which allows the parity bit to determine the if incoming information is an address or data
ADDR9MASK	Determine which bits of data in ADDR9 are used.	Allows more than one address to be used for matching.
IRQ_ENB	Enables one of 8 interrupt sources.	Works in conjunction with Status register to determine when the UART triggers an interrupt.
IRQ_RAW	Unqualified access to all interrupt source status bits	Normally only used for debug purposes to determine what has just happened
IRQ_END	Provides status of all enabled interrupt sources	The UART OR's all interrupt sources together and has one interrupt output to the NVIC. Inside the UART ISR, this register needs to be interrogated to determine which individual interrupt sources are active.
IRQ_CLR	Clears interrupt status information	Only can be used with overflow and error conditions.

RXFIFOIRQTRG	Determines when the FIFO depth will trigger an interrupt	Can be used to reduce number of interrupts for received data if the packet size is greater than one. If single character input must be responded to, this has limited use.
TXFIFOIRQTRG	Determines when the FIFO depth will trigger an interrupt	Can be used to greatly reduce the CPU overhead for transmitting blocks of data. If only transmitting single characters, it has limited use.
RXFIFORTSTRG	Determines when RTS signal triggers based on Rx FIFO depth	Only used when hardware flow control is active.
STATE	Reports state of Rx and Tx state machines and FIFOs	Used during code development and debug only.
PERID	Peripheral ID	Can be used to customize code when future UART from Vorago are available.

2.2 UART block functional partitioning:

Each of the functional blocks of the UART module are briefly described in this section.

- Baud rate generator: The UART bus clock (16x or 8x the baud rate) is created with two pieces: a) a divisor of the system bus clock and b) a fractional portion of divisor. This allows very accurate baud rates to be created for most every CPU clock and baud rate combination.
- Status and control registers: A set of registers is provided to configure the block and to monitor the status during transactions. The most notable settings are:
    - Number of data bits
    - Parity use
    - Number of stop bits
    - Flow control
    - Interrupt enable
- Interrupt logic: Interrupt requests to the CPU can be generated when they are enabled and the event occurs such as a FIFO being full or empty. The full list of 8 interrupt sources is shown here.
        - Receiver FIFO half full (IRQ_RX)
        - Receiver Status Conditions (IRQ_RX_STATUS)
        - Receiver timeout (IRQ_RX_TO)
        - Matched address in 9-bit mode (IRQ_RX_ADDR9)
        - Transmit FIFO half empty (IRQ_TX)

- Transmitter Status conditions (IRQ_TX_STATUS)
- TX FIFO empty (IRQ_TX_STATUS)
- Transmitter interrupt when CTS changes value (IRQ_TX_CTS)

2.3 MCU interrupt path

For an interrupt source to halt normal program flow, a path for it to get to the CPU must be configured via software. As shown below, the UART has 8 interrupt sources that are OR'ed together and tied to the IRQ Selector peripheral. The IRQ Selector peripheral has resources to route the >100 interrupt sources on the MCU to one of the 32 NVIC (nested vectored interrupt controller). The NVIC is part of the ARM core complex and can halt regular code execution and force a vector to be fetched and an interrupt subroutine (ISR) to be executed.

