

Chapter 109. 인터럽트 1

1. 폴링 v.s. 인터럽트

마이크로컨트롤러 프로그램의 main 함수에는 이벤트 루프 또는 메인 루프라고 불리는 무한 루프가 존재한다. 버튼이 눌러지면 LED에 불이 켜지는 프로그램을 생각해 보자. 이벤트 루프에서는 ① 버튼의 상태를 읽고, ② 버튼이 눌러졌는지 검사하여, 버튼이 눌러졌다면 ③ LED에 불을 켜는 과정을 무한히 반복한다. 즉, 데이터의 ① 입력, ② 처리, ③ 출력 과정을 반복하고 있다. 이처럼 (버튼이 눌러졌는지의 여부와 같은) 어떤 특정한 사건이 발생하였는지를 반복적으로 검사하고 사건이 발생하였을 때 (LED를 켜는 것과 같은) 특정한 동작을 수행하는 방식을 폴링(polling) 방식이라고 한다.

코드 1. 폴링 방식의 이벤트 루프 1

```
int main(void){
    while(1)
    {
        // ① 버튼 상태 읽기
        // ② 버튼 상태 판단
        // ③ LED로 버튼 상태 출력
    }
}
```

이러한 폴링 방식은 정해진 순서에 의해 명령을 반복적으로 처리하므로 프로그램을 작성하고 이해하기 쉬운 장점이 있다. 프로그램을 약간 복잡하게 만들어서 두 가지 동작을 수행하도록 만들어 보자. 버튼이 눌러지면 첫 번째 LED에 불을 켜는 것과 동시에, UART 통신을 통해 'O'라는 문자가 수신되면 두 번째 LED에 불을 켜고 다른 문자가 수신된 경우에는 불을 끄고 가정해 보자.

코드 2. 폴링 방식의 이벤트 루프 2

```
int main(void){
    while(1)
    {
        // ① 버튼 상태 읽기
        // ② 버튼 상태 판단
        // ③ LED 1로 버튼 상태 출력

        // ④ UART 문자 수신
        // ⑤ 수신 문자 비교
        // ⑥ LED 2로 상태 출력
    }
}
```

크게 문제가 될 부분은 없어 보인다. 하지만 수신 문자 비교 과정이 아주 오래 걸리는 작업이라고 가정해 보면 문제가 발생할 수 있다. 이벤트 루프는 순차적으로 명령을 실행하므로 UART를 통해 문자를 수신하고 이를 비교하는 도중에는 버튼의 상태를 읽어올 수 없으므로 문자 비교가 진행 중인 경우 버튼이 눌러지면 해당 LED에 불이 켜지지 않는 경우가 발생할 수 있다. 이처럼 2개 이상의 작업이 동시에 진행될 때 하나의 작업은 다른 작업에 의해 실행이 지연되거나 실행할 수 없는 경우가 생긴다. 이처럼 여러 작업이 진행되는 경우, 특히, 버튼이 눌러지면 ‘즉시’ 두 번째 LED에 불이 켜지는 동작이 필요하다면 코드 2에서와 같은 폴링 방식으로는 해결하기가 어렵다. 무엇이 필요할까? 바로 인터럽트가 해결책이다.

인터럽트는 일상생활에서도 흔히 볼 수 있다. 집에서 영화를 보고 있다고 생각해 보자. 우체부가 초인종을 누르면 보고 있던 영화를 일시 정지시키고 현관에 가서 우편물을 받은 후 다시 돌아와서 영화를 계속 본다. 영화를 보는 동작이 이벤트 루프에 해당한다면, 우체부가 초인종을 누르는 동작은 인터럽트에 해당한다. 우체부가 초인종을 누르는 ‘비정상적인’ 사건이 발생하면 우리는 영화를 보는 ‘정상적인’ 동작을 잠시 멈추고 비정상적인 사건부터 해결하려고 한다. 즉, 비정상적인 인터럽트는 먼저 처리되어야 하며 이를 우선순위(priority)가 높다고 이야기 한다.

