Implementing Interrupt driven Stream UART Rx Handling with STM32CubeMX drivers

Kent Swan, White Bird Engineering, Inc.
hkswan@gmail.com
Jan 9, 2018

Objective:

In this case our firmware architecture requires that we be able to listen to an unbroken asynchronous data stream from a GPS receiver. Since this is a low power application we want to use the UART's interrupt to quickly buffer received characters from a specific UART channel without compromising sleep mode. Within a callback from the interrupt handler we also want to detect complete but raw GPS sentences. On end of line we want to burst transfer the received sentence into a queue or transfer buffer then signal the GPS application thread that a new line of GPS data is ready to be processed.

The Current ST Driver Problem:

The major problem with the current ST UART drivers will only work in a block mode and thus cannot be effectively used in a streamed application where the next character IMMEDIATELY follows the current character with no extra stop bit delay. This failure is caused by the fact that the current UART drivers completely shut down the specified UART Rx functionality on receiving the required number of characters.

You think then why not set to receive a buffer of 1 character at a time and use the `HAL_UART_TxCpltCallback() routine` . While this will work for a keyboard application where individual characters are received with lots of idle stop time between them it fails miserably when a generated stream of characters is received nose to tail. This is because, when you reinvoke the UART in receive mode, all of the state and buffering information is cleared which means that all unprocessed characters in the receive hardware buffers will be lost AND/OR any partially received characters are likely to start on false start bits causing framing errors and loss of stream sync.

The Development Environment

In this case we are developing a bare metal product using a STM32L462. The development environment is the Eclipse based STM32 Workbench which installs a GCC arm toolchain. STM32CubeMX is installed as a plugin to Eclipse. Programming and Debugging is provided through a STLink V2 pod using the Serial Wire interface. All drivers and configuration is provided through STM32CubeMX. Yours will likely vary depending on your IDE.

A HAL Solution

In this tutorial we are going to tweak the HAL drivers in a fully consistent manner to allow us to substitute a new streaming UART_RxISR_8BIT handler that will properly handle streaming input on the specific UART channel without affecting other active HAL UART channels.

Assumptions Part 1

1. You have a project in process.
2. You are using STM32CubeMx to generate your startup and base device drivers
3. In STM32CubeMx
    a. On the Tab “Pinout”you’ve selected UARTx into your project
    b. On the Tab “Clock Configuration” you have a HCLK frequency sufficient to divide down to your required async baud rate. (I use 72Mhz for 115200 and because I want my interrupts and processes to use minimum time to get back to sleep quickly.
    c. On the Tab “Configuration->UARTx-> Parameter Settings” you have configured
        i. Baudrate = 9600 to 115200
        ii. Word Length = 8 bits
    d. On the Tab “Configuration->Uartx->NVIC Settings” you have checked Enabled.
    e. On the Tab “Configuration->NVIC” you’ve set the UARTx priority level to less than whatever you are using for systick.
4. You have used “Project->Generate Code” to regenerate and add the new driver code to your project. When the success dialog panel shows select “Open Project” then select the “C/C++” perspective to get back to your code.

Assumptions Part 2

If you look at your application tree and find the Hal Drivers we are going to tweak. The STM directory coding will change depending upon which processor you generated for and possibly which development environment so you may have to poke about. Mine were located at:

```
<project_name>\Drivers\STM32L4xx_HAL_Driver\Src\stm32L4xx_hal_uart.c
<project_name>\Drivers\STM32L4xx_HAL_Driver\Inc\stm32L4xx_hal_uart.h
```

Tweak stm32L4xx_hal_uart.h

Select the above then find and add the following lines and prototypes just below the **HAL_UART_AbortReceiveCpltCallback().**

```
//      __weak define in source to allow override RxISR
void UART_RxISR_8BIT_INDIRECT(UART_HandleTypeDef *huart);
void UART_RxISR_8BIT(UART_HandleTypeDef *huart);
```

The first is a reference to new __weak function that we will add which allows us to extend the HAL_UART_IRQHandler() with the streaming functionality. The second exposes the current block mode RxISR handler so that we our mod preserves the block mode functionality for all other UARTS.

