AVR – HW4

임베디드스쿨1기

Lv1과정

2020. 10. 9

박하늘

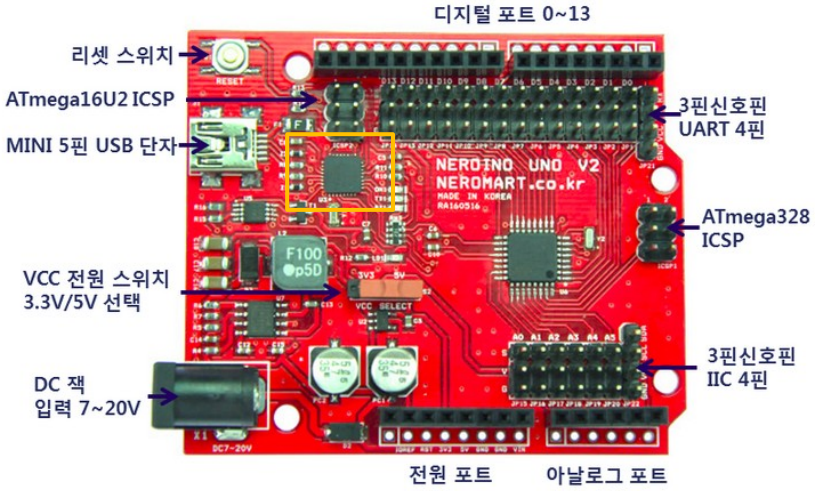
[Review] UART

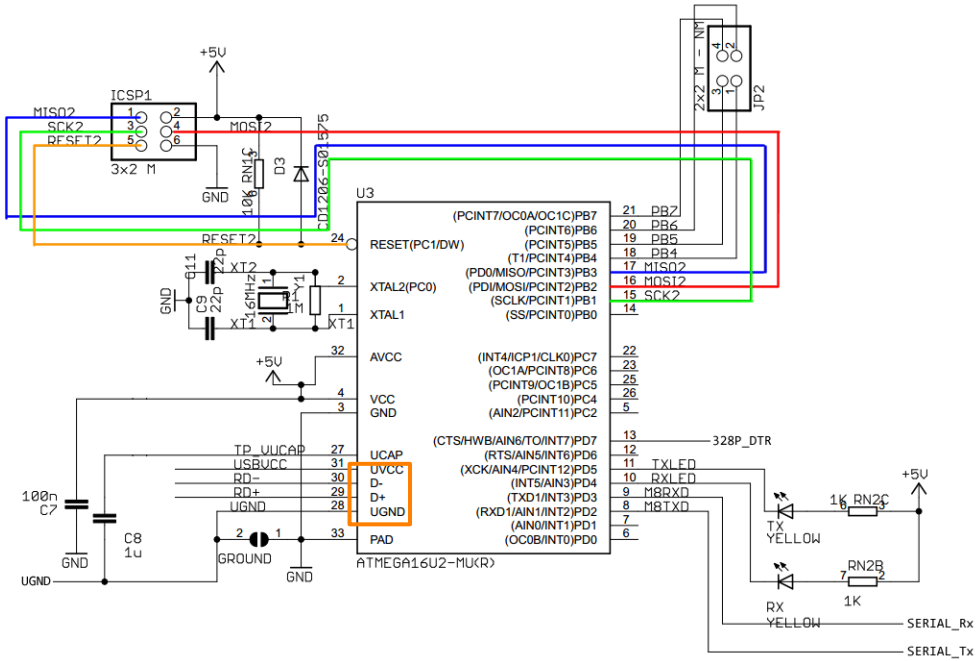
1) UART

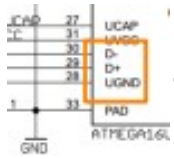
- 가장 간단하게 직렬 RX,TX로 단 2가닥으로 통신하기 때문에 범용적으로 많이 사용됨

2) Atmega328p UART전용 Chip : ATMega16U2

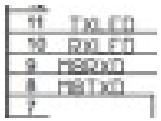
:USB(5V) 와 Atmega328p(3.3V)간 통신을 위해 중간의 별도의 Chip 16U2를 사용함







