KSDK-based Modbus RTU Packet Assembler

How to Implement a KSDK-based Modbus RTU Packet Assembler Running on MQX

I'll start off by disclosing a big mistake I had made when implementing Modbus RTU. Skip to the next paragraph if you don't care to read about it. :) I took my existing knowledge of Modbus TCP and then only looked at the packet structure differences between it and RTU. In other words, I had merely focused on the header differences and the addition of the CRC word, but did not realize early on that there were timing requirements as well. My first Modbus RTU design spoke to a single device only, so I was able to determine the packet structure on the fly by analyzing each byte, figuring out how much data to expect, and then to only read the necessary bytes from the UART. It actually worked great until I had to implement a second RTU device on the same node. Herein lies the big problem -- when Slave 1 responds to the Master, it sends data back that does not contain any packet size information! This very packet is also simultaneously being send to Slave 2, who will be completely unable to determine anything about the incoming data. So the question at this point was, how in the world do I know when a packet is complete?

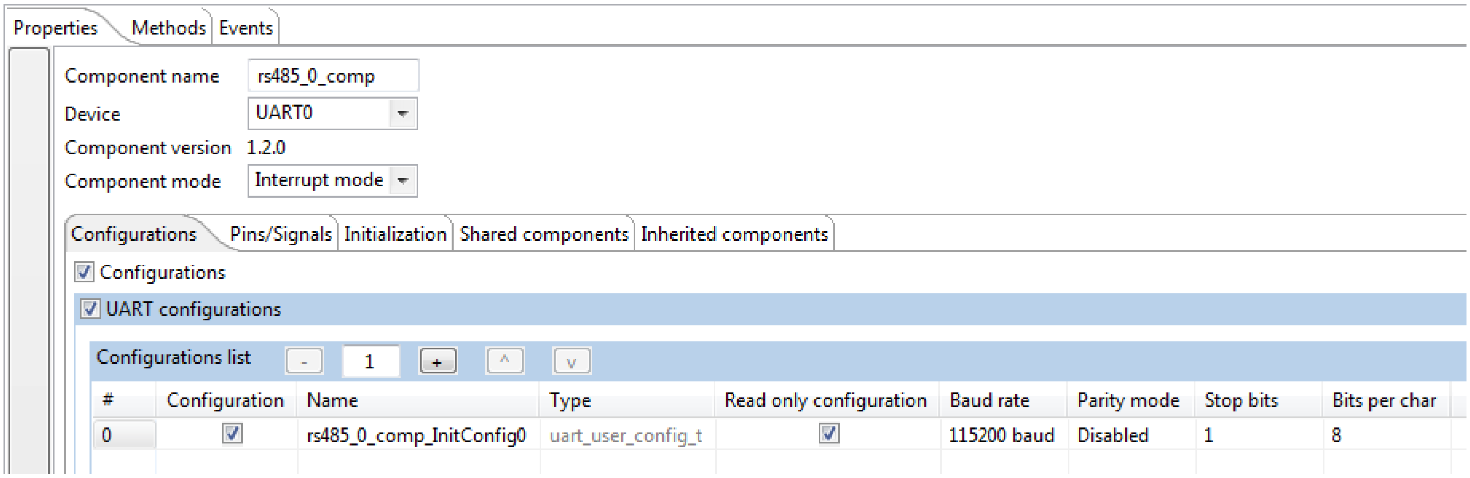
The Modbus RTU reference site I used (not the specification document) actually specifies very clearly (after the packet information) that there are timing requirements that have to be strictly adhered to. One is the intercharacter delay, which is 750us for baud rates >= 19200, and the interpacket delay, which is 1750us for baud rates >= 19200. For slower rates, these delays are referred to as 1.5T and 3.5T, respectively, because they are measured as 1.5 x (character time) and 3.5 x (character time). I'm not interested in anything slower than 115200 baud, so this document is only going to explain how to deal with the fixed timing delays.

