제 2 부

C-언어를 사용한
마이크로컨트롤러 활용기초

C-언어는 수학계산을 위해 개발된 FORTRAN 같은 고급언어들과는 달리 Unix 운영체제를 개발하면서 같이 개발된 고급언어이다. 운영체제의 특성상 C-언어는 다른 고급언어에 비해 컴퓨터의 하드웨어를 직접 제어할 수 있는 능력이 탁월하여 마이크로프로세서의 프로그램에 있어서 어셈블리와 더불어 가장 많이 쓰이는 언어이다.

일반적으로 어셈블리 프로그램이 C-언어 프로그램보다 실행파일의 크기뿐 아니라 실행속도 면에서도 우월하다고 알려져 있으나 이는 두 종류의 프로그램이 최적화 되었을 때 비교일 뿐이므로 항상 옳다고 할 수는 없다. 일반적으로 어셈블리가 C-언어 보다 사용하기 어렵기 때문에 보통 사용자가 최적의 어셈블리 프로그램을 작성하기 어렵다. 아울러 요즘의 C-컴파일러는 성능이 매우 우수하여 C-언어 프로그램이 웬만큼 잘 작성한 어셈블리 프로그램보다 크기 뿐 아니라 속도 면에도 우월하다.

프로그램을 어셈블리로 작성하면 C-언어로 작성된 프로그램에 비해 프로세서의 모든 기능을 사용할 수 있으므로 실행속도가 빠르고 프로그램 크기가 작게 작성할 수 있다. 그러나 고급언어에 비해 프로그램을 작성하기 어려울 뿐 아니라 프로세서마다 문법이 다르기 때문에 프로세서가 바뀌면 어셈블리를 다시 익혀야 한다. C-언어와 같은 고급언어의 문법은 사용하는 프로세서와는 무관하게 정의되어 있어서 프로세서의 특징만 파악하면 어느 프로세서에도 사용할 수 있다. 이런 면에서 가장 효율적인 방법은 C-언어와 같은 고급언어를 사용하여 프로그램을 개발한 후 빠른 실행 속도가 필요하거나 코드의 크기를 줄이고 싶은 부분만을 어셈블리로 작성하는 방법이다. 이 방법을 효율적으로 사용하면 C언어의 간편함과 어셈블러의 강력함을 잘 조화시킬 수 있기 때문에 전문가들이 즐겨 사용하는 방법이다. 이러한 방법을 혼합언어 프로그래밍(mixed-language programming)이라 한다. 그러나 이 방법을 이용하려면 어셈블리 뿐 아니라 C-컴파일러의 상세한 부분에 대해서도 해박한 지식이 있어야 함은 물론이다. 이런 고난이도의 프로그램방법은 이 교재의 범위를 벗어나므로 여기서는 다루지 않고 C-언어를 사용하는 프로그램 방법만 다룬다.

2부에서는 ATmega128 마이크로컨트롤러의 기초적인 활용법을 C-언어를 사용하여 배워보도록 하자.

제 6 장 입출력 포트(I/O Ports)

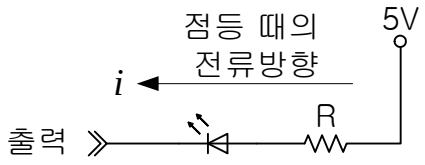
6.1 LED 점등회로 및 스위치 연결회로

마이크로컨트롤러를 배우면 제일 처음 해보는 것이 디지털 입출력 포트를 사용해서 스위치 입력을 받고, LED를 점등하는 것이다. 이는 마이크로컨트롤러의 입출력장치를 이해하기 위한 여러 가지 실습을 간단하게 구성할 수 있기 때문이다.

LED 점등회로

LED 점등회로를 고려하여 보자. LED는 Light Emitted Diode의 약자로 전류가 흐르면 점등되는 다이오드이다. 입출력 장치로 LED를 구동할 때 두 가지 회로를 구성할 수 있다.

첫 번째는 출력장치를 다음 그림과 같이 전류의 싱크(Sink)로 사용하는 것이다.

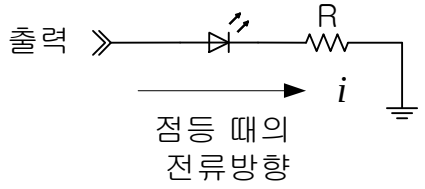


출력	LED
0(0V)	켜짐
1(5V)	꺼짐

☞ 마이크로컨트롤러에서 논리 0을 출력하면 출력 단의 전압은 0V이고 따라서 다이오드에 순방향 바이어스가 걸리고 전류가 흘러 LED가 켜진다.

☞ 논리 1을 출력하면 출력 단의 전압은 5V가 되어 전류가 흐르지 않아 LED가 꺼진다.

두 번째 방법은 다음 그림과 같이 출력장치를 전류의 소스(source)로 사용하는 것이다.



출력	LED
0(0V)	꺼짐
1(5V)	켜짐

☞ 마이크로컨트롤러에서 논리 0을 출력하면 출력 단의 전압은 0V이어서 전류가 흐르지 않고 LED가 꺼진다.
☞ 논리 1을 출력하면 출력 단의 전압은 5V가 되어 순방향 바이어스가 걸려 LED가 켜진다.

LED의 밝기는 전류에 비례한다. LED 다이오드의 전압강하(약 1.5V)를 고려하면

$$i = (5 - 1.5)/R$$

이므로 저항이 작을수록 밝아진다. R=330Ω을 사용하면 흐르는 전류는 약 10mA이다.

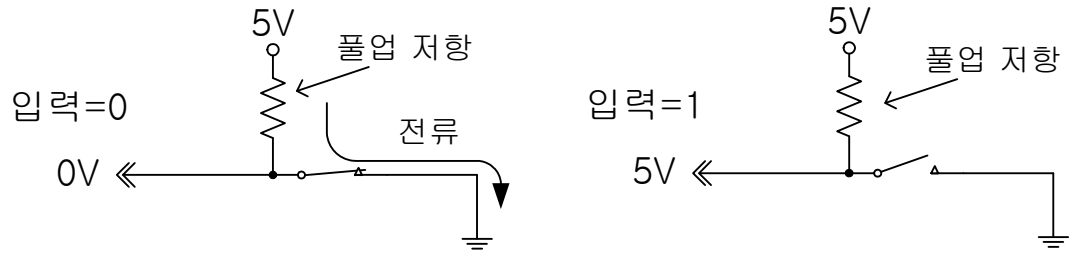
☞ ATmega128의 각 I/O의 최대 싱크/소스 전류는 40mA이므로 어느 경우를 사용하더라도 LED를 구동할 수 있다.
☞ TTL을 사용하여 LED를 구동할 때는 소스형으로 LED를 구동하지 않는다. TTL의 소스 전류는 매우 작기 때문이다.

스위치 입력회로

입력장치의 입력 핀에 0V가 걸리면 마이크로컨트롤러는 논리 0을 읽고 입력 핀에 5V가 걸리면 논리 1을 읽는다. 다음 스위치 입력회로를 고려해보자.

☞ 스위치가 닫히면 입력단에 5V가 걸려 논리 1이 입력된다.
☞ 스위치가 열리면 입력단의 전압이 0V가 아니라 떠 있는(float) 상태가 된다. 이 때 마이크로컨트롤러의 입력은 정의되지 않아서 값을 알 수 없다.

위 회로로는 정확한 스위치 상태를 파악할 수 없으므로 다음 그림과 같이 풀업 저항(pull-up resistor)을 사용해서 스위치 회로를 구성한다.



☞ 스위치가 닫히면 전류가 접지로 흐른다. 입력단은 접지와 연결되어 있으므로 0V가 되고 마이크로컨트롤러는 논리 0을 읽는다.

☞ 스위치가 열리면 전류가 흐르지 않는다. 따라서 풀업 저항 사이의 전압강하가 발생하지 않아 입력단에는 5V가 걸린다. 마이크로컨트롤러는 논리 1을 읽는다.

스위치	입력
닫힘	논리0
열림	논리1

☞ 어떤 경우에도 float상태는 발생하지 않는다.

6.2 입출력 포트 관련 레지스터

ATmega128은 6개의 양방향 8비트 포트(포트 A,B,C,D,E,F)와 1개의 양방향 5비트 포트인 포트 G를 가지고 있다. 내부에서 제공하는 모든 입출력기능은 표 3.1의 I/O 레지스터를 통하여 제어된다. 입출력 포트 제어와 관련된 I/O 레지스터는 다음과 같다.

