

# 마이크로프로세서응용실험 LAB10 결과보고서

## USART

20181536 엄석훈

### 1. 목적

- USART의 기본 동작을 전송속도, parity, stop bits 등을 변경하면서 확인한다.
- Polling에 의한 데이터 전송을 구현하여 각종 flag들의 역할을 이해한다.
- 인터럽트에 의한 데이터 전송을 구현한다.

### 2. 이론

#### 1) USART Introduction

USART는 universal synchronous asynchronous receiver transmitter의 약자로 동기식 비동기식으로 모두 사용가능한 직렬 통신의 방법이다. 동기식 방법은 공통 clock을 사용해 동기화된 타이밍에 서로 신호를 주고받는다. 비동기식 방법은 clock신호 없이 ground만 공유하고 character단위로 전송하며 start bit와 stop bit를 통해서 선로를 유지한다. 아래 [그림 1]은 비동기식 통신에서 장치간의 연결관계이다.

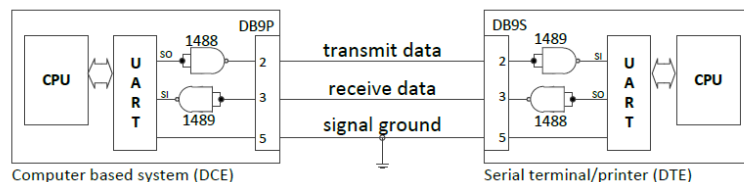


그림 1 비동기 통신에서의 장치간 연결

일반적으로 비동기식 방법에서 전송하지 않을 때는 1을 유지하다가 start bit로 0을 전송하고 stop bit로는 1을 사용한다. 그리고 전송하기 위해 직렬 데이터를 병렬로, 병렬 데이터를 직렬로 변환해야 하는데 이때는 shift register를 활용한다. 또한 1초당 전송되는 bit수인 baud rate를 통해서 데이터의 속도를 이야기한다.

## 2) USART Character Description

USART를 통해 비동기식으로 데이터를 전송할 때 character 단위로 데이터를 전송한다. 이때 USART\_CR1 레지스터의 M bit를 통해 8bits 또는 9bit를 character의 단위로 설정할 수 있다. Idle character는 전체 프레임이 1인 상태를 의미하며 그 뒤에는 start bit인 0이 들어온다. 그리고 break character는 전체 프레임이 0인 상태를 의미하며 뒤에 1 또는 2 bit의 1 데이터가 들어와 start임을 알려준다. 아래 [그림 2]는 8bit와 9bit일 때의 각각의 frame의 구조를 보여준다.

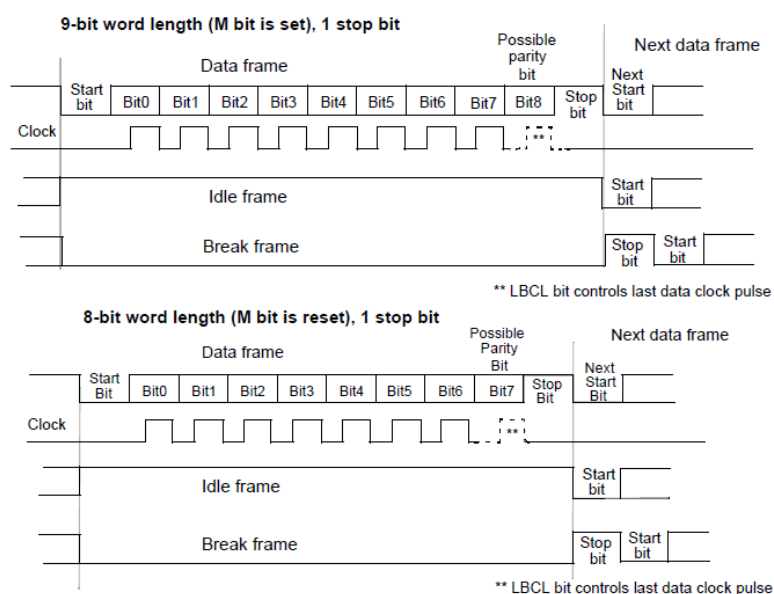
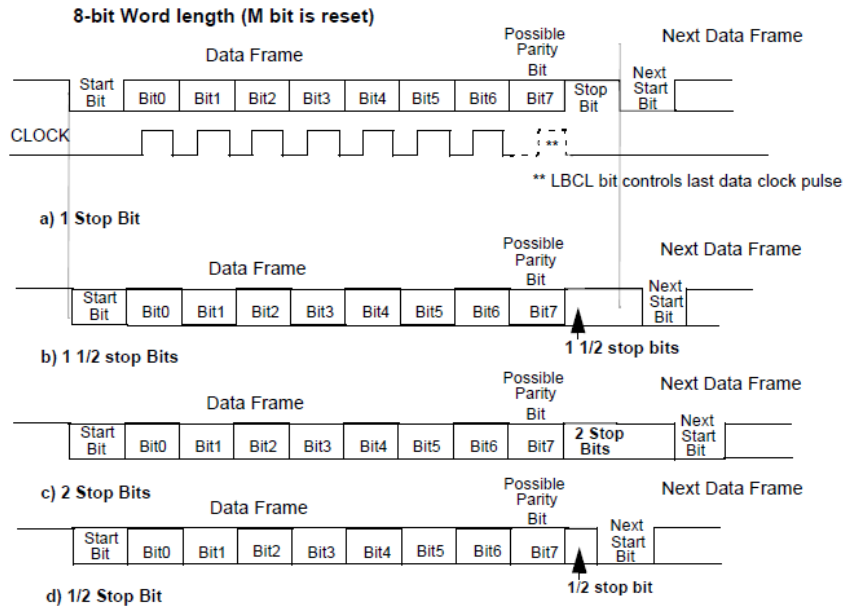


그림 2 Character frame의 구조

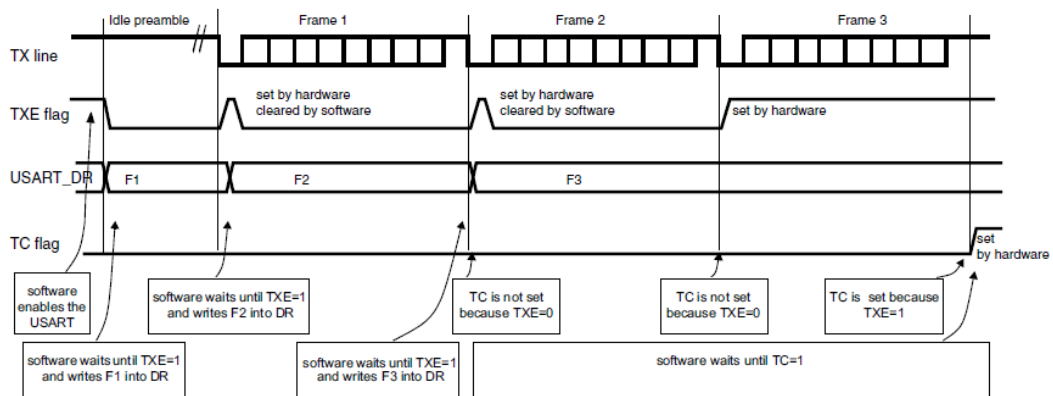
## 3) USART Transmitter

USART의 transmitter에서 데이터를 전송할 때는 shift register를 이용해서 LSB부터 TX pin을 이용해서 전송을 한다. 모든 character는 start bit부터 시작해서 전송을 시작하는데 1bit의 low 신호이다. 그리고 전송 후에는 0.5, 1, 1.5, 2bit의 high 신호인 stop bit가 전송된다. 기본적인 경우에는 1 bit의 stop 신호를 사용한다. 아래 [그림 3]은 여러가지 stop bit에서의 동작을 보여준다.



**그림 3 Stop bit에 따른 신호 구성**

데이터를 전송을 할 때 TXE bit가 매우 중요하게 사용되는데 이 bit가 하드웨어에 의해 1로 set되면 데이터가 TDR에서 shift register로 이동하여 전송이 시작되어 이제 TDR 레지스터가 비어있고 USART\_DR 레지스터의 데이터를 써도 된다는 뜻이다. 그리고 이는 software에 의해 reset된다. 그리고 다음으로 TC bit는 frame이 전부 전송되고 TXE bit가 1로 set되어 있으면 TC bit도 1로 set된다. 아래 [그림 4]는 데이터 전송 시 TXE와 TC bit를 포함한 동작의 예시이다.



**그림 4 데이터 전송 과정 예시**

#### 4) USART Receiver

USART는 USART\_CR1의 M bit에 의해 8bit 혹은 9bit의 데이터를 받을 수 있다. 먼저 데이터를 수신받기 위해서는 start bit를 감지해야 한다. Start bit를 감지하는 과정은 아래 [그림 5]와 같다. 이러한 과정이 없으면 start bit를 감지할 수 없다.

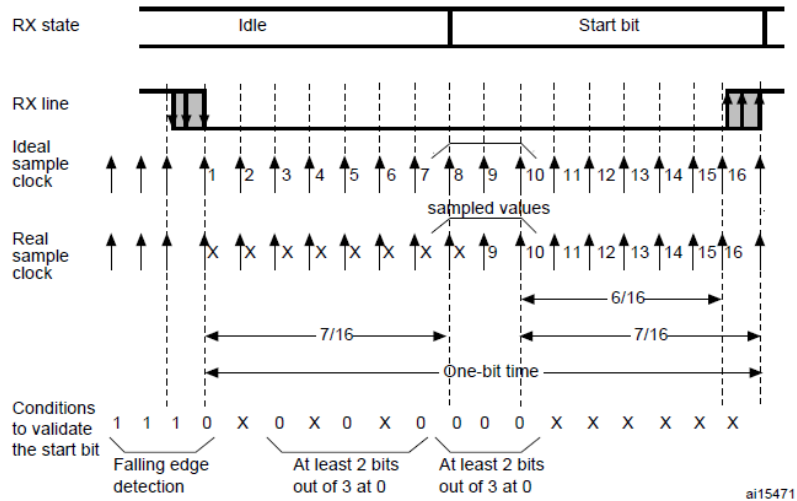


그림 5 Start bit 감지 과정

USART에서 M bit나 baud rate와 같은 기본 설정을 완료하면 RX pin을 통해서 데이터를 입력받을 수 있다. Shift register를 통해서 데이터가 들어오면 기본적으로 USART\_DR 레지스터에 저장된다. 그리고 RXNE bit가 set되고 RXNEIE bit가 set되어 있었으면 인터럽트가 발생하게 된다. 주의해야 할 점으로는 RXNE bit가 software에 의해 clear되지 않으면 데이터가 들어오지 못한다. 또한 노이즈에 의해 noise error나 stop bit가 인지되지 못해서 framing error가 발생할 수 있는데 이 경우 NE나 FE bit가 set되어 인지될 수 있다.

## 5) USART Baud Rate Generation

USART의 TX와 RX모두 동일한 baud rate를 사용해야 정상적으로 통신을 할 수 있다. 이때 USART\_BRR 레지스터에 저장된 값을 fixed point 숫자로 읽은 값인 USARTDIV를 이용해 아래 [그림 6]의 공식을 이용해서 baud값을 구할 수 있다. 그리고 아래 [그림 7]은 일부 목표 baud 값들에 대한 오차율을 보여준다.

$$\text{Tx/ Rx baud} = \frac{f_{\text{CK}}}{(16 * \text{USARTDIV})}$$

그림 6 Baud 계산 공식

Baud rate		$f_{PCLK} = 36 \text{ MHz}$			$f_{PCLK} = 72 \text{ MHz}$		
S.No	in Kbps	Actual	Value programmed in the Baud Rate register	% Error <sup>(1)</sup>	Actual	Value programmed in the Baud Rate register	% Error <sup>(1)</sup>
1.	2.4	2.400	937.5	0%	2.4	1875	0%
2.	9.6	9.600	234.375	0%	9.6	468.75	0%
3.	19.2	19.2	117.1875	0%	19.2	234.375	0%
4.	57.6	57.6	39.0625	0%	57.6	78.125	0.1%
5.	115.2	115.384	19.5	0.15%	115.2	39.0625	0%
6.	230.4	230.769	9.75	0.16%	230.769	19.5	0.16%
7.	460.8	461.538	4.875	0.16%	461.538	9.75	0.16%
8.	921.6	923.076	2.4375	0.16%	923.076	4.875	0.16%
9.	2250	2250	1	0%	2250	2	0%
10.	4500	NA	NA	NA	4500	1	0%

그림 7 Baud 값 계산 예시

## 6) USART Parity Control

USART 통신에서 parity bit의 사용은 USART\_CR1 레지스터의 PCE bit에 따라서 결정된다. 그리고 이때 M bit와의 조합에 따라 USART의 frame이 정해진다. 아래 [그림 8]은 이에 대한 예시이다.

M bit	PCE bit	USART frame
0	0	SB   8 bit data   STB
0	1	SB   7-bit data   PB   STB
1	0	SB   9-bit data   STB
1	1	SB   8-bit data PB   STB

그림 8 USART frame 구성

그리고 USART\_CR1 레지스터의 PS bit가 0이면 even parity가 사용되고 1이면 odd parity가 사용된다. Even parity는 전송하는 데이터와 parity bit를 합쳐서 1이 짝수개가 되도록 parity bit를 설정하고 odd parity는 전송하는 데이터와 parity bit를 합쳐서 1이 홀수개가 되도록 parity bit를 설정하는 방식이다.