Tweak stm32L4xx_hal_uart.c

1. Comment out the static definition to match the following. This prevents conflict with the changes we made in the include matching file.

```
//static void UART_RxISR_8BIT(UART_HandleTypeDef *huart);
```

2. In **HAL_UART_Receive_IT**(..) change the following lines so that we can add the ability to install the streaming extension.

```
huart->RxISR = UART_RxISR_8BIT;
```
to
```
huart->RxISR = UART_RxISR_8BIT_INDIRECT;
```

3. On **UART_RxISR_8BIT**(..) make the entry point public by removing the static attribute

```
void UART_RxISR_8BIT(UART_HandleTypeDef *huart)
```

4. Just after UART_RxISR_8BIT(..) add the following function. This function allows us to extend the HAL_UART_IRQHandler() with the streaming functionality by implementing the _weak override.

```
__weak void UART_RxISR_8BIT_INDIRECT(UART_HandleTypeDef *huart)
        {
        UART_RxISR_8BIT(huart);
        }
```

How Stable is this Tweak

At this point we have modified the ST HAL UART drivers in a way that is transparent to their current usage as block drivers. This works but how stable are these modifications. This is in your project directory so the answer is that if you add any more UARTS or change their fundamental configurations using STM32CubeMX will code regeneration will likely revert the drivers to the reference level by design. At that point you'll have to add the mods back in again… not good. There is another solution though so read on.

A More Stable Tweak

Well that's annoying. How can we install the tweak and still have full access and operation STM32CubeMx. Poking around shows us that the reference files that STM32CubeMx uses are in multiple directories on a per processor type basis. Editing these reference files will be okay as we don't intend to change processors anytime soon. This means we can make the same mods in these STM32CubeMx reference files for our processor so that the tweaks will be stable under any STM32CubeMx code regeneration… that is until, ST decides to update the libraries. Since it's not too difficult to add the tweaks later versions that's acceptable.

First the reference files can be found in a subdirectory off of the base directory.

```
C:/Users/<install_name>/STM32Cube/Repository/STM32Cube_FW_L4_V1.10.0/Drivers/
```

and, for our STM32L4 processor in the sub directories as shown

```
..\STM32L4xx_HAL_Driver\Src\stm32l4xx_hal_usart.c
..\STM32L4xx_HAL_Driver\Inc\stm32l4xx_hal_usart.h
```

Note *that the most active version number is different for different processors and will change as ST updates the libraries but each library downloaded and active will remain invariant even if tweaked so at least within the version it is stable. This is good.*

Application Coding – The Intercept

First let's setup our own code to intercept extend the UART4 interrupt handling so that it will do streaming mode. This is pretty simple as all we have to do is provide an alternate **UART_RxISR_8BIT_INDIRECT**() function we just installed, snaffle off UART4 for stream processing and let the the UART IRS handle the others normally like this:

```
void UART_RxISR_8BIT_INDIRECT(UART_HandleTypeDef *huart)
    {
    if(huart->Instance == UART4) UART_RxISR_8BIT_STREAM(huart);
        // else breakout other stream uarts
    else UART_RxISR_8BIT(huart); // use default handler
    }
```

Application Coding – The Stream Interrupt Handler

Now we have to provide a uart independant stream interrupt handler to provide for the common base handling for ANY uart which needs to display stream Rx behavior. A model for the stream ISR is shown below. The callback is for the application customization on a per stream uart basis.