■ **특수기능 IO 레지스터(Special Function IO Register): SFIOR**

비트	7	6	5	4	3	2	1	0	
	TSM	-	-	-	ACME	PUD	PSR0	PSR321	SFIOR
읽기/쓰기	R/W	R	R	R	R/W	R/W	R/W	R/W	
초기 값	0	0	0	0	0	0	0	0	

• 비트 2 - PUD: 풀업 불능(Pull-up disable)
이 비트가 1이면 I/O 포트의 풀업이 불능이 된다.

☞ 각 포트에 대해서 3개의 레지스터가 연관되어 있는 데 각 포트에 대해 같은 역할을 한다. 다음 레지스터의 표기에 있어서 x는 포트 A,B,C,D,E,F,G를 나타내고 n은 비트 0,1,2,...,7을 나타낸다. 단 포트 G에 대해서는 n에 비트 0,1,2,3,4만 지정된다.

■ **포트 데이터 레지스터(Port Data Register): PORTx**

비트	7	6	5	4	3	2	1	0	
	PORTx7	PORTx6	PORTx5	PORTx4	PORTx3	PORTx2	PORTx1	PORTx0	PORTx
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
초기 값	0	0	0	0	0	0	0	0	

■ **데이터 방향 레지스터(Data Direction Register): DDRx**

비트	7	6	5	4	3	2	1	0	
	DDx7	DDx6	DDx5	DDx4	DDx3	DDx2	DDx1	DDx0	DDRx
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
초기 값	0	0	0	0	0	0	0	0	

■ **포트 입력 핀 레지스터(Port Input Pins Address): PINx**

비트	7	6	5	4	3	2	1	0	
	PINx7	PINx6	PINx5	PINx4	PINx3	PINx2	PINx1	PINx0	PINx
읽기/쓰기	R	R	R	R	R	R	R	R	
초기 값	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

*N/A : Not Available

표 6.1은 각 레지스터의 비트에 따른 입출력 포트의 상태를 보여준다.

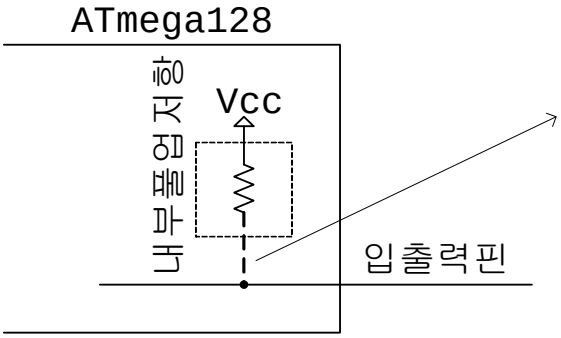
☞ 포트의 핀을 출력으로 하려면 방향 레지스터 DDRx의 해당비트를 DDxn=1로 설정하면 된다. 이 때 PORTxn 비트를 0으로 리셋하면 포트 핀에 0V가 PORTxn 비트를 1로 세트하면 5V가 출력된다.

☞ 방향 레지스터 DDRx의 비트 DDxn=0으로 하면 포트는 입력이 된다. 단 풀업이 되어 있을 때 입력이 가능하다. 6.1절과 같이 외부풀업을 이용하거나, 표6.1에 따라 비트를 설정하여(PORTXn=1, PUD=0) 내부 풀업을 사용할 수 있다. 핀에 0V가 걸렸을 때 PINxn을 읽으면 0이 읽히고, 핀에 5V가 걸렸을 때는 PINxn의 값으로 1이 읽힌다.

표 6.1 포트 출력 핀의 형태

DDxn	PORTxn	PUD (in SFIOR)	입출력	내부 풀업	비고
0	0	X	입력	No	Hi-Z상태†: 외부풀업필요
0	1	0	입력	Yes	입력
0	1	1	입력	No	Hi-Z상태†: 외부풀업필요
1	0	X	출력	No	LOW 출력 - Sink
1	1	X	출력	No	High 출력 - Source

† 전기적인 절연상태로 Tri-State라고도 한다. Z는 Impedance를 뜻한다.



ATmega128의 입출력핀에는 내부풀업저항이 있으며, PORTxn비트와 PUD비트를 설정하여 내부풀업저항의 사용유무를 결정한다.

☞ **레지스터의 비트 속성**

레지스터를 표시할 때 각 비트에 대해 두 가지 속성을 표시한다. 각 속성의 의미는 다음과 같다.

▸ 읽기/쓰기(Read/Write)

R/W : 읽기/쓰기가 가능한 비트

R : 읽기만 가능한 비트

W : 쓰기만 가능한 비트

▸ 초기 값 : 마이크로컨트롤러가 전원이 공급된 직후 또는 리셋 직후 각 레지스터 비트가 갖는 값을 나타낸다. N/A로 표시된 경우는 정의되지 않은 경우로 0 또는 1 어떤 값을 가지는지 모른다.

☞ **레지스터 이름과 각 비트의 이름**

ATmega128의 사용자 메뉴얼은 모든 레지스터에 이름을 부여하고 있으며 레지스터의 각 비트에도 이름을 부여하고 있다. WinAVR C-컴파일러는 헤더파일 "avr/io.h"을 소스파일에 포함시키면 레지스터와 비트 이름을 프로그램에서 직접 사용할 수 있도록 하였다.

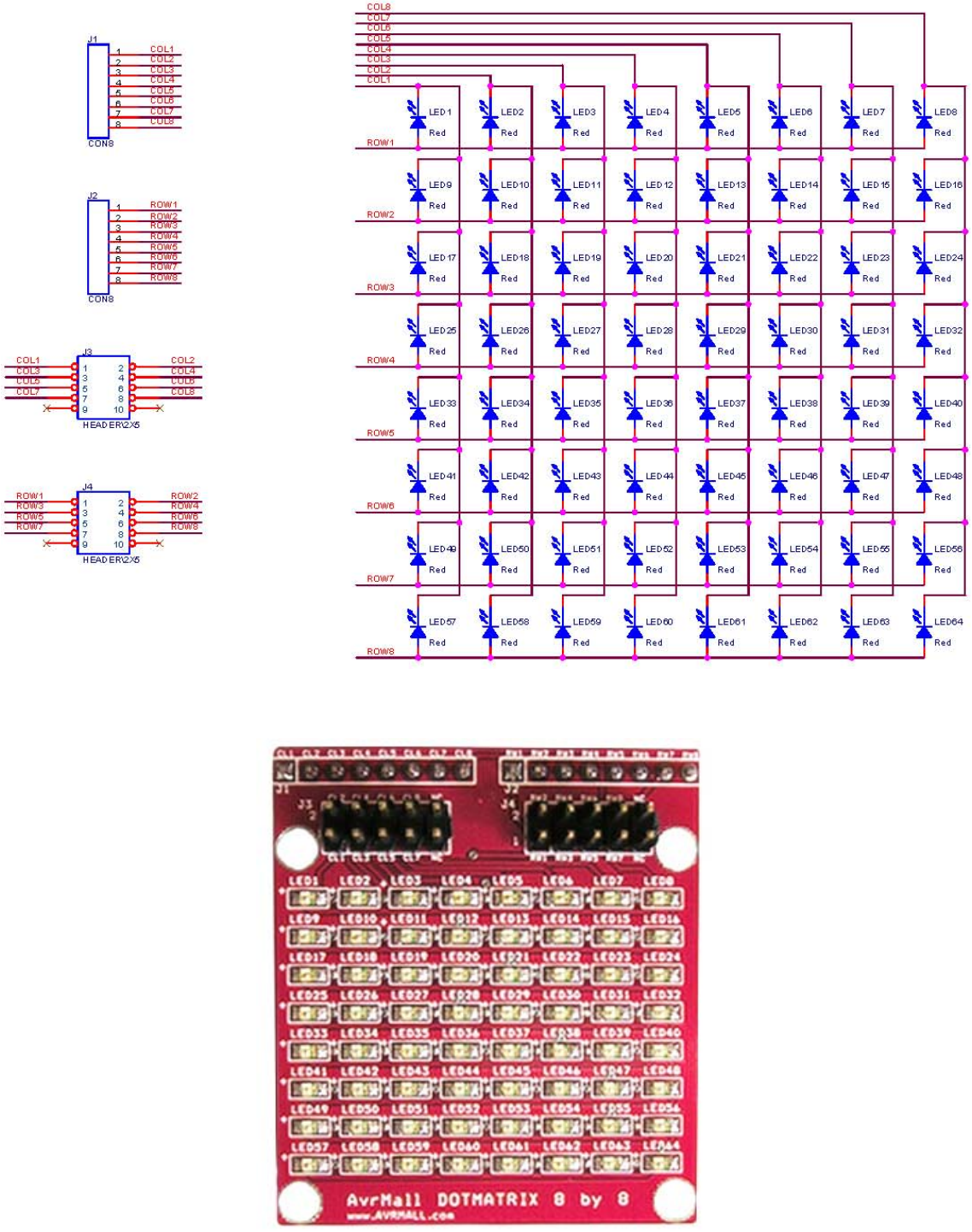


그림 6.1 8×8 LED 도트 매트릭스(dot matrix)