→ USB와 통신시 16U2칩 내부에서 (floating전압을 제외한 high(1), low(0) 인식) 신호레벨을 맞춰준다



→ RS232를 통해 TTL 로 동작시키게 하여 사용가능

[Review] UART

■ UART Code

```c
#define F_CPU 16000000UL

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

#define sbi(PORTX, BitX) (PORTX |= (1 << BitX))

void UART_INIT(void)
{
    sbi(UCSR0A, U2X0);  //U2X0 = 1 -> Baudrate 9600 = 207 , 속도 2배

    UBRR0H = 0x00;
    UBRR0L = 207;       //Baudrate 9600

    UCSR0C |=0X06;

    sbi(UCSR0B , RXEN0);
    sbi(UCSR0B, TXEN0);     //송수신 Enable
}
unsigned char UART_receive(void)
{

    while(!(UCSR0A & ( 1<<RXC0)));  //UCSR0A의 RXC0값이 1인지 확인 작업
    return UDR0;

}
unsigned char UART_tranmit(unsigned char data)
{
    while(!(UCSR0A & (1<<UDRE0)));  //UCSR0A의 UDRE0값이 1인지 확인 작업
    UDR0 = data;
}
int main(void)
{
    /* Replace with your application code */
    unsigned char data;
    UART_INIT();        //UART초기화
    while (1)
    {
        UART_tranmit('c');
        _delay_ms(1000);

    }
    return 0;
}
```

Control

Status, Data

1. 통신 속도 설정

	f_{osc} = 16.0000MHz			
	U2Xn = 0		U2Xn = 1	
Baud Rate (bps)	UBRRn	Error	UBRRn	Error
2400	416	–0.1%	832	0.0%
4800	207	0.2%	416	–0.1%
9600	103	0.2%	207	0.2%

2. 데이터 설정

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

1
2
으로 클리어, 1이 되면 데이터 수신 끝났다는 것
3) UCSR0A & (1<<UDRE0) // 전송버퍼가 비어 있다면 1 값을 가지고 새로운 데이터를 받아서 전송할 준비 완료된 것

3. 송수신 활성화

[Review] UART Printf

- UART Printf Code
  - 디버거없이 printf를 가지고 디버깅을 하기 위함

```c
#define F_CPU 16000000UL

#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define sbi(PORTX, BitX) (PORTX |= (1 << BitX))
#define cbi(PORTX, BitX) (PORTX &= ~(1 << BitX))

#define UART_BUFLEN 10
static int USART_TX_vect(char, FILE*);
static int usartTxChar(char, FILE*);
void UART_INIT(void)
{
    sbi(UCSR0A, U2X0);      //U2X0 = 1 -> Baudrate 9600 = 207

    UBRR0H = 0x00;
    UBRR0L = 207;

    UCSR0C |=0X06;

    sbi(UCSR0B , RXEN0);
    sbi(UCSR0B, TXEN0);
}
unsigned char UART_receive(void)
{

    while(!(UCSR0A & ( 1<<RXC0)));
    return UDR0;

}
unsigned char UART_tranmit(unsigned char data)
{
    while(!(UCSR0A & (1<<UDRE0)));
    UDR0 = data;
}
```

```c
void UART_string_transmit(char *string)     //*string 문자열의 시작주소
{

    while(*string != '\0')                  //문자열 맨 마지막 문자("\0") 확인
    {

        UART_tranmit( *string);
        string++;
    }
}
void UART_PRINT(char *name, long val )
{

    char debug_buffer[UART_BUFLEN] = {'\0'};    //배열 초기화
    UART_string_transmit(name);
    UART_string_transmit(" = ");
    itoa((val),debug_buffer, UART_BUFLEN);      //itoa함수: int데이터를 문자열로 변환시켜주는 함수
    UART_string_transmit(debug_buffer);
    UART_string_transmit("\n");
}
int main(void)
{
    /* Replace with your application code */

    FILE* fpStdio = fdevopen(usartTxChar, NULL);    //함수포인터 및 시스템 함수
    UART_INIT();
    UART_string_transmit("uart_init\n");
    while (1)
    {
        printf("Hello , Double! %lf\r\n",10.205);
        _delay_ms(1000);

    }
    return 0;
}
int usartTxChar(char ch, FILE *fp){
    while(!(UCSR0A  & (1<<UDRE0)));

    UDR0 = ch;

    return 0;
}
```

