DMA

I. Learning Objectives

1. Understand basic DMA knowledge.
2. Learn basic DMA usage, implement variable-length data reception through serial port DMA sending data.

II. DMA Introduction

DMA (Direct Memory Access) is a technology used in computer systems that enables peripherals to directly access memory without requiring CPU involvement for data transfer. It allows efficient data exchange between peripherals (such as ADC, DAC, memory, etc.) and memory, reducing CPU burden and improving overall system performance.

Principle:

- **Trigger data transfer**: DMA operations are typically triggered by external devices (such as ADC, sensors) or internal events (such as timer overflow).
- **Initialize DMA controller**: CPU configures parameters to the DMA controller such as source address, destination address, transfer data size, transfer direction, etc.
- **Direct data transfer**: Once the DMA controller is activated, it can directly transfer data from the source address (peripheral's output buffer or memory) to the destination address (memory or peripheral) without CPU intervention.
- **Notify CPU after transfer completion**: After DMA completes data transfer, it can send an interrupt request to the CPU, telling the CPU that data is ready or the transfer has been successfully completed.

The MSPM0G3507 DMA controller has the following features:

- 7 independent transfer channels;
- Configurable DMA channel priority;
- Supports 8-bit (byte), 16-bit (short word), 32-bit (word) and 64-bit (long word) or mixed size (byte and word) transfers;
- Supports data block transfers of up to 64K arbitrary data types;
- Configurable DMA transfer trigger sources;
- 6 flexible addressing modes;
- Single or block transfer modes; It has a total of 7 DMA channels, each channel can be independently configured, and various data transfer modes can adapt to the data transfer needs of different application scenarios.

By checking TI's data manual, in addition to common address addressing methods between memory and peripherals, the DMA function also provides two extended modes: Fill Mode and Table Mode. DMA channels are divided into two types: basic type and full-featured type.

Table 4-1. Feature Comparison of Basic and Full-Feature DMA Channels

DMA Feature	Full-Feature Channel	Basic Channel
Repeated mode	✓	–
Early IRQ notification	✓	–
Block burst mode	✓	✓
Stride mode	✓	✓
Internal channel as trigger source	✓	✓
Extended Mode (Table and Fill Mode)	✓	–

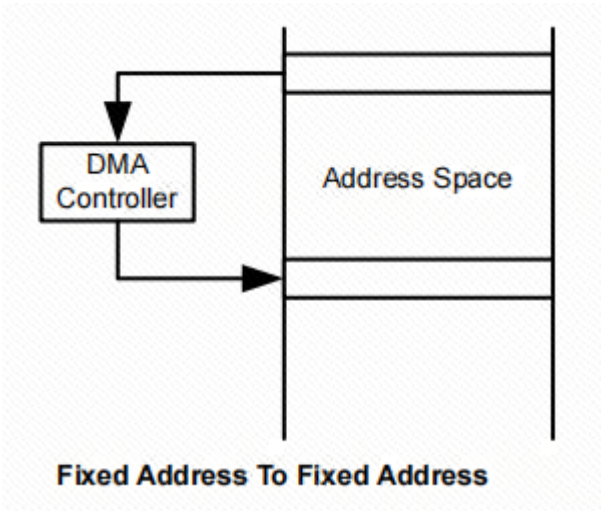
By checking the data manual, only full-featured type DMA channels support repeated transfers, early interrupts, and extended modes. Basic function DMA channels support basic data transfer and interrupts, but are sufficient to meet simple data transfer requirements.

MSPM0G3507's DMA supports four transfer modes, each channel can be individually configured for its transfer mode. For example, channel 0 can be configured for single word or single byte transfer mode, while channel 1 is configured for block transfer mode, and channel 2 uses repeated block transfer mode. Transfer modes are configured independently of addressing modes. Any addressing mode can be used with any transfer mode. Three types of data can be transferred, which can be selected through .srcWidth and .destWidth control bits. Source and destination locations can be byte, short word, or word data. It also supports transmission in byte-to-byte, short word-to-short word, word-to-word, or any combination. As follows:

- Single word or single byte transfer: Each transfer requires a separate trigger. DMA will be automatically disabled when DMAxSZ transfers have been generated.
- Block transfer: An entire block will be transferred after one trigger. DMA will be automatically disabled at the end of block transfer.
- Repeated single word or single byte transfer: Each transfer requires a separate trigger, DMA remains enabled.
- Repeated block transfer: An entire block will be transferred after one trigger, DMA remains enabled.

