

14. 라이브러리

1. 라이브러리

아두이노의 기본 함수들과 기본 클래스는 컴파일 과정에서 해당 헤더 파일이 자동으로 추가되므로 사용을 위해 별도로 추가해야 하는 코드는 없다. 하지만 이외의 기능을 사용하기 위해서는 C/C++에서와 마찬가지로 헤더 파일을 포함시켜야 한다. 예를 들어 LCD 장치를 사용하기 위해서는 'LiquidCrystal.h'를 포함시켜야 하고 EEPROM을 사용하기 위해서는 'EEPROM.h'를 포함시켜야 한다. 아두이노는 이처럼 다양한 장치들을 활용할 수 있는 함수나 클래스를 기능이나 용도에 따라 묶어 라이브러리로 제공하고 있다. 먼저 아두이노 프로그램과 함께 제공되는 기본 라이브러리들을 살펴보자. 기본 라이브러리들은 아두이노 프로그램이 설치된 디렉터리 아래 'libraries' 디렉터리에서 찾아볼 수 있다.

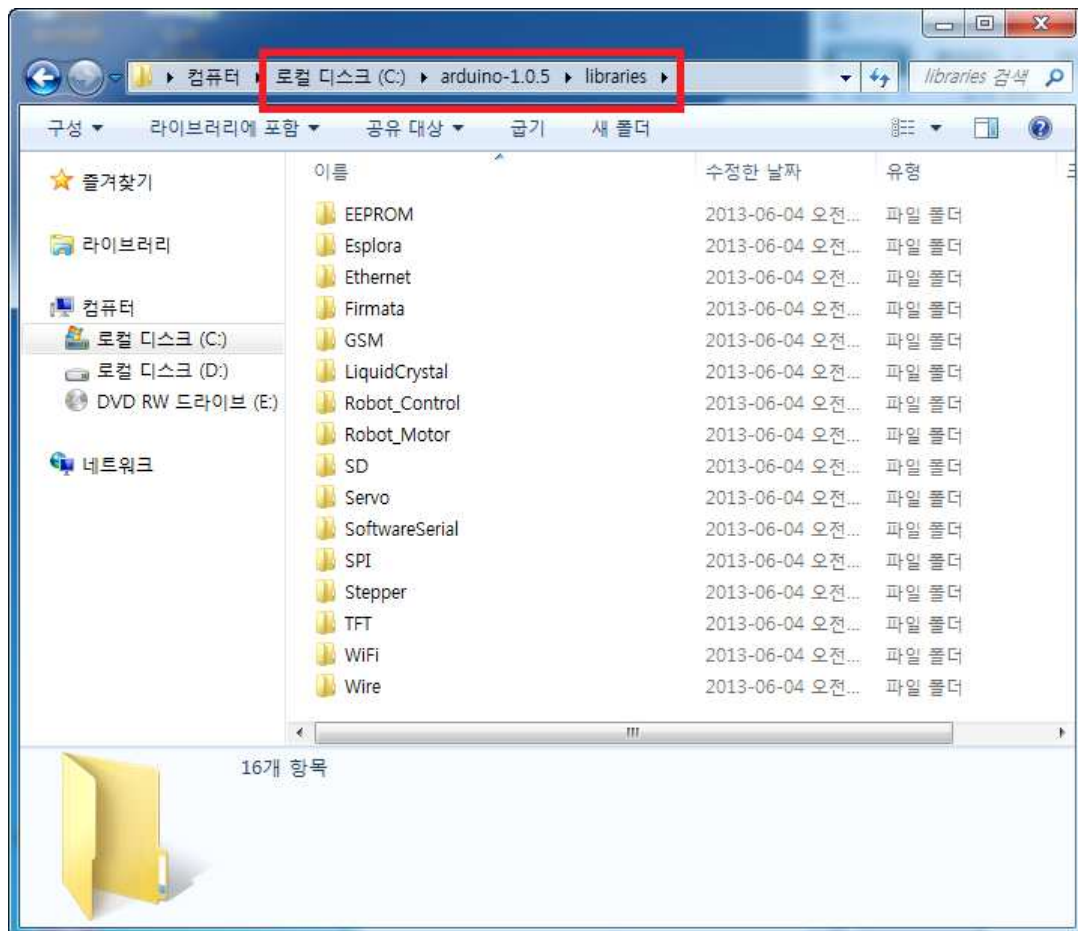


그림 1. 기본 라이브러리 설치 디렉터리

라이브러리는 개별적인 디렉터를 가지며 디렉터리의 이름은 라이브러리 이름과 동일하다. 개별 라이브러리 디렉터리 아래에는 해당 기능을 구현한 헤더(*.h) 파일과 소스(*.cpp) 파일이 위치하며 간단한 라이브러리의 경우 소스 파일 없이 헤더 파일만 존재하는 경우도 있다. 라이브러리를 이용한 예제들은 해당 라이브러리가 위치한 디렉터리 아래에 'examples'라는 이름의 디렉터리 아래에 위치한다. 아두이노 설치 디렉터리의 'libraries\EEPROM' 디렉터를 살펴보면 EEPROM 사용을 위한 클래스를 정의한 EEPROM.h와 EEPROM.cpp 파일이 존재하며 세 개의 예제가 디렉터리 별로 저장되어 있음을 확인할 수 있다. 라이브러리는 아두이노 프로그램에서도 확인 가능하다. '스케치 → 라이브러리 가져오기...' 메뉴를 선택하면 기본 라이브러리에 해당하는 메뉴를 확인할 수 있다.