6.3 단순 LED 점등 실습

6.2절에서 설명한 바와 같이 ATmega128에서 제공하는 입출력 포트는 출력 또는 입력으로 사용할 수 있다. 여기서는 입출력 포트의 출력기능을 배우도록 하자. 그림 6.1은 LED가 8x8 행렬로 장착된 도트 매트릭스를 나타낸다. 그림 6.2와 같이 8x8 도트 매트릭스의 LED를 포트A에 연결하자. 이 경우 1-행의 LED만 점등이 된다. 실습으로 1-행의 LED를 원하는 패턴으로 점등하여 보자. 6.1절에서 설명한 바와 같이 점등하고자 하는 LED의 핀에는 논리 0을, 소등하고자 하는 LED의 핀에는 논리 1을 출력하면 된다.

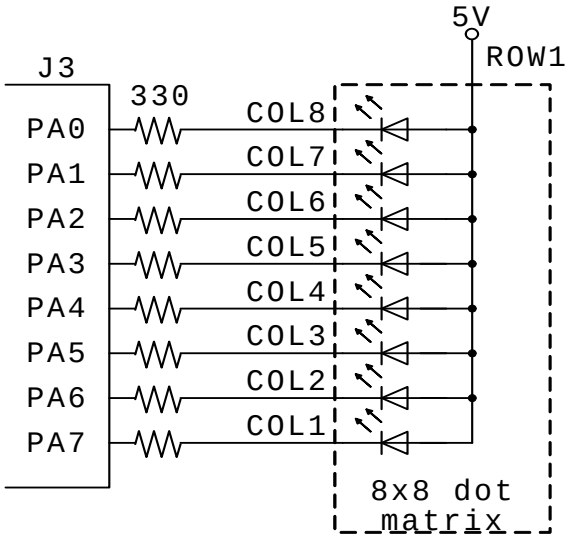


그림 6.2 LED연결회로

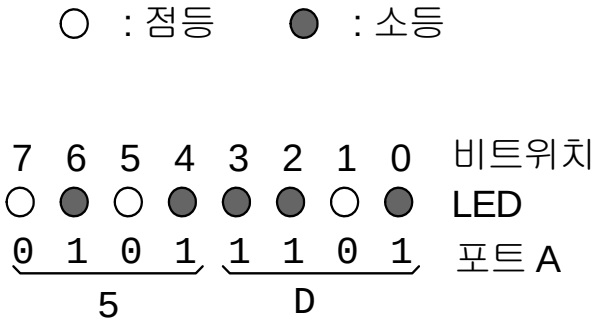


그림 6.3 출력 패턴

그림 6.3과 같은 패턴으로 LED를 점등하려면 포트A를 출력포트로 설정하고 출력 패턴에 해당하는 데이터를 출력포트에 출력하면 된다. 그림 6.3에 따라 포트A에 출력할 데이터는 0x5D이다.

I/O 레지스터 사용 : 포트A의 방향을 설정하려면 1바이트 방향 레지스터인 DDRA에 방향 데이터를 쓰면 되고, 포트A에 데이터를 출력하려면 1바이트 데이터 레지스터인 PORTA에 데이터를 쓰면 된다. WinAVR C-컴파일러에서는 레지스터에 데이터를 쓰려면 헤더파일 "avr/io.h"을 소스파일에 포함시키고 DDRA와 PORTA가 마치 변수인 것처럼 사용하면 된다.(5.3.2절 참조) 따라서 포트A를 출력포트로 설정하고 데이터 0x5D를

출력하려면 다음과 같이 작성하면 된다.(두 레지스터 모두 1바이트 변수이므로 unsigned char형 변수처럼 사용한다.)

```
#include <avr/io.h>
. . .
DDRA  = 0xFF;  // 포트A를 모두 출력포트로 설정
PORTA = 0x5D;  // 패턴으로 LED를 점등
```

☞ **프로그램 언어에서 메모리에 데이터 쓰기/읽기**

- 프로그램언어에서 '='의 의미는 수학에서 의미와는 다르다.
  x = y : 수학에서는 "x와 y가 같다."는 의미
  x = y : 프로그램 언어에서는 "y값을 x에 넣는다."는 의미
- "x = y"를 다시 해석하면
  y값을 읽어서 x에 쓰라는 의미.
- 모든 변수에는 메모리가 할당된다. 따라서 변수에 데이터를 쓰는 것은 해당 메모리에 데이터를 쓰는 것과 동일하고, 변수를 읽는 것은 해당 메모리의 데이터를 읽는 것과 동일하다.
- 데이터를 메모리에 쓸 때
  '='의 왼쪽에 변수를 두면 그 변수의 메모리에 데이터를 쓴다.
  (예) x = 10 : 10을 변수x(실제는 변수x가 있는 메모리)에 쓴다.
- 데이터를 메모리로부터 읽을 때
  '='의 왼쪽에 변수가 위치하는 경우를 제외하고 모든 경우는 변수를 읽는다.
- ATmega128의 I/O레지스터는 모두 메모리번지를 가지고 있으므로 메모리(변수)를 읽는 것과 같은 방법으로 읽기/쓰기를 할 수 있다.

(예) PORTA = 0xFF : 0xFF를 PORTA 레지스터에 쓴다.
(예) if(PINA == 1) : PINA레지스터를 읽어서 1과 같으면 ...
(예) while(x){} : 변수x를 읽어서 그 값이 참이면 루프를 돈다.
(예) x = x + 2 : 변수x를 읽어서 그 값에 2를 더해 x변수에 쓴다. 결과적으로 변수x를 2증가 시킨다.
(예) x +=2 : 이는 x=x+2와 동일함.

따라서 점등을 위해 프로그램 6.1을 작성할 수 있다.

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

int main()
{
     DDRA  = 0xFF;        // 포트A를 모두 출력포트로 설정

     while(1)             // 무한 루프
          PORTA = 0x5D;   // 출력패턴으로 LED를 점등
}
```

프로그램 6.1

☞ 마이크로프로세서의 실행프로그램은 PC와는 달리 전원이 꺼질 때까지 종료되면 안 된다. 따라서 마지막에 while문을 사용하여 무한루프를 구성한다.

☞ while문은 다음과 같은 문법을 가진다.

