제 7 장 인터럽트(Interrupt)

7.1 인터럽트 메커니즘

우리가 어떤 가게의 주인이라고 생각해보자. 가게에 누가 들어왔는지 아닌지 판단하기 위해서는 두 가지 방법을 쓸 수 있다. 하나는 일을 하다가 가끔씩 계속적으로 문을 점검하는 것이고, 다른 방법은 문에 종을 달아 놓아 소리가 나면 문을 점검하는 것이다. 전자와 같이 계속적으로 어떤 사건의 발생을 점검하는 방법을 폴링(Polling)이라고 하고, 후자와 같이 어떤 사건이 발생하면 그 때 CPU에 알려서 처리 요구를 하는 것을 인터럽트라 한다. 인터럽트를 사용하면 사건이 발생할 때만 CPU가 대응하므로 매우 효과적인 방법이라 하겠다.

그림 7.1은 인터럽트가 발생하였을 때 작업수행 경로를 보여준다. 인터럽트가 발생하면 미리 지정된 위치로 분기하여 인터럽트 처리를 한다. 이 때 인터럽트를 처리하는 코드를 인터럽트 서비스루틴이라 한다. 처리가 끝나면 인터럽트 서비스루틴으로 분기하기 전의 위치로 복귀하여 작업을 계속적으로 수행하게 된다.

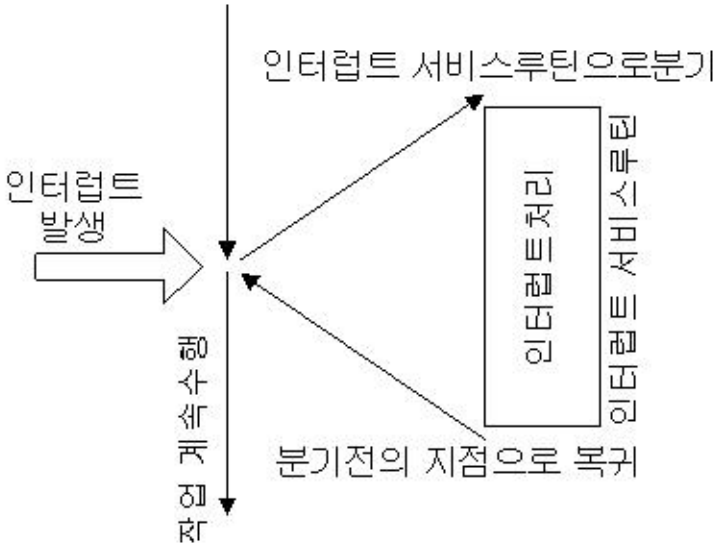

그림 7.1 인터럽트 처리과정

일반적인 함수도 작업수행 중 다른 곳으로 분기한다는 면에서 인터럽트와 유사하다. 그러나 함수는 그림 7.2와 같이 함수의 호출에 의해서 분기를 한다. 함수를 호출할 때 그림 7.2처럼 어셈블리로 "CALL func"을 사용하는 데 이 때 func가 함수가 있는 위치, 즉 주소를 나타낸다. (C-언어에서는 함수의 주소는 함수명과 동일하다. 함수가 func()이면 함수가 위치한 주소는 func이다.) 즉 함수 호출에 의해 프로그램은 함수로 분기하여 작업을 수행한 후 함수호출 다음 문장으로 복귀한다. 함수의 호출은 언제 분기할 지를 사용자가 명확히 알고 있으므로 확정적이라 할 수 있다.

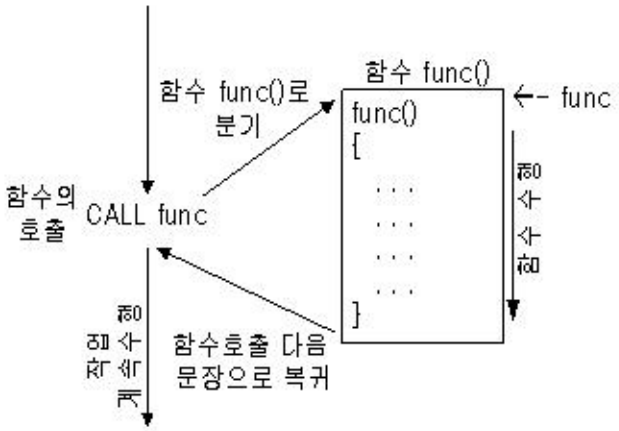

그림 7.2 함수의 수행

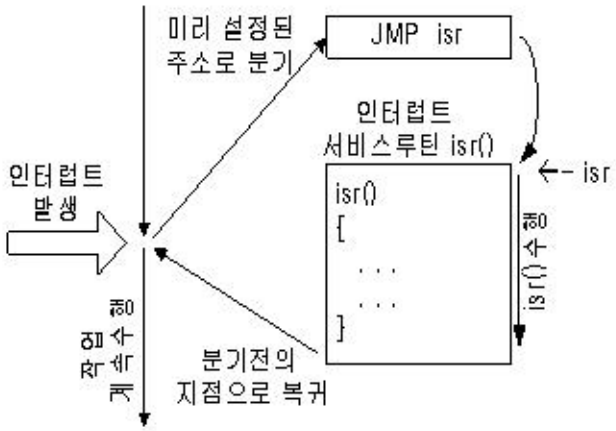

그림 7.3 인터럽트서비스루틴 수행

이에 비해 인터럽트를 발생시키는 사건은 언제 발생할지 모르는 불확정적인 것이다. 이러한 것을 컴퓨터에서는 비동기적(Asynchronous)이라 말한다. 또한 인터럽트를 발생시키는 사건자체는 어디로 분기할 것인가를 말해주지 않는다. 대신 그림 7.3과 같이 특정 인터럽트가 발생하면 미리 정해진 주소를 분기하게 된다. 사용자는 인터럽트 처리를 위해 이 주소의 메모리에 사용자가 작성한 인터럽트서비스루틴 isr()로 분기하라는 어셈블리 명령어 "JMP isr"을 저장한다. 그러면 그림 7.3과 같은 과정을 거쳐 isr()을 수행한 후 인터럽트로 인해 중단되었던 지점으로 복귀하게 된다.

☞ 보통 CPU는 여러 종류의 인터럽트를 처리할 수 있는 데, 각 인터럽트에 고유한 번호가 부여된다. 이를 인터럽트 벡터(Vector)라 부른다.
☞ 인터럽트가 걸리면 프로그램은 인터럽트 벡터테이블이라는 메모리에 지정된 한 부분으로 분기한다. 벡터 테이블에는 "각 인터럽트에 대한 인터럽트서비스루틴으로 점프"하라는 어셈블리 명령어를 위치시킨다.

7.2 ATmega128의 인터럽트

7.2.1 인터럽트 벡터

ATmega128은 35개의 인터럽트 소스에 대해 반응한다. 표 7.1은 각 인터럽트에 대한 인터럽트 벡터, 분기 주소를 나타내고 있다.

표 7.1 리셋 및 인터럽트 벡터

우선순위	벡터번호	분기주소	인터럽트 원인	인터럽트 정의
높음 ⇑ 우선순위 ⇓ 낮음	1	0000H	RESET	리셋
	2	0002H	INT0	외부 인터럽트 요구 0
	3	0004H	INT1	외부 인터럽트 요구 1
	4	0006H	INT2	외부 인터럽트 요구 2
	5	0008H	INT3	외부 인터럽트 요구 3
	6	000AH	INT4	외부 인터럽트 요구 4
	7	000CH	INT5	외부 인터럽트 요구 5
	8	000EH	INT6	외부 인터럽트 요구 6
	9	0010H	INT7	외부 인터럽트 요구 7
	10	0012H	TIMER2 COMP	타이머/카운터 2 비교 일치
	11	0014H	TIMER2 OVF	타이머/카운터 2 오버플로
	12	0016H	TIMER1 CAPT	타이머/카운터 1 캡춰
	13	0018H	TIMER1 COMPA	타이머/카운터 1 비교 일치 A
	14	001AH	TIMER1 COMPB	타이머/카운터 1 비교 일치 B
	15	001CH	TIMER1 OVF	타이머/카운터 1 오버플로
	16	001EH	TIMER0 COMP	타이머/카운터 0 비교 일치
	17	0020H	TIMER0 OVF	타이머/카운터 0 오버플로
	18	0022H	SPI, STC	SPI 전송완료
	19	0024H	USART0, RX	USART0 수신 완료
	20	0026H	USART0, UDRE	USART0 데이터 레지스터 빔
	21	0028H	USART0, TX	USART0 송신 완료
	22	002AH	ADC	ADC 변환 완료
	23	002CH	EE READY	EEPROM 준비
	24	002EH	ANALOG COMP	아날로그 비교기
	25	0030H	TIMER1 COMPC	타이머/카운터 1 비교 일치 C
	26	0032H	TIMER3 CAPT	타이머/카운터 3 캡춰
	27	0034H	TIMER3 COMPA	타이머/카운터 3 비교 일치 A
	28	0036H	TIMER3 COMPB	타이머/카운터 3 비교 일치 B
	29	0038H	TIMER3 COMPC	타이머/카운터 3 비교 일치 C
	30	003AH	TIMER3 OVF	타이머/카운터 3 오버플로
	31	003CH	USART1, RX	USART1 수신 완료
	32	003EH	USART1, UDRE	USART1 데이터 레지스터 빔
	33	0040H	USART1, TX	USART1 송신 완료
	34	0042H	TWI	Two-wire Serial Interface
	35	0044H	SPM READY	Store Program Memory 준비

☞ 각 인터럽트 벡터에 대해 설정된 분기주소가 모여 있는 메모리 블록 0000H~0044H가 인터럽트 벡터테이블이라 한다.