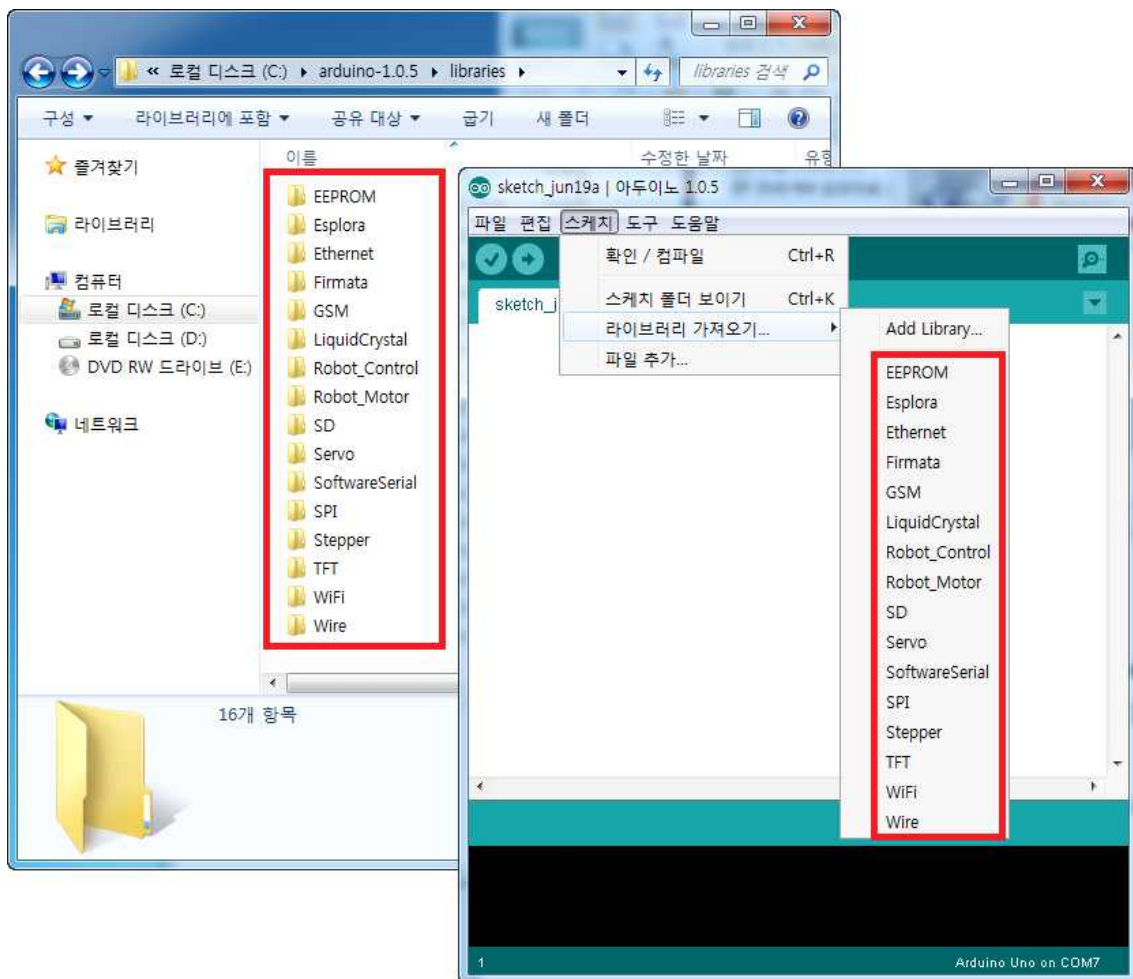


그림 2. 아두이노 프로그램의 라이브러리 가져오기 메뉴

라이브러리는 라이브러리 사용 방법을 보이는 예제가 'examples' 디렉터리에 포함되어 배포되는 것이 일반적이며 라이브러리에 포함된 예제 역시 아두이노 프로그램에서 확인할 수 있다. 아두이노 프로그램에서 '파일 → 예제' 메뉴에서 해당 라이브러리 이름의 메뉴를 선택하면 예제 목록을 확인할 수 있으며, 예제를 선택하면 아두이노 프로그램에서 새 창으로 열린다.

