

no, I am now accepting default 40MHz, and want to run USART this way, what can be missing? is USART interrupt initialized?



To run USART at the default 40MHz, you need to ensure that the following steps are taken:

1. Clock configuration: Make sure that the clock source and frequency are correctly configured for your microcontroller. In this case, it should be 40 MHz.
2. USART configuration: Configure the USART peripheral according to the desired parameters (baud rate, data bits, stop bits, parity, and flow control). To calculate the baud rate, you can use the following formula:
$$\text{Baud rate} = (\text{USART Clock Frequency}) / (\text{USARTDIV})$$
3. USART interrupt initialization: If you want to use USART with interrupt, you need to enable the appropriate interrupt in the NVIC (Nested Vectored Interrupt Controller) and configure the USART interrupt priority.
4. USART interrupt handler: Implement the USART interrupt handler function in your code. This function will be called when the USART receives data or other events happen, such as transmission complete, error, etc.
5. Enable USART and USART interrupt: Finally, enable the USART peripheral and the specific interrupt you want to use (e.g., RXNE for received data, TXE for data transmission, etc.).

Here's a simple example for STM32 microcontrollers using HAL libraries:

c

Copy code

```
// USART Initialization
UART_HandleTypeDef huart; huart.Instance = USART1;
huart.Init.BaudRate = 115200; huart.Init.WordLength = UART_WORDLENGTH_8B;
huart.Init.StopBits = UART_STOPBITS_1; huart.Init.Parity = UART_PARITY_NONE;
huart.Init.Mode = UART_MODE_TX_RX; huart.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart.Init.OverSampling = UART_OVERSAMPLING_16; if (HAL_UART_Init(&huart) !=
HAL_OK) { // Initialization Error
```



no, I have now this HAL-free code only ...

```
#undef USE_HAL_DRIVER
```

```
#include "stm32l4xx.h"
```

```
#include "core_cm4.h"
```

```
volatile uint32_t millis = 0;
```

```
void SysTick_Handler(void)
```

```
{  
    millis++;  
}
```

```
void delay(uint32_t ms)
```

```
{  
    uint32_t start = millis;  
    while ((millis - start) < ms);  
}
```

```
int main(void)
```

```
{
```

```
    SystemCoreClock = 0;
```

```
    SystemCoreClockUpdate(); // Update SystemCoreClock variable
```

```
    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;
```

```
    RCC->APB2ENR |= RCC_APB2ENR_USART1EN;
```

```
    GPIOB->MODER &= ~GPIO_MODER_MODE13_Msk;
```

```
    GPIOB->MODER |= GPIO_MODER_MODE13_0;
```

```
    GPIOA->MODER &= ~(GPIO_MODER_MODE9_Msk | GPIO_MODER_MODE10_Msk);
```

```
    GPIOA->MODER |= (GPIO_MODER_MODE9_1 | GPIO_MODER_MODE10_1);
```

```
    GPIOA->AFR[1] |= (7 << GPIO_AFRH_AFSEL9_Pos) | (7 << GPIO_AFRH_AFSEL10_Pos);
```

```
    // Calculate the BRR value using FBR
```

```
    float baud_rate = 9600.0;
```

```
    float clock_freq = SystemCoreClock;
```

```
    float mantissa = clock_freq / (16.0 * baud_rate);
```

```

float fraction = mantissa - (uint32_t)mantissa;
uint32_t mantissa_int = (uint32_t)mantissa;
uint32_t fraction_int = (uint32_t)(fraction * 16.0 + 0.5); // Round fraction to nearest
integer
USART1->BRR = (mantissa_int << 4) | fraction_int;

//USART1->BRR = (520 << 4) | 5; // Set the baud rate to 9600
//USART1->BRR = (SystemCoreClock / 9600);

USART1->CR1 |= USART_CR1_TE | USART_CR1_RE;
USART1->CR1 |= USART_CR1_UE;

SysTick->LOAD = (SystemCoreClock / 1000) - 1;
SysTick->VAL = 0;
SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_TICKINT_Msk |
SysTick_CTRL_ENABLE_Msk;

__enable_irq();

while (1)
{
    GPIOB->ODR ^= GPIO_ODR_OD13;
    delay(1000);

    uint8_t adataToSend[] = "Hello, USART1!\n";
    for (int i = 0; i < sizeof(adataToSend); i++)
    {
        while (!(USART1->ISR & USART_ISR_TXE));
        USART1->TDR = adataToSend[i];
    }
}
}

