SourceBoost UART Driver

for PICmicro

Reference Manual

SourceBoost UART Driver

SourceBoost UART or RS232 driver is a header based template library that works with hardware UARTs. It uses asynchronous interrupt driven code that operates on user supplied memory in circular buffer fashion. All driver calls are not blocking. The UART driver replaces old rs232 driver from rs232_driver.h. rs232_driver.h is still included into SourceBoost installation for backward compatibility but all new development code should use the new UART driver from uart_driver.h

Driver Features

- Very flexible UI where every UART bit is configured separately
- Asynchronous send and receive
- Any number UARTS can be used in the same application
- All code is header based, no libraries need to be added to project
- Interrupt driven
- All calls are non blocking
- Uses user supplied memory
- Stores data in circular buffers

Rationale

SourceBoost UART driver uses function templates. This provides several advantages over more traditional library based approach. Template arguments pass to the driver code information that is known at compile time and this lets compiler generate very efficient code. Template arguments make code highly configurable where every aspect of hardware UART can be customised: starting from receive and transmit registers and up to the bits that indicate frame errors or data overflow.

Adding UART driver to your code

To add serial driver to your code just include the header file *uart_driver.h*

```
#include "uart_driver.h"
```

Configuration

Driver can be used in single and multiple port modes. The single port mode can be used if user application needs to work with one serial port only. To enable single port mode define SINGLE_PORT_MODE in compiler command line or in source files before serial driver is included. If SINGLE_PORT_MODE is not defined driver will work in multiple port mode:

```
#define SINGLE_PORT_MODE
#include "uart_driver.h"
```

Note that you don't have to use single port mode if your code uses only one UART. Multiple port mode can still be used in applications that work with one UART but such code will use a bit more ram and generate slightly bigger code.

Memory

UART driver does not allocate any memory to deal with serial data but uses application supplied memory instead. Application code must supply receive and transmit buffers and a number of utility variables. In single port mode these variables must have pre-defined names as specified in the table below. For multiple port mode they can have any names. With the exception of *rxCnt* all these variables are used internally by the driver and user code should not access or especially modify any of them.

unsigned char txBuffer[TX_BUFFER_SIZE]	Memory for circular transmit buffer. Can be of any size but power of 2 sizes produce most compact code. Used internally by driver code.
unsigned char txHead	Position in transmit buffer where data to be transmit is written. Used internally by the driver.
unsigned char txTail	Position in transmit buffer to the next data byte to be sent out. Used internally by the driver.
unsigned char rxBuffer[RX_BUFFER_SIZE]	Memory for circular receive buffer. Can be of any size but power of 2 sizes produce most compact code. Used internally by the driver.
unsigned char rxHead	Position in receive buffer where next received data byte is written. Used internally by the driver.
unsigned char rxTail	Position in receive buffer where next received data byte is read from. Used internally by the driver.
unsigned char rxCnt	Number of data bytes in receive buffer. Can be used by user code to check if there is any data in receive buffer.

Helper Macros

Because UART river is header based and uses function templates all its API functions use some template arguments. Template arguments pass to the driver information that is known at compile time and this helps compiler to generate very efficient code. This also makes driver highly configurable. To make code more readable it's a good idea to use defines that will replace template calls with more readable identifiers. For PIC18 targets such defines for UART1 will look like:

```
#define uart1Init \
      rs232Init<PIE1,TX1IE,PIE1,RC1IE,RCSTA,CREN,RCSTA,SPEN>
#define uart1TxInterruptHandler \
      rs232TxInterruptHandler<PIR1,TX1IF,TXREG1,sizeof(txBuffer), \
      TXSTA,TXEN,TXSTA,TRMT>
#define uart1RxInterruptHandler \
      rs232RxInterruptHandler<PIR1,RC1IF,RCREG1,sizeof(rxBuffer), \
      RCSTA,CREN,RCSTA,OERR,RCSTA,FERR>
#define uart1Rx \
      rs232Rx<sizeof(rxBuffer)>
#define uart1Tx   \
      rs232Tx<sizeof(txBuffer),TXSTA,TXEN>
```

and for PIC18 UART2:

```
#define uart2Init \
      rs232Init<PIE3,TX2IE,PIE3,RC2IE,RCSTA2,CREN,RCSTA2,SPEN>
#define uart2TxInterruptHandler \
      rs232TxInterruptHandler<PIR3,TX2IF,TXREG2,sizeof(txBuffer), \
      TXSTA2,TXEN,TXSTA2,TRMT>
#define uart2RxInterruptHandler \
      rs232RxInterruptHandler<PIR3,RC2IF,RCREG2,sizeof(rxBuffer), \
      RCSTA2,CREN,RCSTA2,OERR,RCSTA2,FERR>
#define uart2Rx \
      rs232Rx<sizeof(rxBuffer)>
#define uart2Tx \
      rs232Tx<sizeof(txBuffer),TXSTA2,TXEN>
```

Initialisation

User code is responsible to set up port baud rate and configure interrupts used for UART transmit and receive. After this is done user code must call *rs232Init* driver function. Sample initialisation for PIC18 may look like:

```
      //Configure serial port speed and interrupt
      ipr1.TXIP = 0; ipr1.RCIP = 0;   //use low priority interrupt
      txsta.BRGH = 1;                 //high speed
      spbrg = 64;                     //9600kbps/10Mhz

      //Configure UART pins
      trisc.7 = 1;
      trisc.6 = 0;

      //Init uart driver
      uart1Init();
```