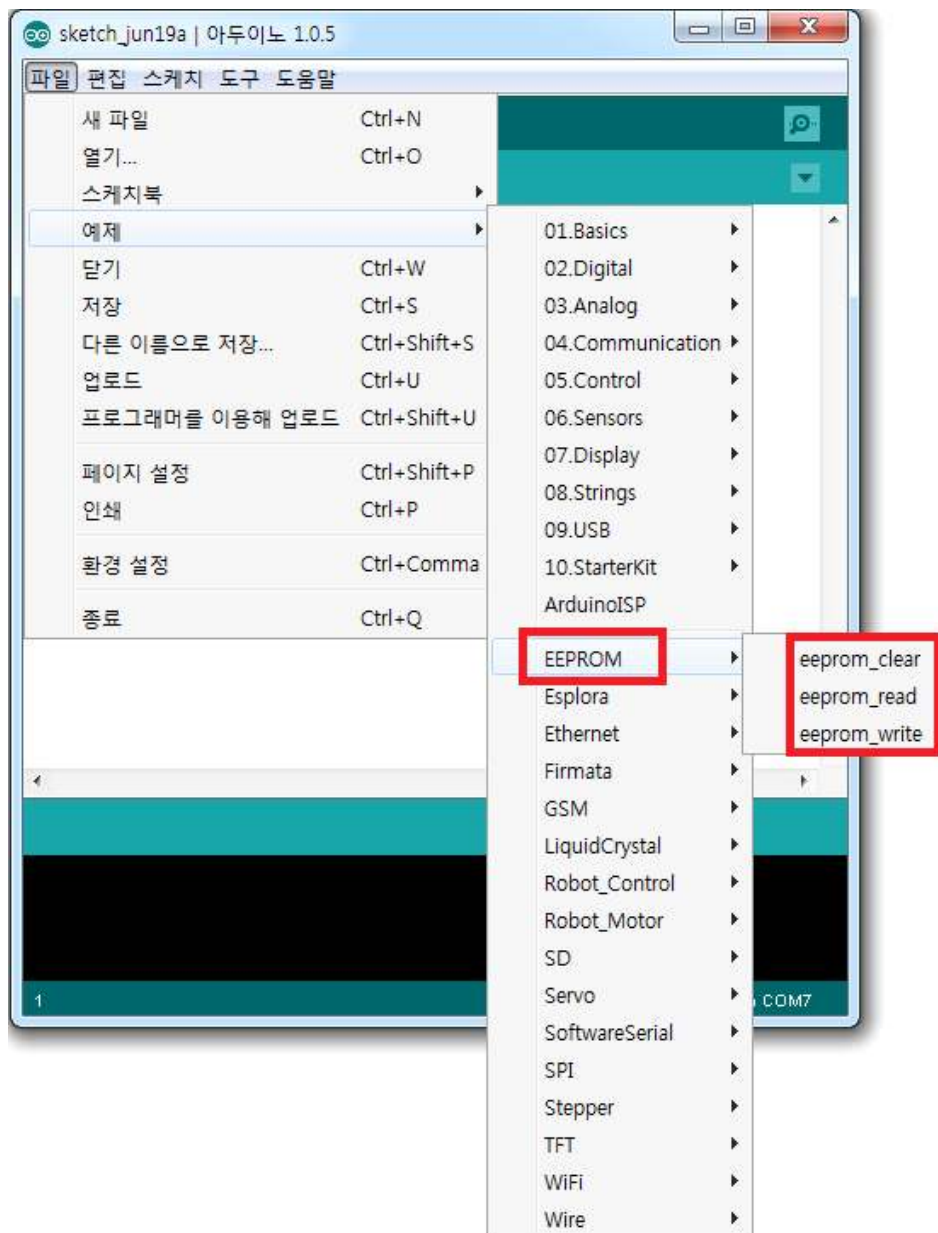


그림 3. 라이브러리에 포함된 예제

아두이노에서 제공하는 기본 라이브러리 이외의 라이브러리는 외부 라이브러리 또는 확장 라이브러리라고 하며 외부 라이브러리를 사용하기 위해서는 먼저 해당 라이브러리를 다운 받아 설치하여야 한다. 라이브러리를 설치하는 위치는 스케치북 디렉터리 아래 'libraries' 디렉터리로 스케치북 디렉터리가 설정되면 그 아래에 'libraries' 디렉터리가 자동으로 생성된다. 스케치북 디렉터리는 '파일 → 환경 설정' 메뉴에서 확인할 수 있다. 확장 라이브러리 역시 기본 라이브러리와 동일한 디렉터리 구조를 가지며 동일한 메뉴를 통해 필요한 헤더 파일을 포함시킬 수 있고 예제를 새 창으로 열 수 있다. 라이브러리 파일이 설치되는 디렉터리가 기본 라이브러리의 경우 아두이노 프로그램 설치 디렉터리 아래인 반면 외부 라이브러리의 경우 스케치북 디렉터리 아래라는 점에서 차이가 있다.

2. EEPROM 라이브러리

아두이노 우노에 사용된 ATmega328 마이크로컨트롤러에는 세 종류의 메모리가 포함되어 있으며 플래시 메모리, SRAM, 그리고 EEPROM이 그것들이다. 사용자가 작성한 스케치를 다운로드 하면 스케치는 비휘발성 플래시 메모리에 저장되며 스케치를 저장하기 위해 사용되는 플래시 메모리를 일반적으로 프로그램 메모리라고 부른다. SRAM(Static Random Access Memory)은 휘발성 메모리로 스케치 실행 중 사용되는 변수 값의 저장을 위해 주로 사용된다. EEPROM(Electrically Erasable Programmable Read-Only Memory)은 비휘발성 메모리로 변수 값을 저장하기 위해 주로 사용되는 점에서는 SRAM과 동일하지만 전원이 없어도 기록된 내용이 보존되는 면에서는 플래시 메모리와 동일하다. 표 1은 ATmega328에 포함되어 있는 메모리의 종류를 요약한 것이다.