```



Your code configures USART1 for transmission without using interrupts. If you want to use USART interrupts for transmission or reception, you should make the following changes to your code:

1. Enable the USART1 interrupt in the NVIC:

c

 Copy code

```
NVIC_EnableIRQ(USART1_IRQn); NVIC_SetPriority(USART1_IRQn, 0); // Set priority as needed
```

2. Enable the TXEIE (Transmit Data Register Empty Interrupt) or RXNEIE (Receive Data Register Not Empty Interrupt) in the USART1 control register:

c

 Copy code

```
USART1->CR1 |= USART_CR1_TXEIE; // Enable TXE interrupt // USART1->CR1 |= USART_CR1_RXNEIE; // Enable RXNE interrupt
```

3. Implement the USART1 interrupt handler function:

c

 Copy code

```
void USART1_IRQHandler(void) { if (USART1->ISR & USART_ISR_TXE) // Check if TXE flag is set { // Transmit data here // ... USART1->ICR |= USART_ICR_TXECF; // Clear the TXE flag } else if (USART1->ISR & USART_ISR_RXNE) // Check if RXNE flag is set { // Receive data here // ... USART1->ICR |= USART_ICR_RXNECF; // Clear the RXNE flag } }
```

4. In your main loop, change the transmission code to utilize the interrupt:

c

 Copy code

```
while (1) { GPIOB->ODR ^= GPIO_ODR_OD13; delay(1000); uint8_t adataToSend[] = "Hello, USART1!\n"; for (int i = 0; i < sizeof(adataToSend); i++) { while (!(USART1->ISR & USART_ISR_TXE)); // USART1->TDR = adataToSend[i]; // Comment out this line // Add the data to be sent to a buffer // Set the TXEIE flag in USART1->CR1 to trigger the USART1_IRQHandler } }
```

Note that in the example above, you need to implement a buffer to store the data to be sent and then send it through the USART1_IRQHandler function when the TXE interrupt is triggered. You can use a simple array, a circular buffer, or any other data structure suitable for your needs.



okay, so this code WITHOUT interrupts would still work, ya? it is checking some status in USART_ISR, probably if byte was sent ... that's okay, but it is NOT sending anything



Yes, your original code should work without interrupts, as it checks the TXE flag (Transmit Data Register Empty) in the USART1 status register (ISR) before sending each byte. This flag indicates that the transmit data register is empty and ready to send another byte.

However, I noticed one issue in your code that could be causing the problem: the size of the `adataToSend` array is not calculated correctly. You should subtract one from the `sizeof(adataToSend)` to avoid sending the null terminator character (`\0`) at the end of the string:

c

Copy code

```
uint8_t adataToSend[] = "Hello, USART1!\n"; for (int i = 0; i <
sizeof(adataToSend) - 1; i++) // Subtract 1 to exclude the null terminator {
while (!(USART1->ISR & USART_ISR_TXE)); USART1->TDR = adataToSend[i]; }
```

If you still can't see any data being transmitted, double-check the hardware connections and make sure you are using the correct pins for USART1 on your specific microcontroller. In your code, you are using PA9 (TX) and PA10 (RX) for USART1, which are the default pins for USART1 in many STM32 devices. However, if your device has a different pin configuration, you might need to update the code accordingly.