```
while(expr1)
     expr2;
```

while문은 expr1이 참값이면 expr2를 수행하고 아니면 while문에서 빠져나가게 된다. 그러나 if문과는 달리 expr2의 수행 후 다시 while문으로 올라가 expr1을 평가하는 반복적 루프를 구성한다. 프로그램1에서는 expr1=1이므로 expr1은 항상 참값이다. 따라서 while문을 빠져나가지 못하고 무한루프를 구성하여 LED 패턴을 순차적으로 점등하는 것을 무한히 반복한다.

실습: 다음 순서에 따라 실습을 수행한다.

ATmega128 보드의 사용 준비:

(1) 그림 4.4에 따라 ATmega128보드와 USBISP를 연결한다.

(2) USBISP와 PC를 USB케이블로 연결한다.

(3) USBISP의 드라이버가 PC에 설치되어 있지 않으면 4.3절에 따라 드라이버를 설치한다.

(4) ATmega128보드에 전원을 공급한다. 전원을 공급하는 방법은 2가지가 있다.

(가) ATmega128보드의 전원단자에 5V 외부전원을 직접 공급한다.

(나) PC의 USB포트 전원을 사용한다. 이를 위해서 USBISP보드의 스위치 SW1을 VBUS로 세팅한다.

※프로그램 개발 단계에서는 USBISP를 사용하므로 외부전원이 필요 없는 방법 (나)가 매우 편리하다. 교재에서는 방법 (나)를 사용한다.

5.4.1절에 따라 프로젝트 구성하고 프로그램 실행하기:

(1) 사용자 작업 폴더를 생성하고 그 곳에 lab6라는 프로젝트를 새로 생성한다. 프로젝트를 생성할 때 초기파일을 생성하지 않도록 한다.

(2) 새로운 소스파일 program6-1.c를 만들고 프로그램 6.1의 내용을 입력한다. (5.4.2절)

(3) 소스파일 program6-1.c를 프로젝트 lab6에 추가한다.(5.4.3절)

(4) 프로젝트 lab6을 빌드한다.(5.4.4절)

(5) 프로그램을 타겟 보드에 다운로드한다.(5.4.5절) 다운로드완료와 동시에 프로그램이 실행된다.

※프로그램은 FLASH메모리에 저장되므로 전원을 꺼도 지워지지 않는다. 따라서 전원을 껐다 켜거나, 리셋 버튼을 누르면 프로그램이 처음부터 다시 수행된다.

다른 패턴 점등하기:

다음 두 가지 패턴으로 LED를 점등하여 보라.

(패턴 1) ○ ○ ● ● ○ ● ● ○ LED

(패턴 2) ○ ○ ● ○ ○ ● ○ ○ LED

6.4 패턴을 변경해 가면서 LED 점등

여기서는 그림 6.4와 같이 패턴을 연속적으로 변경하면서 LED를 점등하여 보자. 각 점등 패턴에 대해 포트A에 출력해야 할 데이터를 써보면

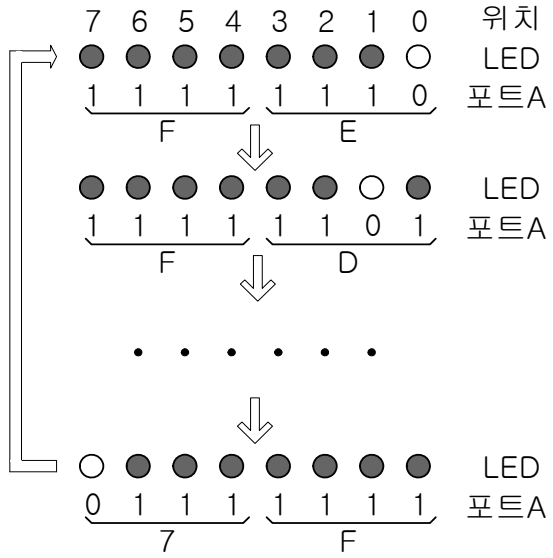


그림 6.4 연속 출력 패턴

```
PA0 점등 ==> 0xFE
PA1 점등 ==> 0xFD
PA2 점등 ==> 0xFB
. . . .
PA7 점등 ==> 0x7F
```

이다. 계속적으로 순환하여 이 패턴들을 점등하여야 하므로 프로그램 6.2와 같이 while문내에 위 데이터를 출력하도록 하면 된다.

```
#include <avr/io.h>
int main()
{
     DDRA  = 0xFF;        // 포트A를 모두 출력포트로 설정

     while(1)             // 무한 루프
     {
          PORTA = 0xFE;  // PA0을 점등
          PORTA = 0xFD;  // PA1을 점등
          PORTA = 0xFB;  // PA2를 점등
          . . .
          PORTA = 0x7F;  // PA7을 점등
     }
}
```

프로그램 6.2

☞ while 루프내에서 수행되어야 할 문장이 여러 문장인 경우는 중괄호 짝 { }내에 수행될 문장을 위치시킨다.

```
while(expr1)
{
     expr2;
     expr3;
     ...
}
```

☞ for 루프는 문법은 다음과 같다.

```
for(expr1; expr2; expr3)
     expr4;
```

위 for 루프문은 다음 while 루프문과 동일하다.

```
expr1;              // 루프의 초기화
while(expr2)        // 루프의 판단
{
     expr4;         // 루프의 몸체
     expr3;         // 루프의 처음으로 가기 전 수행문
}
```

☞ C-언어에서는 while루프, for루프 외에 do-while루프를 제공한다. 이의 문법은 while루프문과 비교해서 다음과 같다.

```
while(expr1)        |        do{
     expr2;         |             expr2;
                    |        }while(expr1);
```

while문는 expr1을 먼저 평가하여 루프를 판단하지만 do-while문은 우선 expr2를 수행한 다음 expr1로 루프를 판단한다.

시간지연함수의 구성

프로그램 6.2를 수행하면 LED 점등패턴이 변하지 않고 희미하게 모두 켜져 있는 것처럼 느껴진다. 이는 패턴 이동이 너무 빨라 패턴의 변화를 눈으로 거의 감지할 수 없기 때문이다. 따라서 점등패턴의 이동사이에 시간지연을 주어야 한다. 정확한 시간지연이 필요할 경우는 마이크로프로세서의 타이머를 사용하여 시간을 측정하여야 하나 현재 실습과 같이 대략의 시간지연이 필요한 경우는 다음과 같은 반복루프를 사용한다.

```
for(i=0; i<1000; i++)
{
     asm("nop"::); asm("nop"::);
}
```

여기서 asm("nop"::)은 단순히 1클록을 쉬는 어셈블리 nop 명령을 C-프로그램에서 수행하도록 하는 것이다. 위 반복루프를 수행하는 데 사용키트에서 약 1msec가 걸리므로 이를 이용하면 사용자가 원하는 시간만큼 시간지연을 줄 수 있는 함수를 다음과 같이 작성할 수 있다.

```
void msec_delay(int n)
{
     int i;
     for(; n >0; n--)              // 1msec루프를 n회 반복
     {
          for(i=0; i<1000; i++)    // 1msec 시간지연 루프
          {
               asm("nop"::); asm("nop"::);
          }
     }
}
```

시간지연 함수

☞ 1msec 시간지연루프를 n회 반복한다.

☞ msec_delay(5)와 같이 함수를 호출하면 msec_delay()함수내의 변수 n에 5가 전달된다. 따라서 1msec 시간지연루프를 5회 반복하므로 5msec 시간지연 후 함수가 리턴 된다.

☞ **시간 지연은 실습환경에 따라 다를 수 있으므로 시간이 맞지 않는 경우는 루프회수를 조정하여야 한다.**

프로그램 6.2를 패턴이 1초마다 이동하도록 하려면 프로그램 6.3과 같이 while루프의 패턴출력 문장 사이마다 msec_delay(1000)을 호출을 하면 된다.

```
#include <avr/io.h>
void msec_delay(int n);
int main()
{
     DDRA  = 0xFF;       // 포트A를 모두 출력포트로 설정
     while(1)            // 무한 루프
     {
          msec_delay(1000);
          PORTA = 0xFE;  // PA0을 점등
          msec_delay(1000);
          PORTA = 0xFD;  // PA1을 점등
          msec_delay(1000);
          PORTA = 0xFB;  // PA2를 점등
          . . .
          msec_delay(1000);
          PORTA = 0x7F;  // PA7을 점등
     }
}
void msec_delay(int n)             // 시간지연 함수
{
     int i;
     for(; n >0; n--)              // 1msec루프를 n회 반복
     {
          for(i=0; i<1000; i++)    // 1msec 시간지연 루프
          {
               asm("nop"::); asm("nop"::);
          }
     }
}
```

프로그램 6.3

☞ 프로그램 6.3에서 main()앞에 시간지연함수를 다음과 같이 선언하였음을 유념하라.

```
void  msec_delay(int n);
```

이는 msec_delay의 정의는 “함수이며 인수로서 int형 하나를 받으며

리턴 값이 없다."이며 이를 컴파일에게 알려주어 msec_delay()를 만나면 이 정의에 따라 컴파일하라는 의미이다.

☞ 프로그램에서 사용하는 함수는 이를 호출하기 전에 미리 이의 속성을 컴파일러에게 알려주어야 한다.

배열을 사용한 점등패턴 테이블 구성

프로그램 6.3에서 LED에 표시되는 형태가 바뀌면 while문내의 LED점등 부분을 일일이 변경하여야 할 뿐 아니라 패턴의 형태가 8가지보다 더 많게 되면 while문내에 패턴 점등부분을 추가하여야 한다. 패턴의 형태가 100개정도 된다고 생각해보자. 이 경우 프로그램 6.3과 같이 작성하는 것은 매우 번거로운 일이다.

프로그램 6.3과 같이 일정형태의 패턴을 반복적으로 사용하여야 하는 경우는 패턴을 배열을 사용한 테이블에 저장하면 편리하다. 출력하고자 하는 패턴은 테이블의 인덱스로 지정하면 된다.

프로그램 6.4는 배열을 사용하여 프로그램 6.3을 다시 작성한 것이다.