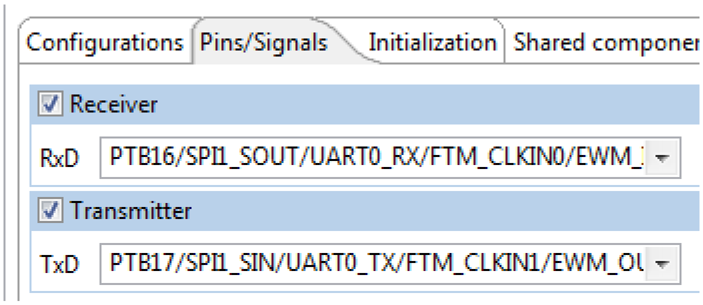
Sometimes gradually building on top of an old implementation causes more pain than it's worth. I tried the following approaches, with unsuccessful results:

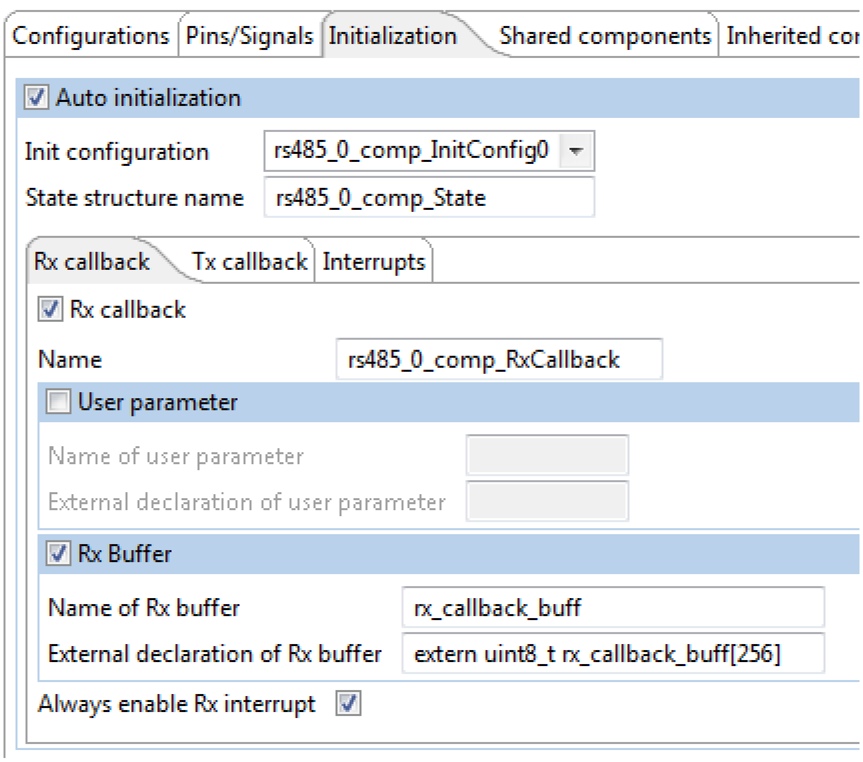
1. MQX task polls for 1 byte at time and keeps track of timing with the various KSDK timer functions that have microsecond precision. Almost worked, but often ran into a possible problem with the KSDK UART read function that doesn't like to read out 1 byte at a time.
2. MQX task runs as a FSM based on the Modbus RTU spec. This was very easy to follow and also very easy to track the intercharacter and interpacket delay requirements. It was less successful than attempt #1. The microsecond time function in KSDK maxes out at 5000, so the timer should *never* rollover when looking for a 750us or 1750us timeout. However, because the FSM advances only once each time the task executes, and because MQX has a 5ms OS tick, packets would "fail" because the rollover would screw up the calculation. Clearly, MQX cannot handle things down at the protocol level.