Also, ensure that your USART receiver (e.g., a serial-to-USB converter or another microcontroller) is set to the correct baud rate and settings to properly receive the data.



it is strange, I corrected this error and was seeing that it IS now sending ok ... but now after rebuilding it is still not sending can't understand



It's puzzling that the code stopped working after rebuilding. Here are a few suggestions to help you diagnose the issue:

1. Clean and rebuild: Sometimes, build artifacts or cached files can cause problems. Try cleaning your project and rebuilding it from scratch.
2. Check optimization settings: Some compilers may aggressively optimize code, which could result in unexpected behavior. If your project uses any optimization flags, try disabling them and rebuilding your code.
3. Confirm the hardware setup: Make sure your hardware connections are stable and correct. If you have access to an oscilloscope or logic analyzer, use it to verify the signals on the TX and RX pins.
4. Debugging: Use a debugger to step through the code and monitor the values of variables and registers. This can help you identify any discrepancies between your expectations and the actual behavior of the code.
5. Reset the microcontroller: If your microcontroller has any reset or power-on issues, it may not execute the code as expected. Try resetting the microcontroller or power-cycling it to see if the issue persists.
6. Ensure the correct clock configuration: Double-check the clock configuration for both the microcontroller and the USART peripheral. Make sure that the `SystemCoreClock` variable is set correctly and that the peripheral clock is enabled.
7. Check for potential conflicts: Review your code for any potential conflicts, such as other peripherals or interrupts that may be affecting USART1.

If you've tried all the suggestions above and still cannot get the code to work consistently, it might be helpful to create a minimal example that only initializes and uses the USART peripheral without any other peripherals or features. This can help isolate the issue and eliminate potential interference from other parts of your code.

Additionally, you may want to consult the reference manual and datasheet for your specific STM32 microcontroller to ensure you are configuring and using USART1 correctly.





what is BLE gatt status 62 ?



The BLE GATT (Bluetooth Low Energy Generic Attribute Profile) status codes are specific to the implementation of the Bluetooth stack that you are using. Status code 62 does not have a standardized meaning across all BLE implementations.

To determine the meaning of status code 62, you will need to refer to the documentation or source code of the particular Bluetooth stack you are using (e.g., Nordic's SoftDevice, BlueNRG, or Zephyr). This information will help you understand what the error code represents and how to resolve it.

However, if you provide more context or information about the specific BLE implementation or library you are using, I may be able to give you more guidance on the meaning of GATT status 62.



it is returning android serial bluetooth terminal" while connecting to JDY-33 connected to my MCU UART at 9600



In the context of Android's Bluetooth stack, GATT status 62 (0x3E) is a proprietary error code used by the Android platform to indicate a "Connection Congested" error. This error can occur when the Bluetooth Low Energy connection has reached its maximum throughput capacity, and the device cannot handle more data at that moment.

The "Connection Congested" error is not related to GATT procedures, but rather to the underlying BLE connection between the Android device and the peripheral device (in this case, the JDY-33 module). This error can be caused by various factors, such as radio interference, heavy traffic in the 2.4 GHz ISM band, or a congested connection due to high data rates.

Here are some suggestions to help resolve the issue:

1. Retry the connection: You can implement a retry mechanism in your Android app to automatically reconnect when the connection congested error occurs.
2. Adjust connection parameters: You can try adjusting the connection parameters (e.g., connection interval and slave latency) to increase the throughput of the BLE connection. Note that increasing the connection interval might help in congested environments but could also increase power consumption.
3. Optimize data transfer: Check the data transfer rate between the Android device and the JDY-33 module. If you are sending too much data too frequently, consider reducing the rate or optimizing the data packets.
4. Check for interference: Ensure there are no significant sources of interference nearby, such as Wi-Fi routers, microwaves, or other devices operating in the 2.4 GHz band.
5. Update firmware: Make sure your JDY-33 module and the Android device have the latest firmware and Bluetooth stack implementations. Updating these components can often help resolve connectivity issues.



how is affected in stm32Cube or generally CMSIS development the defined symbol `USE_HAL_DRIVER`? Is possible to switch this off while generating some CMSIS code?



The **`USE_HAL_DRIVER`** symbol is used in STM32 projects to determine whether to include and use the STM32 Hardware Abstraction Layer (HAL) libraries. When this symbol is defined, the project will include the HAL libraries and their functions, which provide an abstraction layer for the STM32 peripherals and hardware.

If you wish to use the lower-level CMSIS libraries instead of HAL, you can do so by removing or commenting out the **`USE_HAL_DRIVER`** definition in your project. This will disable the HAL libraries, and you'll need to use the CMSIS functions and direct register manipulation for hardware configuration and control.