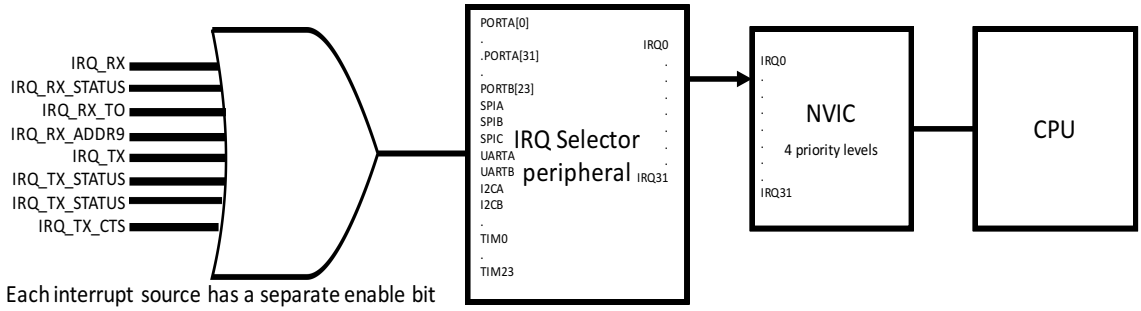


Figure 4 - Path for UART interrupt sources to CPU

2.4 Debug specifics

Understanding how the part works when a debugger is attached is important to comprehend UART behavior. When stopping and starting code execution, most debug probes stop the CPU and then read each register. This can unexpectedly take information from the FIFO. Subsequent code execution will not be able to recover that data.

The peripheral continues to run even when the CPU has been halted by the debugger. It is possible to transmit a character by writing to the DATA register after the enable bit and peripheral select register was setup. Conversely, once the receive function is enabled, the FIFO will continue to be filled as data is applied to the Rx pin. Receive overflow conditions can occur when the CPU is halted and new data is applied to the Rx pin.

Whenever starting and stopping CPU execution, take the information from the above two paragraphs into account.

3 Examples

The following sections provide example software programs to setup and operate the UART block. They grow in complexity sequentially. The code is intended to be run on the REB1 evaluation board. Table 1 provides a summary of the pins used and how to access them on the REB1 evaluation board.

Table 1 - UART pin assignments and REB1 evaluation board access points

Pin Name	UART function	Function Sel Value	REB1 access point
PORTA8	UARTA-RX	2	J10-8
PORTA9	UARTA-TX	2	J10-7
PORTB20	UARTB-RX	1	J14-11
PORTB21	UARTB-TX	1	J14-12
Place jumper between PORTA9 and PORTB20 for UARTA to transmit to UARTB			

3.1 UART peripheral initialization

Out of RESET, the block is disabled and all registers are set to their default value. Prior to writing any UART registers, the clock must be enabled in the System Configuration Peripheral -> Peripheral Clock Enable CTRL register.

The pins used for the UART functions are multiplexed with GPIO and other peripherals. They default to the GPIO function. The function select field (FUNSEL) in IO Configuration (IOCONFIG) peripheral must be setup for each UART pin. The IOCONFIG also has pin filters (FLTTYPE) available. By default, the filter is turned off and each pin is synchronously sampled every clock cycle. For systems with slow changing inputs on the Rx pin, it is advised to turn the filtering on so that a pin is sampled high or low several times before the UART module receives the high / low signal.

The baud rate must be set in the CLKSCALE register. The programmer's manual has a convenient table to reference for 50 MHz operation which is what the evaluation board has. The provided SW automatically calculates this value.

For this example, Hand shaking (flow control) and parity are not used. If they were, fields would need to be set in the UART CTRL register.

Summary of steps to setup the UART module

1. Enable peripheral clocks in SYSCONFIG->PERIPHERAL_CLK_ENABLE
2. Configure GPIO pins for the Rx and Tx function in IOCONFIG peripheral.
3. Configure the UART block
    a. Set UART clock generator CLKSCALE register
    b. Clear both Rx and Tx FIFO in the FIFO_CLR register
    c. Set operating parameters in the CTRL register and
    d. Enable the Tx and/or Rx engines in the ENABLE register

At this point, the module is ready to transmit or receive information.