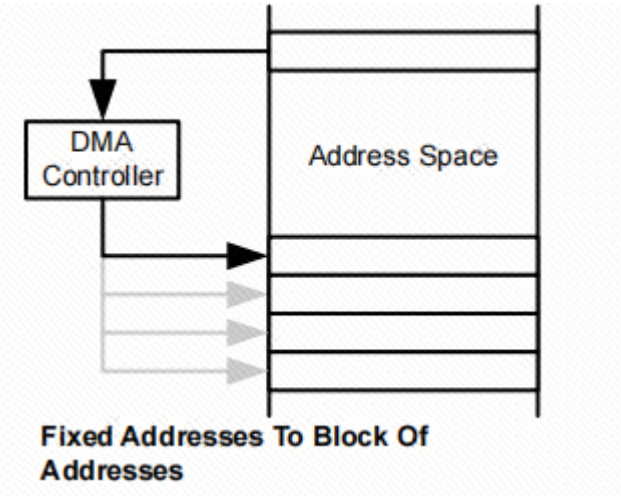
Based on these three modes, MSPM0G3507 provides 6 addressing methods,

Fixed address to fixed address:

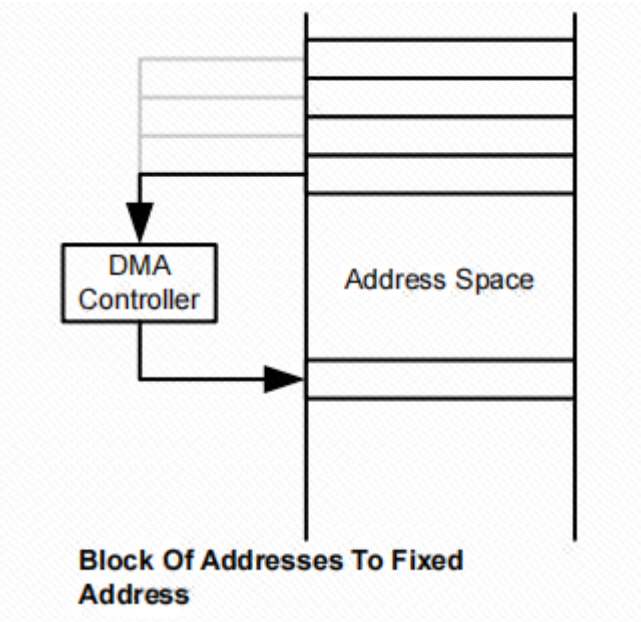


Fixed Address To Fixed Address

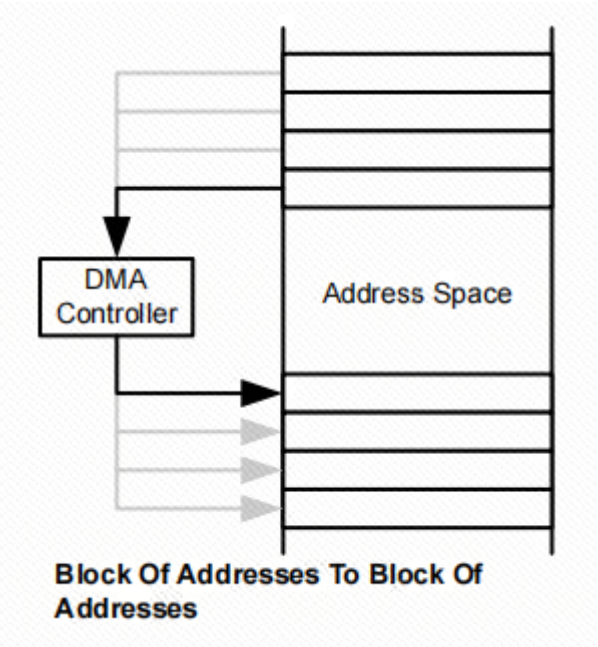
Fixed address to address block:



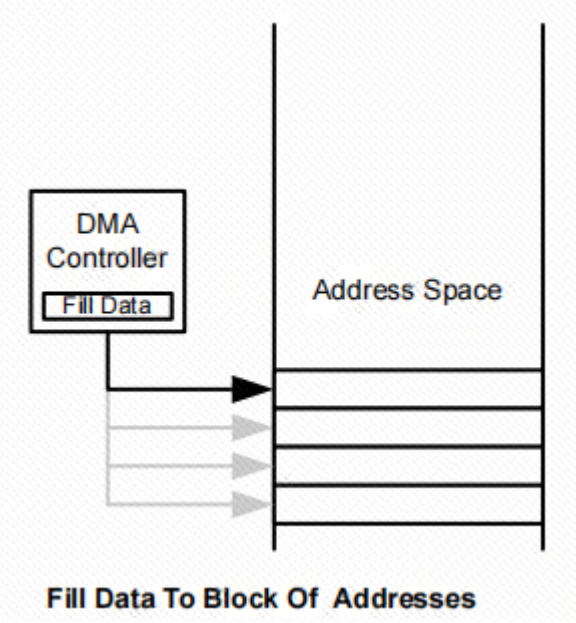
Address block to fixed address:



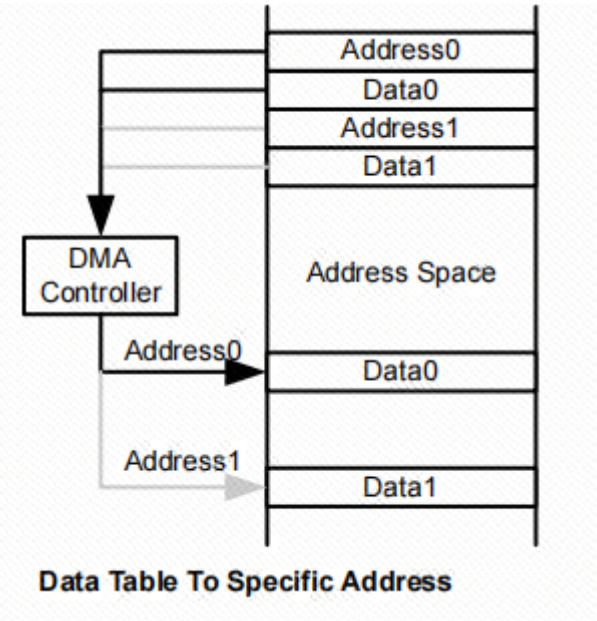
Address block to address block:



Fill data to address block:



Data table to specified address:

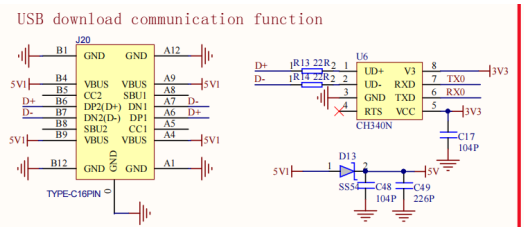


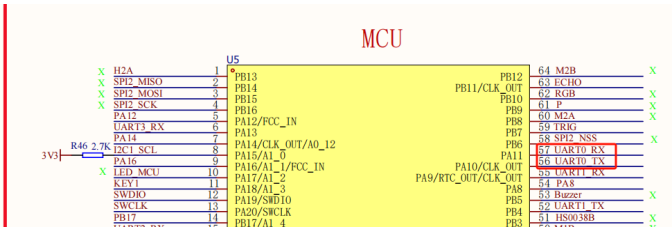
Since the development board has a built-in CH340 serial port circuit, it can be directly implemented through USB cable

III. Hardware Setup

Serial communication, no external USB to TTL module required.