To switch off the **`USE_HAL_DRIVER`** symbol, follow these steps:

1. Find the **`USE_HAL_DRIVER`** definition in your project. It is usually located in the project's preprocessor definitions or in a header file like "stm32xxx.h" or "main.h", where "xxx" refers to the specific STM32 series you are using.

2. Comment out or remove the `#define USE_HAL_DRIVER` line.
3. Modify your code to use CMSIS functions and direct register manipulation instead of HAL functions.
4. Rebuild your project.

Keep in mind that when you disable the `USE_HAL_DRIVER` symbol, you'll need to handle the hardware configuration, initialization, and control manually, using CMSIS functions and registers. This will require a deeper understanding of the STM32 hardware and peripherals.

When generating code with STM32CubeMX or STM32CubeIDE, the default configuration uses HAL libraries. If you want to use CMSIS instead, you'll have to manually modify the generated code to remove the `USE_HAL_DRIVER` definition and replace the HAL functions with CMSIS equivalents.

Please note that this process can be more challenging and time-consuming, especially for complex projects or peripherals.



ummm, yes, I am still unable to end anything over usart1 without hal



I understand that you're having trouble transmitting data over USART1 without using the HAL library. Let's go through the necessary steps to configure and use USART1 with direct register manipulation, assuming you're using an STM32L4 series microcontroller as in your previous code example.

1. Ensure that the GPIOs are configured correctly for USART1. You mentioned using PA9 (TX) and PA10 (RX) for USART1. Make sure the alternate function settings are correct:

c

Copy code

```
// Enable GPIOA and USART1 clocks RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN; RCC->APB2ENR |= RCC_APB2ENR_USART1EN; // Configure PA9 and PA10 as alternate functions for USART1 GPIOA->MODER &= ~(GPIO_MODER_MODE9_Msk | GPIO_MODER_MODE10_Msk); GPIOA->MODER |= (GPIO_MODER_MODE9_1 | GPIO_MODER_MODE10_1); GPIOA->AFR[1] |= (7 << GPIO_AFRH_AFSEL9_Pos) | (7 << GPIO_AFRH_AFSEL10_Pos);
```

2. Configure the USART1 peripheral:

c

 Copy code

```
// Disable USART1 before configuring USART1->CR1 &= ~USART_CR1_UE; // Set the
baud rate uint32_t usartdiv = (SystemCoreClock + (9600 / 2)) / 9600; USART1->BRR
= usartdiv; // Enable transmitter and receiver USART1->CR1 |= USART_CR1_TE |
USART_CR1_RE; // Enable USART1 USART1->CR1 |= USART_CR1_UE;
```

3. Send data over USART1:

c

 Copy code

```
void usart_send(uint8_t data) { // Wait until the transmit data register is
empty while (!(USART1->ISR & USART_ISR_TXE)); // Send the data USART1->TDR =
data; } uint8_t adataToSend[] = "Hello, USART1!\n"; for (int i = 0; i <
sizeof(adataToSend) - 1; i++) { usart_send(adataToSend[i]); }
```

If the code still doesn't work, here are some suggestions to help you diagnose the issue:

Double-check your hardware connections, and ensure you're using the correct pins for USART1 on your specific microcontroller.

Confirm that the clock configuration is correct. Make sure **SystemCoreClock** is set to the correct value and that the peripheral clocks are enabled. Additionally, ensure that the HCLK, PCLK1, and PCLK2 settings are correct in the RCC configuration.

Use a debugger to step through the code and inspect the register values. This can help you identify any discrepancies between your expectations and the actual behavior of the code.

Consult the reference manual and datasheet for your specific STM32 microcontroller to ensure you're configuring and using USART1 correctly.

  



is this ok?

```
GPIOA->MODER &= ~(GPIO_MODER_MODE9_Msk | GPIO_MODER_MODE10_Msk);
```

```
GPIOA->MODER |= (GPIO_MODER_MODE9_1 | GPIO_MODER_MODE10_1);  
GPIOA->AFR[1] |= (7 << GPIO_AFRH_AFSEL9_Pos) | (7 << GPIO_AFRH_AFSEL10_Pos);
```



Yes, this code is correct for configuring PA9 and PA10 as alternate function pins for USART1 in STM32L4 series microcontrollers.