2. 인터럽트

인터럽트란 마이크로컨트롤러가 즉시 특정 작업을 처리하도록 요구하는 이벤트를 말한다. 인터럽트가 발생하면 마이크로컨트롤러는 현재 진행 중인 작업을 멈추고 인터럽트가 요청한 작

업을 수행하기 위해 인터럽트 처리 루틴(Interrupt Service Routine, ISR)으로 이동한다. ISR이 종료되면 마이크로컨트롤러는 수행을 멈춘 곳으로 되돌아가 중지된 작업을 계속한다. 인터럽트 인터럽트는 단어 의미 그대로 정상적인 프로그램의 흐름을 뒤바꾸는 역할을 한다.

ATmega328의 데이터시트에서 인터럽트 장을 찾아보면 표 1을 확인할 수 있다.

벡터 번호	프로그램 주소	소스	인터럽트 정의
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Counter1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

표 1. 인터럽트 벡터 테이블

표 1은 ATmega328에서 가능한 모든 인터럽트의 종류를 나타낸 것으로 ‘인터럽트 벡터 테이블’이라는 이름이 붙어있다. 인터럽트가 발생하면 ISR에서 처리가 이루어진다. 이 때 ISR이 위치하는 메모리 주소를 인터럽트 벡터라고 하며, 인터럽트 벡터를 모아놓은 표 1을 인터럽트 벡터 테이블이라고 한다.

벡터 번호 4번에서 6번까지의 핀 변화 인터럽트(Pin Change Interrupt)는 핀에 가해지는 전압이 VCC에서 GND 또는 그 반대로 변하는 경우 발생한다. 범용 입출력 핀에 풀업 저항을 이용하여 버튼을 연결하였다고 생각해 보자. 버튼을 누르면 VCC에서 GND로 핀에 가해지는 전압이 바뀌고 인터럽트가 발생한다. 인터럽트가 발생하면 해당 인터럽트를 처리할 ISR의 메모리 주소를 인터럽트 벡터 테이블에서 찾아 해당 번지로 실행을 옮긴다. 버튼이 눌러졌을 때 인터럽트를 발생시키고, 해당 인터럽트 처리를 위한 ISR의 주소를 찾아 그 위치로 이동하는 작업은 마이크로컨트롤러에서 자동으로 처리해 주므로 우리가 걱정할 필요는 없다. 버튼이 눌러지면 즉시 어떤 작업을 수행한다. LED에 불을 켤 수도 있고, 음악이 흘러 나오도록 할 수도 있고, 모터를 돌릴 수도 있는 등 수행할 작업은 필요에 따라 달라지므로 우리가 걱정해야 하는 부분은 ‘어떤 작업’을 할 것인가 하는 문제이다. 컴파일러는 인터럽트 벡터 테이블을 작성하여 실행 파일에 추가하고, 마이크로컨트롤러는 인터럽트가 발생한 경우 인터럽트 벡터 테이블을 참고하여 ISR로 이동하는 작업을 자동으로 처리해 준다.

3. 인터럽트 처리

인터럽트가 발생하면 해당 ISR이 호출된다. 이를 위해서는 세 가지 조건이 충족되어야 한다.

- 전역적인 인터럽트 활성화 비트가 세트되어 있어야 한다.
- 인터럽트별로 존재하는 개별 인터럽트 활성화 비트가 세트되어 있어야 한다.
- 인터럽트 발생을 위한 조건이 충족되어야 한다.

전역 인터럽트 활성화 비트는 상태 레지스터(status register)의 7번 비트로, 상태 레지스터는 가장 최근에 산술 논리 연산장치(Arithmetic Logic Unit, ALU)에서 수행된 연산의 결과를 반영하는 레지스터이다. 상태 레지스터의 정보는 직전 연산의 결과에 따라 프로그램의 흐름을 나누기 위해 if와 같은 비교문장 없이 사용할 수 있으므로 프로그램을 보다 작고 빠르게 만드는 것이 가능하다. 하지만 상태 레지스터 값에 따른 분기문은 어셈블리어 수준에서 이루어

지므로 C 언어에서 사용하기에는 어렵고 번거로운 면이 있다. 상태 레지스터의 구조는 그림 1과 같다.