Serial port schematic diagram



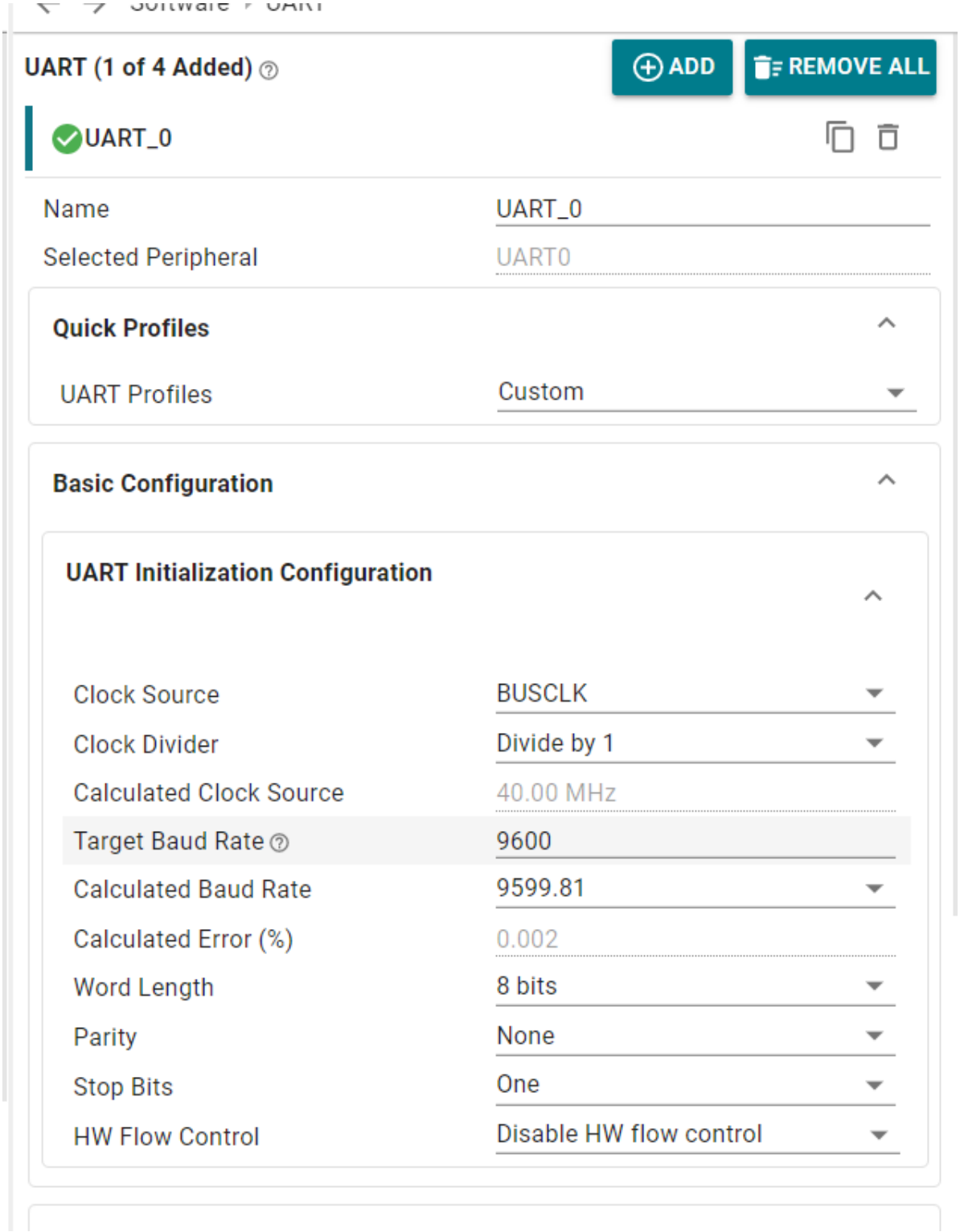


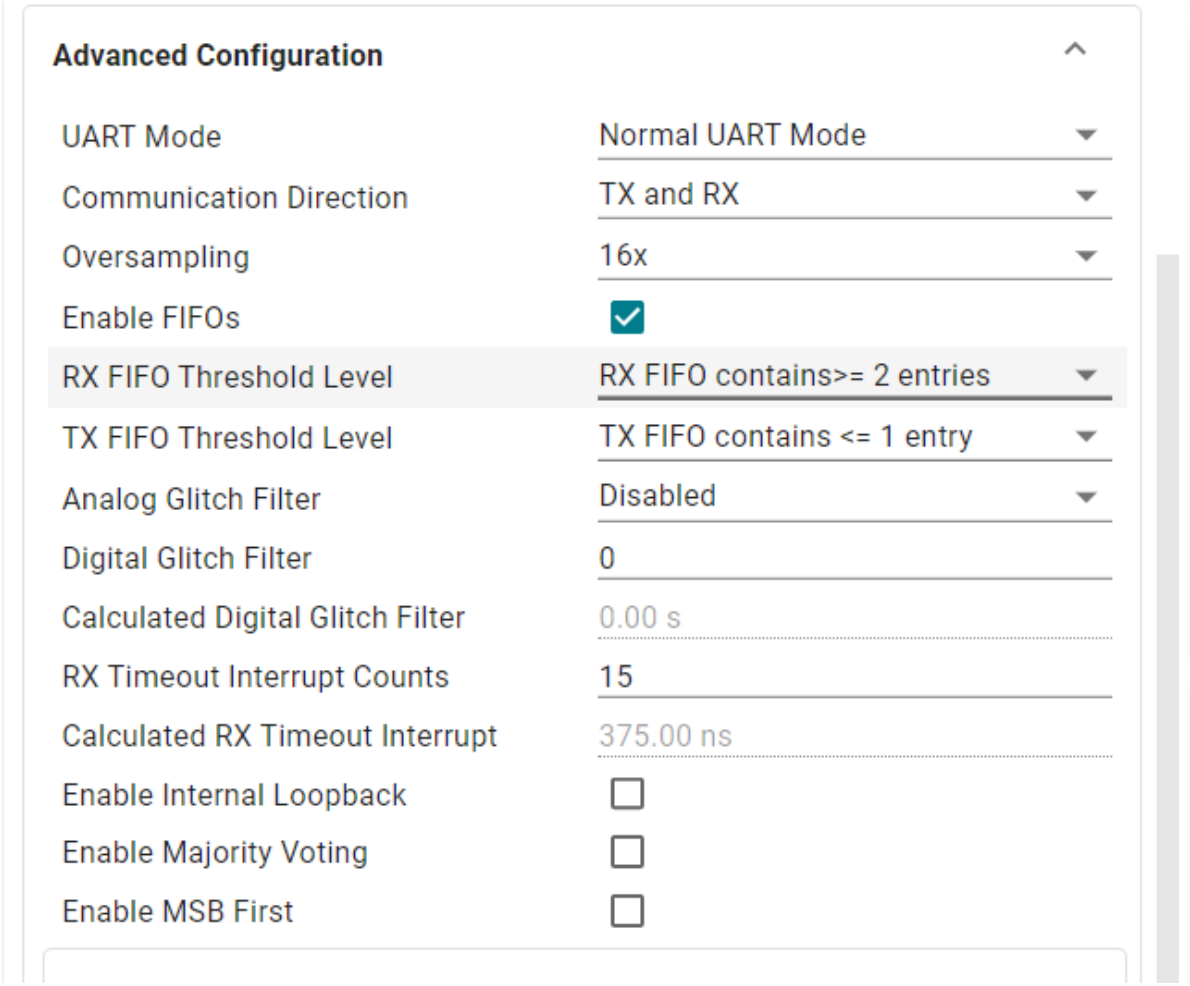
IV. Experiment Steps

This lesson uses the UART0 peripheral on pins PA10 and PA11, combined with the development board's onboard CH340 USB-to-serial chip, to achieve receiving serial port data sent by the host computer, and then sending the data back to the host computer through the serial port sending function.

1. **Open SYSCONFIG configuration tool**

This course is based on migrating previous serial communication code parts. Peripheral configuration modifications are as follows. Parameter definitions can refer to the previous serial communication section introduction.