Interrupt Handler

UART driver uses interrupts to receive and transmit data and its interrupt handlers must be called from relevant interrupt code. Note that single port code does not use any call arguments but in multiple port mode buffers and helper variables must be passed to the handlers as call arguments. Sample code below shows how to write code for single port mode:

```
      … other interrupt code ...
      uart1TxInterruptHandler();
      if( uart1RxInterruptHandler() )
      {
            … new data just arrived ...
      }
      … other interrupt code …
```

and for multiple port mode:

```
      … other interrupt code …
      uart1TxInterruptHandler( txBuffer, txTail, txHead );
      if( uart1RxInterruptHandler( rxBuffer, rxHead, rxCnt ) )
      {
            … new data just arrived ...
      }
      … other interrupt code …
```

Data Receive

To check if there is any data in receive buffer the rxCnt variable can be used. This variable will contain number of data bytes available in receive buffer. Example of the code that check and reads incoming data for single port mode is below:

```
if( rxCnt )
{
     unsigned char data = uart1Rx();
     ... code that handles incoming data ...
}
```

And this is receive code for multiple port mode:

```
if( rxCnt )
{
     unsigned char data = uart1Rx( rxBuffer, rxTail, rxCnt );
     ... code that handles incoming data ...
}
```

Alternatively other mechanism that passes information about received data from interrupt handler can be employed:

```
void interrupt( void )
{
     ...
     if( uart1RxInterruptHandler() )
     {
          // Signal rx semaphore
          SysSignalSemaphoreIsr( hRxSem );
     }
     ...
}
...
unsigned char Rx( void )
{
     //Wait for serial data to come
     Sys_WaitSemaphore( hRxSem, EVENT_NO_TIMEOUT );
     //Read one byte from rx queue
     SysCriticalSectionBegin();
     unsigned char data = uart1Rx();
     SysCriticalSectionEnd();
     return data;
}
```

Data Transmit

To transmit data call *uart1Tx*. For single port mode such code may look like:

```
uart1Tx( data );
```

And for multiple port mode:

```
uart1Tx( data, txBuffer, txHead );
```

rs232Init

Driver initialisation code

Template arguments	
unsigned short TxIrqEnableRegisterAddr	address of the register where tx irq enable flag is located
unsigned char TxIrqEnableBit	zero based position of tx irq enable bit
unsigned short RxIrqEnableRegisterAddr	address of the register where rx irq enable flag is located
unsigned char RxIrqEnableBit	zero based position of rx irq enable bit
unsigned short RxEnableRegisterAddr	address of the register where rx enable flag is located
unsigned char RxEnableBit	zero based position of rx enable bit
unsigned short UartEnableRegisterAddr	address of the register where uart enable flag is located
unsigned char UartEnableBit	zero based position of uart enable bit

Function arguments	
void	none

Return value	
void	none

rs232TxInterruptHandler

Transmit interrupt handler

Template arguments	
unsigned short TxIrqFlagRegisterAddr	address of the register where tx interrupt flag is located
unsigned char TxIrqFlagBit	zero based position of tx interrupt bit
unsigned short TxDataRegisterAddr	address of the data tx register
unsigned char TxBufferSize	size of the tx buffer
unsigned short TxEnableRegisterAddr	address of the register where tx enable flag is located
unsigned char TxEnableBit	zero based position of tx enable bit
unsigned short	address of the register where tx finished flag

TxFinishedRegisterAddr	is located
unsigned char TxFinishedBit	zero based position of tx finished bit

Function arguments	
void	none in single port mode
unsigned char *txBuffer	address of transmit buffer (used in multiple port mode only)
unsigned char &txTail	reference to txTail variable (used in multiple port mode only)
unsigned char &txHead	reference to txHead variable (used in multiple port mode only)

Return value	
void	none

rs232RxInterruptHandler

Receive interrupt handler

Template arguments	
unsigned short RxIrqFlagRegisterAddr	address of the register where rx interrupt flag is located
unsigned char RxIrqFlagBit	zero based position of rx interrupt bit
unsigned short RxDataRegisterAddr	address of the data rx register
unsigned char RxBufferSize	size of the rx buffer
unsigned short RxEnableRegisterAddr	address of the register where rx enable flag is located
unsigned char RxEnableBit	zero based position of rx enable bit
unsigned short RxOverflowRegisterAddr	address of the register where rx overflow error flag is located
unsigned char RxOverflowBit	zero based position of rx overflow error bit
unsigned short RxFrameRegisterAddr	address of the register where rx frame error flag is located
unsigned char RxFrameBit	zero based position of rx frame error bit

Function arguments	
void	none in single port mode
unsigned char *rxBuffer	address of receive buffer (used in multiple port mode only)

unsigned char &rxHead	reference to rxHead variable (used in multiple port mode only)
unsigned char &rxCnt	reference to rxCnt variable (used in multiple port mode only)

Return value	
bool	*true* if new data was received and *false* if no new data was received

rs232Rx

Receive one byte of data

Template arguments	
unsigned char RxBufferSize	size of the rx buffer

Function arguments	
void	none in single port mode
unsigned char *rxBuffer	address of receive buffer (used in multiple port mode only)
unsigned char &rxTail	reference to rxTail variable (used in multiple port mode only)
unsigned char &rxCnt	reference to rxCnt variable (used in multiple port mode only)

Return value	
unsigned char	data byte received over UART

rs232Tx

Transmit one byte of data

Template arguments	
unsigned char TxBufferSize	size of the tx buffer
unsigned short TxEnableRegisterAddr	address of the register where tx enable flag is located
unsigned char TxEnableBit	zero based position of tx enable bit

Function arguments	
unsigned char data	data byte to send
unsigned char *txBuffer	address of receive buffer (used in multiple port mode only)

unsigned char &txHead	reference to rxHead variable (used in multiple port mode only)

Return value	
void	none

Technical support

For example projects and updates please refer to our website: http://www.sourceboost.com

We operate a forum where technical and license issue problems can be posted. This should be the first place to visit:
http://forum.sourceboost.com

Legal Information