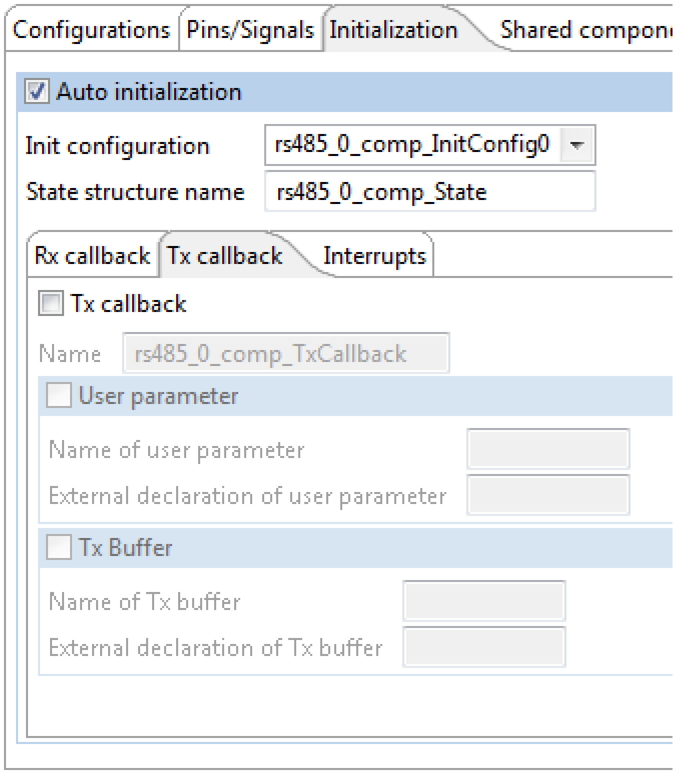
The working solution took a little bit of effort, but the idea behind it is based off of Mark Butcher's uTasker Modbus RTU code. The implementation comes down to a RX callback function, two PITs, and a message queue to share the packet with your MQX task. I will go into those details next.

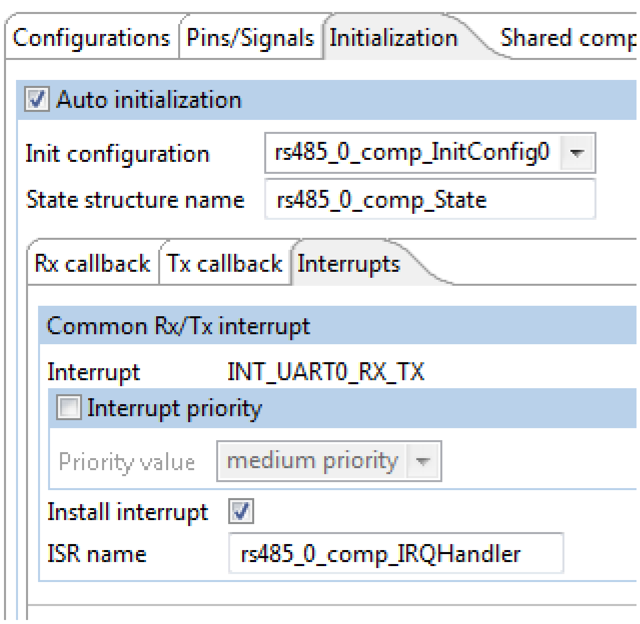
First, obviously you have to have a fsl_uart component added. Here are my settings:


Properties
Methods
Events
Component name
rs485_0_comp
Device
UART0
Component version
1.2.0
Component mode
Interrupt mode
Configurations
Pins/Signals
Initialization
Shared components
Inherited components
Configurations
UART configurations
Configurations list
1
#
Configuration
Name
Type
Read only configuration
Baud rate
Parity mode
Stop bits
Bits per char
0
rs485_0_comp_InitConfig0
uart_user_config_t
115200 baud
Disabled
1
8


Configurations
Pins/Signals
Initialization
Receiver
RxD
PTB16/SPI1_SOUT/UART0_RX/FTM_CLKIN0/EWM_
Transmitter
TxD
PTB17/SPI1_SIN/UART0_TX/FTM_CLKIN1/EWM_O


