

Chapter 114. EEPROM

ATmega328에는 3종류의 메모리가 포함되어 있다고 앞에서 설명하였다. 세 가지 메모리의 특징을 요약하면 표 1과 같다.

	플래시 메모리	SRAM	EEPROM
크기 (KByte)	32	2	1
주 용도	프로그램 저장	데이터 저장	데이터 저장
비휘발성	○	×	○
프로그램 실행 중 변경 가능	불가능	가능	가능
속도	중간	가장 빠름	가장 느림
수명	10,000회 쓰기	반영구적	100,000회 쓰기

표 1. ATmega328의 메모리

이들 메모리 중 플래시 메모리와 EEPROM은 그 특성이 유사하다. EEPROM은 사용자가 내용을 바이트 단위로 읽거나 쓸 수 있어서 SRAM처럼 사용이 가능하다. 하지만 읽기 동작은 SRAM보다 조금 느린 편이지만 쓰기 동작은 1 바이트를 쓸 때마다 수 밀리초의 지연 시간이 발생하므로 실시간으로 데이터를 저장하는 것은 불가능하다.

플래시 메모리는 EEPROM의 변형으로 사용자가 내용을 바이트 단위로 읽을 수 있다는 점은 EEPROM과 동일하다. 하지만 쓰기는 페이지 또는 섹터라고 불리는 블록 단위로만 수행될 수 있다. 블록의 크기는 메모리 소자나 메이커에 따라 차이가 있지만 64 바이트에서 128K 바이트 사이가 일반적으로 사용된다. 플래시 메모리 역시 EEPROM과 마찬가지로 실시간으로 데이터를 저장하는 것은 불가능하며, 블록 단위의 쓰기만 가능하므로 데이터를 저장하는 용도로는 적합하지 않다. 이런 단점에도 불구하고 플래시 메모리는 메모리 구조가 간단하여 EEPROM에 비해 대용량을 메모리 소자를 만드는데 적합하므로 프로그램 저장을 위한 프로그램 메모리로 사용되고 있다.

이 장에서는 ATmega328에 포함되어 있는 1K 바이트 크기의 EEPROM을 사용하는 방법에 대해 알아본다.

1. EEPROM 레지스터

EEPROM을 사용하기 위해 필요한 레지스터는 EEPROM 주소 레지스터, EEPROM 데이터 레지스터, EEPROM 제어 레지스터 등이 있다. ATmega328의 EEPROM은 1K 바이트 크기로 바이트 단위의 주소 레지스터 지정을 위해서는 10비트가 필요하다. 따라서 EEPROM 주소 레지스터 EEAR (EEPROM Address Register) 레지스터는 2바이트 크기를 갖는 레지스터이며, 10비트의 주소는 EEARH 레지스터의 하위 2바이트와 EEARL 레지스터의 8비트로 구성된다. EEAR 레지스터의 구조는 그림 1과 같다.

비트	15	14	13	12	11	10	9	8
EEARH	-	-	-	-	-	-	EEAR9	EEAR8
EEARL	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0
	7	6	5	4	3	2	1	0
읽기/쓰기	R	R	R	R	R	R	R/W	R/W
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초깃값	0	0	0	0	0	0	0	x
	x	x	x	x	x	x	x	x

그림 1. EEAR 레지스터 구조

EEPROM 데이터 레지스터인 EEDR (EEPROM Data Register) 레지스터는 EEPROM에서 읽어오거나 쓸 데이터를 저장하는 레지스터이다.

비트	7	6	5	4	3	2	1	0
	EEDR7..0							
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초깃값	0	0	0	0	0	0	0	0

그림 2. EEDR 레지스터 구조

EEPROM 제어 레지스터인 EECR (EEPROM Control Register) 레지스터는 EEPROM의 읽기와 쓰기 과정을 제어하는 레지스터로, EECR 레지스터의 구조는 그림 3과 같다.