## 7) USART Interrupts

USART에는 다양한 형태의 인터럽트가 발생할 수 있다. 아래 [그림 9]는 USART에서 발생하는 다양한 인터럽트 목록이다. 그리고 아래 [그림 10]은 이러한 인터럽트의 mapping diagram을 보여준다. USART는 하나의 인터럽트 벡터를 가지고 있기 때문에 모든 인터럽트가 하나의 인터럽트처럼 발생하며 그 내부에서 [그림 10]의 event flag를 확인해서 어떤 이유로 인터럽트가 발생하였는지

확인해야 한다.

Interrupt event	Event flag	Enable Control bit
Transmit data register empty	TXE	TXEIE
CTS flag	CTS	CTSIE
Transmission complete	TC	TCIE
Received data ready to be read	RXNE	RXNEIE
Overrun error detected	ORE	
Idle line detected	IDLE	IDLEIE
Parity error	PE	PEIE
Break flag	LBD	LBDIE
Noise flag, Overrun error and Framing error in multibuffer communication	NE or ORE or FE	EIE <sup>(1)</sup>

그림 9 USART 인터럽트 목록

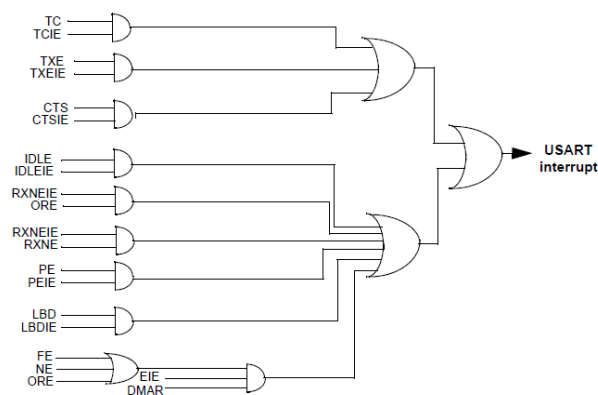


그림 10 USART 인터럽트 mapping diagram

## 8) START Register Map

아래 [그림 11]은 USART 사용에 필요한 레지스터 목록과 초기값이다.

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
0x00	USART_SR	Reserved																						CTS	LBD	TC	TXE	RXNE	IDLE	ORE	FE	PE				
	Reset value	0																						0	0	1	0	0	0	0	0	0				
0x04	USART_DR	Reserved																								DR[8:0]										
	Reset value	0																								0	0	0	0	0	0	0	0	0	0	
0x08	USART_BRR	Reserved																		DIV_Mantissa[15:4]								DIV_Fraction[3:0]								
	Reset value	0																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
0x0C	USART_CR1	Reserved																						UE	WAKE	PCP	PS	PEIE	TXIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SSK
	Reset value	0																						0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	USART_CR2	Reserved																		LINEN	STOP[1:0]	CPOL	CPHA	LBCL	Reserved	LBDIE	LBCL	Reserved	ADD[3:0]							
	Reset value	0																		0	0	0	0	0	0	0	0	0	0	0	0	0				
0x14	USART_CR3	Reserved																						CTSE	RTSE	DMAT	Reserved	SCEN	NACK	HDSEL	IRLP	REN	FIF			
	Reset value	0																						0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	USART_GTPR	Reserved																		GT[7:0]								PSC[7:0]								
	Reset value	0																		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

그림 11 USART 레지스터 목록 및 초기값

### 3. 실험 과정

#### 1) 실험 1

STEP 1: 본 실험에서는 microcontroller에 내장된 USART1을 사용한다.

USART1의 Tx data와 Rx data에 해당하는 USART1\_TX와 USART1\_RX 신호는 각각 PA9와 PA10의 alternate function으로 지정되어 있음을 표 8.5를 통해 확인한다.

Pins		Pin	Main	Alternate function	
100	64	Name	Function	Default	Remap
65	39	PC8	PC8		TIM3_CH3
66	40	PC9	PC9		TIM3_CH4
67	41	PA8	PA8	USART1_CK/ TIM1_CH1/MCO	
68	42	PA9	PA9	USART1_TX/ TIM1_CH2	
69	43	PA10	PA10	USART1_RX/ TIM1_CH3	

그림 12 교재 표 8.5의 일부

위의 [그림 12]에서처럼 실험에서 사용할 USART1의 TX와 RX는 각각 PA9, PA10의 alternate function임을 확인할 수 있다.

STEP 2: Program 10.1은 USART의 설정과 기본적인 동작을 확인하기 위해 작성되었다. 데이터의 송신과 수신 동작을 확인하기 위해 Program 10.1은 송신을 담당하는 보드에서 수행하고, 수신을 담당하는 보드에는 Program 10.1에서 다음 내용을 변경한 프로그램을 수행한다.

● line 4를 char string[10];으로 변경.

● line 15에서 08;을 04;로 변경

● lines 20-23을 다음 내용으로 교체.

```
20     while (1) {  
21         while (!(USART1->SR & 0x020));  
22         string[i++] = USART1->DR;  
23     }
```

이상의 수신을 담당할 보드를 위해 변경된 내용들의 의도를 파악해보자.

이후 설명에서 송신 기능을 수행하는 모듈을 Tx module, 수신 기능을 수행하는 모듈을 Rx module이라 칭한다.

먼저 수신을 담당할 보드를 위해 수정한 코드를 살펴보면 line 4에서는 string에 수신한 데이터를 저장하기 위해서 비워둔 채 공간만 마련을 해 둔 것이다. 사실 송신측 코드처럼 초기값이 있어도 되지만 확실히 데이터가 들어온 것을 확인하기 위해서 초기값이 없도록 코드를 수정하였다. 다음으로 line 15에서 `USART1->CR1 |= 0x00000008;` 대신 `USART1->CR1 |= 0x00000004;` 로 코드를 수정하였는데 `USART1->CR1` 레지스터의 3번 bit는 TE bit로 transmitter enable이고 2번 bit는 RE bit로 receiver enable로 Rx 모듈에서는 TE대신 RE를 set해주어야 하기 때문에 코드를 수정하였다. 마지막으로 lines 20-23의 while문을 수정하였는데 Rx 모듈에서는 계속해서 가장 바깥쪽의 while문을 돌면서 안쪽 while문에서는 `USART1->SR & 0x020`의 값이 0이면 계속해서 while문을 돌도록 하였는데 `USART1->SR`의 5번 bit인 RXNE는 read data register not empty로 shift register의 데이터가 `USART_DR`로 전송되면 set되는 bit로 0이라는 뜻은 `USART_DR`에 아직 데이터가 들어오지 않았다는 뜻임으로 계속해서 while문을 돌면서 기다린다. 그리고 while문이 끝나면 `USART_DR`로 데이터가 송신되었다는 뜻이고 `string[i++] = USART1->DR;` 을 통해서 송신된 데이터를 string에 저장하고 i값을 증가시키는 동작을 하도록 코드를 작성하였다.

**STEP 3: 이 프로그램을 이용하여 프로젝트를 각각 생성한다. 마이크로컨트롤러에 내장된 USART를 사용하기 위한 설정과정과 관련된 다음 사항들에 대해 참고문헌 (STMicroelectronics, 2011)을 참조하면서 프로그램에 포함된 내용을 파악한다.**

- Lines 7-9은 GPIO PA9와 PA10를 alternate function으로 지정하기 위한 설정과정을 보여준다. 이 중 line 7에는 APB2 bus에 연결된 USART1에 clock을 공급하기 위한 설정도 포함된다 ((STMicroelectronics, 2011) 참조). 그림 8.2을 통해 USART1의 bus 연결관계를 파악하고 이 때 clock의 최대 주파수가 72Mhz임도 확인한다.

각 line을 통해 수행되는 설정내용을 참고문헌 (STMicroelectronics, 2011)에 서 확인한다. 이 과정을 통해 alternate function이 어떻게 지정되어 사용되는지 이해한다.

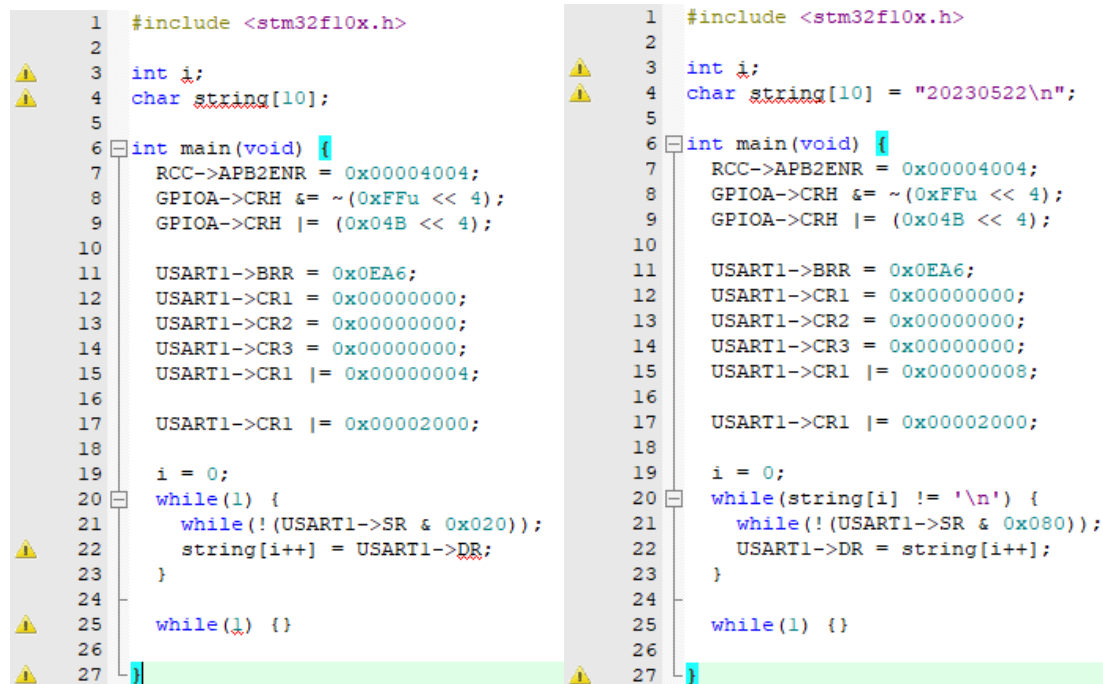
- Line 11은 baud rate를 설정하기 위함이다. 215페이지를 통해 이해한 내용과 수식을 바탕으로 이 설정의 의미를 해석해 본다. 만일 baud rate를 4800으로 설정하려면 이 부분은 어떻게 변경되어야 하는가?

- Line 12-14는 USART1의 동작을 초기화하는 과정이다. 이렇게 초기화했을 때, word length,



parity, stop bits, control flow 등이 어떻게 설정되었는지 확인한다.

Line 15는 본 실험을 위해 USART1의 transmitter만을 enable(TE)시키기 위 함이며 line 17은 USART1 전체를 enable(UE)하기 위한 단계이다. USART1의 동작을 위해 UE는 반드시 설정되어야 한다.



```
1 #include <stm32f10x.h>
2
3 int i;
4 char string[10];
5
6 int main(void) {
7     RCC->APB2ENR = 0x00004004;
8     GPIOA->CRH &= ~(0xFFu << 4);
9     GPIOA->CRH |= (0x04B << 4);
10
11     USART1->BRR = 0x0EA6;
12     USART1->CR1 = 0x00000000;
13     USART1->CR2 = 0x00000000;
14     USART1->CR3 = 0x00000000;
15     USART1->CR1 |= 0x00000004;
16
17     USART1->CR1 |= 0x00002000;
18
19     i = 0;
20     while(1) {
21         while(!(USART1->SR & 0x020));
22         string[i++] = USART1->DR;
23     }
24
25     while(1) {}
26
27 }
```

```
1 #include <stm32f10x.h>
2
3 int i;
4 char string[10] = "20230522\n";
5
6 int main(void) {
7     RCC->APB2ENR = 0x00004004;
8     GPIOA->CRH &= ~(0xFFu << 4);
9     GPIOA->CRH |= (0x04B << 4);
10
11     USART1->BRR = 0x0EA6;
12     USART1->CR1 = 0x00000000;
13     USART1->CR2 = 0x00000000;
14     USART1->CR3 = 0x00000000;
15     USART1->CR1 |= 0x00000008;
16
17     USART1->CR1 |= 0x00002000;
18
19     i = 0;
20     while(string[i] != '\n') {
21         while(!(USART1->SR & 0x080));
22         USART1->DR = string[i++];
23     }
24
25     while(1) {}
26
27 }
```

그림 13 직접 작성한 lab10\_1.c 코드 (좌측: Tx, 우측: Rx)

Line 7-9를 해석하면 다음과 같다. 먼저  $RCC \rightarrow APB2ENR = 0x00004004$ ; 를 통해서  $RCC \rightarrow APB2ENR$  레지스터의 2번 14번 bit를 set 해주었다. 이는 각각 GPIOA와 USART1 clock을 enable 해주는 bit 이다. 그리고 line 8의  $GPIOA \rightarrow CRH \&= \sim(0xFFu \ll 4)$ ; 를 통해서  $GPIOA \rightarrow CRH$  레지스터의 0번부터 4번 bit까지와 20번 bit부터 31번 bit는 전부 1로 set하고 나머지 4번 bit부터 19번 bit는 1로 set하였다. 그 상태에서 line 9의  $GPIOA \rightarrow CRH \mid= (0x04B \ll 4)$ ; 를 통해서  $GPIOA \rightarrow CRH$  레지스터에서  $0x4B0$ 를 OR연산 해주었다. 우리가 관심있는 PA9과 PA10만 보면 PA9는 push pull alternate function output으로 설정하고 PA10은 input floating으로 설정하였다. 이에 대한 근거로 reference manual을 보면 아래 [그림 14], [그림 15]와 같이 각각 TX와 RX는 push-pull alternate function과 input floating으로 설정해야 하는 것을 확인할 수 있다. 그리고 교재에서 가져온 아래 [그림 16]을 통해서 USART1은 APB2 bus에 연결되어 있으며 clock의 최대 주파수가 72Mhz임을 확인할 수 있다.