Example code to show a single UART initialization is shown in Figure 5. UARTA will be used to transmit and UARTB will be used to receive. Note that the module is not enabled in this code. The enable step is performed in the main routine after other subsystems are initialized.

```
/*******************************************************************************************
 *  Init_UARTA_PortA
 *  Purpose:  Initialize UARTA on PORTA[8] and PORTA[9] pins
 *   Input - Baud rate
 *   Return - 1 = success
 *   Other parameters for UART:  Parity = off, Stop bits = 1, Flowcontrol = off
 *******************************************************************************************/
#define OSC_FREQUENCY 50000000
uint32_t Init_UARTA_PortA(uint32_t BaudRate)
{
  VOR_SYSCONFIG->PERIPHERAL_CLK_ENABLE |= (
                                        CLK_ENABLE_UARTA  | CLK_ENABLE_IRQSEL |
                                        CLK_ENABLE_IOMGR | CLK_ENABLE_UTILITY | CLK_ENABLE_PORTIO | CLK_ENABLE_SYSTEM );
  VOR_IOCONFIG->PORTA[9] =  ((0x2)<<IOCONFIG_PORTA_FUNSEL_Pos); // FUNSEL = 2 for UARTA_TX
  VOR_IOCONFIG->PORTA[8] = ((0x3) << IOCONFIG_PORTA_FLTTYPE_Pos | (0x2)<<IOCONFIG_PORTA_FUNSEL_Pos);
                          // filter rx input with 3 counts of system clock

  VOR_UARTA->CLKSCALE= ( (OSC_FREQUENCY / (BaudRate*16))<< 6)\
                          | ( ((OSC_FREQUENCY % (BaudRate*16))*64+(BaudRate*8))\
                              /(BaudRate*16) );
  VOR_UARTA->FIFO_CLR |= (UARTA_FIFO_CLR_TXFIFO_Msk | UARTA_FIFO_CLR_TXFIFO_Msk);
  VOR_UARTA->CTRL |= (0x3<<UARTA_CTRL_WORDSIZE_Pos) ; //set word size to 8 bits, by default parity is off and #stopbit set to 1

  #ifdef AN_Int_Rx_Tx
  VOR_UARTA->TXFIFOIRQTRG=3;
  VOR_IRQSEL->UART[0] = 17   ;  // route UARTA IRQ to NVIC input 17
  NVIC_SetPriority((IRQn_Type)17,1);
  NVIC_EnableIRQ((IRQn_Type)17);      //  NVIC enable
  #endif
  return 1;
}
```

Figure 5 - UART initialization code example

3.2 UART Transmit

Different strategies can be used depending on the number of bytes to transmit, the other tasks of the processor and baud rate relative to the bus frequency. For instance, if all transmit blocks will be under 16-bytes, it is probably best not to use interrupts and just load the FIFO and let the UART stream the data out. However, if large amounts of data, say over 100 bytes need to be sent in each block transfer, implementing an interrupt based

system with a buffer and data pointer is recommended. If there are no other high priority tasks, it is possible to use a polling method to service the UART as more data needs to be loaded in the FIFO.

A basic example with a very small software buffer is shown in this section. A more complex interrupt driven transmit example with a RAM based buffer is shown in section 3.5

Summary of steps:

1. Initialize module as shown in last section
2. Load FIFO with data
3. Start transmitting by setting the Tx enable bit.

```
/*****************************************************************************
 *  Tx_UARTA
 *  Purpose:  send data out
 *   Input - number of bytes to send (must be < 16) and pointer to buffer head.
 *   Return - 1 = success
 *****************************************************************************/
uint32_t Tx_UARTA(uint8_t Num_bytes, uint8_t *Tx_buff_ptr)
{
uint8_t tx_count = 0   ;

  while(tx_count < Num_bytes)      // load FIFO with data from buffer
  {
    VOR_UARTA->DATA = *Tx_buff_ptr++  ;
    tx_count++ ;
  }

  VOR_UARTA->ENABLE = (1<<UARTA_ENABLE_TXENABLE_Pos)  ;  // enable UART TX

  return 1;
}
```

Figure 6 - UART transmit example code

3.3 UART receive operation.

Receive operations are like the transmit operation except instead of writing to the Tx FIFO before the transaction, the received data is pulled from the Rx FIFO after the transaction is complete.