비트	7	6	5	4	3	2	1	0
	-	-	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE
읽기/쓰기	R	R	R/W	R/W	R/W	R/W	R/W	R/W
초깃값	0	0	x	x	0	0	x	0

그림 3. EECR 레지스터 구조

비트	이름	설명
5	EEPM1	EEPROM Programming Mode : EEPE 비트를 세트한 경우 프로그래밍 동작을 결정한다. 쓰기와 지우기를 하나의 동작으로 수행할 수도 있고 분리하여 실행할 수도 있다. EEPM 값은 EEPE 비트가 세트되기 이전에 설정되어 있어야 한다. EEPMn (n = 0, 1) 비트의 값에 따른 프로그래밍 동작은 표 3과 같다.
4	EEPM0	
3	EERIE	EEPROM Ready Interrupt Enable : EEIRE 비트를 세트하는 경우 EEPROM 준비 인터럽트(EEPROM Ready Interrupt)를 활성화 시킨다. EEPROM 준비 인터럽트는 EEPE 비트가 클리어 될 때 발생한다.
2	EEMPE	EEPROM Master Write Enable : EEPE 비트가 세트되는 경우 실제로 EEPROM에 쓰기 동작이 발생할 지의 여부를 결정한다. EEMPE 비트가 클리어 상태인 경우 EEPROM에 쓰기 동작은 발생하지 않는다. 소프트웨어에 의해 EEMPE 비트가 세트된 경우에는 4 클록 이후 하드웨어에 의해 자동으로 클리어 된다.
1	EEPE	EEPROM Write Enable : EEPE 비트가 세트되면 쓰기 동작이 시작된다. EEAR 레지스터와 EEDR 레지스터에 주소와 데이터가 저장되어 있는 경우 EEPE 비트를 세트하면 EEPROM에 데이터를 기록하기 시작한다. 이 때 EEMPE 비트는 EEPE 비트가 세트되기 이전에 세트되어 있어야 실제 쓰기 동작이 발생한다.
0	EERE	EEPROM Read Enable : EERE 비트가 세트되면 읽기 동작이 시작된다. EEAR 레지스터에 주소가 저장되어 있는 경우 EEPROM의 해당 주소로부터 데이터를 읽어온다. EEPROM에 쓰기 동작이 진행 중인 경우 읽기를 수행할 수 없으므로 EEPE 비트가 클리어 되었는지 확인하고 읽기 동작을 수행하여야 한다.

표 2. EECR 레지스터 비트

EEPM1	EEPM0	프로그래밍 시간	동작
0	0	3.4ms	지우고 쓰기
0	1	1.8ms	지우기
1	0	1.8ms	쓰기
1	1	-	예약됨

표 3. EEPmn (n = 0, 1) 비트의 값에 따른 프로그래밍 동작

EEPROM에 데이터를 기록하는 전형적인 순서는 다음과 같다.

- ① EEPE가 0(zero)이 될 때까지 기다린다.
- ② SPMCSR 레지스터의 SPEN 비트가 0(zero)이 될 때까지 기다린다.
- ③ EEAR 레지스터에 EEPROM의 주소를 쓴다.
- ④ EEDR 레지스터에 EEPROM에 기록할 데이터를 쓴다.
- ⑤ EEMPE 비트에 1을 쓴다. 이 때 EEPE 비트는 0이어야 한다.
- ⑥ EEMPE 비트가 설정된 이후 4 클록 이내에 EEPE 비트에 1을 쓴다.

단계 2에서 SPMCSR 레지스터는 Store Program Memory Control and Status Register를 가리키며, SPEN 비트는 Store Program Memory 비트를 가리킨다. EEPROM은 CPU가 플래시 메모리에 데이터를 기록하는 동안에는 사용할 수 없으므로 EEPROM의 쓰기 동작 이전에 CPU에 의한 플래시 메모리 쓰기 동작이 종료되었는지 검사하여야 한다. CPU에 의한 유일한 플래시 메모리 쓰기 동작은 부트로더에 의한 플래시 메모리 쓰기 동작으로 단계 2는 부트로더에 의해 CPU가 플래시 메모리에 쓰기 동작을 수행하고 있는지의 여부를 판단하기 위해 사용된다.