USART pinout	Configuration	GPIO configuration
USARTx_TX <sup>(1)</sup>	Full duplex	Alternate function push-pull
	Half duplex synchronous mode	Alternate function push-pull

그림 14 USART\_TX 설정

USART pinout	Configuration	GPIO configuration
USARTx_RX	Full duplex	Input floating / Input pull-up
	Half duplex synchronous mode	Not used. Can be used as a general IO

그림 15 USART\_RX 설정

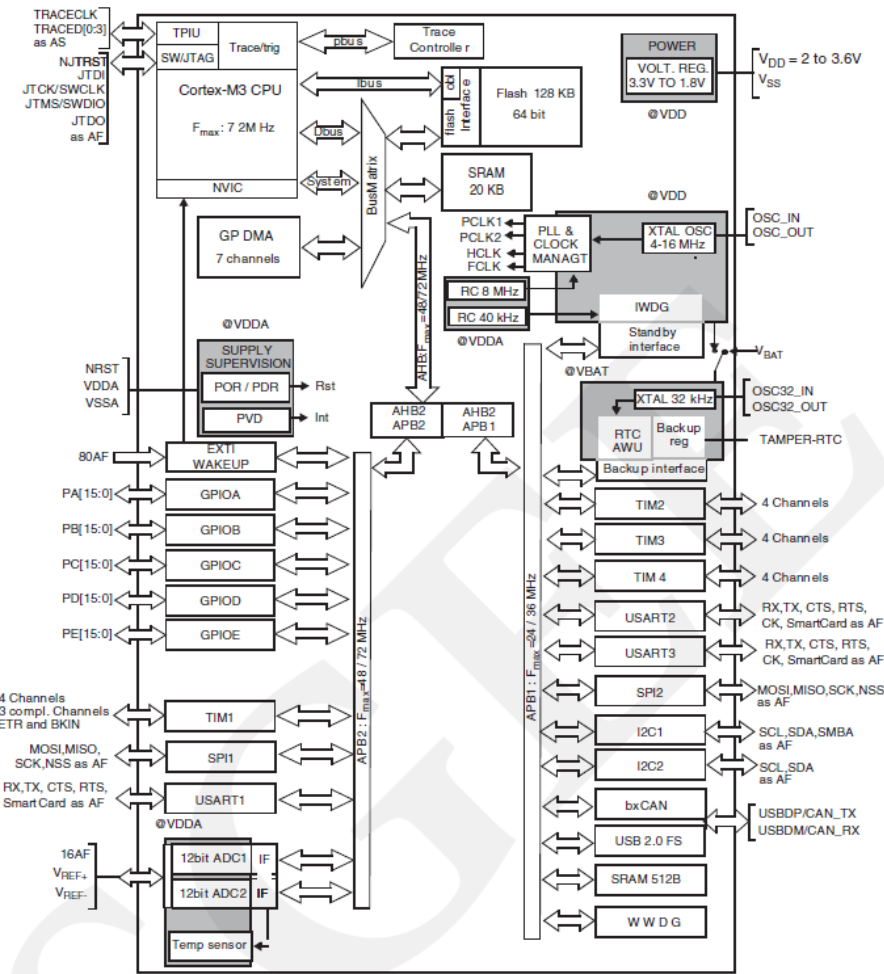


그림 16 교재 그림 8.2

다음으로 line 11의 USART1->BRR = 0x0EA6; 는 USART1의 baud rate를 결정하는 부분으로 이론에서 다룬 내용을 바탕으로 계산해보면 상위 12bit를 통해 구한 mantisa는 234이고 하위 4bit를 통해 구한 fraction은 0.375임으로 USARTDIV는 234.375이고 baud rate를 계산하면 19200이 나온다. 즉 1초에 19200bit가 전송된다. 이때 baud rate를 4800으로 설정하기 위해서는 USARTDIV가 937.5가 되어야 한다. 따라서 mantisa는 937이 되어야 함으로 상위 12bit는 0x3A9이고 fraction은 0.5가 되어야 함으로 하위 4bit는 0x8이 되어야 함으로 BRR값을 0x3A98이 되면 baud rate를

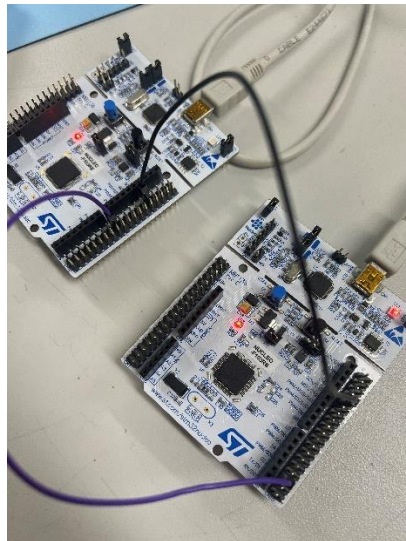
4800으로 설정할 수 있다.

다음으로 lines 12-14에서는 USART1의 CR1, CR2, CR3를 모두 0x00000000으로 설정하였다. 이때 주요한 설정을 살펴보면 word length는 CR1의 M bit가 0이기 때문에 8bit data가 송신된다. 그리고 CR1의 PCE bit도 0이기 때문에 parity control은 disable되어 있고 CR2의 STOP bit는 00임으로 1bit의 stop bit를 가지고 CR3의 CTSE bit가 0임으로 CTS hardware flow control은 disable 되어있다. 또한 RTSE bit도 0이기 때문에 RTS hardware control flow도 disable 되어있다. 그 외에도 각종 interrupt나 clock 모두 disable되어 있다.

그리고 line 15의 USART1->CR1 = 0x00000008; 또는 USART1->CR1 = 0x00000004; 를 통해서 각각 TE bit와 RE bit를 1로 set하여 각각 transmitter와 receiver를 enable 해주었다. 마지막으로 line 17에서 USART1->CR1 = 0x00002000; 을 통해서 CR1레지스터의 13번 bit인 UE를 1로 set하여 USART1을 enable 해주었다.

**STEP 4: 두 boards를 19페이지의 표 1.2를 참고하여 다음과 같이 jump wires로 연결한다.**

- 송신 보드와 수신 보드의 CN10의 9번 핀(GND) 사이를 연결한다.
- Tx module의 CN10의 21번 핀(PA9)을 Rx module의 CN10의 33번 핀 (PA10)에 연결한다.



**그림 17 두 보드를 jump wire로 연결한 모습**

위 [그림 17]과 같이 두 보드의 GND를 연결해주고 Tx 모듈의 PA9과 Rx 모듈의 PA10을 연결하였다.

STEP 5: Tx module의 line 22와 Rx module의 line 22에 각각 break point를 설정한다.

Rx module을 먼저 수행한 후 Tx module을 수행한다. Tx module line 22를 single step으로 수행 하면서 Rx module의 반응을 살펴보자.

Rx module line 22를 single step으로 수행한 후 Rx module의 string[]에 해 당하는 메모리에 저장된 내용이 Tx module에서 전송한 데이터와 일치하는지 확인해보자.

이상의 과정을 반복해서 Tx module의 string[]에 저장된 data가 제대로 전송되는지 확인해보자.

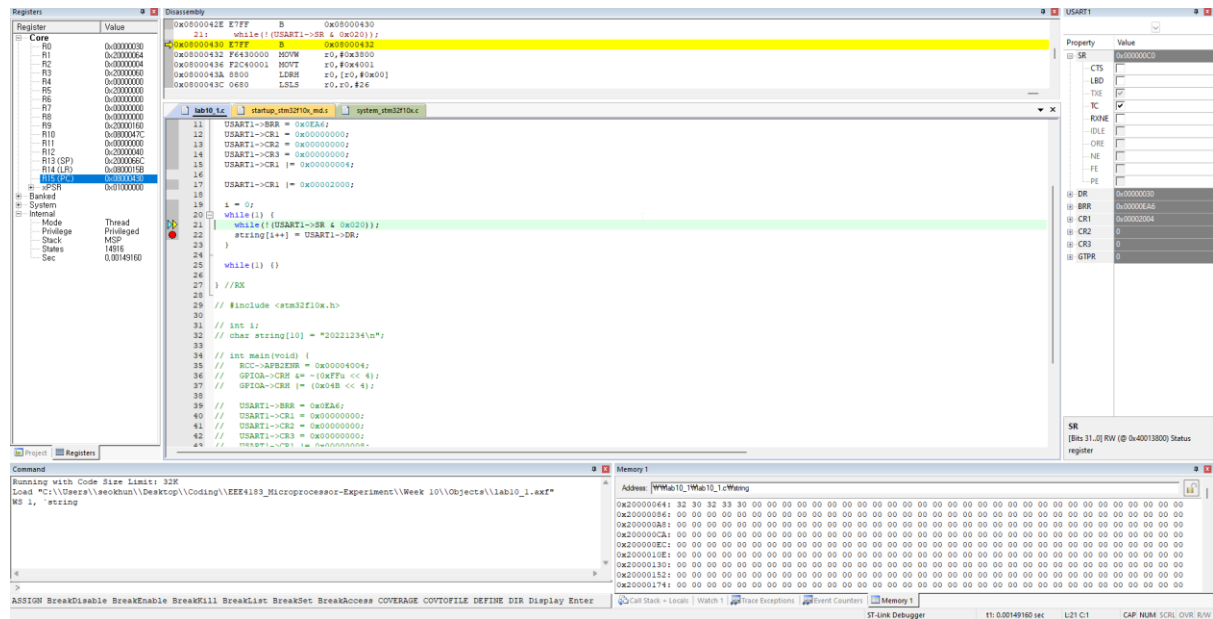


그림 18 Rx 모듈이 line 21에서 루프를 돌고있는 모습

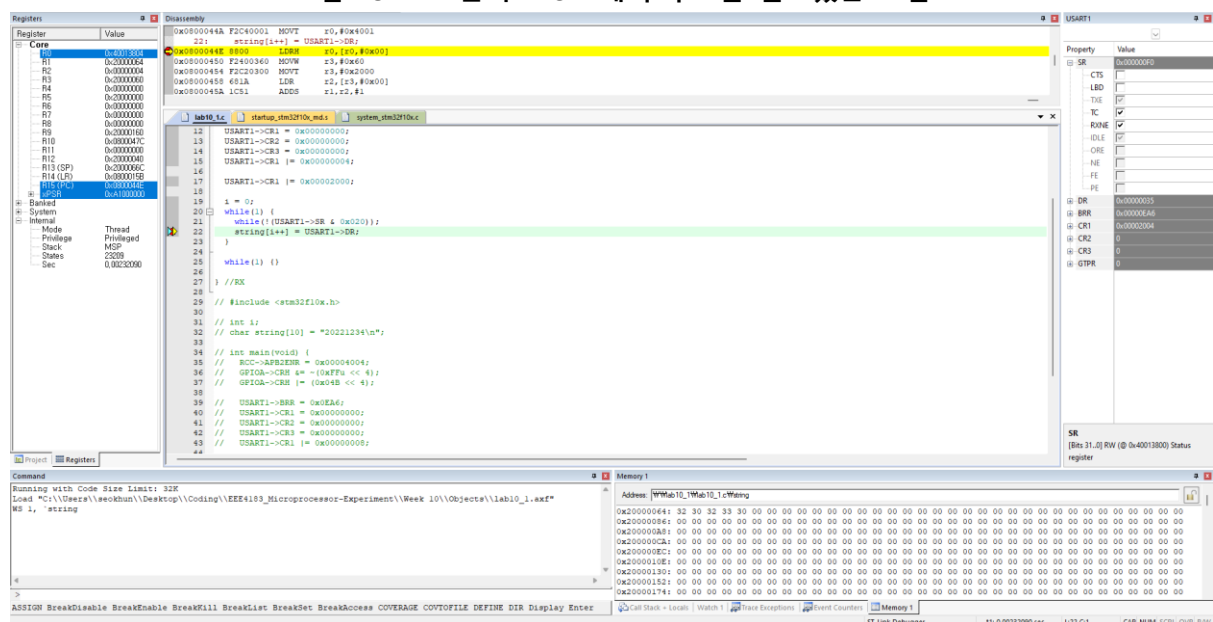


그림 19 Tx 모듈의 line 22 수행 후 Rx 모듈의 반응

Rx 모듈의 line 22에 break point를 걸고 먼저 수행시키면 아직 받은 데이터가 없기 때문에

USART1->SR & 0x20의 결과가 0이고 따라서 [그림 18]과 같이 line 21의 while문에서 무한루프를 돌고 있다. 이때 Tx 모듈의 line 22를 수행하면 Tx 모듈에서 Rx 모듈로 데이터를 전송하였으므로 Rx 모듈의 while문을 탈출하여 [그림 19]와 같이 line 22로 넘어가게 된다. 또한 Rx 모듈도 USART1->DR 레지스터를 보면 새로운 데이터인 35가 들어온 것을 확인할 수 있으며 receive 데이터가 있기 때문에 RXNE 레지스터도 1로 set되어 있는 것을 확인할 수 있다. 그리고 전송이 완료되고 마지막 데이터를 살펴보면 아래 [그림 20]과 같이 Rx에 Tx가 보낸 데이터가 정상적으로 저장된 것을 확인할 수 있다.