```
#include <avr/io.h>
unsigned char pattern[8]
     = {0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F};
int main()
{
     int i=0;            // 패턴 인덱스
     DDRA  = 0xFF;       // 포트A를 모두 출력포트로 설정
     PORTA = pattern[i]; // 처음 패턴으로 LED를 켠다.
     while(1)            // 무한 루프
     {
          msec_delay(1000);  // 1초 시간지연
          if(++i==8)i=0;     //i를 증가하고 결과가 8이면 리셋
          PORTA = pattern[i]; // i-번째 패턴으로 LED 켠다.
     }
}
```

프로그램 6.4

☞ 그림 6.3의 점등패턴은 1바이트(8비트)로 표시되고 부호가 없으므로 점등패턴을 저장할 배열 pattern[]을 unsigned char로 선언하였다.

☞ LED 점등패턴은 8개의 형태를 가지므로 배열 pattern을 8개의 요소를 가지도록 하였다. 배열과 점등패턴은 그림 6.5와 같이 순서대로 대응된다. 따라서 배열의 인덱스로서 점등 패턴을 지정할 수 있다.

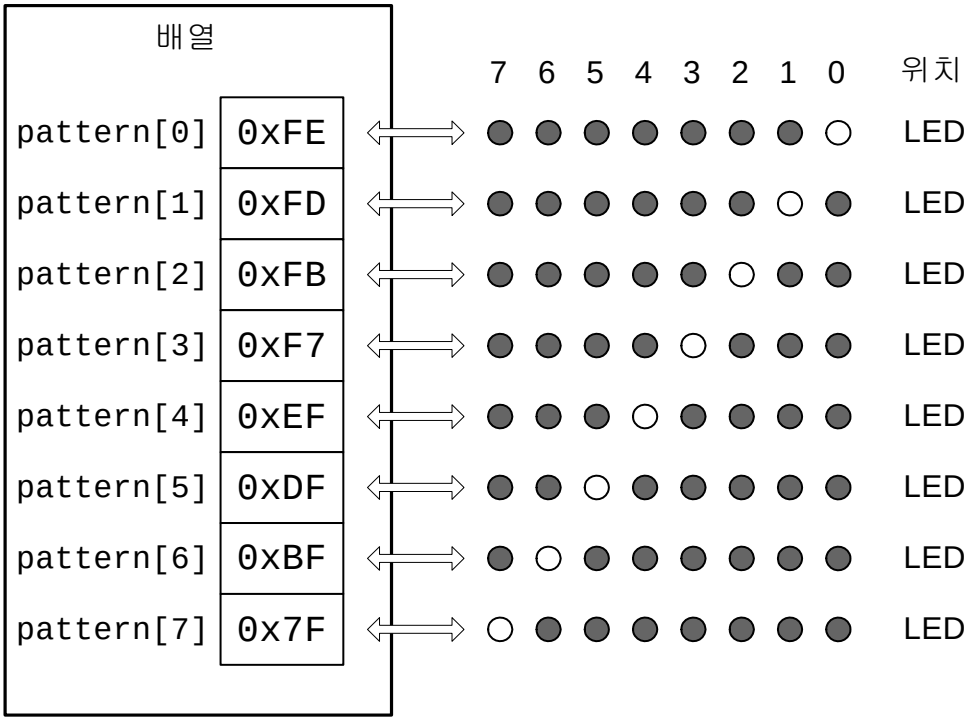


그림 6.5 LED 점등패턴과 배열

☞ PORTA = pattern[i]; 는 포트 A에 patter[i]값을 출력하라는 것이므로 포트 A에 연결된 LED는 i-번째 패턴으로 점등된다.

☞ ++i는 속한 문장을 평가하기 전에 i를 1 증가시키라는 것이므로 if(++i==8) i=0; 는 다음과 동일하다.

```
i++;
if(i==8) i=0;
```

☞ 변수 i는 배열의 인덱스이고 while 루프 내에서 인덱스만 증가한다.

☞ 8개의 점등패턴을 순환시키므로 인덱스가 8이 되었을 때 다시 0으로 리셋 하여야 한다.

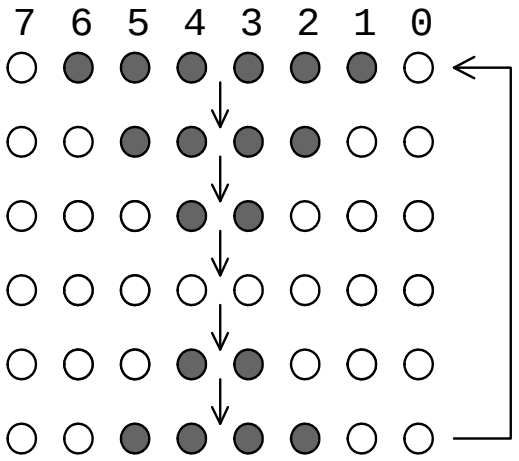
☞ 패턴의 형태가 변하면 패턴을 구성하는 테이블만 변경하면 된다. 또한 패턴의 개수가 변하여도 쉽게 프로그램을 변경할 수 있다. 패턴의 개

수가 8개에서 16개로 변한 경우를 보면 테이블을 크기를 변경하여 구성한 다음 while루프 내에서 인덱스회전을 다음과 같이 하면 된다.

```
unsigned char pattern[16] = {...};   // 16개의 패턴
. . . .
while(1)                 // 무한 루프
{
   if(++i==16)i=0;     // i를 증가하고 결과가 16이면 리셋
    PORTA = pattern[i];     // i-번째 패턴으로 LED 켠다.
}
```

과제

1. 프로그램 prac6-4.c는 프로그램 6.4를 완성한 것이다. 키트에 다운로드하여 프로그램을 수행한다. (5.4.3절을 참조하여 6.3절에서 작성한 프로젝트 lab6에서 소스파일 program6-1.c를 제거하고 prac6-4.c를 첨부한다.) 시간지연시간을 1000msec부터 5msec까지 점점 감소시켜가면서 패턴의 이동을 관찰하고 관찰된 현상을 설명하라.
2. 다음 패턴으로 LED를 점등하도록 프로그램을 작성하라. 단 패턴의 이동은 500msec로 할 것.



[프로그램 prac6-4.c]

```
//=========================================================
// 실습 6.4 배열을 사용하여 LED를 순차적으로 켠다.
//=========================================================
#include <avr/io.h>                  // I/O 레지스터 정의
void msec_delay(int msec);           // 시간지연함수 선언
//--------------------
// LED 점등패턴 테이블
//--------------------
unsigned char pattern[8]
     = {0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F};
int main()
{
     int i=0;

     DDRA = 0xFF;          // 포트 A를 출력으로 설정
     PORTA = pattern[i]; // 처음 패턴으로 LED를 켠다.
     while(1)
     {
          msec_delay(1000);     // 약 1sec 지연
          if(++i==8) i = 0;     // 마지막 패턴에서 인덱스 리셋
          PORTA = pattern[i];   // i-번째 패턴으로 점등
     }
}
//--------------
// 시간지연함수
//--------------
void msec_delay(int n)        // msec 단위 지연
{
     int i;
     for(; n>0; n--)
     {
          for(i=0; i<1000; i++)
          {
               asm("nop"::); asm("nop"::);
          }
     }
}
```

6.5 라이브러리 함수를 사용한 시간지연함수 구성

시간지연은 마이크로컨트롤러 프로그램에서 빈번히 사용되는 기능 중에 하나이다. 6.4절에서 사용한 시간지연함수 msec_delay()는 헛루프를 돌려 시간지연을 얻었다. 그러나 이 때 얻는 시간지연은 컴파일러의 코드 최적화 또는 사용 클록속도에 따라 많은 차이를 보인다. 즉 사용 환경에 따라 지연시간이 변경되므로 좋은 방법은 아니다.

좀 더 신뢰도 높은 시간 지연을 얻기 위해 컴파일러의 라이브러리가 제공하는 시간지연 함수를 사용하는 것이 바람직하다. WinAVR C-컴파일러에는 avr-libc라는 라이브러리를 제공하고 여기에는 msec단위 시간지연 함수와 μsec단위 시간지연함수를 제공한다.