단계 6은 실제 쓰기 동작이 발생하는 단계이지만 한 가지 주의할 점은 EEMPE 비트가 소프트웨어 의해 세트되고 4 클록이 경과하면 하드웨어에 의해 자동으로 클리어된다는 점이다. 따라서 단계 5와 단계 6 사이에 인터럽트가 발생하면 EEPROM에 대한 쓰기 동작은 실행되지 않을 수 있다. 따라서 EEPROM에 대한 쓰기 동작을 실행하기 이전에는 전역 인터럽트 허용 비트를 클리어 해서 인터럽트 발생을 금지시키는 것이 안전하다. 쓰기 시간이 경과하면 EEPE 비트는 하드웨어에 의해 클리어 된다. 따라서 다음 쓰기 동작을 수행하기 전에는 EEPE 비트가 클리어 되었는지 확인하여야 한다.

코드 1은 전형적인 EEPROM에 대한 쓰기 동작을 나타낸 예로 전역적으로 인터럽트 발생이 금지되어 있으며, 부트로더에 의한 플래시 메모리 쓰기 동작은 없는 것으로 가정하고 있다.

코드 1. EEPROM 쓰기 함수

```

void EEPROM_write(unsigned int address, unsigned char data)
{
    while(EECR & (1 << EEPE));    // EEPE 비트가 클리어 될 때까지 대기

    EEAR = address;                // EEPROM의 주소 설정
    EEDR = data;                  // EEPROM에 기록할 데이터 설정

    EECR |= (1 << EEMPE);         // EEMPE 비트 세트

    EECR |= (1 << EEPE);         // EEPE 비트 세트. 실제 쓰기 동작 시작
}

```

코드 2는 전형적인 EEPROM에서의 데이터 읽기 동작을 나타낸 예로, 쓰기과 마찬가지로 전역적으로 인터럽트 발생이 금지되어 있는 것으로 가정하고 있다.

코드 2. EEPROM 읽기 함수

```

unsigned char EEPROM_read(unsigned int address)
{
    while(EECR & (1 << EEPE));    // EEPE 비트가 클리어 될 때까지 대기

    EEAR = address;                // EEPROM의 주소 설정

    EECR |= (1 << EERE);         // EERE 비트 세트. 실제 읽기 동작 시작

    return EEDR;
}

```

2. EEPROM 라이브러리 - GNU 라이브러리

EEPROM 사용은 레지스터를 통해서도 가능하지만 아트멜 스튜디오에서 사용하는 라이브러리인 avr-libc 라이브러리를 통해서도 사용 가능하며 라이브러리를 통해 사용하는 것이 일반적

이다. 라이브러리 사용을 위해서는 먼저 해당 헤더 파일을 포함시켜야 한다.

```
#include <avr/eeprom.h>
```

헤더 파일에는 EEPROM의 데이터 읽기와 쓰기를 위한 함수들이 선언되어 있다. 함수들은 크게 읽기(read), 쓰기(write), 갱신(update)의 3가지 그룹으로 나뉜다. 각 그룹에는 데이터 유형에 따라, 즉, 한 번에 읽거나 쓸 EEPROM의 바이트 수에 따라 5가지 함수가 정의되어 있다.

```
uint8_t eeprom_read_byte (const uint8_t *__p);
uint16_t eeprom_read_word (const uint16_t *__p);
uint32_t eeprom_read_dword (const uint32_t *__p);
float eeprom_read_float (const float *__p);
void eeprom_read_block (void *__flash, const void *__eeprom, size_t __n);

void eeprom_write_byte (uint8_t *__p, uint8_t __value);
void eeprom_write_word (uint16_t *__p, uint16_t __value);
void eeprom_write_dword (uint32_t *__p, uint32_t __value);
void eeprom_write_float (float *__p, float __value);
void eeprom_write_block (const void *__flash, void *__eeprom, size_t __n);

void eeprom_update_byte (uint8_t *__p, uint8_t __value);
void eeprom_update_word (uint16_t *__p, uint16_t __value);
void eeprom_update_dword (uint32_t *__p, uint32_t __value);
void eeprom_update_float (float *__p, float __value);
void eeprom_update_block (const void *__flash, void *__eeprom, size_t __n);
```