☞ 이 주소는 프로그램 메모리의 주소이다. 테이블에서 각 인터럽트 벡터에 대해 2워드(ATmega128의 프로그램 메모리는 기본단위가 워드로 구성되어 있다.)씩 할당되어 있다. 이 2워드 메모리에 2워드 명령어인 "JMP xxxx"를 써 넣을 수 있다. 여기서 xxxx는 인터럽트 서비스루틴의 주소이다.

각 인터럽트에 대한 설명은 이 후 실습 중 필요할 때 설명할 것이다.

※ 퓨즈비트 BOOTRST와 MCUCR레지스터를 사용하여 분기 주소를 부트로더 번지로 하는 모드는 교재에서 다루지 않는다.

7.2.2 인터럽트의 허용/금지

아주 중요한 일을 수행할 때는 외부로부터 방해(인터럽트)를 받지 않도록 할 필요가 있다. 마이크로컨트롤러에서도 경우에 따라서 모든 인터럽트 또는 특정 인터럽트로부터 CPU가 방해받지 말아야 할 필요가 있다. 이것은 그림 7.4에서 보여주는 ATmega128의 인터럽트 허용/금지 메커니즘을 통해 제어된다.

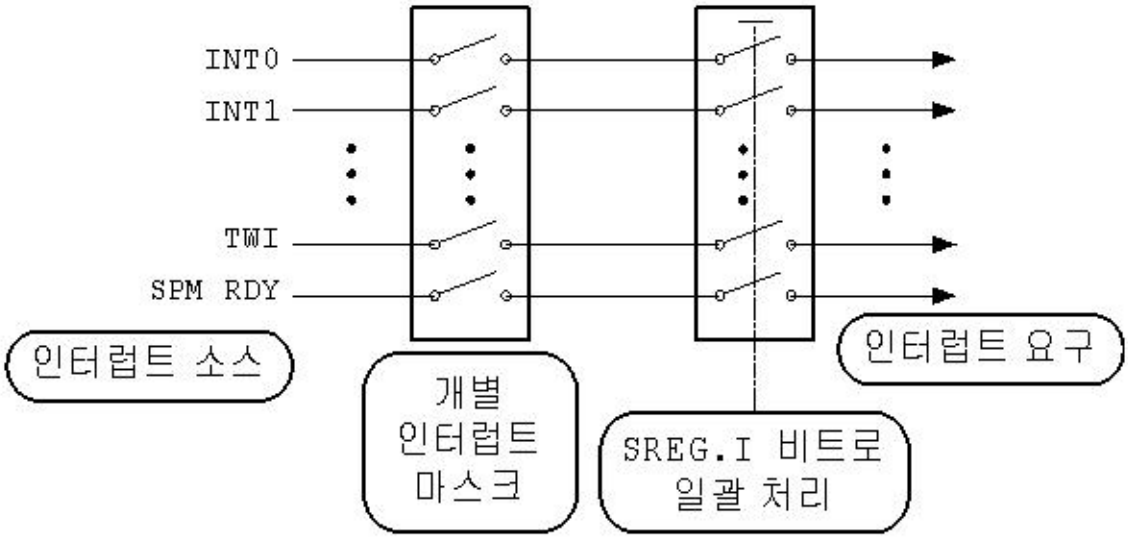

그림 7.4 인터럽트 허용/금지 메커니즘

☞ 각 인터럽트에 대해서는 인터럽트 마스크 비트가 존재하며 이를 사용하여 개별적으로 인터럽트의 허용/금지를 제어할 수 있다.
☞ 상태레지스터의 비트 SREG.I를 리셋하면 모든 인터럽트가 금지된다.
☞ SREG.I를 세트하면 개별적으로 허용된 인터럽트만 허용된다.
☞ WinAVR에서는 헤더파일 "avr/interrupt.h"를 포함하고 다음 함수를 호출하면 인터럽트를 허용/금지할 수 있다.

```
sei();    // 인터럽트 허용
cli();    // 인터럽트 금지
```

7.2.3 인터럽트 처리과정과 인터럽트 플래그(flag)

표 7.1을 보면 각 인터럽트에 대해 우선순위가 정해져 있다. 우선순위는 인터럽트 처리가 동시에 요구될 때 처리되는 순서를 말한다.

☞ 여러 인터럽트가 동시에 처리를 요구하면 우선순위가 가장 높은 인터럽트가 가장 먼저 처리된다.
☞ 인터럽트가 발생하면 인터럽트에 해당하는 플래그 비트가 세트된다. 이 플래그 비트는 (1)프로그램에서 비트를 강제로 리셋 할 수도 있고 (2) 또는 해당 인터럽트 서비스 루틴이 시작되면 자동으로 리셋된다.
☞ 인터럽트 서비스 루틴을 수행할 때 CPU는 자동적으로 SREG.I를 리셋하여 모든 인터럽트를 금지하고 서비스 루틴의 종료와 함께 인터럽트를 허용한다. 따라서 인터럽트 수행 중 우선순위가 더 높은 인터럽트가 발생하더라도 이의 처리는 이미 수행하고 있는 서비스 루틴의 종료까지 연기된다.
☞ 인터럽트 서비스 루틴 수행 중에 발생하는 인터럽트는 당장 처리되지는 않으나 플래그 비트를 세트한다. CPU는 현재 수행 중인 인터럽트 서비스루틴을 종료한 후 모든 플래그 비트를 조사하여 플래그가 세트된 인터럽트 중 가장 우선순위가 높은 인터럽트를 처리하게 된다.
☞ 인터럽트 서비스루틴의 시작과 동시에 CPU가 인터럽트를 금지시키므로 한번 시작된 인터럽트 서비스루틴은 다른 인터럽트에게 선점(Preempt)되지 않는다. 경우에 따라서는 다른 인터럽트의 선점을 허용해야 하는 데, 이는 인터럽트 서비스루틴 내에서 모든 인터럽트를 허용하는 어셈블리 명령 sei를 수행하면 된다. 물론 마스크비트가 세트된 인터럽트만 허용이 된다.

7.3 WinAVR에서 인터럽트 사용

7.1절에서 설명한 것처럼 함수와 인터럽트 서비스루틴의 수행은 차이점이 있어서 인터럽트 서비스루틴을 일반함수처럼 작성할 수 없다. WinAVR 컴파일러에서는 다음과 같이 인터럽트 서비스루틴의 작성하여야 한다.

```
#include <avr/interrupt.h>   // 인터럽트 헤더파일
ISR(vector)
{
     // 여기에 인터럽트 서비스 작업 코드를 삽입할 것
}
```

☞ 인터럽트서비스루틴에는 인터럽트벡터 외에 어떠한 정보도 인자를 사용하여 전달할 수 없다.

☞ 인터럽트서비스루틴은 반환 값이 없는 void형으로 작성된다.

☞ 인자인 vector는 표 7.2에 정의된 매크로를 사용한다.