```
void _delay_ms(double ms);          // msec단위 시간지연
void _delay_us(double us);          // usec단위 시간지연
```

avr-libc 라이브러리의 매뉴얼을 참조하면 위 함수를 사용하기 위해서는 헤더파일 "util/delay.h"를 소스파일에 포함시켜야 한다. 그러나 매뉴얼에 따르면 이 함수는 16MHz 클록을 쓸 때 각각 최대 16msec와 48usec를 지연시킬 수 있다. 16msec이상의 시간지연과 48usec이상의 시간지연을 위해서는 이를 이용해 시간지연함수를 구성할 필요가 있다.

앞에서 작성하였던 시간지연함수 msec_delay()를 _delay_ms()함수를 사용해서 재구성하고 이를 16msec이상의 시간지연에 사용하도록 하자.

```
#include <util/delay.h>

void msec_delay(int n)
{
    for(; n >0; n--)          // 1msec 시간지연을 n회 반복
        _delay_ms(1);         // 1msec 시간지연
}
```

새로운 시간지연 함수

실습 : [프로그램 prac6-4.c]의 시간지연함수를 새로운 시간지연함수로 대체하고 6.4절 과제 1과 과제2를 수행하여 본다.

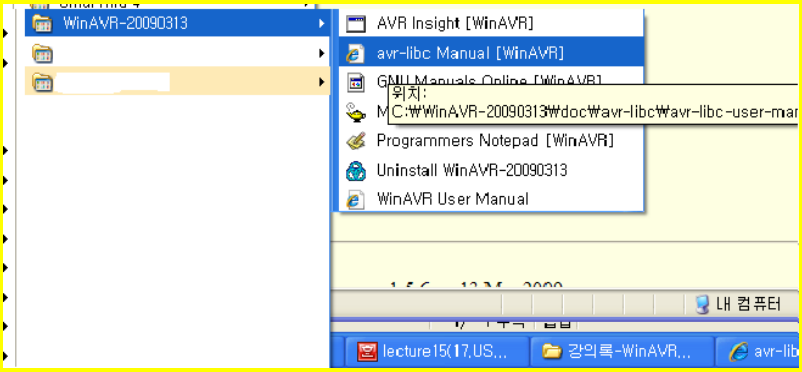
◆ 라이브러리

☞ Library가 책을 한 곳에 모아놓고 책을 대여하는 곳이라면 프로그램 언어에서 라이브러리는 함수를 작성하여 모아놓은 함수의 집합을 말한다. 라이브러리에 있는 함수를 사용하려면(도서관에서 책을 대여하는 것과 유사함) 프로그램에서 라이브러리에서 제공하는 함수를 사용하고, 실행파일을 만드는 빌드과정에서 라이브러리를 링크(Build에서 Link과정)하여야 한다.

☞ C 컴파일러에서는 libc.a(확장자 a는 archives(기록보관소)를 나타낸다.)라는 기본 라이브러리를 제공한다. 이는 별도의 링크 과정 없이 사용할 수 있다. C-언어에서 많이 쓰는 printf()와 같은 함수는 모두 라이브러리 libc.a에서 제공하는 함수이다. 이외의 라이브러리를 사용하고자 할 때는 반드시 링크를 하여야 한다. 17.5.2절에는 기본 라이브러리 이외의 라이브러리를 링크하는 방법을 보여준다.

☞ 라이브러리의 함수를 사용할 때는 반드시 매뉴얼의 사용법에 따라야 한다.

☞ WinAVR에서 제공하는 라이브러리를 모두 통틀어 avr-libc라 한다. WinAVR은 마이크로컨트롤러에 대한 컴파일러이기 때문에 avr-libc는 표준 C에서 제공하는 라이브러리의 함수와 차이가 난다. 따라서 표준 C-언어에서 자주 사용하는 함수라 할지라도 반드시 avr-libc Manual을 통해 사용법을 확인하여야 한다.



6.6 스위치 입력 실습

스위치를 그림 6.6과 같이 연결하고 스위치의 조작에 따라 LED 패턴을 이동시키도록 하자.

프로그램 6.4에 시간지연 함수를 삽입하는 것 대신 다음과 같이 스위치의 누름을 감시하는 프로그램을 삽입하면 스위치를 누를 때 마다 패턴을 이동시킬 수 있다.

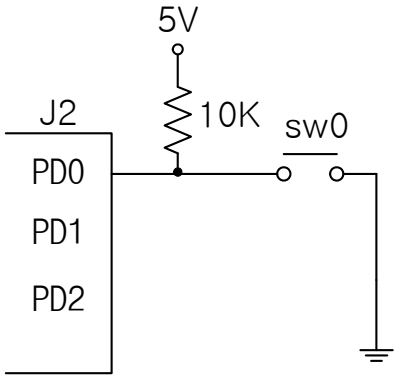


그림 6.6 스위치회로

```
#include <avr/io.h>
unsigned char pattern[8]
     = {0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F};

int main()
{
     int i=0;              // 패턴 인덱스
     DDRA  = 0xFF;         // 포트A를 모두 출력포트로 설정
     DDRD  = 0x00;         // 포트D를 모두 입력포트로 설정
     PORTA = pattern[i]; // 처음 패턴으로 LED를 켠다.

     while(1)              // 무한 루프
     {
          while(!(~PIND  & 0x01)); // 스위치 누름을 기다림
          if(++i==8)i=0;           // 인덱스가 8이면 리셋
          PORTA = pattern[i];      // i-번째 패턴으로 LED 켬
     }
}
```

프로그램 6.5

☞ 스위치를 외부풀업을 하여 포트D에 연결하였으므로 포트D를 입력포트로만 설정하면 된다.

☞ 포트D의 스위치에 연결된 0-번째 핀 외에 다른 핀은 아무런 연결이 없으므로 입력으로 읽어 들인 값들은 의미가 없다. 따라서 스위치의 상태에 따른 입력 값은 다음과 같다. 다음에서 'x'는 어떤 값인지 모른

다는 뜻이다.(스위치가 닫힐 때 PIND의 비트0은 0이 읽히므로 비트별 NOT을 취하여(~PIND) 스위치가 닫힐 때 비트0이 1이 되도록 하였다.)

```
        핀(비트)          :    7 6 5 4 3 2 1 0
   ~PIND(스위치 열림)     :    x x x x x x x 0
   ~PIND(스위치 닫힘)     :    x x x x x x x 1
```

☞ 위에서 sw0의 개폐를 판단할 때 의미 없는 핀의 입력을 판단에서 배제하고 sw0의 입력을 받는 핀0의 데이터만 사용하는 것이 필요하다. 이를 위해 의미 없는 부분을 항상 0으로 리셋하고 핀0의 데이터는 그대로 보존하도록 하자. 이는 다음과 같이 비트별 AND를 사용하여 이룰 수 있다.

```
                  sw0 스위치 열림          sw0 스위치 닫힘
핀                7 6 5 4 3 2 1 0          7 6 5 4 3 2 1 0
~PIND             x x x x x x x 0          x x x x x x x 1
 0x01             0 0 0 0 0 0 0 1          0 0 0 0 0 0 0 1
비트별 AND        --------------           --------------
                  0 0 0 0 0 0 0 0          0 0 0 0 0 0 0 1
                    모두 리셋 ↳보존          모두 리셋 ↳보존
```

☞ 핀 0은 ~PIND에서 비트0에 해당한다. 비트별 AND할 데이터는 판단에 사용할 비트0을 1로 하고, 판단에서 배제할 비트들인 비트 1~비트7은 0으로 설정하여 0x01이 된다.

☞ 따라서 포트D의 입력은 PIND를 읽으면 되므로 (~PIND & 0x01)의 상태(True, False)를 검사하면 스위치 sw0의 상태를 알 수 있다. True이면 스위치가 닫힌 상태이고 False이면 스위치가 열린 상태이다.

프로그램 6.5를 수행하면 스위치를 누르고 있는 동안 패턴이 회전하나 너무 빨라 감지할 수가 없고, 스위치를 떼면 회전이 멈추게 된다. 이를 보완하기 위하여 스위치를 뗐다 누르면 패턴이 회전하도록 하자. 프로그램 6.5에서 스위치가 떨어짐을 점검하는 부분을 첨가하여 while()루프 안을 프로그램 6.6과 같이 변경하자.