쓰기와 갱신의 차이는 데이터를 기록하기 전에 기존에 저장된 값의 확인 유무에 있다. EEPROM은 비휘발성 메모리로 최대 100,000번의 기록이 가능하다. 잦은 데이터 기록은 EEPROM의 수명을 단축시키는 결과를 가져오므로 가능한 쓰기 횟수를 줄이기 위해 갱신 함수는 현재 기록된 값과 쓰고자 하는 값이 동일한 경우에는 데이터를 쓰지 않는다. 데이터를 읽는 속도는 쓰는 속도에 비해 빠르므로 쓰기와 갱신의 속도 차이는 거의 나지 않는다. 쓰기

함수는 이전 버전과의 호환성을 위해 남겨져 있으므로 갱신 함수 사용이 추천되고 있다. EEPROM에서 값을 읽어오거나 쓸 때 주의할 점은 주소를 포인터 형식으로 전달하여야 한다는 점이다. EEPROM의 주소는 정수형으로 주어지지만 이를 사용하고자 하는 함수의 매개변수 형식에 맞게 캐스팅하여 사용하면 된다. 코드 3은 EEPROM에 데이터를 쓰고 이를 읽어오는 예는 코드 3과 같다. float 형 데이터를 읽고 쓰는 함수 역시 존재하지만 마이크로컨트롤러에서 float 연산은 많은 메모리와 시간을 소요하므로 추천되지 않으므로 코드 3에서는 포함시키지 않았다. 블록 단위의 데이터 읽기와 쓰기는 void 형 포인터를 사용한다는 점도 눈여겨보아야 한다. void 형 포인터는 바이트 단위의 데이터를 읽고 쓰기 위해 사용되며 데이터의 의미보다는 데이터의 크기만이 중요한 경우에 사용된다.

코드 3. EEPROM 읽기 - avr-libc 라이브러리

```
#include <avr/io.h>
#include <avr/eeprom.h>
#include "UART.h"

int main(void)
{
    // EEPROM에 쓸 데이터
    uint8_t dataByte = 1;
    uint16_t dataWord = 2;
    uint32_t dataDWord = 3;
    char dataString[] = "ABCDE";

    // EEPROM의 주소
    int addressByte = 0, addressWord = 10, addressDWord = 20, addressString = 30;

    UART_INIT();                // UART 통신 초기화

    // 데이터 쓰기
    eeprom_update_byte ( (uint8_t *)addressByte, dataByte * 10);
    eeprom_update_word ( (uint16_t *)addressWord, dataWord * 10);
    eeprom_update_dword ( (uint32_t *)addressDWord, dataDWord * 10);
    eeprom_update_block ( (void *)dataString, (void *)addressString, 5);
```

```
// 데이터 읽기
dataByte = eeprom_read_byte ( (uint8_t *)addressByte );
dataWord = eeprom_read_word ( (uint16_t *)addressWord );
dataDWord = eeprom_read_dword ( (uint32_t *)addressDWord );
eeprom_read_block ( (void *)dataString, (void *)addressString, 5);

// UART 통신으로 데이터 전송
UART_printString("Byte      : ");           // 1 바이트 데이터
UART_print8bitNumber(dataByte);
UART_printString("\n");
UART_printString("Word      : ");           // 2 바이트 데이터
UART_print16bitNumber(dataWord);
UART_printString("\n");
UART_printString("Double Word : ");       // 4 바이트 데이터
UART_print32bitNumber(dataDWord);
UART_printString("\n");
UART_printString("String    : ");           // 문자열 데이터
UART_printString(dataString);
UART_printString("\n");

while(1);

return 0;
}
```