표 7.2 WinAVR에서 인터럽트 벡터의 매크로

벡터번호	매크로	벡터번호	매크로
2	INT0_vect	19	USART0_RX_vect
3	INT1_vect	20	USART0_UDRE_vect
4	INT2_vect	21	USART0_TX_vect
5	INT3_vect	22	ADC_vect
6	INT4_vect	23	EE_READY_vect
7	INT5_vect	24	ANALOG_COMP_vect
8	INT6_vect	25	TIMER1_COMPC_vect
9	INT7_vect	26	TIMER3_CAPT_vect
10	TIMER2_COMP_vect	27	TIMER3_COMPA_vect
11	TIMER2_OVF_vect	28	TIMER3_COMPB_vect
12	TIMER1_CAPT_vect	29	TIMER3_COMPC_vect
13	TIMER1_COMPA_vect	30	TIMER3_OVF_vect
14	TIMER1_COMPB_vect	31	USART1_RX_vect
15	TIMER1_OVF_vect	32	USART1_UDRE_vect
16	TIMER0_COMP_vect	33	USART1_TX_vect
17	TIMER0_OVF_vect	34	TWI_vect
18	SPI_STC_vect	35	SPM_RDY_vect

7.4 외부인터럽트

ATmega128은 8개의 외부인터럽트 INT0 ~ INT7을 제공한다. 이들은 표7.3과 같이 연관되어 있는 외부 핀들이 인터럽트 소스가 된다.

표 7.3 외부인터럽트 소스 핀

인터럽트	핀의 다른용도	보드 핀	인터럽트	핀의 다른용도	보드 핀
INT0	PD0, SCL	J2,17	INT4	PE4, OC3A	J2,5
INT1	PD1, SDA	J2,18	INT5	PE5, OC3C	J2,6
INT2	PD2, RXD1	J2,19	INT6	PE6, T3	J2,7
INT3	PD3, TXD1	J2,20	INT7	PE7, IC3	J2,8

☞ 인터럽트를 사용할 때 INTn 핀과 공유하는 입출력 핀의 방향은 입력으로 설정하여야 한다.(예: INT0을 사용할 때 DDRD.0 = 0)

☞ 인터럽트를 사용하더라도 공유하는 입출력 핀을 읽으면 인터럽트 신호의 상태를 알 수 있다.

외부 인터럽트와 관련된 레지스터를 요약하면 다음과 같다.

■ 외부 인터럽트 제어 레지스터 A

(External Interrupt Control Register A) : EICRA

비트	7	6	5	4	3	2	1	0	
	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00	EICRA
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	

• 비트 7..0 - 외부 인터럽트 0~3까지의 인터럽트 감지방법을 제어한다. 각 비트의 선택에 따른 인터럽트 감지 방법은 표7.4와 같다.

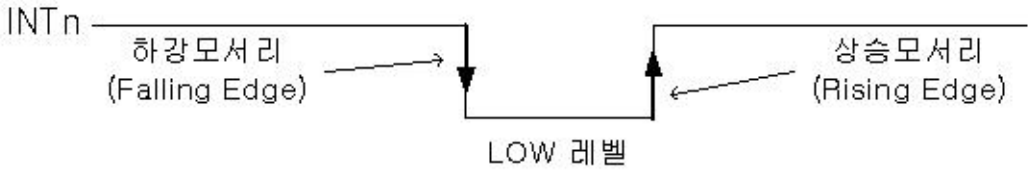

그림 7.5 신호 형태

표 7.4 인터럽트 감지 제어(n=0,1,2,3)

ISCn1	ISCn0	설명
0	0	INTn 핀의 Low 레벨 신호가 인터럽트 요구
0	1	사용하지 않음
1	0	INTn 핀의 하강모서리가 비동기적으로 인터럽트 요구
1	1	INTn 핀의 상승모서리가 비동기적으로 인터럽트 요구

■ **외부 인터럽트 제어 레지스터 B**

(External Interrupt Control Register B) : EICRB

비트	7	6	5	4	3	2	1	0	
	ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40	EICRB
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	

• 비트 7..0 - 외부 인터럽트 4~7까지의 인터럽트 감지방법을 제어한다. 각 비트의 선택에 따른 인터럽트 감지 방법은 표3.5와 같다.

표 7.5 인터럽트 감지 제어(n=4,5,6,7)

ISCn1	ISCn0	설명
0	0	INTn 핀의 Low 레벨 신호가 인터럽트 요구
0	1	INTn 핀의 하강모서리/상승모서리 모두 인터럽트 요구
1	0	INTn 신호의 두 샘플 사이의 하강모서리가 인터럽트 요구
1	1	INTn 신호의 두 샘플 사이의 상승모서리가 인터럽트 요구

여기서 하강모서리와 상승모서리를 인터럽트의 소스로 사용하는 3가지의 감지방법은 I/O 클록을 사용하여 감지한다. 따라서 I/O 클록이 동작을 멈추는 SLEEP상태에서는 인터럽트 소스로 사용할 수 없다. 따라서 SLEEP 상태를 깨우는 인터럽트로는 INT0~3 또는 INT4~5의 레벨트리거방법을 사용하여야 한다.(단 SLEEP상태 중 IDLE모드는 I/O클록이 동작한다.)

■ **외부 인터럽트 마스크 레지스터**

(External Interrupt Mask Register) : EIMSK

비트	7	6	5	4	3	2	1	0	
	INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	EIMSK
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	

• 비트 7..0 - 비트 n을 세트하면 전역으로 인터럽트가 허용되었을 때 (SREG.I가 세트) 외부인터럽트 INTn이 허용된다. 리셋하면 해당 인터럽트가 금지된다.

■ **외부 인터럽트 플래그 레지스터**

(External Interrupt Flag Register) : EIFR

비트	7	6	5	4	3	2	1	0	
	INTF7	INTF6	INTF5	INTF4	INTF3	INTF2	INTF1	INTF0	EIFR
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	

• 비트 7..0 - INTn이 로직변경이 인터럽트를 요구하면 해당 플래그비트가 세트된다. 이 플래그가 세트되어있을 때 전역으로 인터럽트가 허용되고(SREG.I가 세트) EIMSK 레지스터의 해당 마스크 비트가 세트되어 있으면 인터럽트벡터로 분기하여 인터럽트를 수행한다. 이 플래그는 인터럽트 서비스루틴을 수행하면 자동적으로 리셋된다. 또한 해당 플래그에 논리 1을 쓰면 플래그가 지워진다. 레벨 트리거모드에서는 이 플래그는 항상 0으로 리셋되어 있다.

7.5 인터럽트를 사용한 LED 패턴 이동

LED의 패턴을 이동시키기 위해 6.6절에서는 스위치를 계속적으로 감시하는 폴링방법을 사용하였다. 여기서는 스위치를 누를 때 인터럽트가 발생되도록 하는 방법을 사용하여 패턴을 이동시키도록 한다. 그림 6.2의 LED회로를 사용하고 스위치는 그림7.6과 같이 연결한다.

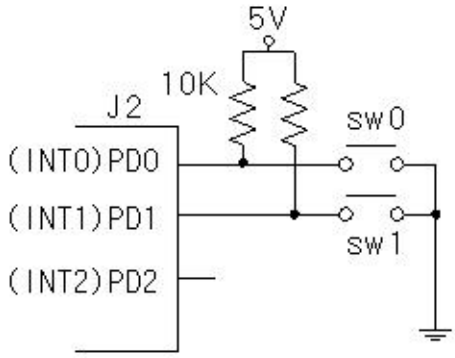

그림 7.6 스위치 회로

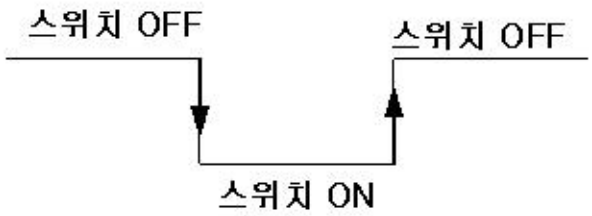

그림 7.7 스위치 조작에 따른 INT0, INT1 신호변화

7.5.1 스위치 인터럽트를 사용한 LED패턴 이동(I)

(가) 외부 인터럽트를 사용하기 위한 준비

☞ 포트 D의 입출력 핀 PD0은 인터럽트 INT0과 공유해서 사용하므로 스위치 입력 SW0을 인터럽트 INT0의 소스로 사용할 수 있다.

☞ 그림 7.7로부터 스위치를 누를 때 인터럽트를 걸려면 하강모서리 트리거를 사용하고 스위치를 뗄 때 인터럽트를 걸려면 상승모서리 트리거를 사용하면 된다. 여기서는 스위치를 누를 때 인터럽트를 건다.

☞ 외부 인터럽트 INT0을 사용하기 위한 순서는

(1) INT0와 공유하는 입출력 핀 PD0은 입력으로 설정되어야 한다. 그러나 마이크로컨트롤러의 리셋 후 입출력 핀들은 입력으로 초기화되므로 다시 설정을 하지는 않는다.

(2) 트리거 모드 선택 : EICRA 레지스터를 하강모서리에서 인터럽트 요구하도록 설정한다.(스위치를 누를 때 인터럽트 발생)

(3) 인터럽트 마스크 레지스터 EIMSK의 해당비트를 세트하여 외부 인터럽트 0을 허용하도록 한다.