Configurations
Pins/Signals
Initialization
Shared components
Auto initialization
Init configuration
rs485_0_comp_InitConfig0
State structure name
rs485_0_comp_State
Rx callback
Tx callback
Interrupts
Rx callback
Name
rs485_0_comp_RxCallback
User parameter
Name of user parameter
External declaration of user parameter
Rx Buffer
Name of Rx buffer
rx_callback_buff
External declaration of Rx buffer
extern uint8_t rx_callback_buff[256]
Always enable Rx interrupt

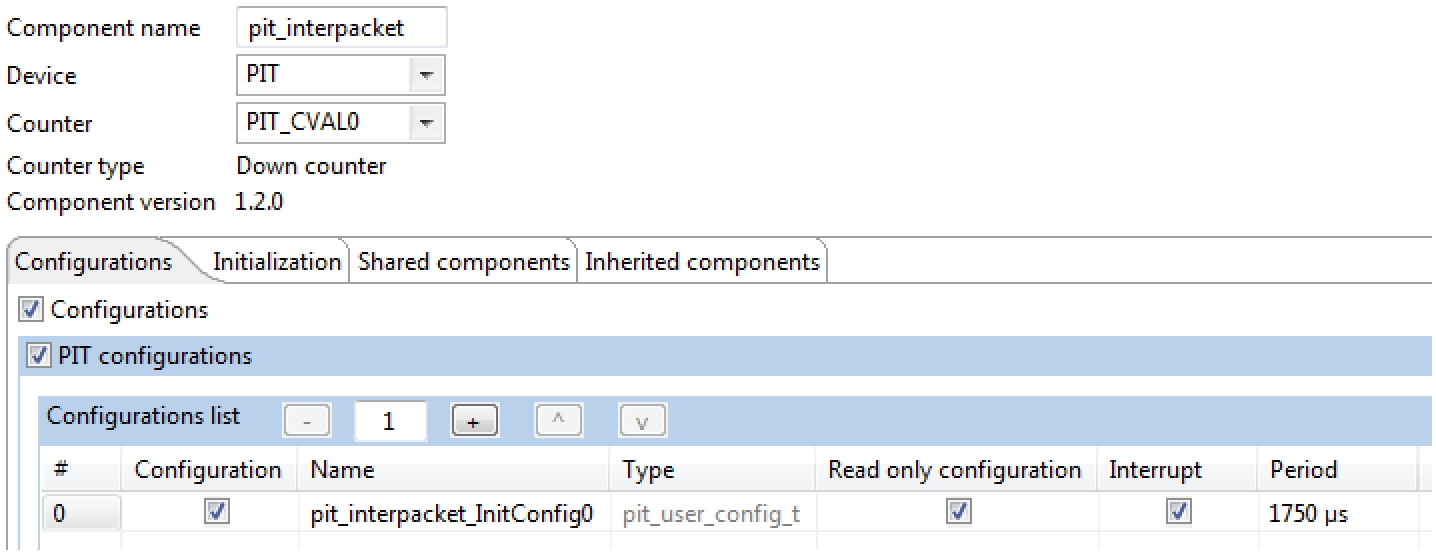


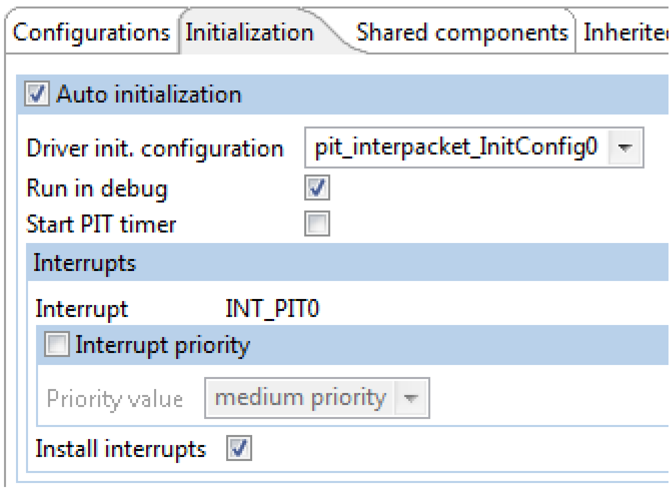


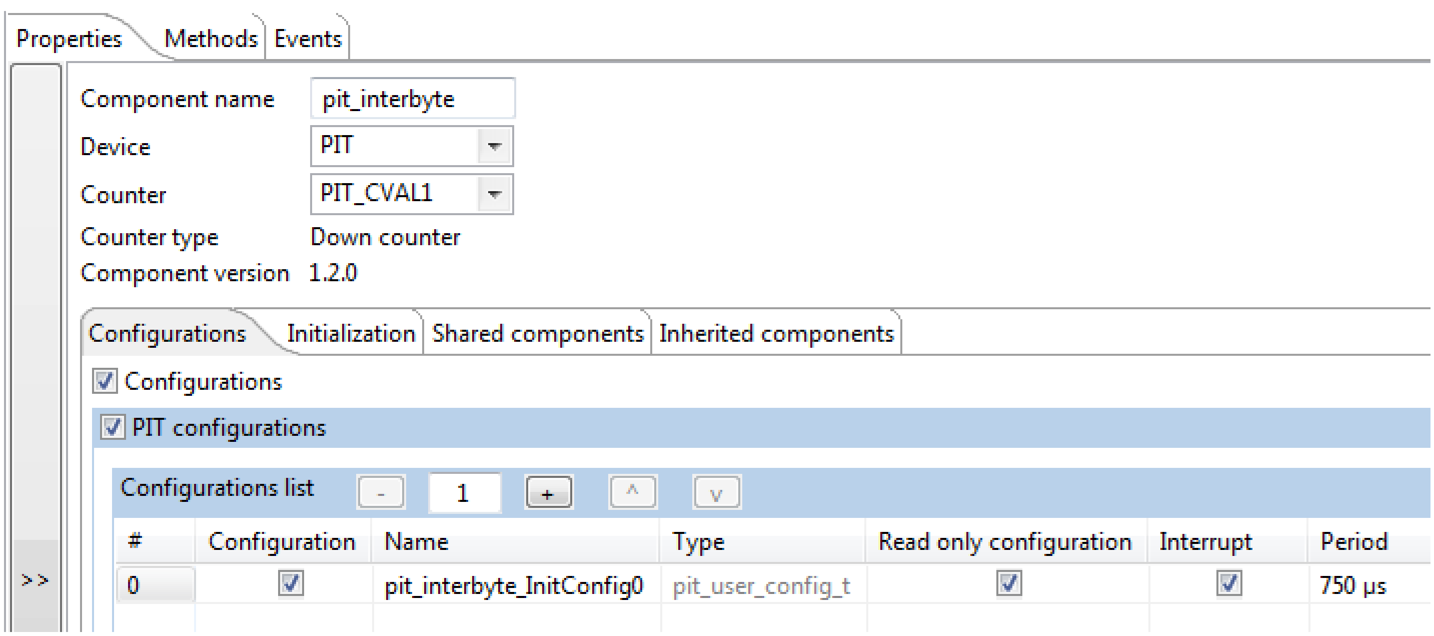
Leave everything else at their default values.