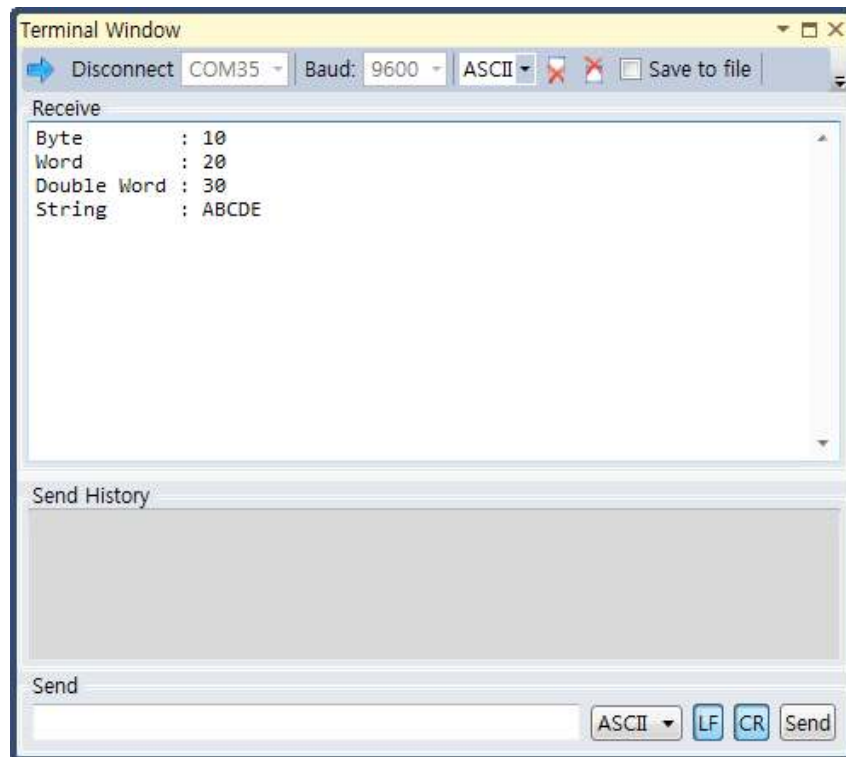


그림 4. 코드 3 실행 결과

EEPROM에 데이터를 쓰는 경우 1바이트 이상의 크기를 가지는 데이터를 쓰는 경우에는 주소 계산에 주의하여야 한다. EEPROM은 바이트 단위로 주소가 주어지므로 2바이트 크기를 가지는 uint16_t 형식의 데이터를 연속해서 쓰는 경우에는 2만큼 차이가 나는 번지를 지정하여야 한다. 하지만 이런 불편한 점은 포인터 연산을 통해 간단히 해결할 수 있다. EEPROM에 데이터를 쓰거나 읽는 경우 주소는 포인터 형으로 변환하여 전달하여야 한다는 점을 기억하는가? uint16_t 형식의 포인터를 생각해 보자. 포인터는 포인터 값이 가리키는 메모리 주소에 저장된 값의 형식을 나타낸다. 즉, uint16_t *p; 에서 p 값이 가리키는 메모리 번지에는 2바이트 크기의 uint16_t 형식의 데이터가 p 번지와 그 다음 번지에 저장되어 있음을 나타낸다. 한 가지 주의할 점은 p가 100번지를 가리키고 있다고 가정하였을 때 p + 1은 101번지가 아니라 uint16_t 형식의 값이 저장된 이후의 번지인 102번지를 가리킨다는 점이다. 코드 4는 1바이트 크기를 갖는 uint8_t 형식과 2바이트 크기를 갖는 uint16_t 형식의 값이 저장되었을 경우 동일한 포인터 연산에서 자동으로 바이트 수에 맞게 주소가 계산되는 예를 보여준다.