Steps to setup the module

1. Initialize UART block as shown in section 3.1
2. Enable reception by setting the Rx enable
4. Periodically check the RDAVL bit to see if data has been received.
    a. Although not necessary it is a good idea to check for framing errors
5. If RDAVL is set, read the DATA register,
6. Repeat step 5 until RDAVL is not set

A short code example to perform these steps is shown here.

```
228
229       UARTB_Rx_Buffer_ptr = UARTB_Rx_Buffer  ;
230       VOR_UARTB->ENABLE = (1<<UARTB_ENABLE_RXENABLE_Pos)  ;  // enable receiver on UARTB
231       UART_stat = Tx_UARTA(12, UARTA_Tx_Buffer_ptr)  ;     // transmit 12 bytes of data out UARTA
232
233       while(1)
234       {
235          if ((VOR_UARTB->RXSTATUS&UARTB_RXSTATUS_RDAVL_Msk) != 0 )
236          {
237              *UARTB_Rx_Buffer_ptr++ = VOR_UARTB->DATA  ;
238              if (UARTB_Rx_Buffer_ptr - UARTB_Rx_Buffer > 32)
239                   {UARTB_Rx_Buffer_ptr = UARTB_Rx_Buffer;} // always keep RX buffer size less than 32
240          }
241       }
```

Figure 7 - Example code for UART Rx

3.4 Using example code from 3.2 and 3.3 on REB1 board.

The code from the previous two examples is meant to work in conjunction. By placing a jumper wire between J10-7 and J14-11, UARTA_Tx is connected to UARTB_Rx. The software project in the accompanying AN1209_SW.7z file has defines and conditional includes. This allows both the simple and interrupt driven routines to be placed in the same reb_main.c file. Please make sure that line 32 is not commented out and that line 33 is commented out. The compiler will include the "simple_Rx_Tx" sections and ignore the "Int_Rx_Tx" sections.

```
32  #define AN_simple_Rx_Tx 1
33  // #define AN_Int_Rx_Tx 1
```

```
197 #ifdef AN_simple_Rx_Tx
198   UART_stat = Init_UARTA_PortA(115000)  ;
199     if(UART_stat == 0)  { while(1)  ;  }  //  hang here if response is negative.
200
```

Compile, download and run the program. After a second or so, stop the code execution and open a memory window in the debugger to show "UARTB_Rx_Buffer. Display the information in ASCII format and you should see something like Figure 8.

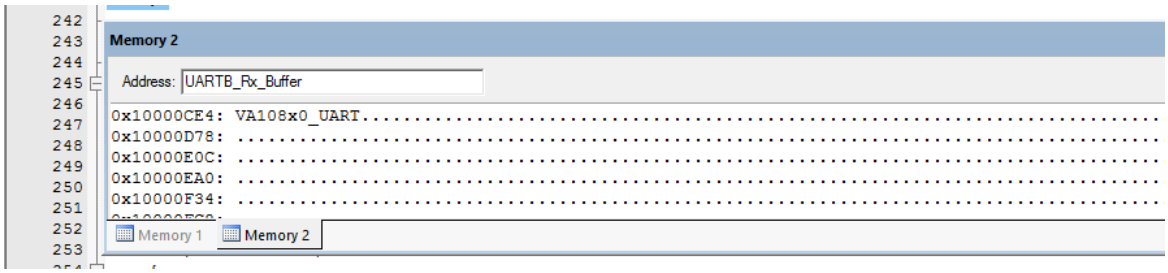


Figure 8 - Screen capture of Keil IDE showing Rx buffer in a memory window.