(4) SREG레지스터의 I비트를 세트하여 인터럽트를 허용한다.

```
EICRA = 0x02;  // ISC01:0=2(하강모서리 트리거)
EIMSK = 0x01;  // INT0비트 세트(INT0 허용)
sei();         // 전역 인터럽트 허용
```

◆ I/O 레지스터의 비트 사용

설명을 위해 다음 EIMSK 레지스터를 고려하자.

7	6	5	4	3	2	1	0	
INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0	EIMSK
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
0	0	0	0	0	0	0	0	

앞에서 언급한 바와 같이 헤더파일 "avr/io.h"를 소스파일에 포함시킴으로써 레지스터 명 EIMSK를 unsigned char형 변수처럼 사용할 수 있다. 여기에 더해 모든 레지스터의 각 비트명도 헤더파일에 정의되어 있다. 예를 들어 EIMSK레지스터에 대해 각 비트명은 다음 매크로로 정의되어있다.

```
#define INT0   0
#define INT1   1
#define INT2   2
. . .
```

이에 대해 왼쪽 쉬프트연산자 "<<"를 사용하여 다음을 고려하여보자.

```
1<<INT2   <==> 1 << 2    <==> 0b00000100
```

즉 EIMSK의 INT2 비트가 1로 세트됨을 알 수 있다. 따라서 레지스터 EIMSK의 비트 INT2를 세트하려면 다음과 같이 프로그램하면 된다.

```
EIMSK     = (1<<INT2);
```

위 항목 (4)의 레지스터는 다음과 같이 프로그램 할 수 있다.

```
EICRA = 0x02;  <==> EICRA = (2<<ISC00);
EIMSK = 0x01;  <==> EIMSK = (1<<INT0);
```

☞ 다수의 비트를 1로 세트할 때는 비트별 OR 연산자를 사용하면 된다. 다음 문장은 비트 INT2와 비트 INT4를 세트한다.

```
(1<<INT2)|(1<<INT4)<==> (1<<2)|(1<<4)
                    <==> (0b00000100)|(0b00010000)
                    <==> (0b00010100)
```

따라서 다음과 같이 작성한다.

```
EIMSK = (1<<INT2)|(1<<INT4); // 비트 INT2, INT4세트
```

이는 INT2와 INT4비트를 제외한 나머지 비트를 모두 0으로 리셋한다.

☞ EIMSK의 각 비트 중 INT2와 INT4를 세트하고 나머지 비트는 원래 값을 보존하도록 하여보자. 6.6절에서 설명한 것을 참고하여 다음과 같이 작성하면 된다.

```
EIMSK = EIMSK | ((1<<INT2)|(1<<INT4));
```

EIMSK를 먼저 읽고 6.6절에서 설명한 바와 같이 이를 ((1<<INT2)|(1<<INT4))와 비트별 OR함으로써 INT2와 INT4비트를 세트하고 나머지 비트는 보존한다. 위 문장은 다음과 동일하다.

```
EIMSK |= ((1<<INT2)|(1<<INT4));
```

☞ EIMSK의 각 비트 중 INT2와 INT4를 0으로 리셋하고 나머지 비트는 원래 값을 보존하도록 하여보자.

```
~((1<<INT2)|(1<<INT4)) <==> ~(0b00010100)
                       <==> 0b11101011
```

이므로 6.6절에서 설명한 것을 참고하여 다음과 같이 작성하면 된다.

```
EIMSK &= ~((1<<INT2)|(1<<INT4));
```

(나) 지역정적변수를 사용한 인터럽트서비스루틴

외부인터럽트 INT0이 발생할 때마다(즉 스위치를 누를 때) 패턴을 회전시켜 LED를 켜야 하므로 인터럽트서비스루틴에서 LED의 패턴을 이동시키도록 한다. 6장의 프로그램 6.5 중 while루프 내에서 LED 패턴을 이동시키는 부분을 인터럽트서비스루틴으로 이동시켜 프로그램 7.1을 작성하여 보자.

```
#include <avr/io.h>
#include <avr/interrupt.h>
unsigned char pattern[8]      // LED 패턴 테이블
     = {0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F};
ISR(INT0_vect)                // 외부인터럽트 0 서비스루틴
{
     int i=0;                 // 패턴 인덱스

     if(++i==8) i=0;          // 증가 값이 8이면 0으로 리셋
     PORTA = pattern[i];      // i-번째 패턴으로 LED를 켠다.
}
int main()
{
     DDRA = 0xFF;         // 포트A를 출력포트로 설정
     PORTA = pattern[0]; // 처음 패턴으로 LED를 켠다.

     EICRA = (2<<ISC00); // ISC01:0=2(하강모서리 트리거)
     EIMSK = (1<<INT0);  // INT0비트 세트(INT0 허용)
     sei();              // 전역인터럽트 허용

     while(1);           // 무한루프
}
```

프로그램 7.1 불완전한 동작

☞ 외부인터럽트 INT0에 해당하는 매크로는 표7.2에서 INT0_vect이다.

☞ main()함수는 인터럽트 사용하도록 한 다음 무한 헛루프를 돌고 있다. 이는 패턴을 회전시키는 작업을 모두 인터럽트 서비스루틴에서 수행하

고 main()함수는 종료해서는 안 되기 때문이다.

☞ 스위치를 누르면 외부인터럽트 INT0이 발생하여 인터럽트서비스루틴 ISR(INT0_vect)가 수행되므로 인덱스 i는 1증가하고 pattern[i]의 변수 값으로 LED가 점등된다.

프로그램 6.3과 프로그램 7.1을 비교하면 while()루프 내 문장이 그대로 인터럽트서비스 루틴으로 이동되었음을 알 수 있다. 그러나 이 프로그램을 수행하여 보면 스위치를 눌러도 점등패턴이 이동하지 않는다. 이유는 무엇일까?

☞ 인터럽트서비스루틴 내의 변수 i를 지역자동변수로 선언하였기 때문이다. 지역자동변수는 변수가 선언된 중괄호(여기서는 인터럽트서비스루틴)를 벗어나면 소멸되므로 인터럽트 서비스루틴인 ISR(INT0_vect) 함수의 수행이 끝날 때 변수 i는 소멸된다. 변수 i는 인터럽트가 발생하여 인터럽트서비스루틴으로 들어올 때마다 새로 생성되고 0으로 초기화된다. LED패턴을 출력하기 전에 변수 i는 1 증가하므로 스위치를 누르면 프로그램 7.1은 항상 pattern[1]의 패턴으로 LED를 점등한다.

☞ 이를 해결하려면 변수 i는 인터럽트서비스루틴이 종료되더라도 소멸되지 않고 값을 유지하여야 한다. 이에 적합한 변수형태는 지역정적변수이다. 다음과 같이 인터럽트 서비스루틴 ISR(INT0_vect)내의 변수 i를 지역정적변수로 변경하면 프로그램 7.1은 스위치의 누름에 따라 패턴을 이동시킨다.

```
int i=0;  ==>  static int i=0;
```

‡ 변수의 범위

그림 7.8에는 여러 가지로 선언된 변수들과 그들의 통용범위를 보여준다. 선언된 변수들은 크게 두 가지로 분류된다.

1. 지역변수

중괄호 { }내에서 선언된 변수는 모두 지역변수이다. 이들 변수는 변수가 선언된 중괄호 { }내에서만 통용이 된다. 그림 7.8에서 다음 변수들은 모두 지역변수이다.

```
함수 func1()의 l_data, ls_data, ll_data, arg
함수 func2()의 l_data, ls_data
```

☞ func1()내의 l_data와 func2()내의 l_data는 같은 이름을 가졌지만 통용되는 범위가 각각 func1(){ }와 func2(){ }내이므로 전혀 다른 변수이다.

☞ 함수 func1()의 변수 ll_data는 func1(){ }에 있지만 선언된 부분이 while(){ }내부이므로 while(){ }에서만 통용되고 func1()의 다른 부분에서는 존재를 알지 못한다.

☞ func1()의 arg변수와 같이 함수의 인자로 넘어오는 변수는 함수의 지역변수로 간주된다.

2. 외부변수

중괄호 { } 밖에서 선언된 변수로서 두 가지로 나뉜다.

2.1 전역변수 : 프로그램의 모든 영역에서 통용되는 변수이다.

e_data : 전역변수로 모든 파일에서 통용된다. 단 file1.c에서 선언된 전역변수를 file2.c에서 사용하려면 그림 7.8과 같이 extern을 사용하여 참조하여야 한다.