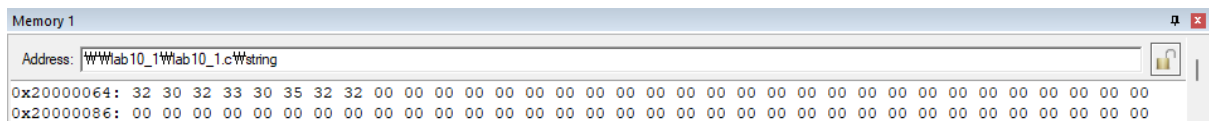


그림 20 Rx에 저장된 데이터

#### STEP 6: Tx module의 line 21과 Rx module의 line 21의 역할은 무엇인가?

만일 이 과정이 두 modules에서 생략된다면 어떤 문제를 예상할 수 있는가? 각 모듈에서 line 21을 주석처리한 후 다음 단계를 통해 이 문제를 확인해보자.

- 앞 단계에서 설정된 break point를 모두 해제한다.
- Rx module을 먼저 수행한 후 Tx module을 수행한다.
- Rx module을 중단하고 string[]에 해당하는 메모리에 저장된 내용을 확인해보자. 앞 단계의 결과에 어떻게 다른가?

Tx 모듈에서 line 21은 USART1->SR 레지스터의 7번 bit가 0이면 계속해서 while문을 도는데 이는 data가 shift register로 이동하지 않았다는 것을 의미함으로 아직 데이터 전송을 하지 않은 상태라는 뜻이다. 그리고 Rx 모듈에서 line 21은 USART1->SR 레지스터의 5번 bit가 0이면 계속해서 while문을 도는데 이는 data가 아직 수신되지 않았다는 뜻이다. 즉 데이터가 수신되어야 해당 while문을 탈출한다.

만일 line 21을 각 프로그램에서 삭제한다면 Tx 모듈에서는 이전 데이터를 송신하지도 않았는데 다음 데이터를 송신하고 Rx 모듈에서는 아직 데이터를 받지 않았는데 데이터를 읽어버리는 일이 발생할 수 있다. 따라서 보내려는 데이터를 제대로 보내지 못하고 받으려는 데이터를 제대로 받지 못하는 일이 발생할 수 있다. 이에 대한 결과는 아래 [그림 21]에서 확인 가능하다. 이를 보면

line 21을 각 코드에서 삭제한 후에 쓰레기 값이 이상한 곳에 퍼져서 저장되어 있는 것을 확인할 수 있다. 이는 Rx 모듈에서 i값이 제한없이 계속 증가하면서 Tx의 데이터를 저장하여 이상한 곳에 데이터가 저장된 것이고 Tx에서는 데이터를 다 보내지도 못했는데 계속 보내거나 Rx에서는 데이터가 다 오지도 않았는데 읽는 일이 발생하여 원치 않는 동작을 확인할 수 있었다.

그림 21 Line 21 삭제 후 결과

```

6 int main(void) {
7     RCC->APB2ENR = 0x00004004;
8     GPIOA->CRH &= ~(0xFFu << 4);
9     GPIOA->CRH |= (0x04B << 4);
10
11     USART1->BRR = 0x3A98;
12     USART1->CR1 = 0x00001400;
13     USART1->CR2 = 0x00003000;
14     USART1->CR3 = 0x00000000;
15     USART1->CR1 |= 0x00000004;
16
17     USART1->CR1 |= 0x00002000;

```

### 그림 22 Tx와 다른 Rx 설정



터가 아니라 0x00으로 잘못된 데이터가 들어온 것을 확인할 수 있다. 그리고 전체적인 흐름을 살펴보기 위해 break point를 제거하고 수행해본 결과 [그림 24]와 같이 데이터가 들어온 것을 확인할 수 있다. 우연의 일치로 Rx 모듈에서 인식하는 데이터의 형태로 데이터가 들어오는 경우에는 쓰레기 데이터가 들어와 저장되었다. 이를 통해서 Tx 모듈과 Rx 모듈의 설정이 다르다면 원하는 데이터를 전송할 수 없는 것을 확인할 수 있었다.

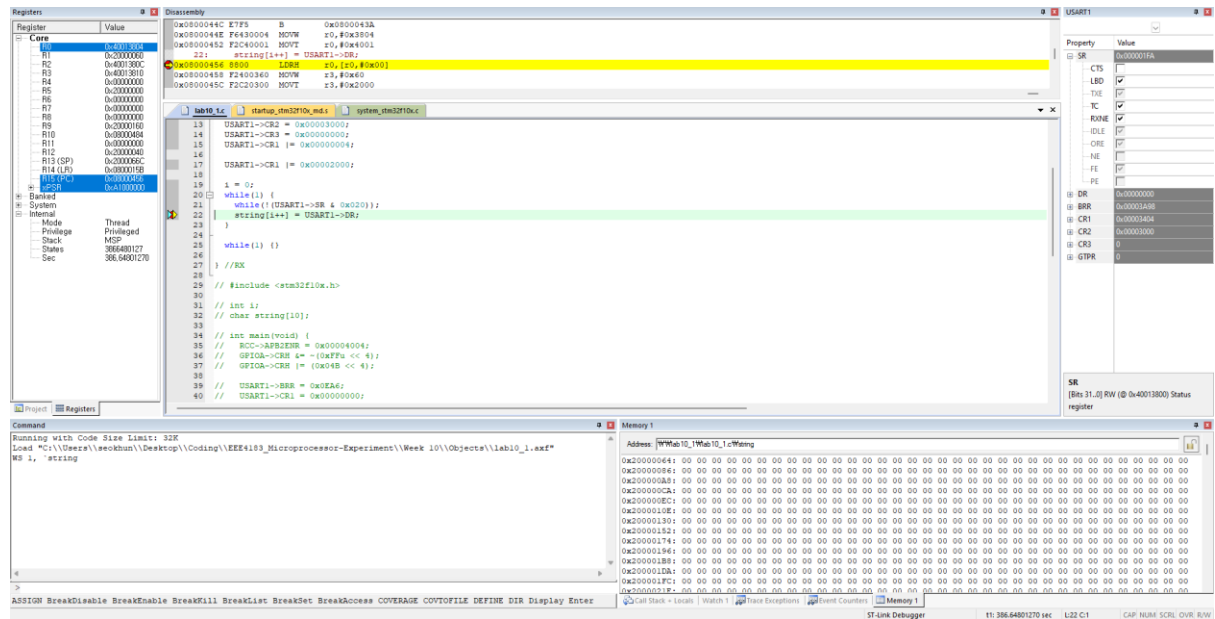


그림 23 Line 22에 break point를 건 뒤 확인한 SR 레지스터 값

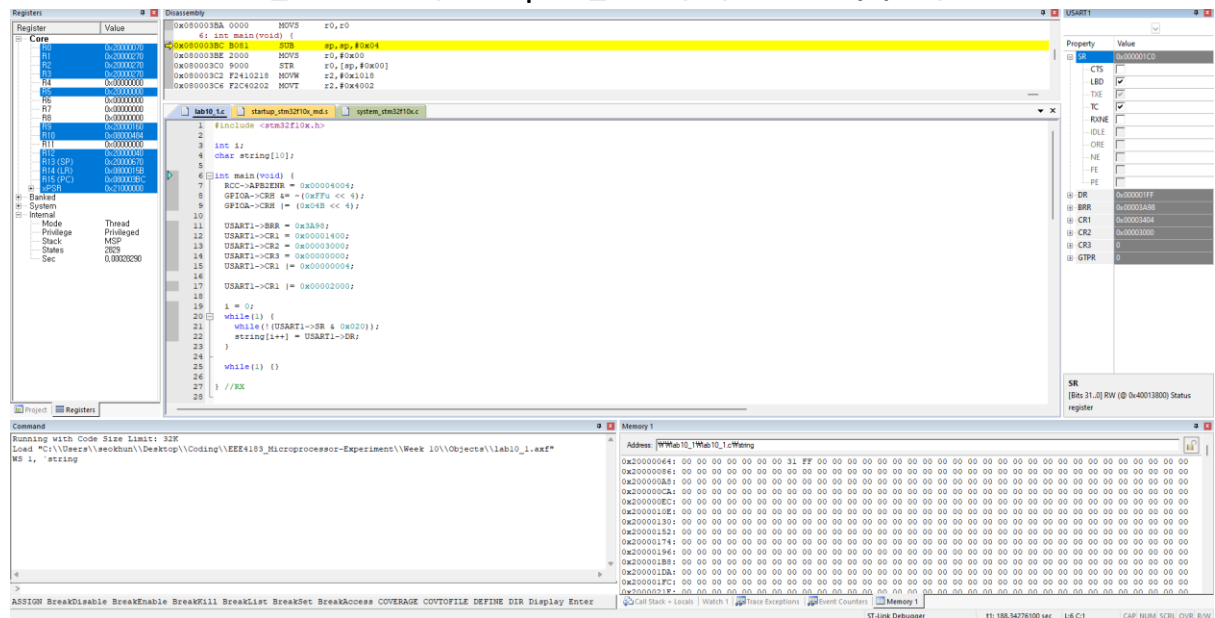


그림 24 전체 수행 후 결과

## 2) 실험 2

**STEP 8: Program 10.2는 인터럽트 방식으로 USART를 구동하는 방법을 보여준다.**

Program 10.1에서와 마찬가지로 송신을 담당하는 보드와 수신을 담당하는 보드로 역할을 구분하여 프로그램을 수행한다.

Program 10.2는 송신을 담당하는 보드에서 수행하고 수신을 담당하는 보드에서는 이 프로그램에서 다음과 같이 내용을 변경한 프로그램을 수행한다.

- line 4를 `char string[10];`으로 변경.
- line 15에서 08;을 04;로 변경
- line 19에서 80;을 20;로 변경
- lines 25-32의 USART1 IRQHandler를 다음으로 교체.

```
26     void USART1_IRQHandler (void){
27         if (USART1->SR & 0x20 )
28             string[i++] = USART1->DR;
29     }
```

Rx module을 위해 변경된 내용들의 의도를 파악해보자.

이상의 내용을 바탕으로 송신 보드와 수신 보드에서 수행할 프로젝트를 각각 생성한다.

Rx 모듈을 위한 코드에서 Tx 모듈과 다른 부분을 먼저 분석하면 다음과 같다. 먼저 line 4를 `char string[10];` 으로 바꾼 것은 실험 1에서와 같이 Rx는 초기값이 필요 없기 때문에 그냥 공간 할당을 위해 선언만 해둔 것이다. 그리고 line 15에서 08대신 04로 변경한 것은 실험 1과 동일하게 USART1을 receiver mode로 설정한 것이다. 그리고 line 19의 `USART1->CR1 |= 0x00000080;` 을 `USART1->CR1 |= 0x00000020;` 으로 변경한 것은 Tx 모듈에서 사용하는 USART1->CR1의 7번 bit인 TXE interrupt enable대신 5번 bit인 RXNE를 1로 set하여 RXNE interrupt를 enable해주었다. 그리고 USART1의 핸들러에서 line 27은 USART1의 핸들러가 호출되었을 때 데이터가 수신되어 RXNE bit가 1로 set되어 있는지 확인한 뒤 데이터가 수신된 것이 맞다면 USART1->DR에 저장된 수신된 데이터를 `string[i]`에 저장하고 i를 증가시켜 준다.

이를 바탕으로 Tx 모듈과 Rx 모듈을 위해 작성한 프로그램은 아래 [그림 25]에서 확인 가능하다.



```

1 #include <stm32f10x.h>
2
3 int i = 0;
4 char string[10] = "20221234\n";
5
6 int main(void) {
7     RCC->APB2ENR = 0x00004004;
8     GPIOA->CRH &= ~(0xFFu << 4);
9     GPIOA->CRH |= (0x04B << 4);
10
11     USART1->BRR = 0x0EA6;
12     USART1->CR1 = 0x00000000;
13     USART1->CR2 = 0x00000000;
14     USART1->CR3 = 0x00000000;
15     USART1->CR1 |= 0x00000008;
16     USART1->CR1 |= 0x00002000;
17
18     NVIC->ISER[1] |= (1 << 5);
19     USART1->CR1 |= 0x00000080;
20
21     while(1) {}
22 } //end main
23
24 void USART1_IRQHandler(void) {
25     if(USART1->SR & 0x80) {
26         if(string[i] == '\n')
27             USART1->CR1 &= ~0x80; //disable Tx interrupt
28         else
29             USART1->DR = string[i++];
30     }
31 }
32

```

```

1 #include <stm32f10x.h>
2
3 int i = 0;
4 char string[10];
5
6 int main(void) {
7     RCC->APB2ENR = 0x00004004;
8     GPIOA->CRH &= ~(0xFFu << 4);
9     GPIOA->CRH |= (0x04B << 4);
10
11     USART1->BRR = 0x0EA6;
12     USART1->CR1 = 0x00000000;
13     USART1->CR2 = 0x00000000;
14     USART1->CR3 = 0x00000000;
15     USART1->CR1 |= 0x00000004;
16     USART1->CR1 |= 0x00002000;
17
18     NVIC->ISER[1] |= (1 << 5);
19     USART1->CR1 |= 0x00000020;
20
21     while(1) {}
22 } //end main
23
24 void USART1_IRQHandler(void) {
25     if(USART1->SR & 0x20)
26         string[i++] = USART1->DR;
27 }
28

```

그림 25 직접 작성한 lab10\_2.c 코드 (좌측: Tx, 우측: Rx)

STEP 9: Rx module을 먼저 수행하고 Tx module을 수행한다.

Rx module을 중단하고 string[]에 해당하는 메모리에 저장된 내용을 확인해보자. 인터럽트에 의해 정상적으로 전송이 수행되었는가?