비트	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
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 1. SREG (Status Register) 레지스터 구조

상태 레지스터 각 비트의 의미는 표 2와 같다.

비트 번호	비트 이름	설명
7	I	전역적인 인터럽트 발생을 허용한다. (Global Interrupt Enable)
6	T	비트 복사를 위한 BLD (Bit Load), BST (Bit Store) 명령에서 사용한다. BST 명령에 의해 비트 값을 비트 T에 저장할 수 있으며, BLD 명령에 비트 T의 내용을 읽어올 수 있다. (Bit Copy Storage)
5	H	산술 연산에서의 보조 캐리 발생을 나타낸다. 보조 캐리는 바이트 단위 연산에서 하위 니블(nibble)로부터 발생하는 캐리를 말한다. (Half Carry Flag)
4	S	부호 비트로, 음수 플래그(N)와 2의 보수 오버플로 플래그(V)의 배타적 논리합(XOR)으로 설정된다. (Sign Bit, $S = N \oplus V$)
3	V	2의 보수를 이용한 연산에서 자리 올림이 발생하였음을 나타낸다. (2's Complement Overflow Flag)
2	N	산술 연산이나 논리 연산에서 결과가 음수임을 나타낸다. (Negative Flag)
1	Z	산술 연산이나 논리 연산에서 결과가 영(zero)임을 나타낸다. (Zero Flag)
0	C	산술 연산이나 논리 연산에서 캐리가 발생하였음을 나타낸다. (Carry Flag)

표 2. 상태 레지스터 비트

상태 레지스터의 I 비트를 세트 또는 클리어 하기 위해서는 sei 또는 cli 함수를 사용할 수 있다. sei 함수는 전역적으로 인터럽트를 허용하는 'Set Interrupt' 함수이며, cli 함수는 전역적으로 인터럽트를 금지하는 'Clear Interrupt' 함수이다.

전역적으로 인터럽트가 허용되어도 인터럽트는 발생하지 않으며 인터럽트에 따라 개별적으로 인터럽트를 허용해 주어야 한다. AD 변환에서 AD 변환이 완료되면 이를 알려주는 인터럽트를 허용하기 위해 ADCSRA (ADC Control and Status Register A) 레지스터의 ADIE (ADC Interrupt Enable) 비트를 설정해 주어야 한다는 것을 기억하는가? 이처럼 특정 인터럽트를 발생시키기 위해서는 두 개의 비트를 설정해 주어야 한다. 마지막으로 기대하는 사건이 발생하기를 기다리면 된다. 예를 들어 AD 변환이 종료되었을 때 I 비트가 세트되어 있고 AD 변환 종료 인터럽트를 허용하는 ADIE 비트가 세트되어 있으면 인터럽트 벡터 테이블에서 22번 ADC Conversion Complete 인터럽트가 발생하고 22번 인터럽트 처리를 위한 ISR로 이동하게 된다.

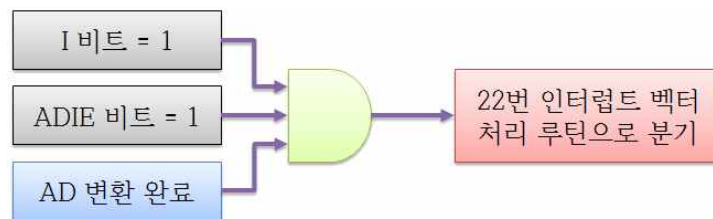


그림 2. AD 변환 인터럽트 처리

각 인터럽트 처리 루틴은 프로그래머의 몫이다. AD 변환 완료에 대한 인터럽트 처리 루틴은 코드 3과 같이 작성한다.