메모리	크기 (KB)	용도	특성	실행 중 내용 변경
플래시 메모리	32	스케치 설치	비휘발성	불가능
SRAM	2	변수값 저장	휘발성	가능
EEPROM	1	변수값 저장	비휘발성	가능

표 1. ATmega328의 메모리

개인용 컴퓨터(Personal Computer, PC)에서 메모리는 휘발성 메모리로 프로그램은 하드 디스크에 설치되고 프로그램이 실행되는 시점에서만 메모리에 적재되어 실행된다. 하지만 PC는 여러 개의 프로그램을 설치하고 선택적으로 실행시킬 수 있는 반면 마이크로컨트롤러에는 단 하나의 프로그램만 설치될 수 있는 차이가 있다. 플래시 메모리의 경우 스케치가 실행 중인

상태에서 그 내용을 변경할 수 없으므로 스케치 실행 중 변경되는 값들은 휘발성 메모리인 SRAM에 저장된다. EEPROM 역시 데이터 값의 보존을 위해 사용할 수 있는 작은 크기의 메모리로 스케치 실행 중에 값을 변경할 수 있으며 비휘발성 메모리이므로 전원이 꺼져도 값이 보존된다. 하지만 EEPROM은 한 번에 1바이트씩만 데이터를 쓰거나 지울 수 있기 때문에 그 속도가 느리며 약 10만 번 정도 내용을 기록할 수 있다.

아두이노 프로그램에는 EEPROM에 데이터를 읽고 쓸 수 있는 EEPROM 라이브러리가 포함되어 있다. EEPROM을 사용하기 위해서는 ‘스케치 → 라이브러리 가져오기 → 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. EEPROMClass에는 바이트 단위의 데이터를 읽고 쓸 수 있도록 두 개의 멤버 함수가 정의되어 있다.

```
uint8_t read(int address)
void write(int address, uint8_t value)
```

EEPROM은 바이트 단위로 데이터를 읽고 쓰기 때문에 기록할 값의 범위를 벗어나지 않도록 주의해야 한다. 스케치 1은 ATmega328의 EEPROM의 0번지에서 9번지까지 데이터를 기록하고 이를 다시 읽어 시리얼 모니터로 출력하는 예이다.

스케치 1. EEPROM에 데이터 쓰기과 읽기

```
#include <EEPROM.h>

void setup(){
    Serial.begin(9600);          // 시리얼 통신 초기화
}
```

```
void loop(){
  for(int i = 0; i < 10; i++){      // EEPROM에 값 쓰기
    EEPROM.write(i, i);  // write(주소, 값)
    delay(100);
  }

  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(100);
  }

  while(true);
}
```

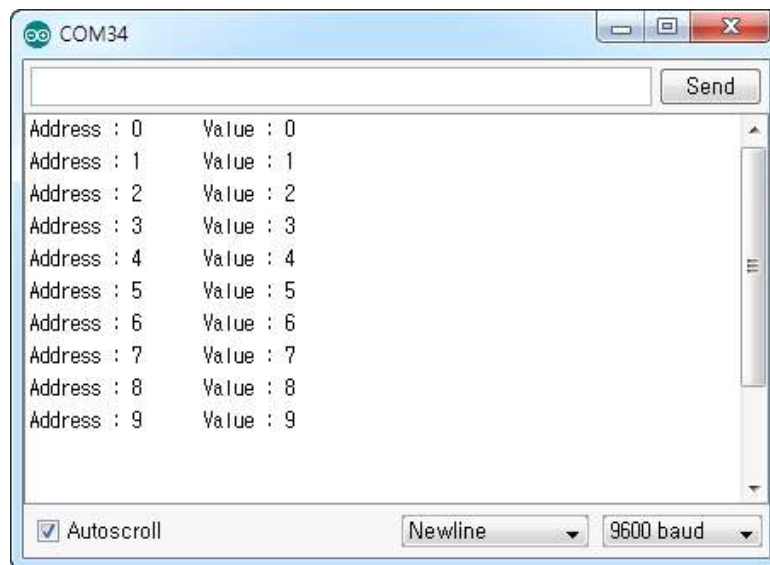


그림 4. 스케치 1 실행 결과

EEPROM은 바이트 단위로 데이터를 입출력하므로 1바이트 이상의 크기를 갖는 정수나 문자열 등의 데이터는 한 번에 EEPROM에 기록할 수 없으며 바이트 단위로 나누어 여러 번 쓰고 읽기를 반복하여야 한다. 스케치 2는 2바이트 크기를 갖는 정수를 EEPROM에 기록하고 읽는 예이다. 정수의 경우 2바이트 크기이므로 EEPROM에 두 개 이상의 정수를 기록하는 경우 연

속된 번지를 지정하여서는 안된다.