DMA configuration part is as follows

Channel ID: DMA channel, default channel 0

Address Mode: Address mode, we set it to from block address to fixed address

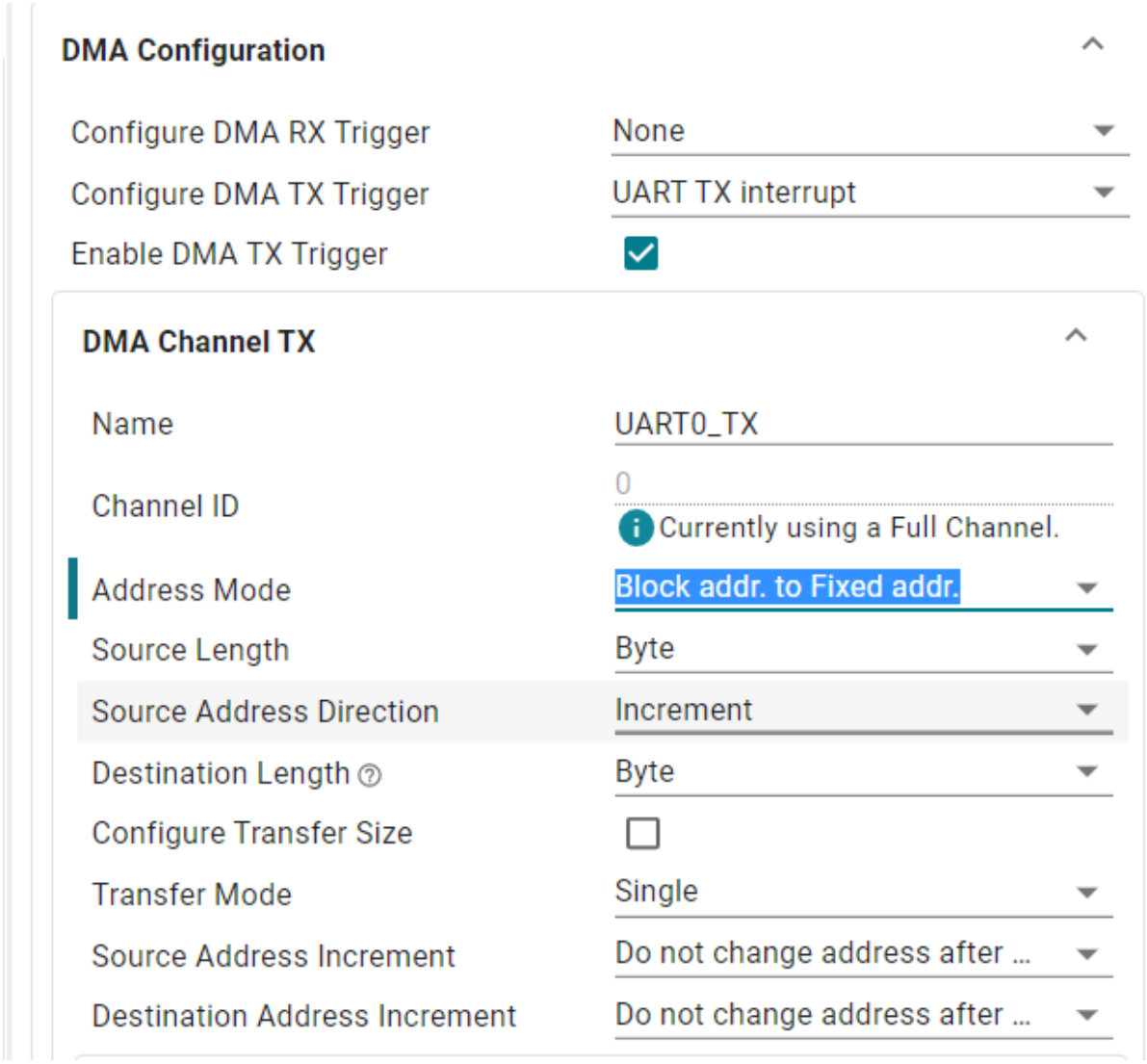
Source Address Direction: Source address direction, set to address increment

Source Length: Source transfer size, the source transfer size determines the size of data transfer from source to destination, as well as the increment per transfer (if configured as block).