Here's a breakdown of the code:

1. **GPIOA->MODER &= ~(GPIO_MODER_MODE9_Msk | GPIO_MODER_MODE10_Msk);** - This line clears the bits for PA9 and PA10 in the MODER register to set them as input pins temporarily.
2. **GPIOA->MODER |= (GPIO_MODER_MODE9_1 | GPIO_MODER_MODE10_1);** - This line sets the appropriate bits in the MODER register to configure PA9 and PA10 as alternate function pins.
3. **GPIOA->AFR[1] |= (7 << GPIO_AFRH_AFSEL9_Pos) | (7 << GPIO_AFRH_AFSEL10_Pos);** - This line sets the appropriate bits in the AFRH register (AFR[1]) to select the alternate function 7 (USART1) for PA9 and PA10.

Make sure that you have enabled the clocks for both GPIOA and USART1 before this configuration, as shown in the previous code examples:

c

Copy code

```
// Enable GPIOA and USART1 clocks  
RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN; RCC->APB2ENR |= RCC_APB2ENR_USART1EN;
```

With this configuration, PA9 will be the TX pin, and PA10 will be the RX pin for USART1.



which is this form of uart BRR for 40MHz at 9600 ?

```
USART1->BRR = (520 << 4) | 5;
```



The formula for calculating the USART BRR value is as follows:

$$\text{BRR} = (\text{USARTDIV} * 16)$$

where USARTDIV is the division factor that is calculated from the system clock frequency and the desired baud rate.

For a system clock frequency of 40 MHz and a desired baud rate of 9600, you can calculate the USARTDIV as follows:

$$\text{USARTDIV} = 40,000,000 / (9600 * 16) = 260.41667$$

The BRR value is composed of the mantissa and the fraction parts, and it's represented as:

$$\text{BRR} = (\text{Mantissa} < < 4) | \text{Fraction}$$

To calculate the mantissa and fraction parts, you can use the following:

$$\text{Mantissa} = \text{floor}(\text{USARTDIV}) = 260$$

$$\text{Fraction} = \text{round}((\text{USARTDIV} - \text{Mantissa}) * 16) = \text{round}(0.41667 * 16) = 7$$

So, the BRR value for a 40 MHz system clock and a 9600 baud rate would be:

$$\text{USART1} \rightarrow \text{BRR} = (260 < < 4) | 7;$$

The value $(520 < < 4) | 5$ that you've provided seems to be for a system clock of 80 MHz, not 40 MHz.



well, it seems will be far better to let MX generate all including HAL and use this with optimizations finally enabled for now ... this works, probably everywhere easily on stm32



Yes, using STM32CubeMX or STM32CubeIDE to generate the initial configuration and code, including the HAL libraries, is often the most convenient approach for most developers, especially when getting started with STM32 microcontrollers. This method has several advantages:

1. **Abstraction:** The HAL libraries provide an abstraction layer that simplifies hardware configuration and control, allowing developers to focus on their application logic.
2. **Portability:** The HAL libraries are designed to be portable across different STM32 families, which makes it easier to migrate your code to different STM32 microcontrollers.
3. **Time-saving:** Using the graphical interface of STM32CubeMX or STM32CubeIDE, you can quickly configure your microcontroller's peripherals and generate the initialization code automatically. This can save a considerable amount of time compared to manual configuration and register manipulation.
4. **Updates and bug fixes:** The HAL libraries are maintained by STMicroelectronics, so you can benefit from regular updates, bug fixes, and improvements in the libraries.

While using HAL libraries might result in a slightly larger code size and marginally lower performance compared to using CMSIS and direct register manipulation, the trade-off is often acceptable for many applications. Additionally, enabling compiler optimizations can help reduce the impact of these disadvantages.

If you need more fine-grained control over the hardware or want to optimize your code further, you can always switch to CMSIS and direct register manipulation later on in your development process. However, starting with HAL libraries and generated code from STM32CubeMX or STM32CubeIDE is a good choice for most projects.