2.2 정적변수 : 변수를 선언할 때 지시어 static을 사용한다. 통용범위는 같은 파일 내에서 변수가 선언된 아래 부분 모두이다.

se_data : 외부정적변수로 file1.c에서 선언되었으므로 파일 file2.c에서는 변수의 존재를 모른다. file1.c의 맨 처음에 선언되었으므로 file1.c 모든 부분에서 통용된다.

‡변수의 수명

변수는 변수의 선언과 함께 생성된다. 생성된 변수는 static으로 선언되어 있으면 프로그램이 끝날 때까지 소멸하지 않고, 그렇지 않은 변수는 변수가 선언된 중괄호 { }내에서만 존재한다. 즉 프로그램이 변수가 선언된 중괄호의 닫는 부분 "}"를 만나면 해당변수는 소멸된다.

1 지역변수

지역변수는 이의 수명에 따라 두 가지로 구분된다.

1.1 자동변수 : 변수가 선언된 중괄호 { }내에서만 존재한다.

ex) func1()의 l_data, arg : 변수가 선언된 중괄호는 함수 func1()의 중괄호이므로 함수 func1()을 벗어나면 소멸된다. 이 변수는 함수 func1()에 들어 올 때마다 새로 생성된다.

ex) func1()의 ll_data : 변수가 선언된 중괄호는 함수 while() 루프의 중괄호이므로 while()루프를 벗어나면 소멸된다.

1.2 정적변수 : 지역변수 중 static으로 선언된 변수이다. static으로 선언되어 있으므로 프로그램이 끝날 때까지 존재한다.

ex) func1()의 ls_data : 이 변수는 함수 func1()에 처음 들어올 때 생성된다. 이 변수는 한번 생성되면 소멸되지 않으므로 func1()에 다시 들어올 때는 변수를 새로 생성하지 않고 이미 생성된 변수를 사용한다.

2. 외부변수

외부변수는 정적변수와 전역변수 모두 이를 감싸고 있는 중괄호가 없으므로(즉 중괄호의 닫는 부분을 만나지 않으므로) 한 번 생성되면 소멸되지 않는다. 변수는 프로그램이 시작될 때 한 번만 생성된다.

‡ 참고사항

☞ 변수의 범위와 수명은 항상 일치하는 것은 아니다.

- 지역정적변수 ls_data의 범위는 함수 func1()이지만 이 함수를 벗어나더라도 소멸되지 않는다.

☞ 변수를 선언할 때 부득이 한 경우를 제외하고는 지역자동변수를 사용하는 것이 바람직하다.

☞ 그림 7.8에서 "extern int e_data;"는 변수의 선언을 하는 것이 아니다. extern은 변수 e_data가 외부파일에 선언되어 있으므로 참조하라는 뜻이다. 따라서 변수가 새로 생성되지 않는다.

☞ 프로그램 7.1에서 LED 패턴 테이블, pattern[]은 소멸되지 말아야 하므로 외부정적변수로 선언하였다.

☞ 전역변수는 가능하면 사용하지 않는 것이 바람직하다. 전역변수를 꼭 사용해야 할 경우 "i"와 같이 자주 사용하는 이름은 전역변수로 사용하지 말아야 한다.(전역변수로 사용된 이름이 지역변수 이름과 같을 때는 프로그램에 혼란을 초래할 수 있다.)

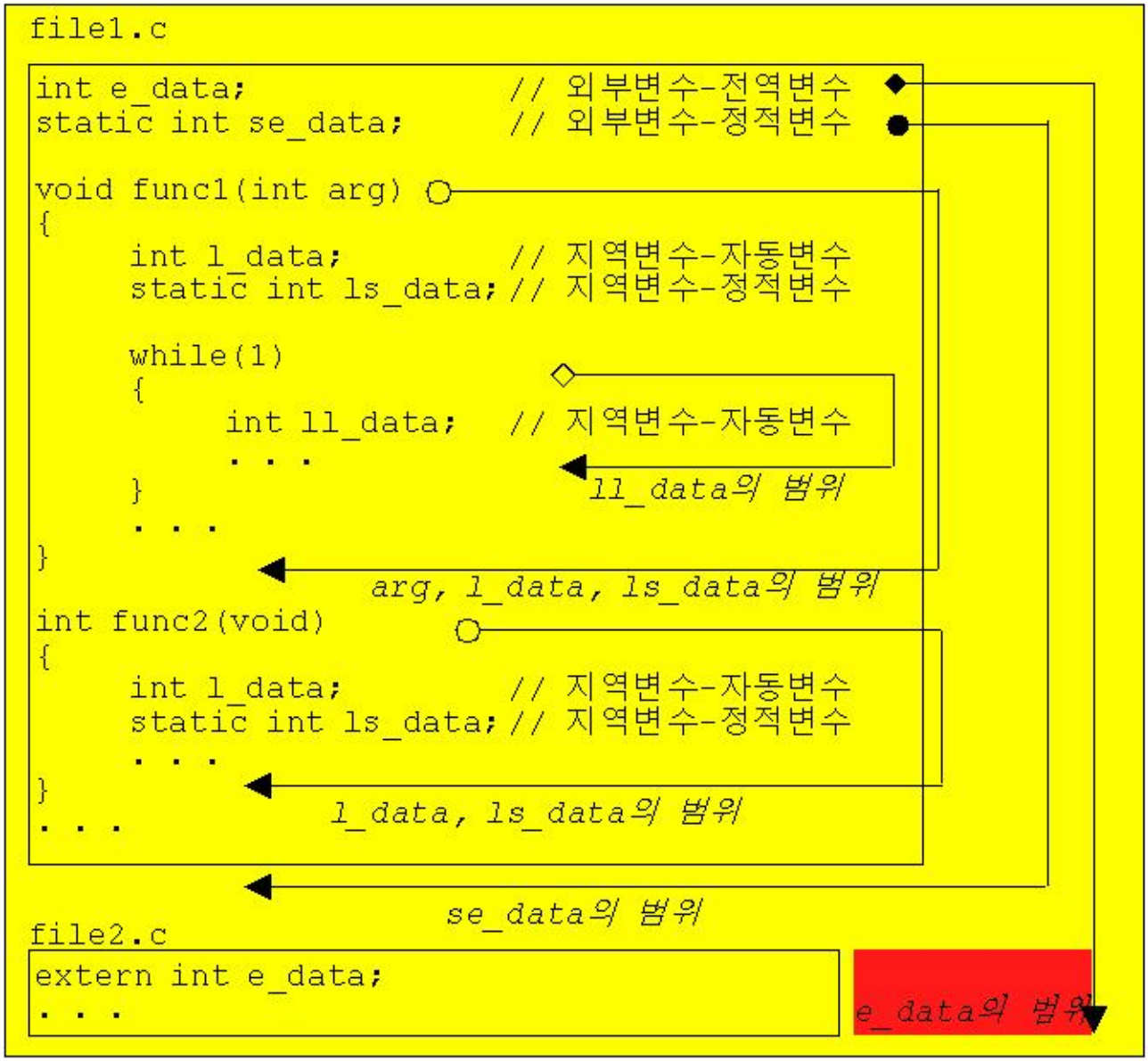

그림 7.8 변수의 범위

(다) 채터링 방지 대책

신호에 채터링이 발생하면 신호의 하강모서리가 여러 번의 인터럽트를 발생시키게 된다. 인터럽트 신호 역시의 채터링이 발생하므로 디바운싱을 하여야 한다.

☞ 이를 위해 그림 7.9와 같이 스위치를 누를 때(인터럽트 서비스 루틴 진입) 시간지연을 두어 디바운싱을 한다.

☞ 누를 때만 디바운싱하면 스위치를 뗄 때 채터링에 의해 다시 인터럽트가 걸릴 수 있으므로 스위치를 누를 때와 뗄 때 모두 디바운싱 한다.