Uneven transfer sizes can be configured. If source is larger than destination, only the lower half of the short word is transferred, ignoring the upper half. If destination is larger, the upper half of source will be zero-padded to fill the entire segment.

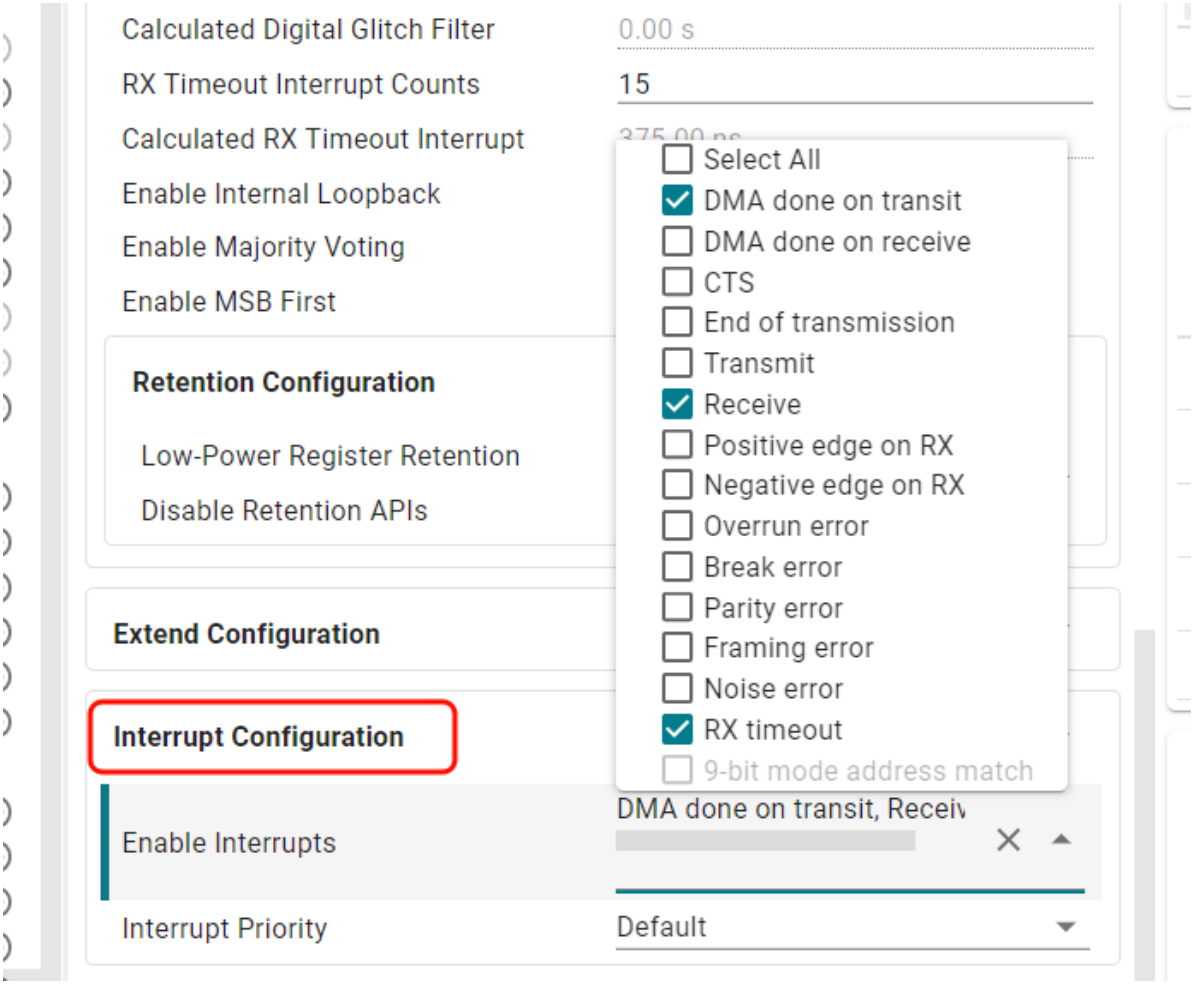
Destination Length: Destination transfer size, the destination transfer size determines the size of data transfer from source to destination, as well as the increment per transfer (if configured as block).