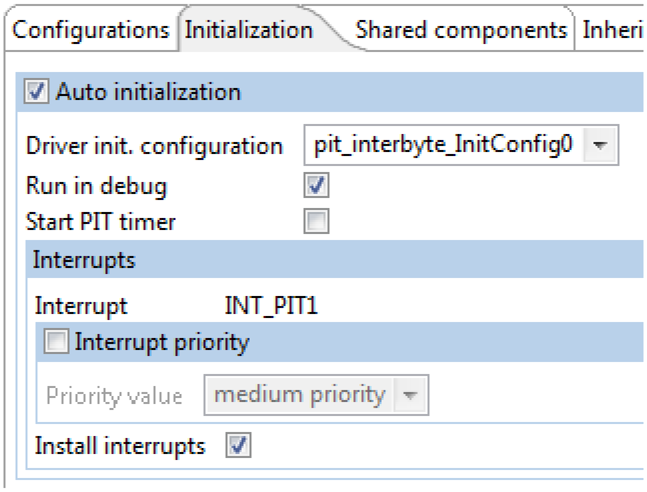
With this configuration, every time a byte comes in, the ISR rs485_0_comp_IRQHandler will get called. This in turn will call your RX handler rs485_0_comp_RxCallback. You have to have the Rx buffer defined, or the callback function will not get called.

Next, add 2 PITs. I called mine pit_interpacket and pit_interbyte.


Component name
pit_interpacket
Device
PIT
Counter
PIT_CVAL0
Counter type
Down counter
Component version
1.2.0
Configurations
Initialization
Shared components
Inherited components
Configurations
PIT configurations
Configurations list
1
#
Configuration
Name
Type
Read only configuration
Interrupt
Period
0
pit_interpacket_InitConfig0
pit_user_config_t
1750 µs


Configurations
Initialization
Shared components
Inherite
Auto initialization
Driver init. configuration
pit_interpacket_InitConfig0
Run in debug
Start PIT timer
Interrupts
Interrupt
INT_PIT0
Interrupt priority
Priority value
medium priority
Install interrupts


Properties
Methods
Events
Component name
pit_interbyte
Device
PIT
Counter
PIT_CVAL1
Counter type
Down counter
Component version
1.2.0
Configurations
Initialization
Shared components
Inherited components
Configurations
PIT configurations
Configurations list
1
#
Configuration
Name
Type
Read only configuration
Interrupt
Period
0
pit_interbyte_InitConfig0
pit_user_config_t
750 µs
>>



It is **critical** that you uncheck the "Start PIT timer" checkboxes in both PITs. The reason for this is that these are periodic timers. They would otherwise start counting immediately, and you'll probably never be able to assemble a full packet.

One more thing, I also manually configured my RX FIFO to be 1 byte deep since the ISR is fast enough to keep up, and I'm a little nervous about some FIFO things I've seen in the past. I'd rather have an occasional missed packet than to be stuck in an endless loop where I can never assemble one.

The ISRs will deal with the starting and stopping of each timer. The overall idea is very simple -- when a byte is received (rx callback is called), append the byte to the packet and then reset (stop, then start) the intercharacter timer. In the intercharacter ISR, start the interpacket timer and set the packet state to CharacterTerminating, which is a flag that says if another byte arrives, it is invalid. If the rx callback gets called again and this flag is set, then mark the packet as FrameNotOk and stop the intercharacter timer. In the interpacket ISR, stop the intercharacter timer and the interpacket timer. If the packet is marked FrameNotOk, then clear the buffer index, reset the packet state to Idle, and flush the UART. Otherwise, the packet is marked FrameOk and is copied to the message queue.