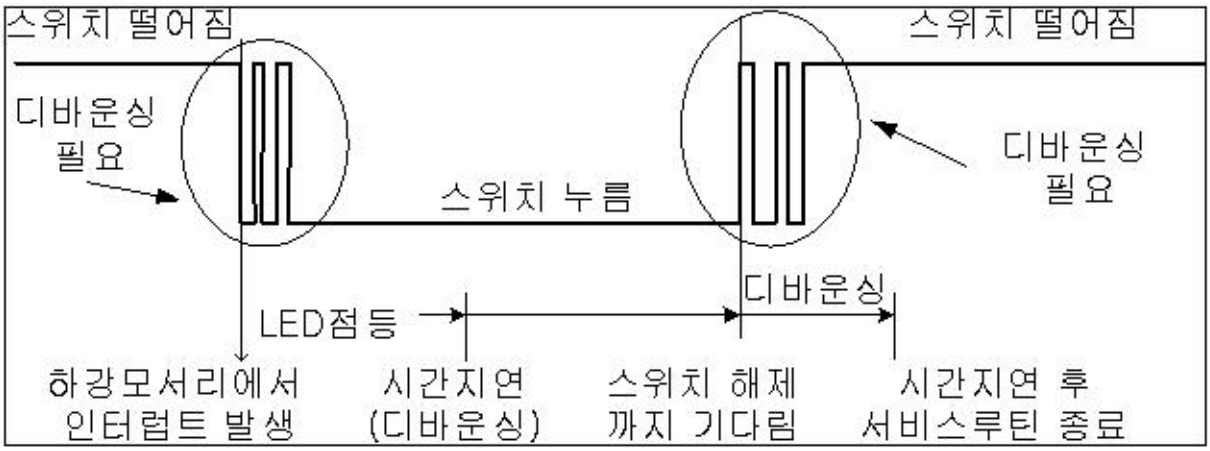

그림 7.9 인터럽트신호의 채터링

프로그램 7.2는 이를 고려하여 i를 지역정적 변수로 변경한 프로그램 7.1의 인터럽트 서비스루틴 부분을 다시 작성한 것이다.

```
ISR(INT0_vect)                  // 외부인터럽트 0 서비스루틴
{
    static int i=0;             // 패턴 인덱스

    if(++i == 8) i=0;           // 증가 값이 8이면 0으로 리셋
    PORTA = pattern[i];         // i-번째 패턴으로 LED를 켠다.

    msec_delay(20);             // 스위치 누름에 대한 디바운싱
    while(~PIND & 0x01);        // 스위치 해제를 기다림
    msec_delay(20);             // 스위치 뗌에 대한 디바운싱
}
```

프로그램 7.2 불완전한 채터링 방지

그러나 스위치 입력을 해보면 디바운싱에도 불구하고 채터링 방지가 완전하지 않은 것이 LED 패턴의 이동에서 관찰된다. 이는 인터럽트 처리 방식 때문이다. 그림 7.10은 불완전한 채터링 방지의 원인을 보여준다.

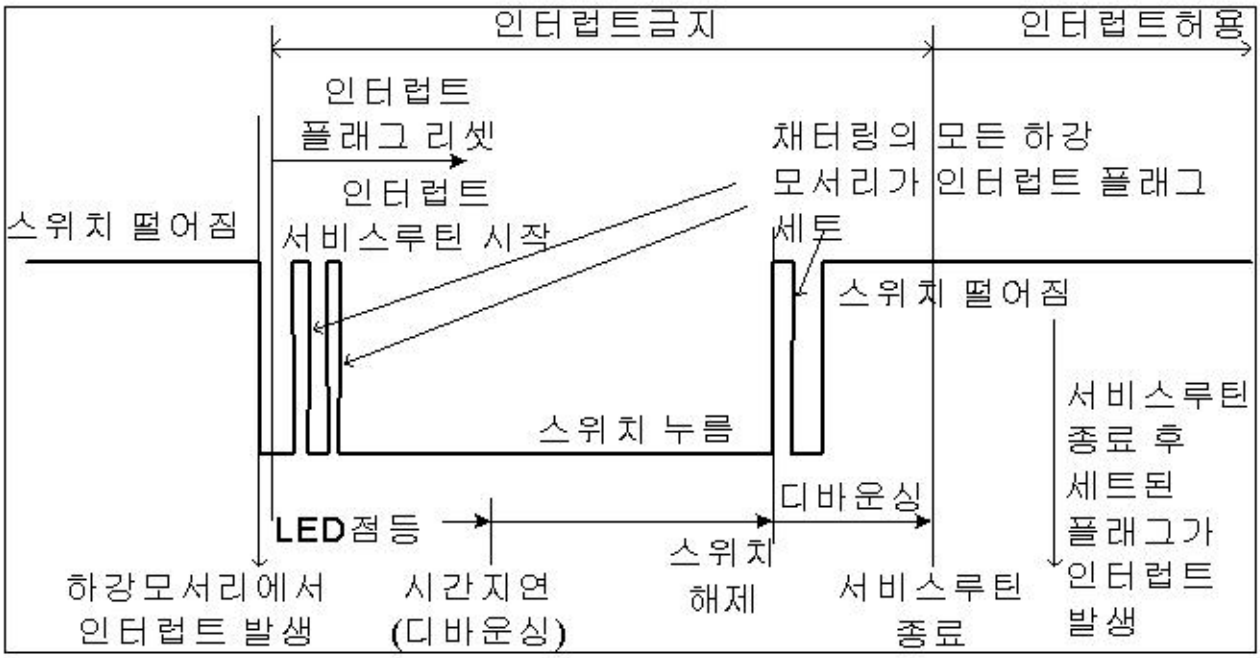

그림 7.10 인터럽트 플래그에 의한 불완전한 채터링 방지

☞ 인터럽트 플래그는 인터럽트가 발생하면 세트되지만 인터럽트서비스루틴이 시작되면서 자동적으로 리셋된다. 아울러 인터럽트 서비스루틴이 시작되면 자동으로 인터럽트는 전역으로 금지된다.

☞ 인터럽트가 금지되어 있어도 신호의 하강모서리에서 인터럽트 플래그는 세트된다. 따라서 채터링에 따른 모든 하강모서리에서 인터럽트 플래그가 세트된다.

☞ 현재 수행 중인 인터럽트서비스루틴이 종료된 후 인터럽트는 다시 허용된다. 이때 채터링에 의해 인터럽트 플래그가 세트되어 있으므로 인터럽트가 발생하게 된다. 즉 스위치를 누르지 않았음에도 불구하고 인터럽트 서비스루틴으로 재 진입하게 되어 채터링이 방지되지 않는다.

☞ 인터럽트플래그를 리셋하고 인터럽트서비스루틴을 종료하면 이를 방지할 수 있다.

☞ 외부 인터럽트 INT0의 플래그는 EIFR 레지스터의 0-번째 비트에 1을 쓰면 플래그를 리셋할 수 있다.

```
EIFR = 0x01;   // 또는 EIFR = (1<<INTF0);
```

[프로그램 prac7-5I.c]는 채터링 방지를 하면서 외부 인터럽트 INT0을 사용하여 LED의 패턴을 이동시키는 프로그램이다.

실습 :

1. 프로그램 7.1을 수행하면서 스위치 입력에 따른 LED 패턴의 이동을 관찰하라.
2. 변수 i를 지역정적변수로 변경한 프로그램 7.1을 수행하면서 스위치 입력에 따른 LED패턴의 이동을 관찰하라.
3. 프로그램 7.2를 수행하면서 스위치 입력에 따른 LED패턴의 이동을 관찰하라. 디바운싱의 지연시간을 길게 하면 채터링이 방지되는 지 관찰한다.
4. 프로그램 prac7-5I.c를 수행하면서 스위치 입력에 따른 LED 패턴의 이동을 관찰하라.

과제 : 1. 프로그램 7.1에서 배열 pattern[]는 어떠한 변수인가?
(1) 지역자동변수 (2) 지역정적변수 (3) 외부정적변수 (4) 전역변수

2. 프로그램 prac7-5I.c에서 배열 pattern[]는 어떠한 변수인가?
(1) 지역자동변수 (2) 지역정적변수 (3) 외부정적변수 (4) 전역변수

◆ **[프로그램 7.2]와 [프로그램 prac7-5I.c]의 인터럽트서비스루틴은 인터럽트 메커니즘을 설명하기 위해서 작성한 것으로 인터럽트서비스루틴으로 적합하지 않다는 것을 명심하여야 한다.**

☞ 인터럽트서비스루틴이 동작되는 동안은 다른 프로세스가 중단이 되므로 인터럽트서비스루틴은 아주 짧은 시간 내에 종료가 되도록 작성하여 다음 동작에 영향을 미치지 않는 것이 중요하다.

☞ [프로그램 7.2]와 [프로그램 prac7-5I.c]에서는 스위치의 채터링에 대응하기 위해 디바운싱과 함께 스위치가 떨어질 때까지 기다린다. 이를 위해 인터럽트서비스루틴에서 소요되는 시간은 디바운싱을 위한 수십 msec와 스위치를 누르고 있는 시간을 합한 시간이다. 인터럽트서비스루틴으로 동작하기에는 너무 긴 시간을 소비하고 있어 인터럽트서비스루틴으로 적합하지 않다.