스케치 2. EEPROM에 int 형 데이터 쓰기와 읽기

```
#include <EEPROM.h>

void setup(){
    Serial.begin(9600);                // 시리얼 통신 초기화
}

void write_int_EEPROM(int address, int value){
    EEPROM.write(address, value >> 8);    // 상위 바이트
    EEPROM.write(address + 1, value & 0x00FF); // 하위 바이트
}

int read_int_EEPROM(int address){
    byte byte_high = EEPROM.read(address);    // 상위 바이트
    byte byte_low = EEPROM.read(address + 1); // 하위 바이트

    return (byte_high << 8 | byte_low);        // 정수값 조합
}

void loop(){
    Serial.println("Write 1234 to EEPROM");
    write_int_EEPROM(10, 1234);

    Serial.println("Read from EEPROM");
    Serial.println(String(" >> ") + read_int_EEPROM(10));

    while(true);
}
```

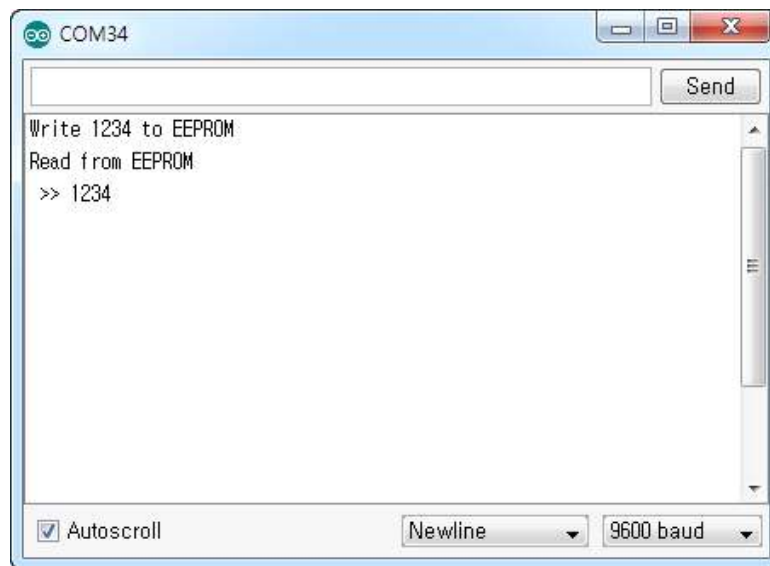


그림 5. 스케치 2 실행 결과

3. RTC (Real Time Clock) 라이브러리

컴퓨터에는 배터리에 의해 동작하는 시계가 메인 보드에 내장되어 있어 컴퓨터 전원을 내렸다가 올린 뒤에도 날짜와 시간 정보가 유지되며 이를 RTC라 한다. 아두이노의 경우 RTC가 포함되어 있지 않으므로 마이크로컨트롤러의 타이머를 이용하는 소프트웨어적인 방법이나 별도의 하드웨어 모듈을 이용하는 하드웨어적인 방법으로 날짜와 시간을 사용할 수 있다. 소프트웨어 라이브러리를 사용하는 경우에는 마이크로컨트롤러 내의 타이머를 이용하여 날짜와 시간을 알아낼 수 있지만 아두이노의 전원을 끄면 정보가 유지되지 않으므로 아두이노를 켤 때 매번 시간을 설정하여야 하는 불편함이 있다. RTC를 이용하게 되면 이런 불편은 없지만 별도의 하드웨어가 필요한 단점이 있다. 여기서는 소프트웨어 라이브러리를 이용하여 시간 및 날짜를 사용하는 방법을 살펴본다.

아두이노 우노 보드에 사용되는 ATmega328에는 timer0, timer1, timer2의 3개 타이머가 내장되어 있다. 이 중 timer0과 timer2는 8비트 타이머이며 timer1은 16비트 타이머이다. 타이머는 일반적으로 마이크로컨트롤러의 클럭 주파수에 영향을 받으므로 아두이노 우노의 경우 16MHz로 동작하게 된다. 타이머를 명시적으로 사용하지는 않지만 많은 아두이노의 함수들이 타이머에 의존하고 있다. 시간 지연을 위한 delay 함수, 실행 시간을 알려주는 millis 함수, PWM 신호를 출력하는 analogWrite 함수, 소리 재생을 위한 tone 함수 등이 그 예이다. 특히 아두이노 우노에서 PWM 출력의 경우 디지털 5번 및 6번 핀은 timer0을, 디지털 9번 및 10번 핀은 timer1을, 디지털 11번 및 3번 핀은 timer2를 이용하므로 타이머 기능을 사용하

는 경우 타이머와 연관되어 있는 핀의 PWM 출력은 사용할 수 없다.

타이머	Arduino UNO PWM 핀	타이머 사용 함수/라이브러리
timer0	5, 6	delay, millis, micros
timer1	9, 10	서보 모터 라이브러리
timer2	11, 3	tone

표 2. 타이머와 PWM 출력 핀의 연관성

타이머를 이용한 소프트웨어 RTC를 지원하는 여러 라이브러리를 아두이노 홈페이지의 Playground에서 찾아 볼 수 있다.¹⁾ 그 중 먼저 swRTC 라이브러리를 이용하여 날짜와 시간을 다루는 방법을 살펴본다. ATmega328 마이크로컨트롤러의 경우 swRTC 클래스는 timer2를 사용하므로 timer2를 이용하는 PWM 출력인 디지털 3번 및 11번 핀은 사용할 수 없다. swRTC 라이브러리를 다운받아²⁾ 스케치북 디렉터리의 'libraries' 디렉터리에 설치한다. swRTC 라이브러리를 사용하기 위해서는 먼저 해당 헤더 파일을 포함시켜야 한다. '스케치 → 라이브러리 가져오기 → swRTC' 메뉴를 실행시키면 자동으로 해당 라이브러리를 사용할 수 있도록 include 문장이 추가된다.

```
#include <core_build_options.h>
#include <swRTC.h>
```

헤더 파일이 포함되면 swRTC 사용을 위해 객체를 생성하여야 한다.

```
swRTC rtc;
```

생성된 swRTC는 시간과 날짜를 초기화시켜야 하며 setTime 함수와 setDate 함수를 통해 가능하다. 초기화 함수는 지정한 날짜와 시간이 범위를 벗어나는 값이면 false를 반환한다.