1. itoa()함수 코드

■ itoa()란?
:정수형을 문자열로 변환 (리눅스에서는 itoa함수가 인식되지 않으므로 sprintf 함수로 대체)
→ sprintf(버퍼, :형식지정자",값)

```
#ifdef  __DOXYGEN__
extern char *itoa(int val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *itoa (int __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix)) {
    extern char *__itoa (int, char *, int);
    return __itoa (__val, __s, __radix);
    } else if (__radix < 2 || __radix > 36) {
    *__s = 0;
    return __s;
    } else {
    extern char *__itoa_ncheck (int, char *, unsigned char);
    return __itoa_ncheck (__val, __s, __radix);
    }
}
#endif
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
        int i = 4;
        char tmp[100];

/*      strcat(str,itoa(a,tmp,10));     //정수형 -> 문자열 반환
        printf("%s\n", str);            //strcat: 문자열 덧붙임
*/
        strcat(sprintf(tmp, "%d", i));
        return 0;
}
```

(리눅스에서)

1. 매개변수
- val: 변환할 정수값 명시, 이 값을 기준으로 변환
- string: 변환된 문자열이 저장될 배열, 포인터를 명시
- radix: 변환할때 사용할 진법을 명시

2. 함수의 반환값
- 자신이 넘겨준 배열 or 포인터의 시작 주소값이 반환됨
(string변수에 명시한 주소 값이 그대로 반환)

3. 해더 및 사용예제 →
#include “stdlib.h”

```
#include "stdlib.h"

void main()
{
   char temp_data[33];

   itoa(196, temp_data, 2);
   printf("196 을 2진수로 표시하면 %s 이다.\n", temp_data);

   itoa(196, temp_data, 8);
   printf("196 을 8진수로 표시하면 %s 이다.\n", temp_data);

   itoa(196, temp_data, 16);
   printf("196 을 16진수로 표시하면 %s 이다.\n", temp_data);
}
```

출력 결과 :

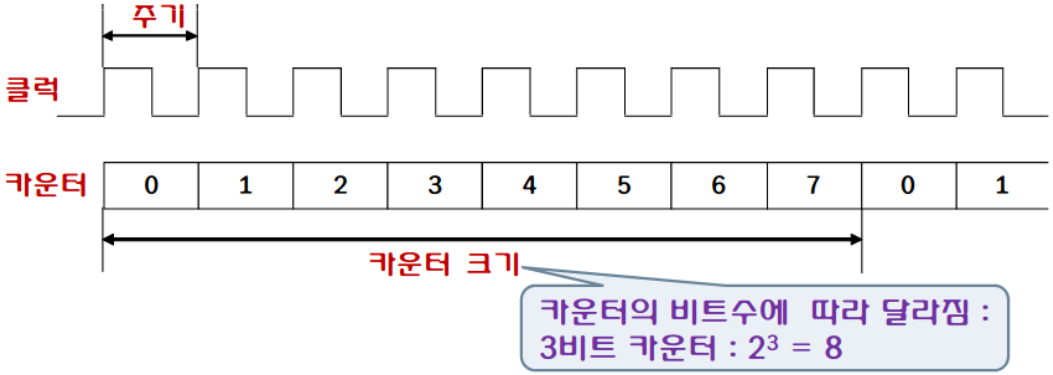
```
196 을 2진수로 표시하면 11000100 이다.
196 을 8진수로 표시하면 304 이다.
196 을 16진수로 표시하면 c4 이다.
```