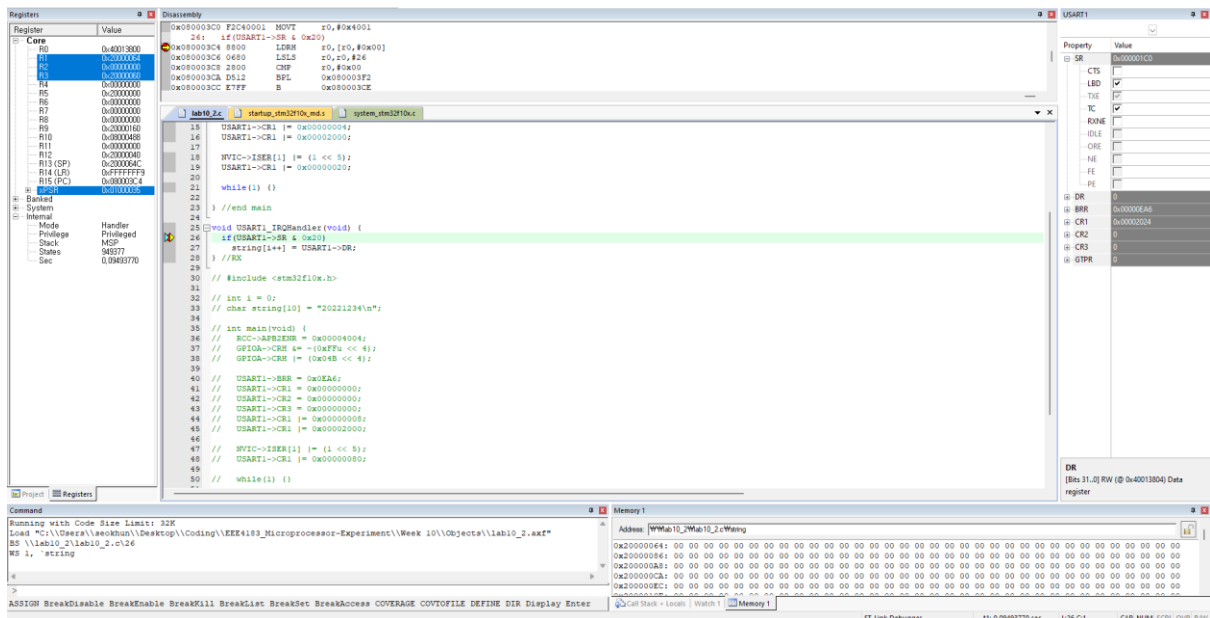


그림 26 인터럽트를 기다리는 Rx 모듈

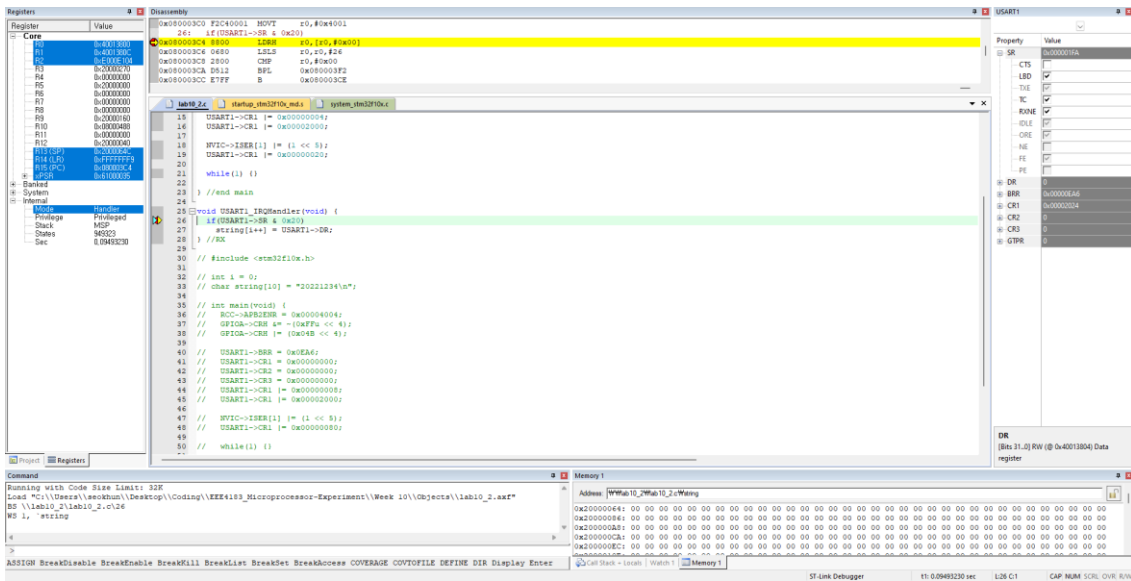


그림 27 인터럽트가 발생한 Rx 모듈

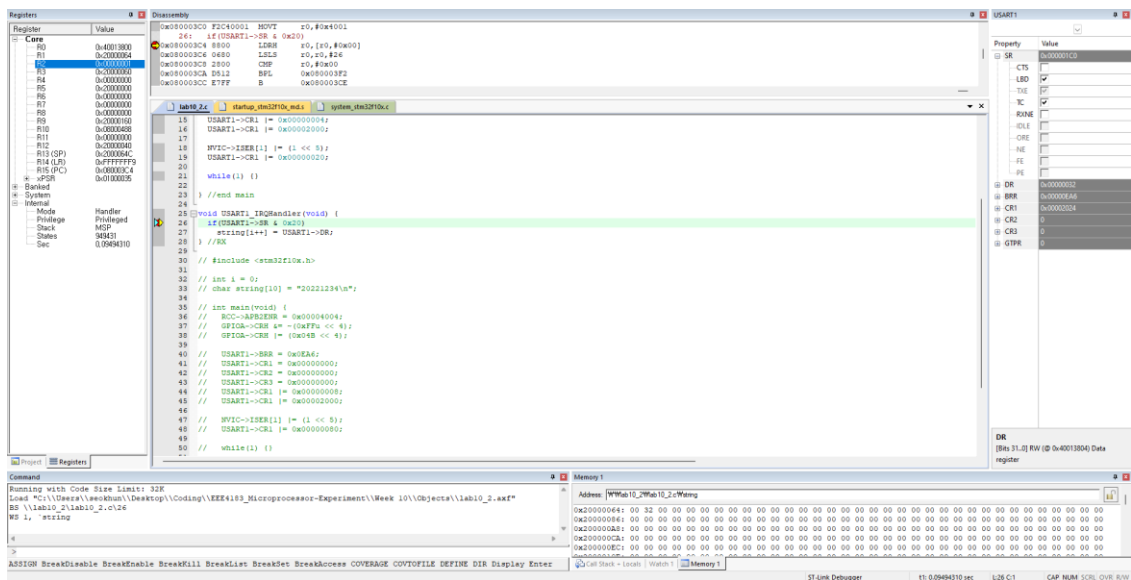


그림 28 데이터를 수신한 Rx 모듈

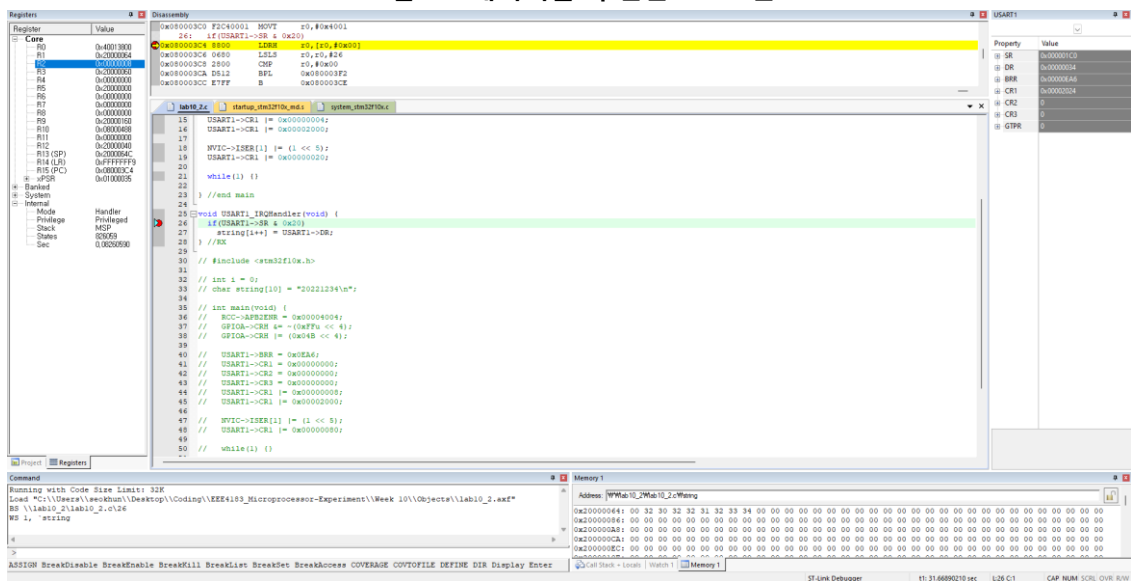


그림 29 전체 데이터가 들어온 Rx 모듈

먼저 [그림 26]와 같이 데이터가 들어와 인터럽트가 발생하기 전에 Rx 모듈은 인터럽트를 기다리고 있다. 이때 Tx 모듈이 동작해 [그림 27]과 같이 데이터가 전송되면 Rx 모듈에서 인터럽트가 발생하게 된다. 이때 Rx 모듈의 USART1->SR 레지스터를 보면 RXNE bit가 1로 set되어 있는데 즉 receive data is ready to read라는 뜻으로 데이터가 들어와서 읽을 준비가 완료되었다는 뜻이다. 따라서 [그림 28]과 같이 USART1->DR 레지스터의 0x32라는 데이터를 읽어서 string에 저장한 것을 확인할 수 있다. 그리고 마지막으로 [그림 29]과 같이 전체 데이터의 전송이 완료된 후에는 전송한 데이터 20221234가 잘 저장되어 있는 것을 확인할 수 있다.

다만 이때 Rx 모듈의 string 공간의 메모리를 살펴보면 20221234라는 데이터 앞에 0이 저장되어 있는 것을 확인할 수 있다. 정확한 원인을 파악할 수는 없지만 다음과 같이 생각해볼 수 있다. 우선 string의 앞부분에 0이라는 데이터가 저장되었다는 것은 인터럽트 핸들러로 들어갔고 RXNE bit도 1로 set되어 있었으며 그때 Rx 모듈의 DR 레지스터에는 0이라는 값이 저장되어 있었다. 즉 Tx 모듈에서 실제로 데이터를 전송하지는 않았지만 어떠한 이유에서 인지 Rx 모듈에서 데이터가 수신되었다고 판단하여 RXNE bit로 1로 set하였고 따라서 0이라는 데이터를 읽어서 string에 저장하고 i가 증가된 것을 알 수 있다.

**STEP 10: 각 모듈에서 USART1의 ISR은 어떻게 반복적으로 수행되어 string[]의 데이터를 송신하고 수신하는지 분석해보자. 확인을 위해 break points를 어떻게 설정 했는지와 확인한 내용에 대해 설명한다.**

먼저 Rx 모듈에서 main함수의 while문에서 인터럽트가 발생할 때까지 기다리고 있다. 그리고 Tx 모듈에서는 TDR이 비어있어 데이터를 전송할 수 있으면 인터럽트가 발생한다. Tx 모듈에서 인터럽트가 발생하면 인터럽트 핸들러로 들어와 다음 보내야 할 string의 데이터가 'Wn'인지 확인한다. 만약 'Wn'이면 string의 데이터를 다 전송하였으므로 Tx 모듈의 인터럽트가 발생하지 않도록 해준다. 그리고 다음 보낼 string 데이터가 'Wn'이 아니라면 USART1->DR에 전송할 데이터를 저장한다. 그리고 shift register를 통해 Tx의 데이터가 Rx로 전송된다. 그러면 Rx 모듈에서 RXNE bit가 set되면서 인터럽트가 발생한다. 그리고 Rx 모듈에서는 USART1->DR에 저장되어 있는 송신한 데이터를 string의 i번째 공간에 저장한다. 그리고 마지막으로 i를 증가시켜준다. 이러한 과정에 따라 Tx에서 데이터를 송신할 준비가 되면 Rx 모듈에서 데이터가 수신되면 인터럽트가 발생하며 데이

터가 전송된다. 이를 확인하기 위해서는 위 코드에서 Rx 모듈과 Tx 모듈 모두에서 인터럽트 핸들러를 들어오고 바로 다음 명령어인 line 26에서 break point를 걸어서 인터럽트가 발생하는 것을 확인하였다.

**STEP 11: Program 10.1에 비해 어떤 측면에서 효율적으로 데이터 전송이 이뤄진다고 볼 수 있는가? 프로그램을 비교하며 구체적으로 설명해보자.**

프로그램 10.1에 비해서 CPU의 자원 사용 측면에서 효율적으로 데이터 전송이 이뤄진다고 볼 수 있다. 왜냐하면 프로그램 10.1의 경우는 Rx 모듈에서 데이터가 들어왔는지 무한루프를 돌면서 계속해서 확인해주었다. 즉 CPU가 다른 작업을 할 수 없었다. 하지만 프로그램 10.2에서는 Rx 모듈에 데이터가 들어왔을 때만 인터럽트가 발생하여 들어온 데이터를 처리하고 다시 원래의 흐름으로 돌아오기 때문에 필요하다면 CPU가 다른 작업을 수행할 수 있다. 또한 Tx 모듈에서는 프로그램 10.1과 10.2의 차이는 크지 않지만 프로그램 10.1은 데이터를 전부 전송하기 전까지는 다른 작업을 수행할 수 없는데 반해 프로그램 10.2는 인터럽트에 의해 동작하기 때문에 USART의 느린 속도에 맞춰서 CPU가 작업을 수행하지 않고 USART의 동작을 기다리는 동안 다른 작업을 할 수 있다. 따라서 Tx와 Rx 모듈 전부 다 인터럽트를 사용하는 방식이 CPU가 USART의 동작을 기다리지 않고 필요할 때만 USART 동작을 수행하면 되기 때문에 더 효율적으로 CPU를 활용할 수 있다.