1) <http://playground.arduino.cc//Main/GeneralCodeLibrary>

2) <http://www.leonardomiliani.com/arduino/>

```
boolean setTime(byte hourT, byte minuteT, byte secondT)
boolean setDate(byte dayT, byte monthT, int yearT)
```

초기화가 끝나면 startRTC 함수로 시계가 동작하도록 할 수 있으며, stopRTC 함수로 시계의 동작을 일시 멈출 수 있다.

```
void startRTC(void)
void stopRTC(void)
```

현재 시간과 날짜를 알아내기 위해서는 다음 함수들을 사용하면 된다.

```
byte getSeconds(void)
byte getMinutes(void)
byte getHours(void)
byte getDay(void)
byte getMonth(void)
int getYear(void)
```

스케치 3은 swRTC 클래스에서 제공하는 메서드들을 사용하는 예를 보여주고 있다. 스케치 3에서 알 수 있듯이 swRTC 라이브러리는 초기 시간과 날짜를 설정한 후 필요한 시점에서 멤버 함수를 호출하여 날짜와 시간을 알아낼 수 있다.

스케치 3. swRTC 사용

```
#include <core_build_options.h>
#include <swRTC.h>

swRTC rtc;                                // swRTC 인스턴스 생성

void setup() {
```

```
rtc.stopRTC();           // RTC 정지
rtc.setTime(15, 0, 0);    // 15시 0분 0초로 설정
rtc.setDate(5, 5, 2014);  // 2014년 5월 5일로 설정
rtc.startRTC();           // RTC 시작
Serial.begin(9600);       // 시리얼 통신 초기화
}

void loop() {
  Serial.print(rtc.getYear(), DEC);    // 년
  Serial.print("/");
  Serial.print(rtc.getDay(), DEC);     // 일
  Serial.print("/");
  Serial.print(rtc.getMonth(), DEC);   // 월
  Serial.print(" -- ");

  Serial.print(rtc.getHours(), DEC);   // 시
  Serial.print(":");
  Serial.print(rtc.getMinutes(), DEC); // 분
  Serial.print(":");
  Serial.println(rtc.getSeconds(), DEC); // 초

  delay(1000);
}
```

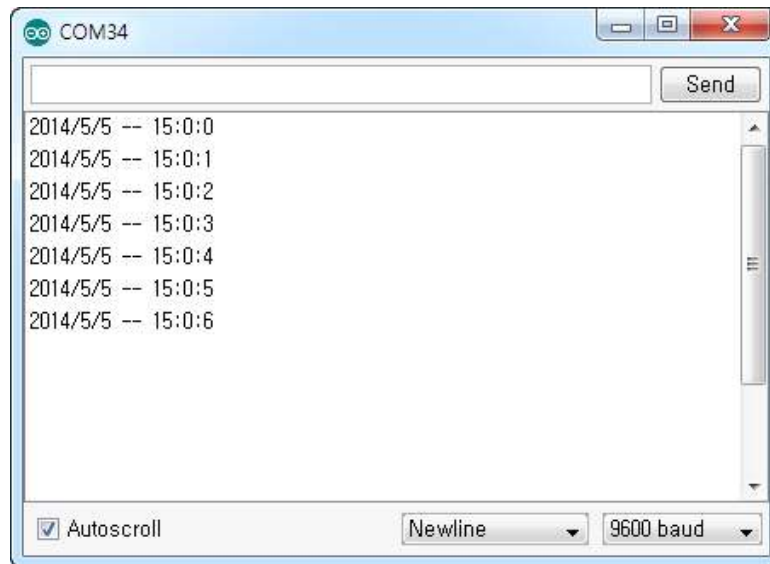


그림 6. 스케치 3 실행 결과

4. 타이머 라이브러리

아두이노를 사용하다 보면 일정한 간격으로 정해진 작업을 수행해야 하는 경우가 있다. 예를 들어 시계를 구현하고 싶다면 1초 간격으로 시간 표시 장치를 갱신하여야 한다. 이 때 사용할 수 있는 가장 단순한 방법은 delay 함수를 이용하는 것이다. 하지만 delay 함수는 delay 함수가 실행 중인 동안 대부분의 작업을 진행할 수 없으며 delay(1000) 함수를 실행하면 지연 시간만 1초가 되고 다른 코드의 실행 시간은 고려하지 않으므로 정확한 시간 간격을 지정하기 어렵다. 따라서 보다 정확한 시간 간격을 설정하기 위해서 millis 함수를 사용하여 지정한 시간을 경과하였는지를 비교하는 방법을 사용하였다.