코드 3. 포인터 연산

```
#include <avr/io.h>
#include <avr/eeprom.h>
#include "UART.h"

int main(void)
{
    UART_INIT();          // UART 통신 초기화

    uint8_t *p1, *p2;
    p1 = 0;
    p2 = p1 + 1;          // p1 번지에 uint8_t 데이터 형의 바이트 수를 더한 값 = 1

    UART_printString("P1 : ");
    UART_print16bitNumber((uint16_t)p1);
    UART_printString("\n");
    UART_printString("P2 : ");
    UART_print16bitNumber((uint16_t)p2);
    UART_printString("\n");

    uint16_t *p3, *p4;
    p3 = 0;
    p4 = p3 + 1;          // p3 번지에 uint16_t 데이터 형의 바이트 수를 더한 값 = 2

    UART_printString("P3 : ");
    UART_print16bitNumber((uint16_t)p3);
    UART_printString("\n");

    UART_printString("P4 : ");
    UART_print16bitNumber((uint16_t)p4);
    UART_printString("\n");

    while(1);

    return 0;
}
```

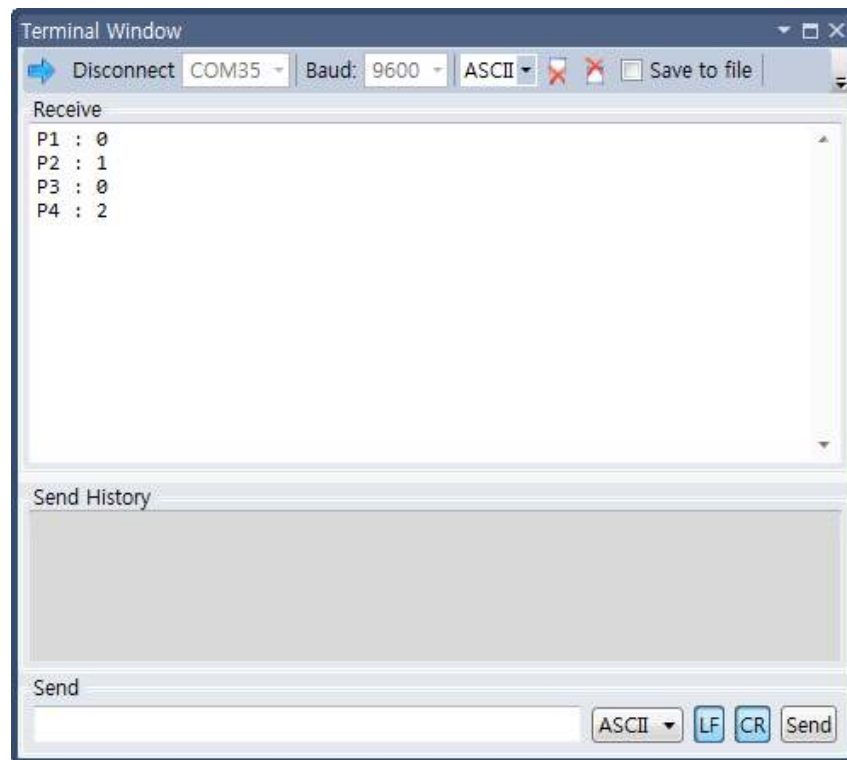


그림 5. 코드 4 실행 결과

3. EEPROM 라이브러리 - 아두이노

아두이노에는 EEPROM에 데이터를 읽고 쓸 수 있도록 EEPROM 라이브러리를 기본 라이브러리 중 하나로 포함하고 있다. EEPROM을 사용하기 위해서는 '프로젝트 → Add/Import Sketch Library → Core → EEPROM'을 선택하여 해당 헤더 파일을 포함시켜야 한다.

```
#include <EEPROM.h>
```