### 3) 실험 3

**STEP 12: 여기서부터는 Program 10.3을 이용하여 송신과 수신을 모두 처리할 수 있는 방식(full duplex)에 대해 살펴본다. 즉 Program 10.2의 Tx module과 Rx module의 기능을 결합하여 수행한다.**

연결된 두 보드에서 송신 시점을 결정하기 위해 Program 8.1에서 사용했던 USER button을 이용한다.

```

1  #include <stm32f10x.h>
2
3  int txcnt = 0, rxcnt = 0;
4  char txstring[10], rxstring[10] = "20221234\n";
5
6  int main(void) {
7      RCC->APB2ENR = 0x00004015;
8      GPIOA->CRH &= ~(0xFFu << 4);
9      GPIOA->CRH |= (0x04B << 4);
10     GPIOC->CRH = 0x00800000;
11
12     USART1->BRR = 0x0EA6;
13     USART1->CR1 = 0x00000000;
14     USART1->CR2 = 0x00000000;
15     USART1->CR3 = 0x00000000;
16     USART1->CR1 |= 0x0000000C;
17     USART1->CR1 |= 0x00002000;
18
19     NVIC->ISER[1] = (1 << 5);
20
21     EXTI->FTSR = 0x2000;
22     EXTI->IMR = 0x2000;
23     AFIO->EXTICR[3] = 0x20;
24     NVIC->ISER[1] = (1 << 8);
25
26     USART1->CR1 |= 0x00000020;
27
28     while (1) {}
29
30 } //end main
31
32 void USART1_IRQHandler(void) {
33     if(USART1->SR & 0x80) {
34         if(txstring[txcnt] == '\n')
35             USART1->CR1 &= ~0x80; //DISABLE Tx interrupt
36         else
37             USART1->DR = txstring[txcnt++];
38     }
39     if(USART1->SR & 0x20) {
40         rxstring[rxcnt++] = USART1->DR;
41     }
42 }
43
44 void EXTI15_10_IRQHandler(void) {
45     USART1->CR1 |= 0x00000080;
46     EXTI->PR = 0x2000;
47 }

```

그림 30 직접 작성한 lab10\_3.c 코드

STEP 13: STEP 4의 기존 연결에 다음 wire 연결을 추가한다.

- Tx module의 CN10의 33번 핀(PA10)을 Rx module의 CN10의 21번 핀(PA9)에 연결한다.

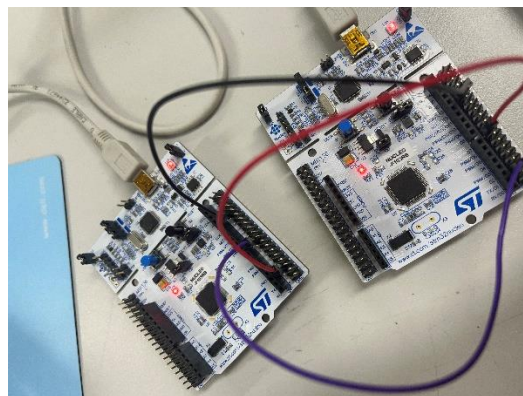


그림 31 실험 3을 위한 보드 연결

#### STEP 14: 프로그램의 전체적인 구조를 살펴보자.

- Lines 7의 설정에 대해 해석해보자.
- Line 16의 설정에 대해 해석해보자.
- Lines 21-24의 설정은 무엇을 위함인가?
- Lines 32-42의 ISR에서 수행되는 내용을 파악해보자.
- Lines 44-47의 ISR은 USART1의 동작과 어떻게 연관되는가?

먼저 line 7의 `RCC->APB2ENR = 0x00004015;` 를 통해서 `RCC->APB2ENR` 레지스터의 0번, 2번, 4번, 14번 bit를 1로 set해주었다. 이는 실험에서 사용할 AFIO, GPIOA, GPIOC, USART1을 enable해준 것이다. 다음으로 line 16의 `USART1->CR1 |= 0x0000000C;` 를 통해 `USART1->CR1` 레지스터의 2,3번 bit를 1로 set하였는데 이를 통해 transmitter와 receiver를 모두 enable해주었다. 즉 이전 실험들과 다르게 Tx와 Rx 모두 사용하는 full duplex로 사용하기 위한 설정이다. 다음으로 line 21의 `EXTI->FTSR = 0x2000;` 을 통해 EXTI13이 falling edge에서 trigger되도록 하고 line 22의 `EXTI->IMR = 0x2000;` 을 통해 EXTI13에서 interrupt가 발생할 수 있도록 해주었다. Line 23의 `AFIO->EXTICR[3] = 0x20;` 을 통해서 PC13을 EXTI13으로 연결하였고 `NVIC->ISER[1] = 0x00000020;` 을 통해 EXTI15\_10 인터럽트가 발생할 수 있도록 허용해주었다.

다음으로 lines 32-42는 USART1에 대한 인터럽트 핸들러인데 `if(USART1->SR & 0x80)`를 통해 TDR 레지스터가 empty 상태라면 해당 if문 내부를 수행하는데 `txstring[txcnt]`가 'Wn'으로 마지막 전송할 문자라면 Tx 모듈을 위한 인터럽트가 발생하지 못하도록 해주고 'Wn'이 아니라면 `txstring[txcnt]`의 데이터를 USART1->DR에 저장하여 데이터를 전송하고 `txcnt`값을 하나 증가시킨다. 그 다음에는 `if(USART1->SR = 0x20)`을 통해서 RXNE bit가 1로 set되어 있는지 확인하여 receive된 데이터가 있다면 `rxstring[rxcnt]`에 USART1->DR의 내용을 저장하고 `rxcnt`를 하나 증가시킨다.

마지막으로 line 44-47은 EXTI15\_10 인터럽트 핸들러인데 해당 인터럽트가 발생하였다면 `USART1->CR1 |= 0x00000080;` 을 수행하여 TXE 인터럽트가 발생할 수 있도록 허용해주고 `EXTI->PR = 0x2000`을 통해 EXTI13의 pending 상태를 해제해준다. 즉 해당 인터럽트 핸들러가 수행되면 USART1의 TXE 인터럽트가 발생할 수 있도록 해서 TDR이 비어있다면 인터럽트가 발생해 데이터를 송신할 수 있도록 해준다.

**STEP 15:** 두 보드에 Program 10.3을 장착하되 통신 결과의 확인을 위해 각각의 프로그램에서 line 4의 txstring[]에 저장되는 내용을 다르게 한다.

두 프로그램을 실행한 후 보드의 USER Button을 눌러 송신을 시작한다.

디버거를 사용하는 보드에서는 txstring의 데이터는 12342022로 설정하고 다른 보드에서는 txtring의 데이터를 20221234로 설정하였다. 따라서 디버거를 켜 보드에서 수신하는 데이터는 20221234가 될 것이다.

**STEP 16:** 프로그램을 중단한 후 각각의 보드에서 rxstring[]에 해당하는 메모리 내용을 살펴보고 txstring[]에 저장된 내용이 온전하게 전송되었는지를 확인한다.

USER Button를 누르는 시점의 선후를 변경하면서 위 동작여부를 확인한다.

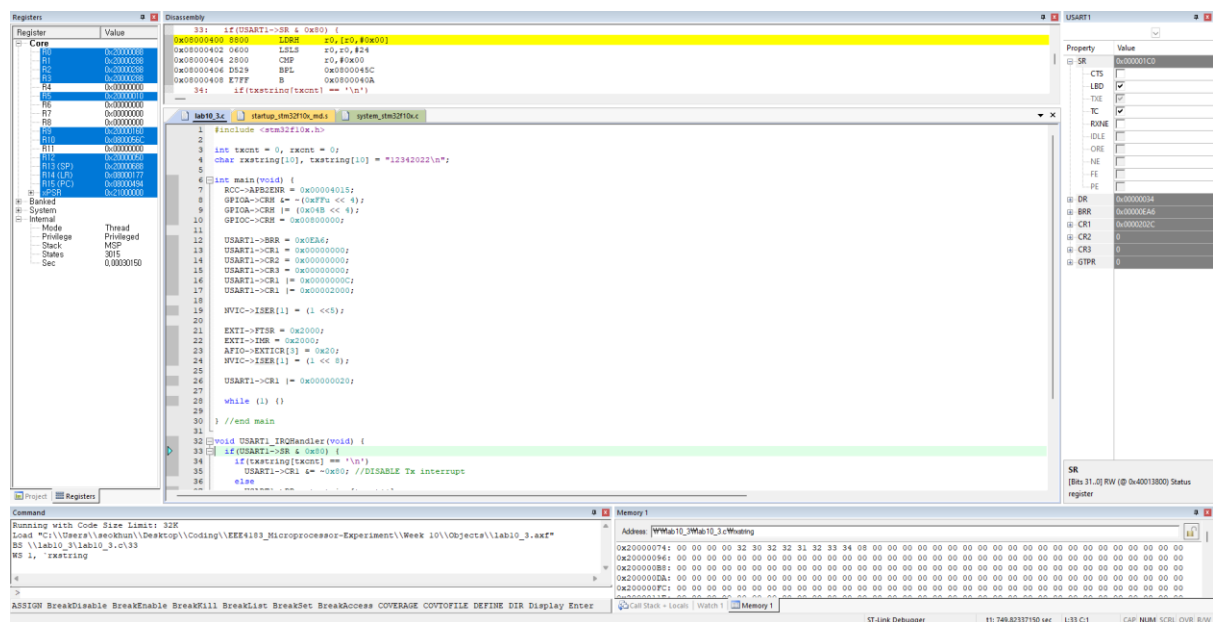


그림 32 USER Button을 먼저 눌렀을 때의 결과

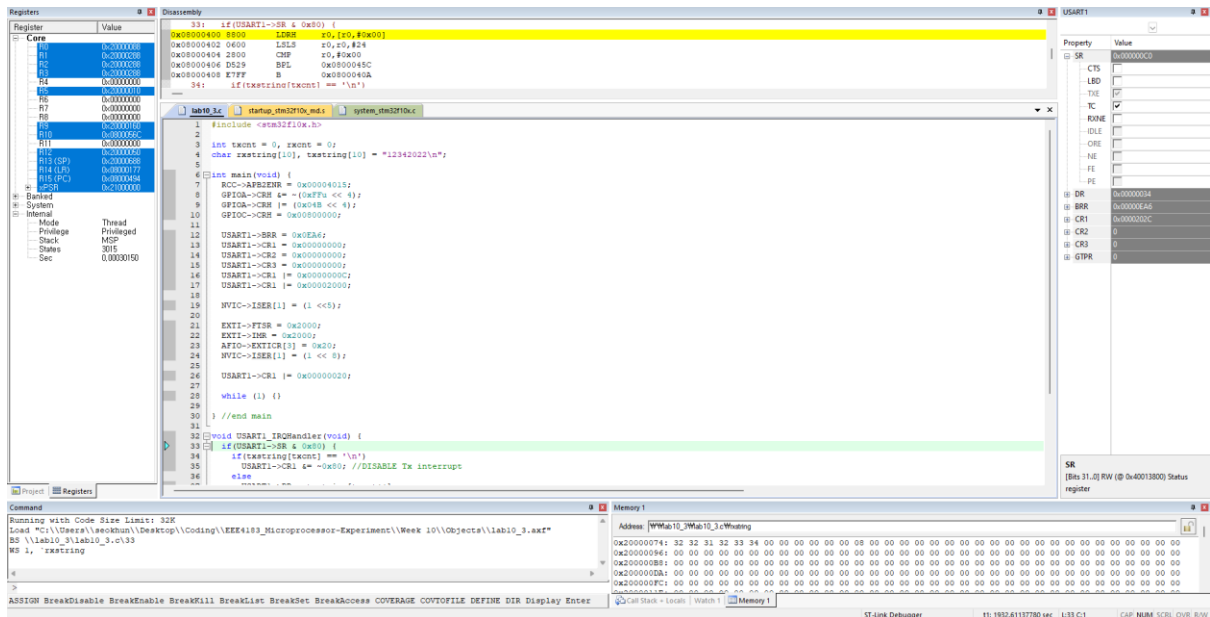


그림 33 USER Button을 늦게 눌렀을 때의 결과

먼저 [그림 32]는 디버거를 켜고 보드의 USER BUTTON을 먼저 눌렀을 때의 결과이다. 다른 보드에서 전송하는 데이터는 20221234로 데이터가 정상적으로 들어온 것을 확인할 수 있다. 다만 이전 실험 2에서와 마찬가지로 앞부분에 0이 몇 byte만큼 저장된 것을 확인할 수 있다. 이에 대한 정확한 원인은 파악할 수 없었지만 프로그램을 리셋하고 수행할 때 마다 0이 저장되는 byte가 다른 것으로 보아 Tx 모듈과 Rx 모듈의 타이밍 차이로 인한 문제로 예상된다. 하지만 이러한 문제를 제외하면 정상적으로 데이터가 수신되었다고 생각할 수 있다.