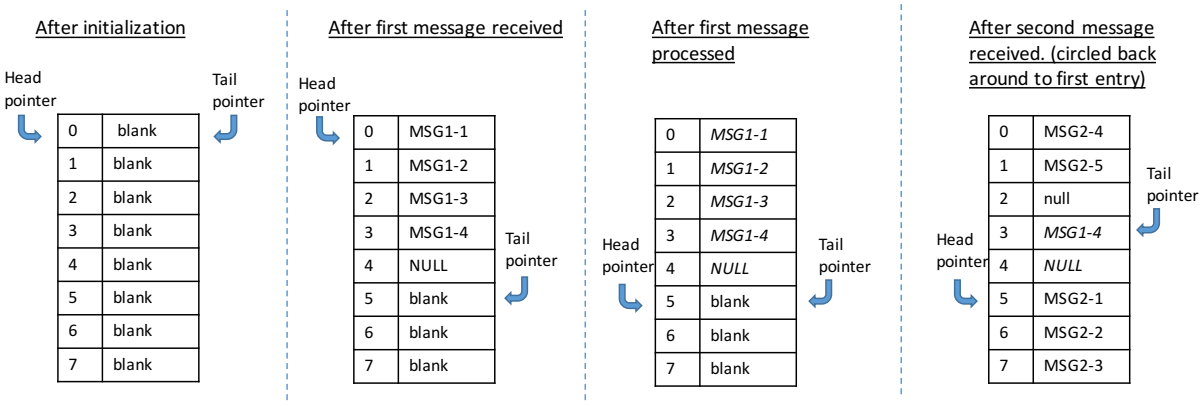
3.5 Interrupt driven string transmission

Most UART applications are interrupt driven. This will limit wasted CPU bandwidth waiting for an event to occur. This section provides information on how to setup the MCU to allow interrupt driven actions to transmit a buffer loaded with ASCII information.

3.5.1 Circular buffer concept

A commonly used structure with serial interfaces is a circular buffer. This allows one software task to load the buffer and another task to unload it when an event occurs. The task for loading the buffer uses a "tail pointer" as an index into the buffer. The task reading the buffer uses a "head pointer" to track the last location it has written. The graphic in Figure 9 explains the concept further.

8-entry Circular buffer implementation for receiving UART information



Notes:

1. Tail pointer always points to next location to place incoming data.
2. Head Pointer always points to first unread data in buffer.
3. ISR uses tail pointer to track where to store data
4. Main routine uses head pointer to read data from the buffer
5. Head pointer can never lead tail pointer
6. Both pointers roll over to zero when they reach the buffer size value.
7. Overflow errors occur if the tail pointer increments to a value equal to the head.

Figure 9 - Circular buffer explanation

The appropriate size of a buffer is dependent upon: a) baud rate, b) interval the software can process buffer data, and c) message length and interval. The below table provides the maximum bytes that could be sent or received for common baud rates and loop intervals.

Table 2- Maximum number of bytes transferred for various baud rates and intervals.

	Buffer Check interval		
Baud Rate	10msec	100msec	1 sec
9600	9.6	96	960
115200	115.2	1152	11520
1000000	1000	10000	100000

3.5.2 MCU configuration for UART Tx interrupts

Steps to setup the module for generating interrupts on a FIFO half empty condition

1. Initialize the UART block as described in section 3.1.
2. Set the FIFO half empty trigger level (default is 8)
3. Set the IRQ_Tx bits in the IRQ_ENB register
4. Assign the UART interrupt to an NVIC input (input 17 is used for this example) in the IRQSEL block.

5. In the NVIC, set priority level of IRQ17 and enable interrupts on the NVIC OC17.

Example code is shown Figure 5. See the section near the bottom beginning with "#ifdef AN_int_Rx_Tx".