[프로그램 prac7-5I.c]

```
//=======================================================
// 실습 7.5.1 외부 인터럽트 INT0을 사용하여 LED 패턴이동
//=======================================================

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#define DEBOUNCING_DELAY     20
void msec_delay(int n);              // 시간지연함수

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

ISR(INT0_vect)                    // 외부인터럽트 0 서비스루틴
{
     static int i=0;              // 패턴 인덱스

     if(++i==8) i=0;              // 증가 값이 8이면 0으로 리셋
     PORTA = pattern[i];          // i-번째 패턴으로 LED를 켠다.

     msec_delay(DEBOUNCING_DELAY);// 디바운싱
     while(~PIND & 0x01);         // 스위치 해제를 기다림
     msec_delay(DEBOUNCING_DELAY);// 디바운싱

     EIFR = (1<<INTF0);           // 인터럽트 플래그 INTF0을 리셋
}

int main()
{
     DDRA = 0xFF;           // 포트A를 출력포트로 설정
     PORTA = pattern[0]; // 처음 패턴으로 LED를 켠다.

     EICRA = (2<<ISC00); //ISC01:0=2 (하강모서리 트리거)
     EIMSK = (1<<INT0);  // INT0비트 세트(INT0 허용)
     sei();              // 전역인터럽트 허용
```

```
    while(1);           // 무한루프
}

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

7.5.2 스위치 인터럽트를 사용한 LED패턴 이동(II)

7.5.1절에서 작성한 prac7-5I.c를 보면 인터럽트서비스루틴에서 패턴 인덱스 증가와 포트 A에 LED패턴의 출력 두 가지를 수행하고 있는 데 이를 살펴보면

(1) 스위치를 누를 때 마다 패턴을 이동시켜야 하므로 인덱스의 증가는 인터럽트서비스루틴에서 이루어져야 한다.
(2) 현재의 인덱스만 알면 포트 A에 패턴을 출력하는 것은 main()함수에서도 할 수 있다.

위의 관점에 따라 인터럽트서비스루틴에서 포트 A에 패턴을 출력하는 부분을 main()함수로 넘기기로 하고 다음과 같이 작성하여 보자.

```
ISR(INT0_vect)                  // 외부인터럽트 0 서비스루틴
{
    static int i=0;             // 패턴 인덱스

    if(++i==8) i=0;             // 증가 값이 8이면 0으로 리셋

    msec_delay(20);             // 버튼을 누른 것에 대한 디바운싱
    while(~PIND & 0x01);        // 스위치 해제를 기다림
    msec_delay(20);             // 버튼을 뗀 것에 대한 디바운싱

    EIFR = 0x01;                // 인터럽트 플래그 INT0을 리셋
}
```

프로그램 7.3

☞ main()함수에서 패턴을 출력하려면 인덱스 값 i를 알아야 하나 이는 지역정적변수이므로 통용범위가 ISR(INT0_vect)함수 내부이고 main()함수에서 i 값을 알 수 없다.
☞ 인터럽트서비스루틴으로 사용되는 함수는 인수를 받지 못할 뿐 아니라 데이터를 반환할 수도 없으므로 인덱스 i를 main()함수에 전달할 수 없다.

인터럽트서비스루틴의 인덱스를 main()함수에 전달하는 방법은 인덱스 변수 i의 통용범위를 main()함수까지 넓히는 것이다. 이를 위해 인덱스 변수를 다음과 같이 변경하자.

(1) 인덱스변수를 외부변수로 선언하여 범위를 파일전체로 확장한다. 한 파일 내에서만 사용되면 되므로 외부전역변수보다는 외부정적변수로 선언한다.(static사용)
(2) 인덱스 변수의 범위가 확장됨에 따라 변수 명을 i와 같이 범용으로 사용하는 이름은 쓰지 않는 것이 좋다. 변수 명을 index로 변경한다.

이에 따라 인터럽트서비스루틴을 작성하면

```
static int index=0;       // 패턴 인덱스

ISR(INT0_vect)            // INT0 인터럽트서비스루틴
{
     if(++index==8) index=0; // 증가 값이 8이면 0으로 리셋

     msec_delay(20);          // 버튼을 누른 것에 대한 디바운싱
     while(~PIND & 0x01);     // 스위치 해제를 기다림
     msec_delay(20);          // 버튼을 뗀 것에 대한 디바운싱
     EIFR = 0x01;             // 인터럽트 플래그를 리셋
}
```

프로그램 7.4

변수 index는 외부정적변수이므로 main()함수에서도 값을 알 수 있다. 프로그램 prac7-5II.c는 완성된 프로그램을 보여준다.

☞ 배열 pattern[] 역시 외부정적변수이므로 main() 함수에서 값을 알 수 있다.

실습 : 프로그램 prac7-5II.c를 수행한다. 패턴이 이동하지 않는 경우는 그림 7.11의 "Project Options" 창에서 "Optimization" 항목을 "-O0"로 변경하고 수행한다.

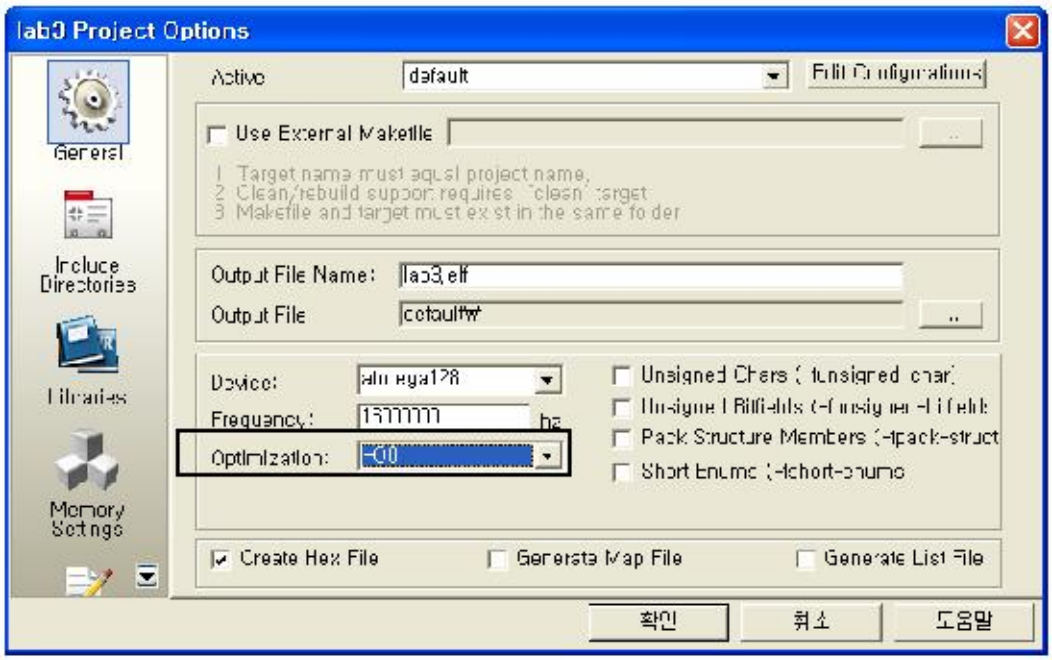

그림 7.11 최적화(Optimization)옵션 변경

[프로그램 prac7-5II.c]

```
//=========================================================
// 실습 7.5.2 외부 인터럽트 INT0을 사용하여 LED 패턴이동
//=========================================================

#include <avr/io.h>             // I/O 레지스터 정의
#include <avr/interrupt.h>      // 인터럽트 정의
#include <util/delay.h>         // 시간지연 함수 헤더파일