```
while(1)
{
   while(!(~PIND & 0x01)); // 스위치 누름을 기다림
   if(++i==8) i=0;         // i를 증가하고 8이면 리셋한다.
   PORTA = pattern[i];     // 현재 패턴을 출력한다.
   while(~PIND & 0x01);    // 스위치 떨어짐을 기다림
}
```

프로그램 6.6

◆ 데이터 마스크(Mask)

☞ 위와 유사하게 특정비트를 0으로 리셋하고 나머지는 원상태를 보존하려면 특정비트가 0이고 나머지는 1이 되는 수와 비트별 AND를 하면 된다. 변수 var의 비트1과 비트6을 리셋하고 나머지를 보존하려면 :

```
   7 6 5 4 3 2 1 0      비트위치
   x x x x x x x x      var
   1 0 1 1 1 1 0 1      0xBD = ~0x42 = ~0b01000010
-------------------     비트별 AND
   x 0 x x x x 0 x      var & ~0x42
     ↳리셋     ↳리셋
```

x: 이 비트들은 연산 후에도 그대로 보존됨

☞ 위와는 달리 특정비트를 1로 세트하고 나머지는 원상태를 보존하려면 특정비트가 1이고 나머지는 0이 되는 수와 비트별 OR를 하면 된다. 변수 var의 비트1과 비트6을 1로 세트하고 나머지를 보존하려면 :

```
   7 6 5 4 3 2 1 0      비트위치
   x x x x x x x x      var
   0 1 0 0 0 0 1 0      0x42
-------------------     비트별 OR
   x 1 x x x x 1 x      var | 0x42
```

☞ 위와 같이 필요한 비트만 선별적으로 뽑아내는 것을 "**마스크(Mask)한다**"라고 한다. 마치 가면의 구멍을 통해 제한된 영역만을 외부에 알려주는 것과 같다.

스위치 입력 채터링 방지

프로그램 6.6을 수행하면 스위치를 누르고 뗄 때마다 패턴이 회전하는 것을 쉽게 알 수 있다. 그러나 한 번의 스위치 동작에 패턴이 두 번 이상 회전이 자주 발생하는 것을 알 수 있다. 이런 현상은 필히 방지하여야 한다.

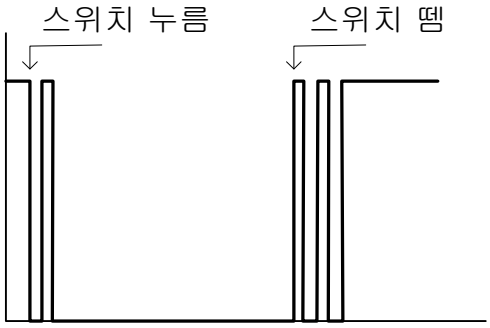


그림 6.7 스위치 입력신호의 채터링

스위치를 조작하게 되면 그림 6.7과 같이 한 번의 기계적인 개폐동작에 대해 전기적으로는 여러 번 개폐동작이 발생하기도 한다. 이러한 것을 채터링이라 한다. 마이크로프로세서가 아주 빠른 속도로 포트입력을 읽으면 스위치 조작을 한번 하지만 마이크로프로세서는 여러 번의 개폐동작이 일어난 것으로 판단하게 된다. 이런 현상이 발생하면 한 번의 스위치 조작에 2번 이상의 패턴회전이 발생하게 된다.

채터링을 방지하는 것을 스위치 디바운싱(Debouncing)이라하고 하드웨어적인 방법과 소프트웨어적인 방법이 있다. 여기서는 소프트웨어적으로 처리하는 방법을 고려한다.

그림 6.7에서보면 스위치 상태변화가 일어난 후 신호가 얼마간 채터링을 한 후 안정화된다. 이를 이용하여 신호의 변화가 감지되면 입력이 안정화될 때까지 기다린 후 다음 신호입력을 검사하도록 한다. 안정화 될 때까지는 아무런 입력을 받지 않으므로 채터링 신호를 읽지 않을 수 있다. 이와 같은 채터링 방지 기능을 프로그램 6.6에 삽입하여 프로그램 6.7을 작성한다.

☞ 안정화 될 때까지의 기다려야 하는 시간은 일정하지 않으므로 테스트를 통하여 경험적으로 설정하여야 한다.

☞ 디바운싱을 위한 지연시간의 설정을 길게 하면 확실히 채터링을 방지할 수 있다. 그러나 스위치 입력감도가 떨어진다. 지연시간을 짧게 하면 스위치 입력이 민감해지는 대신 디바운싱이 완전하지 않을 수 있다.

```
while(1)
{
   while(!(~PIND & 0x01)); // 스위치 누름을 기다림
   msec_delay(20);         // 디바운싱을 위한 시간 지연

   if(++i==8) i=0;         // 패턴을 회전한다.
   PORTA = pattern[i];

   while(~PIND & 0x01);    // 스위치 떨어짐을 기다림
   msec_delay(20);         // 디바운싱을 위한 시간 지연
}
```

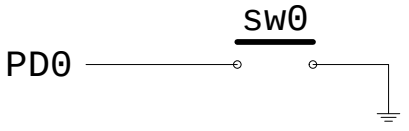
프로그램 6.7

프로그램 prac6-6.c는 프로그램 6.7을 완성한 것이다.

실습 : 프로그램 6.6, 6.7, prac6-6.c를 수행하여 차이점을 확인하라.

과제 :

1. 프로그램 prac6-6.c에서 디바운싱을 위한 시간지연을 짧게 변경하면서 채터링 방지효과를 관찰하라.(매크로 DEBOUNCING_DELAY의 값을 변경하면 시간지연이 변한다.)
2. 스위치회로에 풀업 저항을 제거하면 신호가 floating상태가 된다. 프로그램을 수행하고 현상을 관찰하라. 표6.1을 참조하여 내부적으로 풀업이 되도록 프로그램을 작성하여 수행하고 정상적으로 작동함을 관찰하라.



풀업저항을 제거하면 스위치 Off일 때 floating상태가 됨

‡ C-언어에서 매크로(Macro)의 사용

매크로는 매우 다양한 용도로 사용될 수 있으나 여기서는 문장을 대체하는 용도에 대해 [프로그램 prac6-5.c]를 예로 들어 설명한다.

매크로는 선행처리기 지시어 #define을 사용하여 정의한다.

```
#define DEBOUNCING_DELAY   20   //디바운싱 지연시간(msec)
```

여기서 DEBOUNCING_DELAY를 매크로라고 한다. 매크로를 정의한 후 다음의 왼 쪽 문장은 오른 쪽 문장과 동일하다.

```
msec_delay(DEBOUNCING_DELAY); <==> msec_delay(20);
```

☞ 매크로 DEBOUNCING_DELAY는 20으로 대체되었음을 알 수 있다.

☞ 위의 좌우 문장은 동일한 것이지만 왼쪽 문장의 의미가 더욱 확실하다. 왼쪽은 디바운싱을 위한 지연시간(DEBOUNCING_DELAY)동안 지연하는 의미를 알 수 있지만, 오른쪽 문장에서는 20이라는 수의 의미를 알기 힘들다. 즉 매크로를 사용함으로써 프로그램의 의미를 확실하게 전달할 수 있다.

☞ [프로그램 prac6-6.c]에서 디바운싱을 위한 시간지연은 스위치 떨어짐과 누름을 검사할 때 두 번 사용된다. 시간지연을 50msec로 변경하는 것을 고려해보자. 매크로를 사용하지 않으면 지연시간을 변경하기 위해 두 문장의 값을 바꾸어야 한다. 그러나 매크로를 사용하면 매크로를 정의한 한 문장만을 변경하면 된다.