To demonstrate the receive implementation described in the next section handles all message lengths, this transmit example increments the transmit string length by 1 until it reaches the maximum value. The code to setup the transmission is shown in Figure 10 and the ISR code is shown in Figure 11. The project AN1209_SW.7z has all this code also.

```
307         if(count > 100000/3)   //  occurs approximately every 14 msec
308         {
309           tx_char_count = tx_char_count + 1  ;
310           if(tx_char_count > 104) { tx_char_count = 5  ; }
311           uartTxBufHead=uartTxBuf;   // set head pointer to start
312           uartTxBufTail=uartTxBuf+tx_char_count;  // set tail to different lengths
313           VOR_UARTA->FIFO_CLR |= UARTA_FIFO_CLR_TXFIFO_Msk ;
314           VOR_UARTA->ENABLE = (1<<UARTA_ENABLE_TXENABLE_Pos)  ;  // enable UARTA TX
315           VOR_UARTA->IRQ_ENB= UARTA_IRQ_ENB_IRQ_TX_Msk  ;  // enable interrupts
316
317           count = 0 ;
318         }
```

Figure 10 - Example code for initiating an interrupt driven buffer transmission

```
137 /*****************************************************************************************
138  *  ISR_Tx_UARTA
139  *  Purpose:  send data out from Tx buffer until head pointer = tail pointer
140  *             load FIFO until it is full.
141  *   Input - called when half full mark reached
142  *   Return - none  (head_pointer is incremented once for each byte written to DATA register.
143  *
144  *****************************************************************************************/
145
146 void OC17_IRQHandler(void)
147 {
148     while((uartTxBufTail > uartTxBufHead) && (VOR_UARTA->TXSTATUS & UARTA_TXSTATUS_WRRDY_Msk)) // "1" indicates FIFO not full
149     {
150        VOR_UARTA->DATA = *uartTxBufHead++  ;
151     }
152     if ( uartTxBufTail == uartTxBufHead)
153         {
154           VOR_UARTA->IRQ_ENB = 0x00 ;  // end of data reached -> shut off ints
155         }
156 }
157
```

Figure 11 - Interrupt service routine to load transmit FIFO from previously filled buffer.

3.6 Interrupt driven receive operation

Several receive interrupt sources are available and several software strategies can be used. One of the most common strategies for receiving information that is randomly spaced and with random length is twofold: 1) handle the heavy lifting (receiving lots of data in a short interval) with an interrupt and 2) check for remnants (small number of data bytes in the FIFO) periodically in one of the control loops.

The receive interrupt source for this example is the FIFO half full condition. The half full level is set to 12. The ISR will empty the FIFO using the RDAVL flag to determine when it is empty. Figure 12 shows example code for the Rx ISR. Until there are 12 bytes on information in the FIFO, no interrupt will occur. This could cause a stall or long delay in processing the information if the main routine does not periodically check the RAVL flag to see if there is any FIFO data available. Figure 13 shows example code for checking the Rx FIFO and processing the Rx buffer data. This code is executed every 10 msec.

```
/*******************************************************************************
*  ISR_Rx_UARTB
*  Purpose:  move data from out from FIFO to Rx buffer until FIFO is empty
*   Input - none
*   Return - none  (head_pointer is incremented once for each byte written to DATA register.
*
*******************************************************************************/

void OC18_IRQHandler (void)
{
       if ((VOR_UARTB->RXSTATUS&UARTB_RXSTATUS_RDAVL_Msk) != 0 )
       {
           *uartRxBufTail++ = VOR_UARTB->DATA  ;
           if (uartRxBufTail - uartRxBuf > (UARTB_RX_BUFFER_LEN-1))
           { uartRxBufTail = uartRxBuf;} // circular buffer - when reach end, point back to beginning

           RxBuffNotEmpty =  1 ;  //  set flag to inform main routine that data is available

           NVIC_ClearPendingIRQ((IRQn_Type)18);
       }
}
```