#define DEBOUNCING_DELAY        20

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

```
static int index = 0;    // 패턴 인덱스
static unsigned char pattern[8]   // LED 패턴 테이블
     = {0xFE, 0xFD, 0xFB, 0xF7, 0xEF, 0xDF, 0xBF, 0x7F};

ISR(INT0_vect)                  // 외부인터럽트 0 서비스루틴
{
     if(++index==8) index=0; // 증가 값이 8이면 0으로 리셋

     msec_delay(DEBOUNCING_DELAY);// 디바운싱
     while(~PIND & 0x01);          // 스위치 해제를 기다림
     msec_delay(DEBOUNCING_DELAY);// 디바운싱

     EIFR = (1<<INTF0);       // 인터럽트 플래그 INT0를 리셋
}

int main()
{
     DDRA = 0xFF;         // 포트A를 출력포트로 설정
     PORTA = pattern[0]; // 처음 패턴으로 LED를 켠다.

     EICRA = (2<<ISC00); //ISC01:0=2(하강모서리 트리거)
     EIMSK = (1<<INT0);  // INT0비트 세트(INT0 허용)
     sei();              // 전역 인터럽트 허용

     while(1)            // 무한루프
     {
          PORTA = pattern[index];
     }
}

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

7.5.3 WinAVR에서 코드 최적화와 volatile 지시어

2007년 이후 버전의 WinAVR에서는 최적화(Optimization) 옵션 "-Os"가 디폴트로 설정되기 때문에 실습 7.5.2에서 최적화 옵션을 변경을 하여 프로그램을 수행하였다. 여기서는 컴파일러의 최적화 옵션에 대해 설명하고자 한다.

마이크로컨트롤러 프로그램을 작성할 때 다음 두 가지 사항을 고려하여야 한다.

- 실행 코드의 크기
- 실행 코드의 실행속도

이상적인 것은 코드의 크기를 작게 하면서 실행속도를 높이는 것이다. 이에 대한 일차적인 책임은 프로그램 작성자에게 있다. 어떤 일을 수행하는 순서 및 방법을 알고리듬(Algorithm)이라 하는 데, 프로그래머가 사용하는 알고리듬에 따라 실행코드의 크기 및 속도가 크게 좌우된다.

일단 소스프로그램이 작성되면 두 번째로 고급언어로 작성된 프로그램을 기계어로 번역하는 컴파일방법에 따라 실행코드의 크기와 속도가 달라진다. 영어를 한글로 번역할 때 단 하나의 방법만 있는 것이 아닌 것처럼 컴파일러도 기계어로 번역할 때 여러 가지 방법을 사용할 수 있다. 사용자는 컴파일러의 최적화 옵션으로 번역방법을 선택한다. WinAVR에서는 그림 7.11의 "Project Option"창에서 Optimization 옵션을 선택한다.

최적화 옵션으로 5가지를 선택할 수 있다.

- -O0 : 최적화를 수행하지 않는다.
- -O1 : 컴파일 시간을 많이 요구하는 최적화를 제외하고 코드의 크기와 실행속도를 줄이도록 최적화를 수행한다.
- -O2 : 코드크기와 실행속도간의 Trade-off를 하지 않는 범위에서 -O1옵션보다 더 최적화를 수행한다.
- -O3 : -O2옵션보다 더 최적화를 수행한다.
- -Os : 코드크기를 최적화한다. 코드크기를 증가시키지 않도록 -O2 최적화를 한다.

☞ -O0 옵션을 선택하여 생성된 코드는 다른 옵션에 비해 크기도 크고 실행속도도 늦다. 이는 디버거를 사용할 때 사용되는 옵션이다. 이외의 옵션에 대해서는 컴파일러가 최적화를 수행한다. 최적화를 수행한 코드는 옵션에 관계없이 -O0 옵션으로 생성된 코드보다는 크기 및 실행속도에서 우월하다.

☞ 최적화 번호가 높다고 해서 좋은 코드를 생성한다고 말할 수는 없다. 최적화란 기준에 따라 다르기 때문이다. 코드크기를 기준으로 하였을 때 최적화라 함은 실행속도에 관계없이 작은 실행코드를 생성하는 것이고 실행속도를 기준으로 하였을 때는 코드크기에 상관없이 빠른 실행속도를 나타내는 코드가 최적이기 때문이다. 최적화 옵션 선택은 사용자가 자기의 기준에 맞도록 선택하여야 한다.

여기서 프로그램 prac7-5II.c가 최적화 옵션을 "-O0"가 아닌 다른 옵션을 선택하였을 때 동작하지 않는 이유를 살펴보자.

프로그램 prac7-5II.c의 외부정적변수 index는 인터럽트서비스루틴에서는 값이 변하지만 main()함수에서 index변수의 값은 변하지 않는다. 최적화 옵션을 선택하면 컴파일러는 index변수와 같이 main()함수에서 변하지 않는 변수는 최초 한번만 값을 평가하고 이후는 사용하지 않도록 컴파일한다. 따라서 인터럽트서비스루틴에서 index값이 변하지만 이는 main()함수에서는 반영되지 않는다.

volatile지시어는 인터럽트와 같은 외부요인에 의해 변수가 변할 수 있음, 즉 휘발성임을 컴파일러에게 알려서 최적화 과정에서 변수가 소외되지 않도록 한다. 프로그램 prac7-5II.c에서 index변수 선언을 다음과 같이 변경한다.

```
static int index;
    ==> static volatile int index;
```

이후 최적화를 수행하는 옵션 "-O1", "-O2", "-O3", "-Os"등으로 설정하여

도 프로그램 prac7-5II.c는 정상적으로 수행된다.

☞ 외부변수이면서 인터럽트 서비스루틴과 일반함수에서 공용으로 사용하는 변수는 volatile 지시어를 사용하는 것이 바람직하다.

실습 : 최적화 옵션을 "-Os"로 선택한 다음 prac7-5II.c의 변수 index를 volatile변수로 변경하여 프로그램을 수행한다.

과제 :

1. 7-세그먼트는 그림 7.12와 같이 LED를 배열하여 글자를 표현할 수 있도록 한 장치이다. 7-세그먼트는 Anode 공통형과 Cathode 공통형이 있다. 포트 A에 Anode 공통형 7-세그먼트를 그림 7.13과 같이 연결하고 sw0을 누를 때마다 7-세그먼트의 글자가 0 -> 1 -> 2 -> ... -> 9 -> A -> b -> c -> d -> E -> F로 순환하고 sw1을 누르면 반대방향으로 순환하는 프로그램을 작성하고 수행하라.(최적화 옵션을 "-Os"로 설정할 것)

2. 그림 6.9와 같이 연결된 도트 매트릭스에 sw0을 누를 때마다 도트 매트릭스에 표시되는 글자는 A->B->C->D로 순환하도록 프로그램을 작성하라.

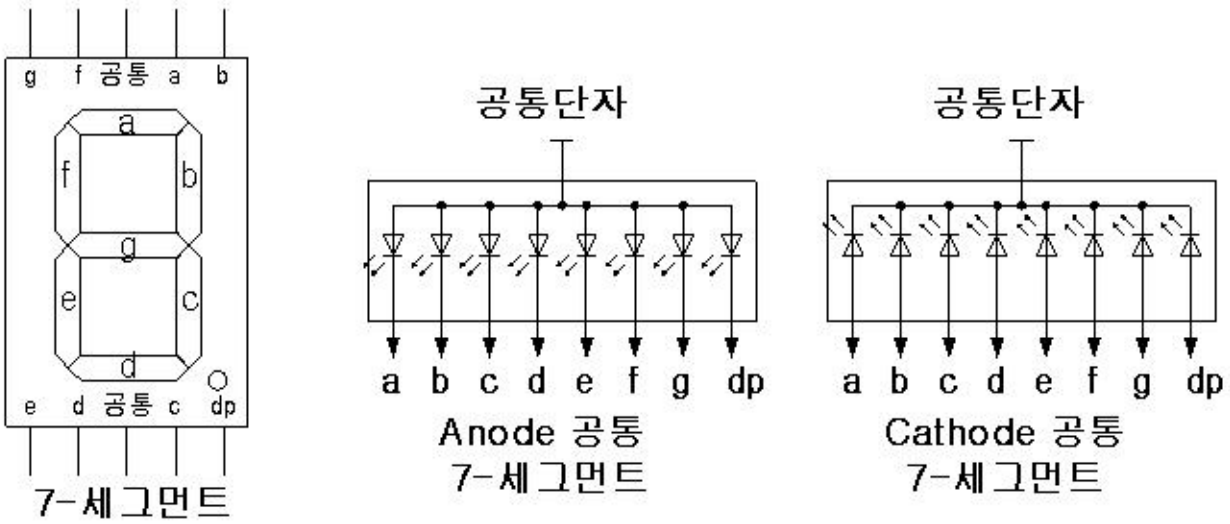

그림 7.12 7-세그먼트

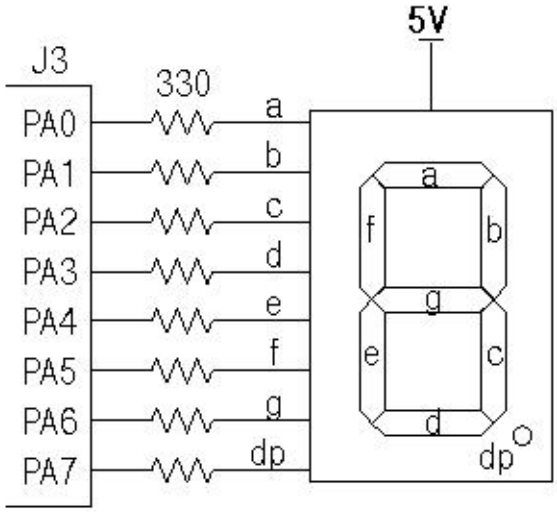

그림 7.13 Anode 공통 7-세그먼트 연결