일정 시간 간격으로 반복 작업을 수행하는 방법에는 폴링(polling) 방식과 인터럽트(interrupt) 방식이 있다. 폴링 방식은 일정한 시간 간격으로 지정한 시간이 경과하였는지 검사하는 방식이며 인터럽트 방식은 지정한 시간이 경과하면 자동으로 알려주는 방식이다. 친구와 만날 약속 장소를 이메일로 받기로 한 경우를 생각해 보자. 이메일이 도착하였는지 알아보는 방법은 5분마다 메일함을 열어보는 방법과 이메일이 도착하면 소리를 내어 알려주는 프로그램을 설치하고 소리가 나기를 기다리는 방법이 있을 수 있다. 전자가 폴링 방식에 해당하고 후자가 인터럽트 방식에 해당한다. 인터럽트 방식이 보다 편리한 방식으로 보이지만 추가적인 이메일 알리미 프로그램을 설치하여야 하는 단점이 있다. 마이크로컨트롤러의 내장 타이머는 알리미 역할을 수행하는 부품으로 인터럽트 방식의 사용을 가능하게 해 준다. 하지만 타이머를 이용하기 위해서는 복잡한 레지스터 설정 과정이 필요하므로 아두이노에서 간단히 주기적

인 작업을 진행하기 위해 millis 함수를 사용하여 경과 시간을 검사하는 폴링 방식이 많이 사용된다.

이처럼 일정한 시간 간격으로 반복 수행되는 작업을 위해 사용되는 기능을 구현해 놓은 라이브러리를 타이머 라이브러리라 하며 아두이노 사이트에는 여러 종류의 타이머 라이브러리가 공개되어 있으며 폴링 방식의 Metro 라이브러리와 인터럽트 방식의 MsTimer2 라이브러리 사용 방법을 살펴본다.

4-1. Metro 라이브러리

Metro 라이브러리는 millis 함수를 이용하는 폴링 방식의 타이머 라이브러리이다. Metro 라이브러리는 Metro 클래스를 정의하고 있으며 경과 시간별로 하나의 Metro 개체를 생성할 수 있으므로 여러 개의 타이머를 동시에 사용할 수 있다. 단, Metro 라이브러리는 millis 함수를 사용하므로 loop 함수 내에서 시간 경과를 계속적으로 검사하여야 한다.

Metro 라이브러리를 다운 받아³⁾ 설치하고 ‘스케치 → 라이브러리 가져오기... → Metro’ 메뉴를 선택하여 필요한 헤더 파일을 추가한다.

```
#include <Metro.h>
```

Metro 클래스의 객체를 생성한다. 생성자에서 autoreset이 true로 설정되면 경과시간이 초과 되는 경우 자동으로 리셋되므로 check 함수를 자주 호출하여 확인하지 않으면 경과 시간이 만료된 것을 지나칠 수 있다. 디폴트값은 false이다.

```
Metro(unsigned long interval_millis)
Metro(unsigned long interval_millis, uint8_t autoreset)
    interval_millis : 경과 시간
    autoreset : 경과시간 초과 시 자동 리셋 여부
    반환값 : 없음 (생성자)
```

객체가 생성된 이후 새로운 경과 시간을 설정하기 위해서는 interval 함수를 사용하면 된다.

3) http://www.pjrc.com/teensy/td_libs_Metro.html

```
void interval(unsigned long interval_millis)
    interval_millis : 새로운 경과 시간
    반환값 : 없음
```

loop 함수 내에서는 check 함수를 통해 지정한 시간이 경과하였는지의 여부를 확인할 수 있다. check 함수는 지정한 시간의 경과 여부를 0 또는 1로 반환한다.

```
char check(void)
    반환값 : 경과 시간 만료 여부 (0 또는 1)
```

타이머를 다시 시작하고 싶은 경우에는 reset 함수를 사용하여 타이머를 초기화할 수 있다.

```
void reset(void)
    반환값 : 없음
```

스케치 4는 0.5초와 1초 간격으로 만료되는 두 개의 타이머 객체를 생성하고 loop 함수 내에서 타이머의 만료를 검사하는 스케치의 예를 나타낸다.

스케치 4. Metro 라이브러리 사용

```
#include <Metro.h>                // Metro 라이브러리

Metro timer1(500);                // 0.5초 간격 타이머
Metro timer2(1000);               // 1초 간격 타이머

void setup(){
    Serial.begin(9600);
}
```

```

void loop(){
    if(timer1.check())          // 0.5초 경과
        Serial.println(">> Timer 1 : 0.5 second");
    if(timer2.check())          // 1초 경과
        Serial.println("** Timer 2 : 1 second");
}