Figure 12 - Interrupt service routine to move data from Rx FIFO into buffer

```
      while ((VOR_UARTB->RXSTATUS&UARTB_RXSTATUS_RDAVL_Msk) != 0 )  // check UARTB RX for data
                              //   need to do this to check on case where #char < FIFO half full level
      {
          *uartRxBufTail++ = VOR_UARTB->DATA  ;
          RxBuffNotEmpty =  1 ;  //  set flag to inform main routine that data is available
          if (uartRxBufTail - uartRxBuf > (UARTB_RX_BUFFER_LEN-1))
             { uartRxBufTail = uartRxBuf;} // circular buffer - when reach end, point back to beginning
      }

      if (RxBuffNotEmpty != 0 )   //  process data in Rx Buffer.  (just read and discard)
      {
          while (uartRxBufTail != uartRxBufHead)
          {
           temp = *uartRxBufTail   ;
           uartRxBufHead ++ ;
           if (uartRxBufHead - uartRxBuf > UARTB_RX_BUFFER_LEN-1)
              { uartRxBufHead = uartRxBuf;} // circular buffer - when reach end, point back to beginning
          }
          RxBuffNotEmpty = 0  ;
      }
```

Figure 13 - Example code to check Rx FIFO and to process data in the Rx buffer.

3.6.1 MCU configuration for UART Rx interrupts

Steps to setup the module for generating interrupts on a FIFO half empty condition

1. Initialize the UART block as described in section 3.1.
2. Set the FIFO half full trigger level to 1 (default is 8)

3. Set the IRQ_Rx bit in the IRQ_ENB register
4. Assign the UART interrupt to an NVIC input (input 18 is used for this example) in the IRQSEL block.
5. In the NVIC, set priority level of IRQ18 and enable interrupts on the NVIC OC18.

Example code is shown in Figure 14.

```
115     #ifdef AN_Int_Rx
116     VOR_UARTB->RXFIFOIRQTRG=1;
117     VOR_UARTB->IRQ_ENB= UARTB_IRQ_ENB_IRQ_RX_Msk  ;  // enable interrupts
118
119     VOR_IRQSEL->UART[1] = 18   ;  // route UARTB IRQ to NVIC input 18
120     NVIC_SetPriority((IRQn_Type)18,1);
121     NVIC_EnableIRQ((IRQn_Type)18);      //  NVIC enable
122     #endif
123
124     return 1;
125   }
```

Figure 14 - Example code for configuring MCU for UART Rx interrupts

3.7 Using example code from 3.5 and Error! Reference source not found. on REB1 board

The code examples from the previous two examples are meant to work in conjunction. By placing a jumper wire between J10-7 and J14-11, UARTA_Tx is connected to UARTB_Rx. The software project in the accompanying AN1209_SW.7z file has defines and conditional includes. This allows both the simple and interrupt driven routines to be placed in the same reb_main.c file. Please make sure that line 33 is not commented out and that line 32 is commented out as shown in Figure 15.

```
32  // #define AN_simple_Rx_Tx 1
33   #define AN_Int_Rx 1
```

Figure 15 - Compiler directive to use interrupt driven, "AN_Int_Rx_Tx" example code.

```
115     #ifdef AN_Int_Rx
116     VOR_UARTB->RXFIFOIRQTRG=1;
117     VOR_UARTB->IRQ_ENB= UARTB_IRQ_ENB_IRQ_RX_Msk  ;  // enable interrupts
118
119     VOR_IRQSEL->UART[1] = 18   ;  // route UARTB IRQ to NVIC input 18
120     NVIC_SetPriority((IRQn_Type)18,1);
121     NVIC_EnableIRQ((IRQn_Type)18);      //  NVIC enable
122     #endif
```

Figure 16 - Example code showing conditional inclusion of "AN_Int_Rx_Tx" implementation

Compile, download and run the program. After a second or so, stop the code execution and open a memory window in the debugger to show "uartTxBuf" and uartRxBuf". Display the information in ASCII format. It should be similar to what is shown in Figure 17.