Here is the code:

```c
void UtaskerModbusImpl( uint32_t instance, void * uartState)
{
    uart_state_t* uart = (uart_state_t*)uartState;

    if( g_modbus_state == CharacterTerminating) {
        g_modbus_state = FrameNotOk;
        PIT_HAL_StopTimer(g_pitBase[FSL_PIT_INTERBYTE], FSL_PIT_INTERBYTE_CHANNEL);
        return;
    }
    g_rxbuff[index++] = uart->rxBuff[0];
    // start the timer over for T1.5 since we received a character OK
    PIT_HAL_StopTimer(g_pitBase[FSL_PIT_INTERBYTE], FSL_PIT_INTERBYTE_CHANNEL);
    PIT_HAL_StopTimer(g_pitBase[FSL_PIT_INTERPACKET], FSL_PIT_INTERPACKET_CHANNEL);
    PIT_HAL_StartTimer(g_pitBase[FSL_PIT_INTERBYTE], FSL_PIT_INTERBYTE_CHANNEL);
}

void rs485_0_comp_RxCallback(uint32_t instance, void * uartState)
{
  /* Write your code here ... */
    //SlightlyOlderModbusImpl( instance, uartState);
    UtaskerModbusImpl( instance, uartState);
}

void pit_interpacket_IRQHandler(void)
{
  /* Clear interrupt flag.*/
  PIT_HAL_ClearIntFlag(g_pitBase[FSL_PIT_INTERPACKET], FSL_PIT_INTERPACKET_CHANNEL);
  /* Write your code here ... */
  // stop all timers
  PIT_HAL_StopTimer(g_pitBase[FSL_PIT_INTERBYTE], FSL_PIT_INTERBYTE_CHANNEL);
  PIT_HAL_StopTimer(g_pitBase[FSL_PIT_INTERPACKET], FSL_PIT_INTERPACKET_CHANNEL);

  if( g_modbus_state != FrameNotOk) {
      g_modbus_state = FrameOk;
      // copy message to queue
      _lwmsgq_send( (void*)packet_queue, (_mqx_max_type_ptr)g_rxbuff, 0);//LWMSGQ_SEND_BLOCK_ON_FULL);
  }

  // get ready to receive a new packet after success or even failure
  index = 0;
  g_modbus_state = Idle;
  FlushUartRx( FSL_RS485_0_COMP);
}

void pit_interbyte_IRQHandler(void)
{
  /* Clear interrupt flag.*/
  PIT_HAL_ClearIntFlag(g_pitBase[FSL_PIT_INTERBYTE], FSL_PIT_INTERBYTE_CHANNEL);
  /* Write your code here ... */
  g_modbus_state = CharacterTerminating;
  PIT_HAL_StartTimer(g_pitBase[FSL_PIT_INTERPACKET], FSL_PIT_INTERPACKET_CHANNEL);
}
```

Now, on to the message queue, which is how the interpacket ISR is going to share the packet with your MQX task. The sending code is shown above, but first you have to initialize the message queue. I did that in my MQX task that is going to process the Modbus packet.

```
//----------------------------------------------------------------------------
void ModbusRtuCommandDispatcher::Listen()
{
    _mqx_uint msg[64]; // max number of messages we'll support in message queue is 1 command packet
    _mqx_uint result;

    // disable RX FIFO so ISR handles only one byte at a time
    DisableFifo( _rs485_instance);
    OSA_TimeDelay( 1000); // this is due to a possible bug in KSDK / MQX found by David Seymour
    FlushUartRx( _rs485_instance);
    result = _lwmsgq_init( (void*)packet_queue, 1 /* number of message */, 64 /* message size */);

    if( MQX_OK != result) {
        assert( !"what should we do about this?");
    }

    while (1) {
        OSA_TimeDelay( 10);
        // get packet from message queue
        _lwmsgq_receive( (void*)packet_queue, msg, LWMSGQ_RECEIVE_BLOCK_ON_EMPTY, 0, 0);

        // make it easier to deal with bytes of data instead of uint32_t
        uint8_t *rxbuff = (uint8_t*)msg;
        //**********************************************************************
```

and packet queue is just this:

```
#define MAX_MODBUS_BUFFER              256
#define MODBUS_SEND_TIMEOUT            100
#define MODBUS_RECEIVE_WAITFOREVER     -1
#define MODBUS_MESSAGE_SIZE            (MAX_MODBUS_BUFFER / sizeof(_mqx_))
uint32_t packet_queue[sizeof(LWMSGQ_STRUCT)/sizeof(uint32_t) + 1 * (MAX_MODBUS_BUFFER / sizeof(_mqx_uint))];
uint8_t rx_callback_buff[256]; // required for RX callback function to work (or ISR will crash)
```

This should be just about all you need to make it work. Please let me know if I have missed anything, or if this doesn't work for you. Maybe I'll be able to help you figure it out.