다음으로 [그림 33]은 디버거를 켜고 보드의 USER BUTTON을 늦게 눌렀을 때의 결과이다. 그림에서 보았을 때는 20221234의 전체 데이터 중에서 앞에 2byte인 20이 잘리고 221234의 데이터만 저장된 것을 볼 수 있다. 이는 receive 모드에서 데이터를 아직 읽기도 전에 Tx 모듈이 먼저 전송하여 RDR의 데이터를 덮어써서 이러한 문제가 발생한 것으로 확인할 수 있다. 이 또한 디버거를 사용할 때 Tx와 Rx 모듈 사이에 타이밍 차이로 인한 문제로 예상된다.

이상적인 결과로는 버튼을 누르는 순서에 상관없이 각 보드에서 txstring에 저장되어 있는 데이터가 다른 보드의 rxstring에 저장되어야 한다고 생각하는데 디버거를 사용할 때는 타이밍이 맞지 않아 이상적인 결과가 나오지 않는 것 처럼 보인다.

**STEP 17: USART1 IRQHandler에서 송신과 수신이 어떻게 인터럽트에 의해 수행되는지 분석해보자.**



먼저 송신의 경우에는 user button이 눌러 TXE 인터럽트가 발생할 수 있게 된 이후에 TDR이 비어있으면 인터럽트가 발생한다. TXE 인터럽트가 발생하면 TDR 레지스터에 송신하려는 txstring의 데이터를 저장하고 해당 데이터는 shift register를 통해 Rx 모듈로 데이터가 송신된다. 다음으로 수신은 shift register를 통해서 데이터가 들어오면 RXNE bit가 1로 set되고 인터럽트가 발생하게 된다. 그렇게 되면 RDR에서 데이터를 읽어들이어 rxstring에 데이터를 저장하게 된다. 즉 인터럽트에 의해 데이터가 송신, 수신이 된다.

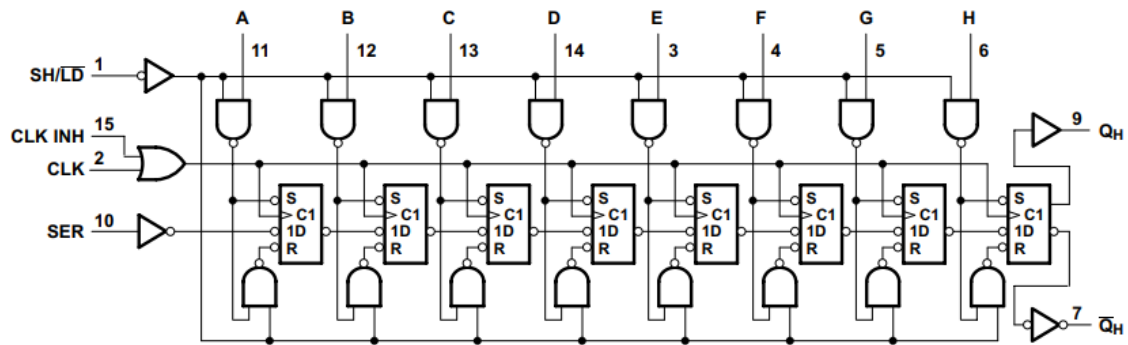
#### 4. Exercises

**1) USART 동작의 중심에는 shift register가 있다. 디지털논리회로 수업/실험을 통해 배운 shift register들의 datasheet를 통해 병렬데이터가 어떻게 직렬데이터로 (또는 그 반대로) 변환될 수 있는지 확인해보자. 특히 이 소자들에서 clock의 역할을 구체적으로 알아보자.**

아래 [그림 34]와 [그림 35]은 TI사의 SN74HC595라는 shift register 소자의 데이터 시트로 이를 통해서 USART 동작의 중심에 있는 직렬데이터를 병렬데이터로 변환해주는 shift register의 block diagram과 function table을 확인할 수 있다. USART에서 수신할 때는 수신된 직렬데이터가 shift register로 들어와 병렬데이터로 변환되어 RDR에 저장되어 데이터가 수신된다. 이의 가장 중요한 원리를 살펴보면 SRCLK 클럭이 high가 되면 SER로 들어온 데이터가 shift register의 가장 앞에 저장되고 다른 데이터는 모두 다음 레지스터로 넘어간다. 이를 통해 직렬데이터가 클럭의 rising edge 마다 하나씩 SER로 들어오고 그때마다 shift register에서 한칸씩 밀려가다가 전체 데이터가 다 들어오면 Q로 병렬데이터가 출력된다. 즉 클럭마다 데이터가 들어오고 데이터가 밀려나면서 직렬데이터가 병렬데이터로 변환되기 때문에 클럭이 매우 중요한 역할을 한다.



shift되어서 Q<sub>H</sub>로 데이터가 출력되게 된다. 즉 병렬데이터가 들어와 클락마다 하나씩 밀려나면서 데이터가 직렬로 출력되는 동작을 수행한다.



Pin numbers shown are for the D, DB, J, N, NS, PW and W packages.

그림 36 SN54HC165 block diagram

INPUTS			FUNCTION
SH/LD	CLK	CLK INH	
L	X	X	Parallel load
H	H	X	No change
H	X	H	No change
H	L	↑	Shift <sup>(1)</sup>
H	↑	L	Shift <sup>(1)</sup>

그림 37 SN54HC165 function table

즉 두종류의 shift register모두 클락에 따라서 shift register에서 하나씩 shift가 일어나서 직렬데이터를 병렬로 바꿔주거나 병렬데이터를 직렬데이터로 바꾸는 역할을 수행한다. 즉 클락은 shift register의 동작을 제어하는 가장 중요한 기능을 수행한다고 말 할 수 있다.

**2) Program 10.1에서 입력되는 문자들의 도착여부를 확인하는 과정과 출력한 문자의 출력 종료 여부를 확인하는 과정을 볼 수 있다. 이와 같은 방식을 polling이라고 한다. 이를 위해 어떠한 내부적인 장치들이 제공되고 있는가? 이 검토사항을 bit band aliasing과 연관지어 설명할 수 있는가?**

USART가 송신을 할 때는 SR레지스터의 TXE bit를 확인해서 1이면 데이터를 shift register로 이동하여 출력한 문자의 출력이 종료된 것을 알 수 있다. 또한 수신할 때는 SR레지스터의 RXNE bit를 확인해서 1이라면 shift register를 통해서 들어온 데이터가 RDR레지스터에 저장되어 문자가 도착하였다는 것을 확인할 수 있다. 이번 실험에서는 SR레지스터의 7번 bit또는 5번 bit를 masking해서 1인지 0인지 확인하였다. 이때 bit band aliasing된 영역을 활용하면 해당 bit만 정확히 확인하거나 수정할 수 있다. 따라서 polling을 할 때 해당 bit가 1인지 0이지만 확인하면 되기 때문에 편

하게 문자의 도착여부나 문자의 출력 종료 여부를 확인할 수 있다.

### **3) Program 10.2의 인터럽트에 의한 방식이 Program 10.1의 polling 방식에 비해 더 효율적이라고 여기는 이유는 무엇인가?**

인터럽트 방식의 경우는 polling 방식에 비해서 CPU의 사용 측면에서 효율적이다. 왜냐하면 polling방식은 계속해서 TXE bit나 RXNE bit를 확인해서 송신이나 수신을 할 준비를 마치면 확인하고 동작을 하는 방식이다. 따라서 아직 준비가 되지 않았더라도 계속해서 CPU가 확인을 해야한다. 반면 인터럽트에 의한 방식은 TXE bit나 RXNE bit가 1로 set되어 송신이나 수신할 준비를 마치면 인터럽트가 발생하여 송신이나 수신 동작을 할 수 있도록 알려준다. 따라서 송신이나 수신할 준비가 되지 않았다면 polling방식과 다르게 계속해서 확인할 필요 없이 CPU는 필요한 작업을 수행하면 된다. 따라서 인터럽트 방식이 polling방식에 비해 CPU의 사용측면에서 효율적이다.

### **4) 본문의 서술에 따르면 통신 중에 여러 유형의 error들이 발생할 수 있으며 이들은 status register 또는 interrupt를 통해 확인할 수 있음을 알 수 있다. 실험 과정은 통신 중 error를 가정하지 않고 있는데, 언제든 발생할 수 있는 이 errors는 일반적으로 어떻게 처리 (handling)되는지 알아보자.**

USART의 SR레지스터에서 0~3번 bit는 차례대로 parity error, framing error, noise error, overrun error가 발생하였을 때 set되어 에러가 발생하였음을 알 수 있다. 또한 USART->CR1 RXNEIE또는 PEIE bit를 1로 set해둔 경우 overrun error나 parity error가 발생하였을 때 인터럽트가 발생하도록 설정할 수도 있다. 이와 같이 SR레지스터의 bit를 확인하거나 인터럽트를 통해서 에러가 발생한 것을 확인할 수 있다. 이러한 에러는 통신 과정에서 언제든 발생할 수 있는데 일반적으로는 재전송을 요청하거나 error correction을 통해 에러를 수정하여 데이터를 받는다. 가장 중요한 것이 error detection인데 이를 위해서 USART에서는 parity를 활용할 수 있도록 해주고 있다. 보통 에러를 감지하면 송신자 측에 재전송을 요청하는데 이 신호 또한 에러가 발생하여 제대로 도착하지 않을 수 있기 때문에 안정화된 통신을 위해서는 송신자 측에서 타이머를 준비해 일정시간이 지나거나 재전송을 요청하는 신호가 도착하면 재전송을 하는 등 다양한 방법을 활용해서 에러를 처리

하도록 하고 있다.

**5) 직렬 데이터 통신과정에서 전송속도를 높이는 것이 유리하다. 전송속도를 높이는데 어떤 요소들이 제약사항으로 작용하는지 알아보자.**

전송속도를 높이면 같은 시간동안 더 많은 데이터를 전송할 수 있으므로 전송속도는 통신에서 매우 중요하다. 가장 먼저 통신에서 신호 대 잡음의 비율인 SNR이 낮으면 에러가 발생할 확률이 높기 때문에 전송속도를 낮춰 안정적인 전송이 되도록 해야 한다. 또한 전송 거리가 멀수록 신호가 더 많이 왜곡되거나 감쇠될 수 있기 때문에 전송속도를 낮춰 안정적인 전송이 되도록 해야 한다. 또한 통신에 사용하는 하드웨어의 스펙에 따라 전송속도 또한 제한된다. 그 외에도 전송속도를 높이는데 다양한 제약사항이 존재한다.

**6) 교재에서 설명한 RS-232C 방식 이외에 RS-422 방식도 데이터 통신에 자주 사용된다. 두 방식에 어떤 차이가 있는지 사용되는 buffer (driver/receiver)의 차이점을 중심으로 알아보자.**

RS-232C 방식의 경우는 다른 핀도 있지만 기본적으로 TXD, RXD, GND 총 3개의 핀을 중심으로 1대1 통신을 하는데 사용된다. 이때 버퍼에는 driver와 receiver역할을 둘 다 수행할 수 있는 하나의 버퍼를 이용해 데이터를 송신할 때는 버퍼에 데이터를 쓰고 수신할 때는 버퍼에서 데이터를 읽어들인다. 반면 RS-422 방식의 경우는 차등신호를 이용한 통신으로 기본적으로 TXD+, TXD-, RXD+, RXD-, GND 총 5개의 핀을 사용하며 다양한 수신자에게 데이터를 전송할 수 있는 multi drop이 사용 가능하다. 따라서 버퍼도 driver용 버퍼와 receiver용 버퍼를 따로 사용하여 송신할 때와 수신할 때 서로 다른 버퍼를 사용하기 때문에 multi point로 송신이 가능하다는 차이점이 있다.