EEPROM.h 파일에는 EEPROMClass 클래스가 정의되어 있고 EEPROMClass의 객체인 EEPROM이 정의되어 있으므로 별도의 객체 생성 없이 'EEPROM'을 통해 데이터를 읽고 쓸 수 있다. EEPROM 클래스에는 바이트 단위의 읽기와 쓰기를 위한 두 개의 멤버 함수만이 정의되어 있으며 이들 함수는 avr-libc 라이브러리의 함수인 eeprom_read_byte와

EEPROM_write_byte 함수를 통해 구현되어 있다. 2바이트 이상의 데이터를 읽거나 쓰기 위해서는 별도의 함수를 작성하거나 avr-libc 라이브러리의 함수를 사용하여야 한다.

```
uint8_t EEPROMClass::read(int address)
{
    return eeprom_read_byte((unsigned char *) address);
}

void EEPROMClass::write(int address, uint8_t value)
{
    eeprom_write_byte((unsigned char *) address, value);
}
```

코드 4는 EEPROM의 0번에서 10번까지의 메모리에 값을 기록하고 이를 다시 읽어 들여 출력하는 예이다.

코드 5. EEPROM에 데이터 읽고 쓰기 - 아두이노 EEPROM 라이브러리

```
#include <EEPROM.h>

void setup(){
    Serial.begin(115200);           // 시리얼 통신 초기화
}

void loop(){
    for(int i = 0; i <= 10; i++){   // EEPROM에 값 쓰기
        EEPROM.write(i, i);        // write(주소, 값)
    }

    for(int i = 0; i <= 10; i++){   // EEPROM에서 값 읽기
        int value = EEPROM.read(i); // read(주소)
        Serial.println("Address : " + String(i) + "\t Value : " + value);
        //delay(50);
    }
}
```

```
while(true);  
}
```

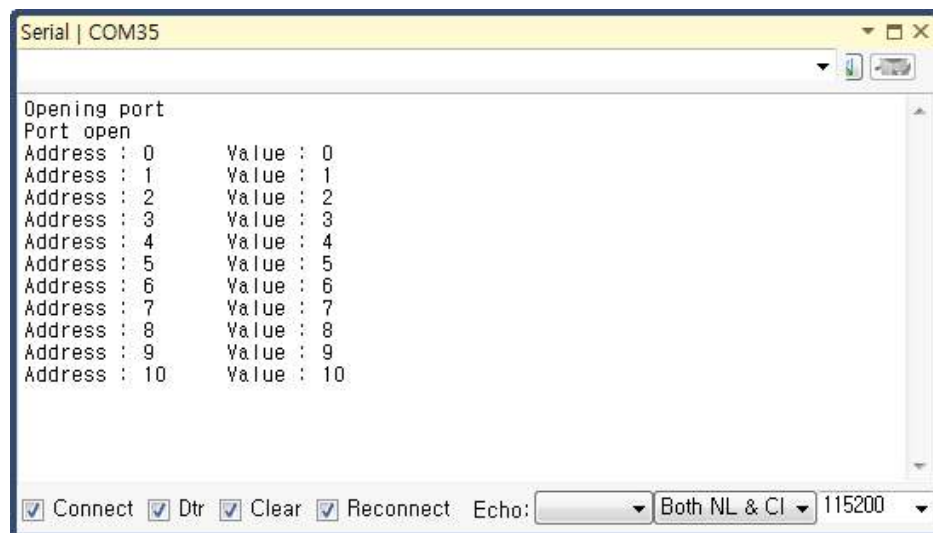


그림 6. 코드 5 실행 결과