```
#define DEBOUNCING_DELAY     20
     ==>  #define DEBOUNCING_DELAY     50
```

이와 같이 하나의 상수를 여러 군데 사용할 때는 매크로를 사용하는 것이 편리하다.

☞ 매크로는 변수가 아니므로 메모리를 차지하지 않는다.

☞ C-언어에서 변수는 주로 소문자를 많이 사용하므로 매크로를 일반 변수와 구분하기 위해 대문자를 사용하여 정의하는 것이 일반적이다.

[프로그램 prac6-6.c]

```
//=======================================================
// 실습 6.6 스위치 입력을 사용하여 LED를 순차적으로 켠다.
//=======================================================
#include <avr/io.h>                 // I/O 레지스터 정의
#include <util/delay.h>             // 시간지연 함수용 헤더파일

#define   DEBOUNCING_DELAY   20   // 디바운싱 지연시간(msec)
void msec_delay(int n);             // 시간지연함수
//=======================================================
// LED 점등패턴 테이블
//=======================================================
unsigned char pattern[8]
     = {0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F};
int main()
{
     int i=0;

     DDRA = 0xFF;              // 포트 A를 출력으로 설정
     DDRD = 0x00;              // 포트 D를 입력으로 설정
     PORTA = pattern[i];       // 처음 패턴으로 LED를 켠다.
     while(1)
     {
          while(!(~PIND & 0x01)); // 스위치 누름을 기다림
          msec_delay(DEBOUNCING_DELAY);     // 시간지연

          if(++i==8) i=0;          //마지막 패턴에서 인덱스리셋
          PORTA = pattern[i];      // i-번째 패턴으로 점등

          while(~PIND&0x01);       // 스위치 떨어짐을 기다림
          msec_delay(DEBOUNCING_DELAY);     // 시간지연
     }
}
void msec_delay(int n)
{
     for(; n >0; n--)          // 1msec 시간지연을 n회 반복
          _delay_ms(1);        // 1msec 시간지연
}
```

6.7 도트 매트릭스에 글자 표시하기

6.4절에서는 8x8 도트 매트릭스의 1행에 위치한 LED에 주어진 패턴으로 점등을 하였다. 여기서는 그림 6.8과 같이 LED의 패턴을 변경할 때 도트 매트릭스의 행을 이동시켜보기로 하자.

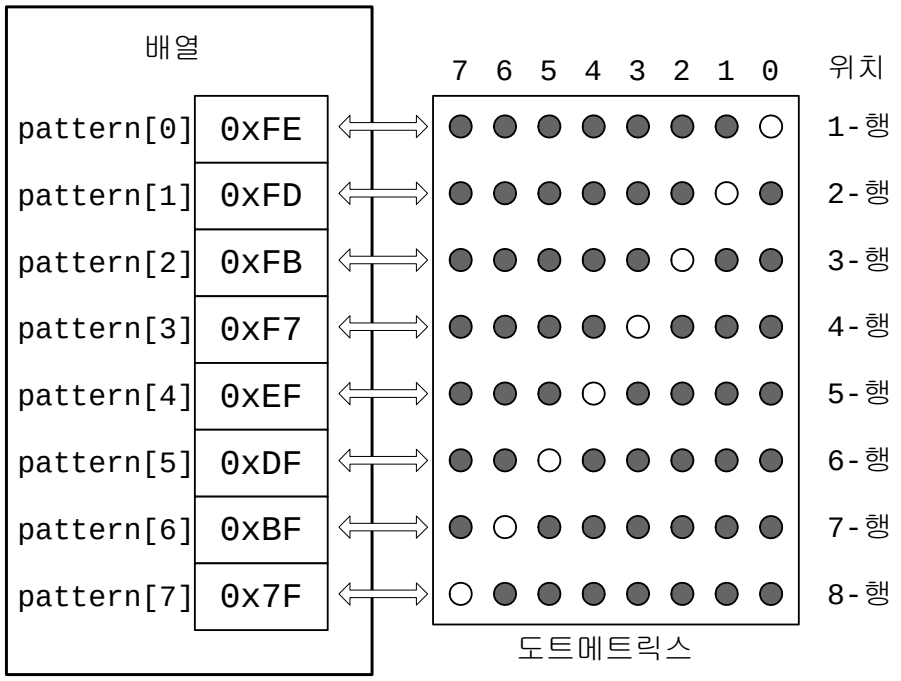


그림 6.8 도트 매트릭스 패턴표시

이를 위해 도트 매트릭스와 ATmega128을 그림 6.9와 같이 연결한다.

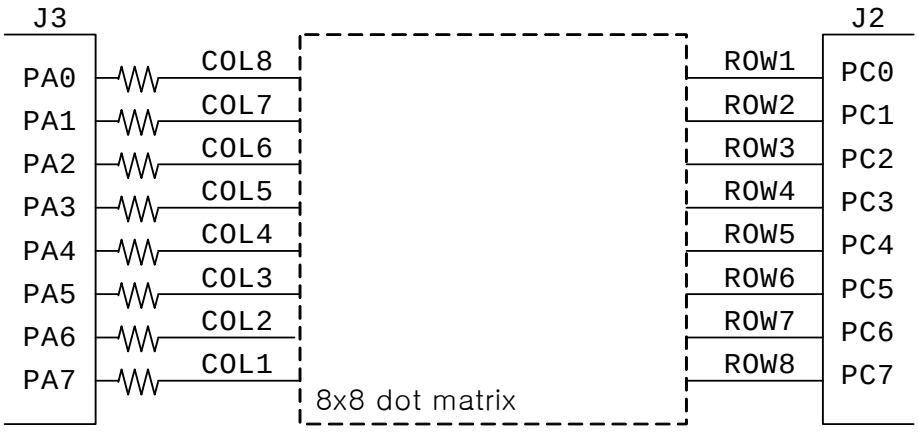


그림 6.9 글자표시를 위한 도트 매트릭스 연결

도트 매트릭스의 ROW1핀이 5V에 연결된 그림 6.2와는 달리 ATmega128의 PC0핀에 연결되어있으므로 PC0에 논리1(5V)을 출력하면 도트 매트릭스의 1행에 원하는 패턴으로 LED를 켤 수 있다. 2행을 사용하려면 PC1핀에 논리1을 출력하면 된다. LED 패턴을 변경할 때 행을 이동하려면 LED를 켤 행에 논리1을 출력하고 원하는 패턴을 포트 A에 출력하면 된다. prac6-7.c는 (i+1)-번째 행에 i-번째 패턴을 순차적으로 점등하는 프로그램이다.

☞ (i+1)-번째 행을 켜려면 포트 C의 i-번째 비트에 논리 1을 출력하면 된다.

☞ i-번째 비트를 1로 세트하려면 왼쪽 쉬프트연산자 "<<"를 사용하면 편리하다.

```
0-번째 비트를 세트 : 0b00000001 <==> 0x01 << 0
1-번째 비트를 세트 : 0b00000010 <==> 0x01 << 1
      . . .
i-번째 비트를 세트 :                  0x01 << i
```

☞ 포트 C의 i-번째 비트를 1로 세트하려면

```
PORTC = 0x01 << i;        // (i+1)행을 켠다.
```

실습 : prac6-7.c의 시간지연은 1000msec로 설정되어 있다. 이를 50, 20, 10, 5msec로 줄이면서 프로그램을 수행한다. 시간지연이 줄어들면서 도트 매트릭스의 모든 행이 켜져 있는 것처럼 보인다.

과제 :

1. 도트 매트릭스에 나타나는 패턴이 깜박거림이 없어지는 시간지연을 구하라. 그리고 이를 이용하여 도트 매트릭스에 깜박거림이 없이 문자 'A'를 표시하도록 프로그램을 작성하라.
2. 도트 매트릭스에 문자 'A'를 1초간 깜박거림이 없이 표시하고, 다음 1초간은 COL8을 끈 'A'를 표시, 다음 1초간은 COL7을 끈 'A'를 표시하며, 같은 방법으로 COL6, COL5, COL4, COL3, COL2, COL1을 각각 끈 'A'를 1초간 표시한 후 다시 COL8을 끈 'A'를 표시하는 것으로 돌아가서 같은 동작을 반복하도록 프로그램을 작성하라.

[프로그램 prac6-7.c]

```
//=======================================================
// 실습 6.7 도트 매트릭스 켜기
//=======================================================
//
#include <avr/io.h>                 // I/O 레지스터 정의
#include <util/delay.h>
void msec_delay(int msec);          // 시간지연함수 선언

unsigned char pattern[8]
     = {0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F};

int main()
{
     int i=0;

     DDRA = 0xFF;         // 포트 A를 출력으로 설정
     DDRC = 0xFF;         // 포트 C를 출력으로 설정
     PORTC = 0x01 << i;   // 1-행을 켠다.
     PORTA = pattern[i]; // 처음 패턴으로 LED를 켠다.

     while(1)
     {
          msec_delay(1000);      // 약 1sec 지연
          if(++i==8) i = 0;      // 마지막 패턴에서 인덱스 리셋
          PORTC = 0x01 << i;     // (i+1)-번째 행을 점등
          PORTA = pattern[i];    // i-번째 패턴으로 점등
     }
}

void msec_delay(int n)        // msec 단위 지연
{
     for(; n >0; n--)         // 1msec 시간지연을 n회 반복
          _delay_ms(1);       // 1msec 시간지연
}
```