2. Timer / Counter

1) 정의

- 일정한 개수만큼 클럭을 세어 정해진 시간이 되면 인터럽트를 발생시키는 역할
- 타이머는 필요한 시간을 미리 레지스터에 설정하여, 다른 작업과 병행하게 타이머가 동작하고, 설정한 조건에서 인터럽트가 발생하게 함 → MCU효율을 위함

2) 클럭

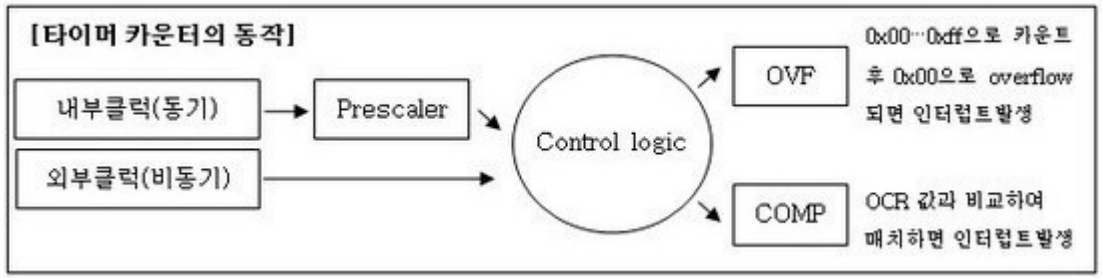


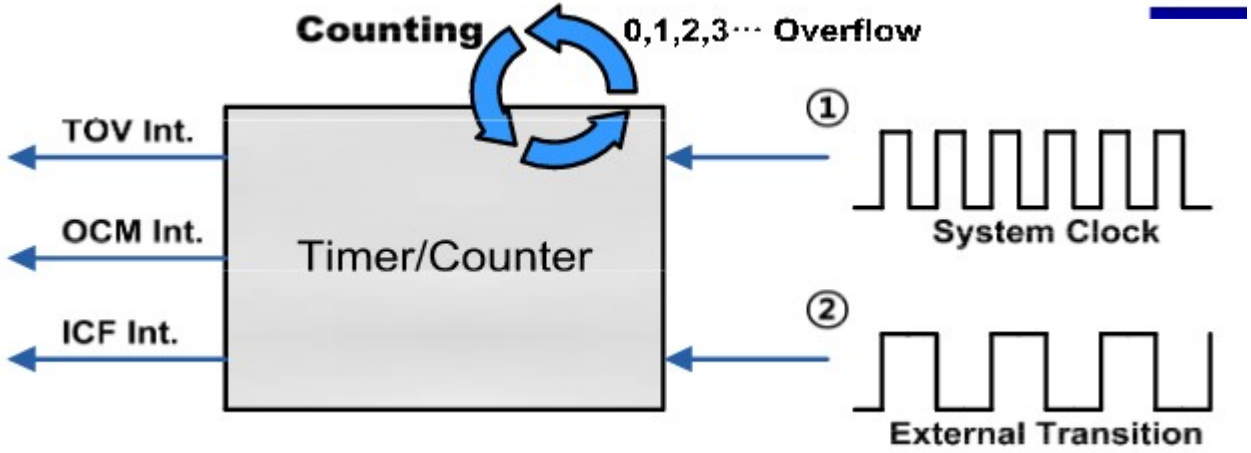
- 클럭: 1MHz → 초당 1,000,000번 0과 1이 반복되어 주어진 일을 정확한 시간내에 처리

2. Timer/Counter - Prescaler?

3) Timer/Counter의 분류

- 입력되는 파형의 변화를 감지하는 것
- 입력파형의 어디에서 오느냐에 따라 Timer와 Counter로 나뉨
  - 타이머: 시스템 클럭의 변화를 감지
  - 카운터: 외부입력의 변화를 감지