## **5. 추가 실험**

### **1) USART Race Condition**

```

1  #include "stm32f10x.h" // Device header
2
3  char str1[] = "HAPPY";
4  char str2[] = "MicroProcessor";
5
6  void TIM1_UP_IRQHandler(void) {
7      NVIC_DisableIRQ(TIM2_IRQn);
8      for(int i=0; str1[i]!='\0'; i++) {
9          while(!(USART1->SR & USART_SR_TXE)); // Wait until transmit data register is empty
10         USART1->DR = str1[i]; // Send character
11     }
12     TIM1->SR &= ~TIM_SR_UIF; // Reset the update interrupt flag
13     NVIC_EnableIRQ(TIM2_IRQn);
14 }
15 void TIM2_IRQHandler(void) {
16     NVIC_DisableIRQ(TIM1_UP_IRQn);
17     for(int i=0; str2[i]!='\0'; i++) {
18         while(!(USART1->SR & USART_SR_TXE)); // Wait until transmit data register is empty
19         USART1->DR = str2[i]; // Send character
20     }
21     TIM2->SR &= ~TIM_SR_UIF; // Reset the update interrupt flag
22     NVIC_EnableIRQ(TIM1_UP_IRQn);
23 }
24 void USART1_Init(void) {
25     RCC->APB2ENR |= RCC_APB2ENR_USART1EN; // Enable USART1 clock
26     USART1->BRR = 0x1D4C; // Set baud rate to 9600 [assuming 72MHz clock]
27     USART1->CR1 |= USART_CR1_UE | USART_CR1_TE; // Enable USART, set TE (Transmitter enabled)
28 }
29 void TIM1_Init(void) {
30     RCC->APB2ENR |= RCC_APB2ENR_TIM1EN; // Enable TIM1 clock
31     TIM1->PSC = 7199; // Set prescaler [assuming 72MHz clock, 7200-1 for 1ms]
32     TIM1->ARR = 10 - 1; // Set auto reload register for 10ms
33     TIM1->DIER |= TIM_DIER_UIE; // Enable update interrupt
34     TIM1->CR1 |= TIM_CR1_CEN; // Start timer
35     NVIC_EnableIRQ(TIM1_UP_IRQn); // Enable TIM1 interrupt
36 }
37 void TIM2_Init(void) {
38     RCC->APB1ENR |= RCC_APB1ENR_TIM2EN; // Enable TIM2 clock
39     TIM2->PSC = 7199; // Set prescaler [assuming 72MHz clock, 7200-1 for 1ms]
40     TIM2->ARR = 13 - 1; // Set auto reload register for 13ms
41     TIM2->DIER |= TIM_DIER_UIE; // Enable update interrupt
42     TIM2->CR1 |= TIM_CR1_CEN; // Start timer
43     NVIC_EnableIRQ(TIM2_IRQn); // Enable TIM2 interrupt
44 }
45 int main() {
46     RCC->APB2ENR = 0x00004004;
47     GPIOA->CRH &= ~(0xFFu << 4);
48     GPIOA->CRH |= (0x04B << 4);
49
50     USART1_Init();
51     TIM1_Init();
52     TIM2_Init();
53
54     while(1) {}
55 }

```

그림 38 lab10\_4.c 코드

서로 다른 인터럽트 핸들러에서 USART를 이용해서 데이터를 송신하는 경우 인터럽트가 중첩되어 발생하는 경우 다른 인터럽트에서 전송하던 흐름을 침범하여 race condition이 발생하여 동기화 문제가 발생할 수 있다. 이를 방지하기 위해서 주어진 코드에서 USART를 사용하는 TIM인터럽트에 들어가면 다른 인터럽트가 발생하지 못하도록 disable해주었다. 그리고 USART를 이용해서 송신이 종료된 후에 다른 인터럽트가 발생할 수 있도록 enable하도록 코드를 작성하였다. 그리고 추가적으로 GPIOA를 초기화하는 코드를 추가하여 핀을 이용해서 통신이 가능하도록 위 [그림 38]과 같이 코드를 작성하였다. Rx 모듈의 코드는 실험 2의 코드에서 BRR값만 0x1D4C로 수정하여

[illegible]

확인해보면 MicroProcessor와 HAPPY가 합쳐지지 않고 잘 나뉘어서 저장된 것을 확인할 수 있다.

이를 통해 USART가 하나의 DR레지스터를 공유하기 때문에 race condition이 발생할 수 있는데 한 번에 하나의 USART만 동작할 수 있도록 다른 interrupt를 disable해주어서 동기화 문제를 해결하였다.

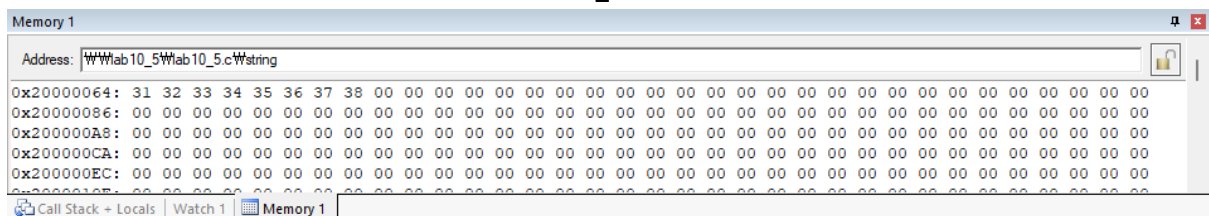
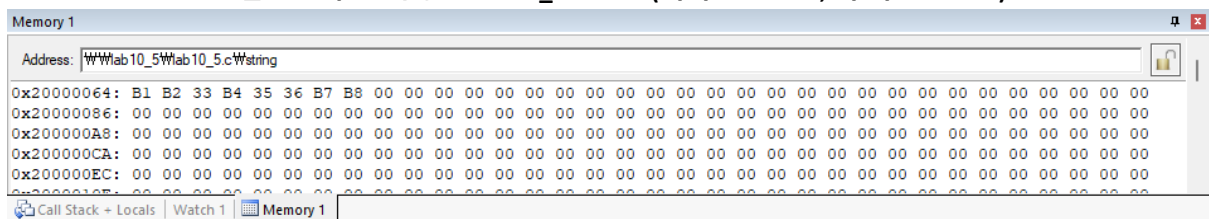
통신에서 error detection을 위한 가장 쉬운 방법으로 parity check가 있다. 우리가 사용하는 USART에서는 parity bit를 포함해서 데이터를 전송하도록 설정할 수 있다. USART의 CR1 레지스터의 10번 bit인 PCE를 1로 set해두면 parity bit를 포함해서 데이터를 송수신 한다. 그리고 CR1의 9번 bit인 PS bit를 통해 odd parity와 even parity를 선택할 수 있는데 0이면 even parity, 1이면 odd parity가 된다. 또한 CR1 레지스터의 8번 bit인 PEIE를 1로 set해두면 parity error가 발생하였을 때 interrupt가 발생하도록 설정할 수 있다. 실험 2의 코드를 활용하여 even parity bit를 포함해 데이터 송수신이 가능하도록 아래 [그림 40]과 같이 Tx 모듈과 Rx 모듈을 위한 코드를 작성하였다.

이 코드에서 송신한 데이터는 "12345678"인데 수신 측에서는 '1'을 0x31이 아니라 0xB1로 받았다. 왜냐하면 parity bit를 포함하여 0x31을 7bit로 나타내면 0110001인데 여기에 even parity를 포함하여 10110001이 송신되어 수신 측에서 10110001 = 0xB1로 수신하였다. 다른 데이터도 동일하게 parity bit가 포함되어 수신되었다. 이와 관련하여 실험을 진행하다 M을 1로 set하여 원래 8bit 데이터에 parity가 붙어 9bit가 되면 어떻게 데이터가 들어올지 궁금하여 약간 코드를 수정하여 한번 더 실험을 진행해보았다. 그 결과는 아래 [그림 42]와 같다. 9bit 데이터가 저장되는 것은 아니고 parity bit는 제거된 채로 데이터가 저장되는 것을 확인할 수 있었다.

```

1 #include <stm32f10x.h>
2
3 int i = 0;
4 char string[10];
5
6 int main(void) {
7     RCC->APB2ENR = 0x00004004;
8     GPIOA->CRH &= ~(0xFFu << 4);
9     GPIOA->CRH |= (0x04B << 4);
10
11     USART1->BRR = 0x0EA6;
12     USART1->CR1 = 0x00000400;
13     USART1->CR2 = 0x00000000;
14     USART1->CR3 = 0x00000000;
15     USART1->CR1 |= 0x00000004;
16     USART1->CR1 |= 0x00002000;
17
18     NVIC->ISER[1] |= (1 << 5);
19     USART1->CR1 |= 0x00000020;
20
21     while(1) {}
22 } //end main
23
24 void USART1_IRQHandler(void) {
25     if(USART1->SR & 0x80)
26     if(string[i] == '\n')
27         USART1->CR1 &= ~0x80; //disable Tx interrupt
28     else
29         USART1->DR = string[i++];
30 }

```



### 3) USART Abstraction



```

#define USART_EVENPARITY_SETTING 0x400

#define USART_ODDPARITY_SETTING 0x600

#define USART_WORDLENGTH_SETTING 0x1000


#define USART_PE_INTERRUPT 0x100
#define USART_TXE_INTERRUPT 0x80
#define USART_TC_INTERRUPT 0x40
#define USART_RXNE_INTERRUPT 0x20
#define USART_IDEL_INTERRUPT 0x10


void USART1_Init(int baud, int setting, int interrupt) {

    RCC->APB2ENR |= 0x00004004;

    GPIOA->CRH &= ~(0xFFu << 4);
    GPIOA->CRH |= (0x04B << 4);


    double temp = (double)4500000 / baud;
    USART1->BRR = temp * (1 << 4);


    USART1->CR1 |= setting;
    USART1->CR1 |= interrupt;
    if(interrupt) {
        NVIC->ISER[1] |= (1 << 5);
    }


    USART1->CR1 |= 0x0000200C;
}

```

```

int main(void) {

    RCC->APB2ENR = 0x00000011;

    GPIOC->CRH = 0x00800000;

    EXTI->FTSR = 0x2000;

    EXTI->IMR = 0x2000;

    AFIO->EXTICR[3] = 0x20;

    NVIC->ISER[1] = (1 << 8);


    USART1_Init(9600, NULL, USART_RXNE_INTERRUPT);


    while(1) {}

} //end main


void USART1_IRQHandler(void) {

    if(USART1->SR & 0x80) {

        if(txstring[txcnt] == '\n')

            USART1->CR1 &= ~0x80;

        else

            USART1->DR = txstring[txcnt++];

    }

    if(USART1->SR & 0x20) {

        rxstring[rxcnt++] = USART1->DR;

    }

}

```

```

void EXTI15_10_IRQHandler(void) {

    USART1->CR1 |= 0x00000080;

    EXTI->PR = 0x2000;

}

```

이번에도 지난번과 비슷하게 USART의 설정을 위 코드와 같이 함수로 추상화해보았다. USART의 기본적인 baud rate나 parity, word length에 대한 세팅값을 받고 인터럽트는 어떻게 처리할지를 함수의 인자로 받아 미리 #define을 해둔 값을 통해 USART의 레지스터를 초기화 하도록 함수를 작성하였다. 그리고 이 함수를 이용해서 main 함수 내부에서는 USART관련된 설정은 한 줄로 간소화하여 실험 3에서 사용한 코드를 작성해보았다. 아무래도 추상화하는 함수를 포함하다 보니 코드의 길이는 길어졌지만 나중에 재사용하기 편하고 사용할수록 코드의 길이는 감소하는 효과가 있을 것이고 가독성 또한 더 좋아진다는 장점이 있다. 위의 코드를 이용해서도 아래 [그림 43]처럼 실험 3과 동일한 동작을 수행하는 것을 확인할 수 있었다.

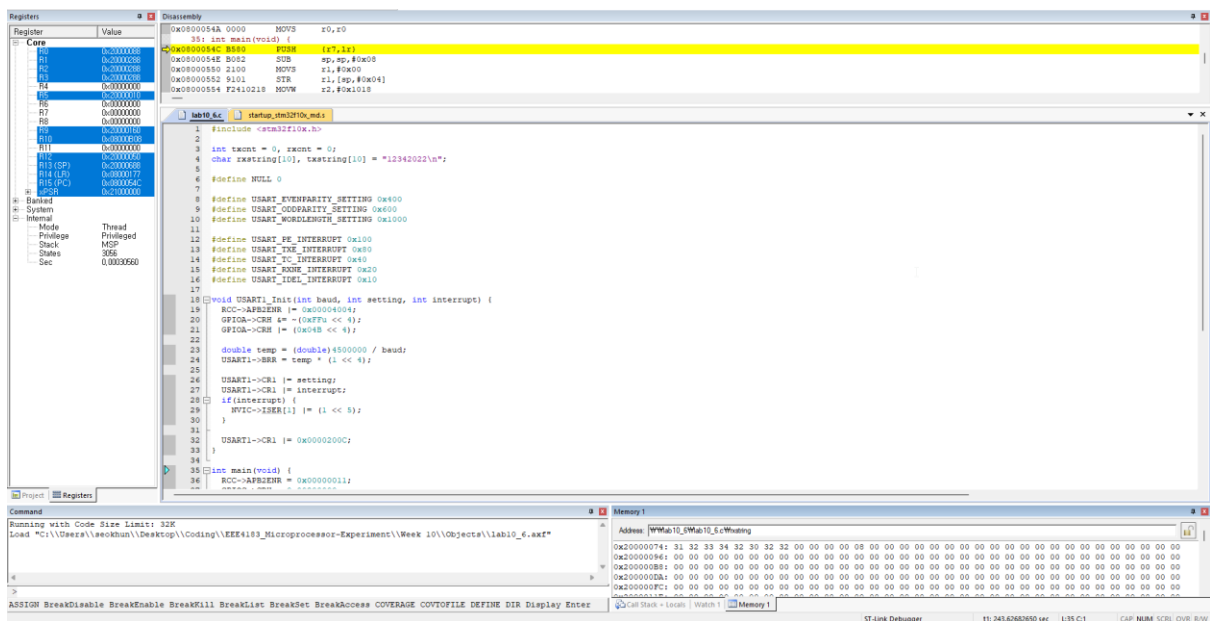


그림 43 lab10\_6 수행 결과

## 6. 결론

이번 주차 실험을 통해서 STM32F103RB 보드를 이용해서 통신을 하는 방법 중 가장 기본적인 USART 통신의 사용 방법을 익혀보았다. 직렬 통신의 가장 기본인 USART의 동작 원리를 알아보고 USART 통신에서 사용되는 다양한 신호들의 의미와 인터럽트가 발생하는 이유들에 대해 알아

보았다. 또한 USART 통신을 이용해 다른 보드와 직접 통신해 봄으로써 구체적인 통신의 과정을 살펴보았다. 또한 polling 방식과 interrupt 방식의 차이를 비교하고 인터럽트의 장점을 다시한번 느낄 수 있었다.

## 7. 참고문헌

서강대학교 전자공학과 (2023). 마이크로프로세서개론.

STMicroelectronics (2017). STM32F10xxx/20xxx/21xxx/L1xxx Cortex-M3 Programming Manual (Rev 6)

STMicroelectronics (2021). STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced ARM-based 32-bit MUCs Reference Manual (Rev 21).

Yiu, J. (2010). The Definitive Guide To The ARM Cortex-M3 (2<sup>nd</sup> Edition)

히언. (2021). EMBEDDED RECIPES.