Uneven transfer sizes can be configured. If source is larger than destination, only the lower half of the short word is transferred, ignoring the upper half. If destination is larger, the upper half of source will be zero-padded to fill the entire segment.



RX Timeout Interrupt Counts: Set the timeout time for receive timeout interrupt. To make the timeout function effective, the RTOUT interrupt needs to be set.

In the interrupt settings, we add DMA transmit interrupt and receive timeout interrupt based on the original receive interrupt. Priority can be set to default.



Then we save and generate code

V. Code Analysis

```
 * @brief DMA方式发送字符串
 * @brief Send string via DMA
 *
 * @param str 要发送的字符串指针 / Pointer to string to send
 * @param len 字符串长度 / String length
 * @return uint8_t 1=发送成功, 0=发送失败 / 1=success, 0=failure
 */
uint8_t UART0_sendStrDMA(const char* str, uint16_t len) {
    // Check send conditions: DMA idle, length valid, pointer valid

    if (!UART0TxDMADone || len == 0 || str == NULL) {
        return 0; // Send failed
    }

    // Set DMA transfer parameters

    UART0TxDMADone = 0; // Mark DMA as busy
    // Set source address
    DL_DMA_setSrcAddr(DMA, UART0_TX_CHAN_ID, (uint32_t)str);
    // Set destination address
    DL_DMA_setDestAddr(DMA, UART0_TX_CHAN_ID, (uint32_t)(&UART_0_INST->TXDATA));
     // Set DMA send data size
    DL_DMA_setTransferSize(DMA, UART0_TX_CHAN_ID, len);
     // Enable DMA channel
    DL_DMA_enableChannel(DMA, UART0_TX_CHAN_ID);
```

```
    return 1; // Send success
}
```

```
void UART0_IRQHandler(void) {
    uint8_t receivedData = 0;
    // Get interrupt source
    switch (DL_UART_getPendingInterrupt(UART_0_INST)) {
        // DMA send complete interrupt
        case DL_UART_IIDX_DMA_DONE_TX:
            UART0_DMADoneTxCallback();
            break;

        // Receive timeout interrupt (frame end)
        case DL_UART_MAIN_IIDX_RX_TIMEOUT_ERROR:
            // Clear remaining data in RX FIFO
            while (!DL_UART_Main_isRXFIFOEmpty(UART_0_INST)) {
                uint8_t data_from_fifo = DL_UART_Main_receiveData(UART_0_INST);

                // Store in buffer (check overflow)
                if (recv0_length < (RE_0_BUFF_LEN_MAX - 1)) {
                    recv0_buff[recv0_length++] = data_from_fifo;
                } else {
                    // Reset when buffer is full
                    recv0_length = 0;
                }
            }
            // Add terminator and mark reception complete
            recv0_buff[recv0_length] = '\0';
            UART0RxDone = 1;
            break;

        // Regular receive interrupt
        case DL_UART_IIDX_RX:
            receivedData = DL_UART_Main_receiveData(UART_0_INST);

            // Check buffer space
            if (recv0_length < RE_0_BUFF_LEN_MAX - 1) {
                // Store in receive buffer
                recv0_buff[recv0_length++] = receivedData;

                // Detect line end character (newline)
                if (receivedData == '\n') {
                    // Add terminator
                    recv0_buff[recv0_length] = '\0';
                     // Mark reception complete
                    UART0RxDone = 1;
                }
            } else {
                // Buffer overflow handling
                recv0_length = 0;
                recv0_buff[0] = '\0';
            }
            break;

        // Other unhandled interrupts
        default:
```

```
            break;
        }
    }
```

VI. Experiment Phenomenon

After the program is flashed, connect the Type-C to the car, then configure the serial port as follows. After receiving data each time, we will copy a copy of the received data to the buffer and send it back, then start receiving again. We can send "Hello,Yahboom", the program processes and echoes data through timeout interrupt, or manually add a line break after "Hello,Yahboom", as shown in the figure below, to process data in specified format.