코드 3. 인터럽트 처리 루틴

```

#include <avr/interrupt.h>

ISR(ADC_vect)
{
    // 인터럽트 처리 코드
}
  
```

인터럽트 처리 루틴 인터럽트 벡터의 이름을 매개변수로 가지며, 인터럽트 벡터의 이름은

io.h 파일에 정의되어 있다. 실제로 io.h 파일은 마이크로컨트롤러의 종류에 따라 서로 다른 헤더 파일을 포함시키는 역할만 수행하며 ATmega328의 경우 iom328p.h 파일을 포함하도록 되어 있다. 표 1의 인터럽트에 대하여 정의된 벡터의 이름은 표 3과 같다.

벡터 번호	벡터 이름	인터럽트 정의
1	-	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	INT0_vect	External Interrupt Request 0
3	INT1_vect	External Interrupt Request 1
4	PCINT0_vect	Pin Change Interrupt Request 0
5	PCINT1_vect	Pin Change Interrupt Request 1
6	PCINT2_vect	Pin Change Interrupt Request 2
7	WDT_vect	Watchdog Time-out Interrupt
8	TIMER2_COMPA_vect	Timer/Counter2 Compare Match A
9	TIMER2_COMPB_vect	Timer/Counter2 Compare Match B
10	TIMER2_OVF_vect	Timer/Counter2 Overflow
11	TIMER1_CAPT_vect	Timer/Counter1 Capture Event
12	TIMER1_COMPA_vect	Timer/Counter1 Compare Match A
13	TIMER1_COMPB_vect	Timer/Counter1 Compare Match B
14	TIMER1_OVF_vect	Timer/Counter1 Overflow
15	TIMER0_COMPA_vect	Timer/Counter0 Compare Match A
16	TIMER0_COMPB_vect	Timer/Counter0 Compare Match B
17	TIMER0_OVF_vect	Timer/Counter0 Overflow
18	SPI_STC_vect	SPI Serial Transfer Complete
19	USART_RX_vect	USART Rx Complete
20	USART_UDRE_vect	USART, Data Register Empty
21	USART_TX_vect	USART, Tx Complete
22	ADC_vect	ADC Conversion Complete
23	EE_READY_vect	EEPROM Ready
24	ANALOG_COMP_vect	Analog Comparator
25	TWI_vect	2-wire Serial Interface
26	SPM_READY_vect	Store Program Memory Ready

표 3. 인터럽트 벡터 이름

4. 인터럽트 사용에서의 주의 사항

(1) 중첩된 인터럽트

인터럽트는 디폴트로 ‘인터럽트’ 될 수 없다. 인터럽트가 발생하고 ISR을 실행하기 이전에 ATmega328은 전역 인터럽트 활성화 비트, I 비트를 자동으로 클리어 하고, ISR에서 반환하기 이전에 I 비트를 자동으로 세트하여 ISR 내부에서는 인터럽트가 발생하지 않도록 한다. 또한 ISR로 분기하기 전에 마이크로컨트롤러는 현재 실행 상태를 스택에 저장해(push) 놓으며 ISR에서 반환하면 스택에서 데이터를 복구해서(pop) ISR 실행 이전 상태로 되돌린다. 따라서 인터럽트 실행 도중 또 다른 인터럽트를 실행하게 되면 스택에 저장해야 할 데이터가 증가하게 되고 한정된 메모리를 가진 마이크로컨트롤러는 곧 스택이 넘치는 상황을 맞이하게 될 수 있다. ISR 내에서 전역 인터럽트 허용 비트를 세트하여 인터럽트 처리 중에 또 다른 인터럽트 발생을 허용할 수도 있지만 바람직하지 않으며 권장되지 않는다. ISR 내에서는 인터럽트가 발생하지 않아 중요한 인터럽트를 놓칠 수 있으므로 ISR은 가능한 짧게 작성하고 바로 반환하는 것이 바람직하다.

(2) 인터럽트 우선순위

인터럽트는 이벤트 루프 내의 코드에 비해 먼저 처리된다. 즉, 인터럽트 처리의 우선순위가 이벤트 루프 내의 코드에 비해 높다. 하지만 ATmega328에서만 하더라도 26개의 인터럽트가 정의되어 있다. 2번 인터럽트인 외부 인터럽트 0번과 22번 인터럽트인 AD 변환 완료 인터럽트가 동시에 발생한다면 어떻게 될까? 인터럽트들 사이에도 우선순위가 존재하며 낮은 번호의 인터럽트가 우선순위가 높다. 따라서 2번 외부 인터럽트 0번이 22번 AD 변환 완료 인터럽트보다 먼저 실행된다.