```

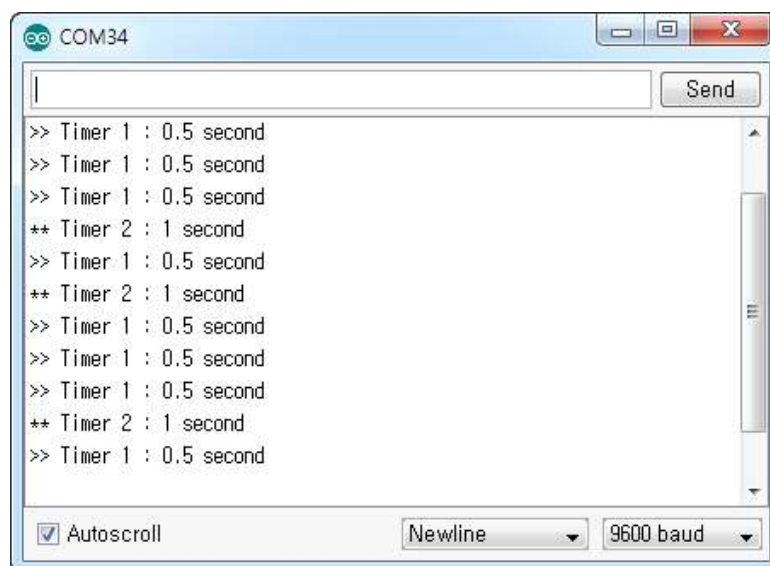


그림 7. 스케치 4 실행 결과

4-2. MsTimer2 라이브러리

MsTimer2 라이브러리는 마이크로컨트롤러의 하드웨어 타이머를 이용하는 인터럽트 방식의 타이머 라이브러리로 설정된 타이머가 만료되면 인터럽트가 발생하고 인터럽트 처리 함수인 인터럽트 서비스 루틴(Interrupt Service Routine, ISR)이 자동으로 호출된다. MsTimer2 라이브러리는 클래스를 정의하고 있지 않으며 MsTimer2 네임 스페이스에 타이머 관련 함수들을 정의하고 있으므로 함수 호출을 위해서는 'MsTimer2::function_name'과 같은 방식으로 사용하여야 한다. MsTimer2 라이브러리는 하드웨어 타이머를 사용하므로 millis 함수를 사용하는 Metro 라이브러리에 비해 보다 정확한 타이머 기능을 사용할 수 있지만 만료 시간이 다른 여러 개의 타이머를 사용할 수는 없으며 하나의 타이머만을 사용할 수 있는 한계가 있다. 또한 MsTimer2 라이브러리는 timer2를 사용하므로 timer2를 이용하는 디지털 3번 및 11번

핀의 PWM 출력을 사용할 수 없다는 점도 유의하여야 한다.

MsTimer2 라이브러리를 다운 받아⁴⁾ 설치하고 ‘스케치 → 라이브러리 가져오기... → MsTimer2’ 메뉴를 선택하여 필요한 헤더 파일을 추가한다.

```
#include <MsTimer2.h>
```

클래스로 정의되어 있지 않으므로 객체를 생성할 필요는 없으며 만료 시간은 set 함수를 통해 설정한다. 만료 시간을 설정할 때에는 지정한 시간이 만료된 경우 인터럽트에 의해 자동으로 호출되는 함수 역시 설정한다. 이 때 자동으로 호출되는 함수는 매개변수를 가질 수 없다.

```
void set(unsigned long ms, void (*f>())
    ms : 경과 시간
    void (*f()) : 인터럽트 처리 함수에 대한 포인터
    반환값 : 없음
```

만료 시간이 설정되면 타이머를 start 함수를 호출하며 인터럽트를 활성화시키거나 stop 함수를 호출하여 인터럽트를 비활성화 시킬 수 있다.

```
void start(void)
void stop(void)
```

스케치 5는 1초 간격으로 만료되는 타이머를 설정하고 사용하는 방법을 보여준다. 인터럽트 방식에 의해 interruptFunction이 자동으로 호출되므로 loop 함수 내에는 아무런 코드도 존재하지 않는다는 점에 유의하여야 한다.

스케치 5. MsTimer2 라이브러리 사용

4) http://www.pjrc.com/teensy/td_libs_MsTimer2.html

```
#include <MsTimer2.h>

int count = 0;

void interruptFunction(){           // 인터럽트 처리 함수
    Serial.println(String("From ISR : ") + count++);
}

void setup(){
    Serial.begin(9600);

    MsTimer2::set(1000, interruptFunction);           // 1초 간격 타이머 설정
    MsTimer2::start();                               // 인터럽트 활성화
}

void loop(){                                     // loop 함수는 비어있다.
}
```

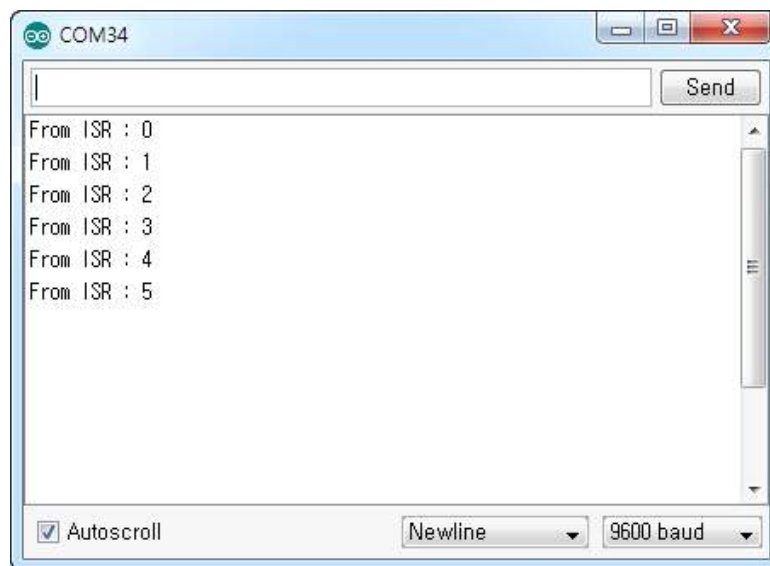


그림 8. 스케치 5 실행 결과