- **UART_RxISR_8BIT_STREAM** is called for each character received which is stored in the huart->pRxBuffer. The variable huart->RxBufferCount keeps track of how many characters are in the buffer.
- U**ART_RxISR_8BIT_STREAM_ReadyCallback** is called for each character in the interrupt context for each character allowing the user to handle or examine the received characters stored in the receive buffer.
- The expectation is that when **UART_RxISR_8BIT_STREAM_ReadyCallback** decides to accept the characters in the buffer it processes them out then resets huart->RxXferCount=0
- Note that if an error occurs within **UART_RxISR_8BIT_STREAM_ReadyCallback** it simply sets RxState = HAL_UART_STATE_ERROR which terminates the receive session.

```
__weak void UART_RxISR_8BIT_STREAM_ReadyCallback(UART_HandleTypeDef *huart)
    {
    // Safety code for default if not overridden.
    If(huart->RxXferCount >= huart->RxXferSize) huart->RxXferCount=0;
    }

void UART_RxISR_8BIT_STREAM(UART_HandleTypeDef *huart)
    {
    uint16_t uhMask = huart->Mask;
    uint16_t  uhdata;

    /* Check that a Rx process is ongoing */
    if (huart->RxState == HAL_UART_STATE_BUSY_RX)
        {
        uhdata = (uint16_t) READ_REG(huart->Instance->RDR);
        huart->pRxBuffPtr[huart->RxXferCount++] = (uint8_t)(uhdata & (uint8_t)uhMask);

        UART_RxISR_8BIT_STREAM_ReadyCallback(huart);

        if (huart->RxState == HAL_UART_STATE_ERROR)
            {
            // The following code duplicates UART_EndRxTransfer(UART_HandleTypeDef *huart)
            // Terminating the current UART Interrupt Receive session.
            /* Disable the UART Parity Error Interrupt and RXNE interrupt*/
#if defined(USART_CR1_FIFOEN)
            CLEAR_BIT(huart->Instance->CR1, (USART_CR1_RXNEIE_RXFNEIE | USART_CR1_PEIE));
#else
            CLEAR_BIT(huart->Instance->CR1, (USART_CR1_RXNEIE | USART_CR1_PEIE));
#endif
            /* Disable the UART Error Interrupt: (Frame error, noise error, overrun error) */
            CLEAR_BIT(huart->Instance->CR3, USART_CR3_EIE);
            /* Rx process is completed, restore huart->RxState to Ready */
            huart->RxState = HAL_UART_STATE_READY;
            /* Clear RxISR function pointer */
            huart->RxISR = NULL;
            }
        }
    else
        {
        /* Clear RXNE interrupt flag */
        __HAL_UART_SEND_REQ(huart, UART_RXDATA_FLUSH_REQUEST);
        }
    }
```

Application Coding – The Stream UART dispatch Callback

Since all UART's are routed through a single ISR, we need to break out the specific UART to provide the custom handling of that UART's data. We instantiate UART_RxISR_8BIT_STREAM_ReadyCallback handler to (a) detect the UARTs associated with stream handling then call the custom code handler associated with that UART.

```
void UART_RxISR_8BIT_STREAM_ReadyCallback(UART_HandleTypeDef *huart)
    {
    if(huart->Instance == UART4) APP_UART4_ReadyCallback(huart);
    // breakout other stream uarts ignoring any we don't want to handle
    }
```

Application Coding – The Stream Application Handler Callback

Now we need to provide the ability to custom handle the stream like we want. Remember we are still in the HAL_UART_ISR_Handler context. Per out specific requirement we need to detect whether a full line has been read into the input buffer. For GPS the first character is a '$' and the last is a LF code. In this case we are using a FreeRTOS task thread to process the received data and a ring buffer to transfer it from the ISR to the GPS thread.

```
// GPS Stream Rx Interrupt Handler for UART4
void APP_UART4_RxReadyCallback(UART_HandleTypeDef *huart)
    {
    if(huart->pRxBuffPtr[0]=='$')
        {
        if(huart->pRxBuffPtr[huart->RxXferCount-1] == '\n')
            {
            if(xRingbuf_PutArray(&UART4_RxRB, (huart->pRxBuffPtr),huart->RxXferCount))
                {
                // Tell the GPS RX Thread that there's a line of GPS Data Ready
                osMessagePut(GPSSerialRxQueueHandle, GPS_DATA_READY, 1000);
                }
            else
                {
                // Solid RED Ring buffer returned false meaning buffer is full error
                huart->RxState = HAL_UART_STATE_ERROR;
                }
            huart->RxXferCount = 0;
            }
        }
    else
        {
        huart->RxXferCount = 0;
        }
    }
```