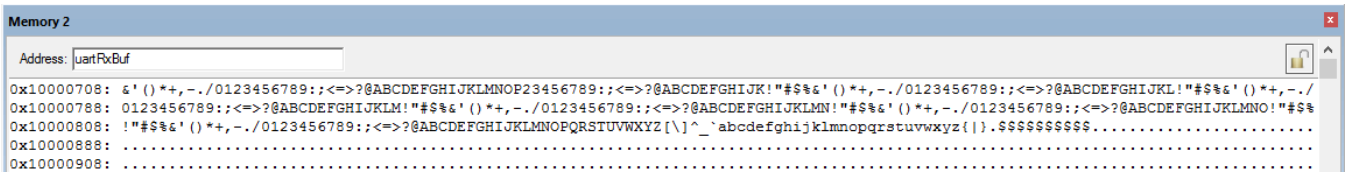


Figure 17 - Memory Window inside debugger showing Rx Buffer starting at 0x10000708 and Tx Buffer starting at 0x10000808

4 Conclusions

The VORAGO UART block has many options and a 16-word FIFO that makes it very flexible and capable of very efficient transaction management for a wide variety of UART message transfers and protocols. VORAGO has provided several forms of software drivers to utilize the UART peripheral.

This application note has provided several example operations for reading and writing to different UART devices using both polling and interrupt driven methods. You should be able to quickly adapt one of these examples to fit your application needs.

5 Common questions and issues

1. Can I invert the polarity of the Rx and Tx signals?
    a. Yes, via the IO configuration register (bit name = INVOUT). This is in the IO Configuration peripheral which is separate from the UART module.

2. Can I create differential physical layers like RS422?
    a. Yes, but this requires a separate hardware transceiver.
3. Is it possible to build a multipoint drop system like RS485?
    a. Yes, with appropriate software control. The UART peripheral does not have the capacity for addressing and disabling Tx built in.
4. How do you control color of terminal screen characters?
    a. Color monitor and terminal windows use special ASCII characters to control color. Here is a list of commands that can be embedded in a printf statement to set color.
        i. ANSI_COLOR_RED "\x1b[31m"
        ii. ANSI_COLOR_GREEN "\x1b[32m"
        iii. ANSI_COLOR_YELLOW "\x1b[33m"
        iv. ANSI_COLOR_BLUE "\x1b[34m"

5. Can you provide recommendations on when and when not to use pin filtering?
    a. Anytime an input signal has a rise or fall time > 5 MCU clock cycles is a good time to use pin filtering to remove potential glitches as the input switches states.
6. My code sets up all the UART registers but nothing comes out the port pin when a character is transmitted. What is going on?
    a. This most likely is caused by one of two things:
        i. The peripheral clock for the UART not being enabled
        ii. The port pin not being set to the UART function in the Function Select register.
7. I can see data at the output of my transmitter but the receiver never responds. What can be going wrong?
    a. This quite possibly is a problem with the Rx and Tx lines not being cross wired. The transmitter (TX1) of one device is connected to the receiver (RX2) of the other device. Similarly, RX1 is connected to TX2. See the null modem diagram in section 1.
8. Does the VA108x0 have support for DMA transfers with the UART?
    a. No. The VA108x0 does not have a DMA engine.
9. How can I send 9-bit wide data if the FIFO is only 8-bits wide?
    a. 9-bit mode uses the 9th bit to designate the 8-bit transfer is either an address or data. For transmitting, parity (PAREN = 1) and manual parity (PARMAN=1) must be enabled. The DPARITY bit in the data register must be set for each character loaded into the FIFO.

6 Other Resources

VORAGO VA108x0 programmers guide & VORAGO MCU products:
http://www.voragotech.com/VORAGO-products

VORAGO Application notes: http://www.voragotech.com/resources

VORAGO VA108xx REB1board user guide: Part of Board Support Package (BSP)
http://www.voragotech.com/products/reb1

LIN sub-bus specification: https://www.iso.org/standard/61229.html

Revision log:

May 18, 2017 – Initial release