(3) 최적화 방지

코드 4를 보자. 어딘가 잘못된 점이 보이는가?

코드 4. 정상 동작을 보장하지 못하는 인터럽트 처리

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

int value = 0;
```

```

ISR(INTERRUPT_vect)
{
    value = (value + 1) % 2;
}

int main(void)
{
    SetupInterrupt();

    while(1){
        if(value == 1) doSomethingWhenInterruptOccurred();
    }
}

```

코드 4는 첫 번째, 세 번째 등 인터럽트가 2번 발생할 때마다 어떤 작업을 수행하도록 의도된 프로그램으로 별다른 문제가 없어 보인다. 하지만 인터럽트가 발생하더라도 아무런 동작을 수행하지 않을 수 있다. 이는 프로그래머의 잘못이 아니라 컴파일러의 잘못이다. 컴파일러는 가능한 작고 빠른 실행 파일을 만들어 내고자 하며 main 함수 내에서 value의 값이 변하지 않으므로 value는 영(zero)의 값을 갖는 것으로 간주해 버린다. 즉, value는 결코 1이 될 수 없다. 이를 해결하는 방법은 변수 value를 volatile로 선언하는 것이다. volatile로 변수를 선언하는 것은 변수의 값이 보기와는 다르게 다른 곳에서 변할 수 있음을 컴파일러에게 알려주어 if 문을 실행할 때마다 value의 값을 다시 확인하도록 한다.

변수 value를 volatile로 선언함으로써 문제를 해결한 것으로 보이지만 한 가지 문제점이 더 있다. value는 16비트, 즉, 2바이트 값을 가진다. ATmega328은 8비트 마이크로컨트롤러이므로 2바이트 데이터를 한 클럭에 읽어올 수 없다. value의 값을 읽어오는 중간에 인터럽트가 발생하면 어떻게 될까? value에 원하는 값의 절반만 읽혀진 상태에서 인터럽트가 발생하고 ISR에서 value의 값을 바꾸게 되면 value에는 전혀 다른 값이 저장된다. 이를 방지하기 위해서는 value에 값을 읽어오는 동안에는 인터럽트가 발생하지 못하도록 해주어야 하며 이를 위해 사용할 수 있는 매크로가 ATOMIC_BLOCK이다. ATOMIC_BLOCK 매크로는 코드 블록을 형성하며 코드 블록을 시작할 때 전역 인터럽트 플래그를 클리어하고 코드 블록을 끝낼 때 전역 인터럽트 플래그를 세트하여 블록 내 코드가 실행되는 동안 인터럽트의 방해를 받지 않도록 해준다. ATOMIC_BLOCK 매크로는 인자로 ATOMIC_RESTORESTATE와

ATOMIC_FORCEON을 가질 수 있다. 두 인자 모두 ATOMIC_BLOCK 블록 내의 코드가 단어 그대로 'atomic'하게 실행될 수 있도록 보장하지만 ATOMIC_RESTORESTATE는 ATOMIC_BLOCK 블록 내의 코드가 시작되기 전과 끝난 후에 상태 레지스터인 SREG 값이 동일함을 보장하지만 ATOMIC_FORCEON은 ATOMIC_BLOCK 블록 내의 코드에 의해 SREG 레지스터의 값이 바뀔 수 있는 점에서 차이가 있다. 코드 4를 수정한 코드가 코드 5이다.

코드 5. 정상 동작을 보장하도록 수정된 인터럽트 처리

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

volatile int value = 0;

ISR(INTERRUPT_vect)
{
    value = (value + 1) % 2;
}

int main(void)
{
    int local_value;
    SetupInterrupt();

    while(1){
        ATOMIC_BLOCK(ATOMIC_RESTORESTATE){
            local_value = value;
        }
        if(local_value == 1) doSomethingWhenInterruptOccurred();
    }
}
```