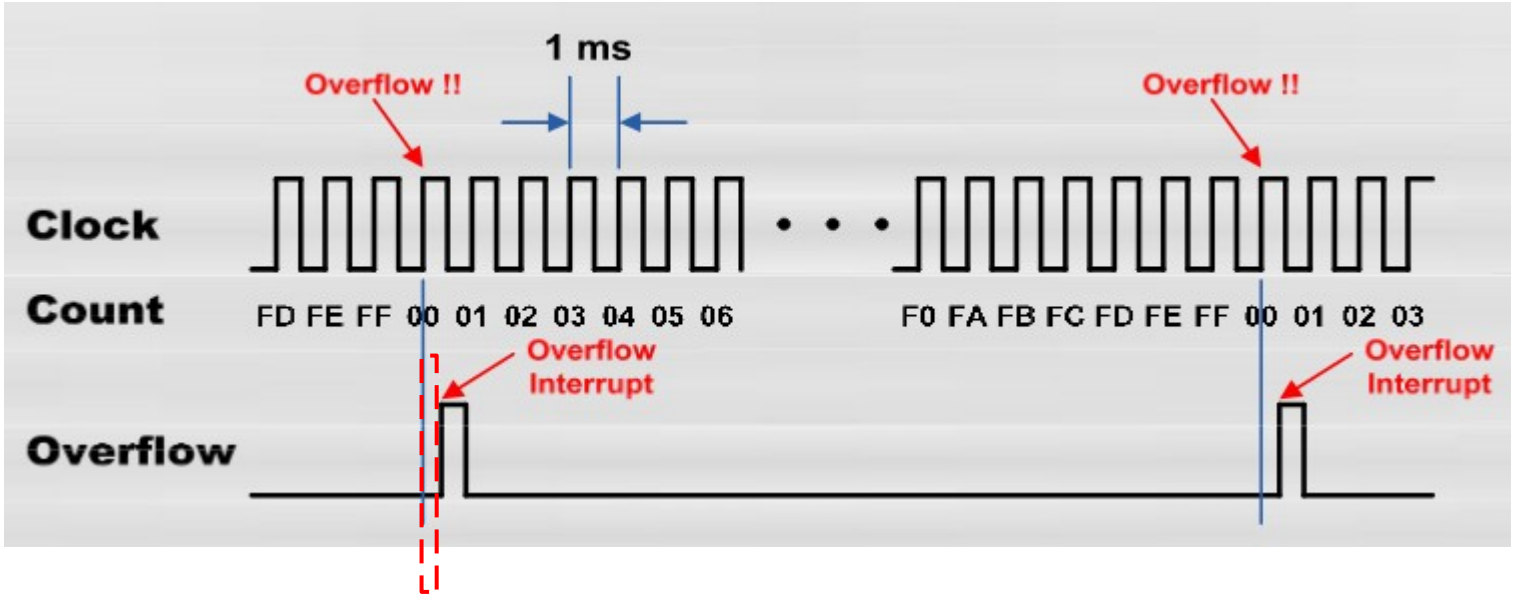


■ Prescaler

- 고속의 클럭을 사용하여 타이머를 동작시킬때 나타나는 문제를 해결하기 위함.
  - 클럭을 분주하여 더 느린 타이머 클럭을 만듦
  - 10비트, 프리스케일러 보유 (최대 1024배 가능)

2. Timer/Counter - Timer Overflow?

- Timer Overflow
  - Count가 (MAX+1)에서 발생
  - 8bit 타이머/카운터의 경우 최대값은 0xFF, 16bit일 경우 0xFFFF



→ 약간의 지연시간? 하드웨어적 지연 및 인터럽트 서비스 루틴 처리 시간 때문
인터럽트 서비스 루틴 시간은 예측 불가능해 노멀모드인 오버플로우인터럽트 보다 CTC모드에서의 출력 비교 인터럽트를 사용하는 것이 좋음

2. Timer/Counter - 동작

■ 동작

- 동작모드 결정
- 타이머에 사용할 클럭소스와 프리스케일러 결정
- 원하는 타이머 주기 및 그 주기동안의 시간을 정확히 세기 위한 타이머 클럭의 수 결정

$$\text{Timer Period} = \frac{1}{\text{Clock Frequency / Prescaler}} \cdot \text{TCNT}$$

- 카운터 레지스터의 카운터 시작 값 설정(오버플로우 인터럽트 사용)

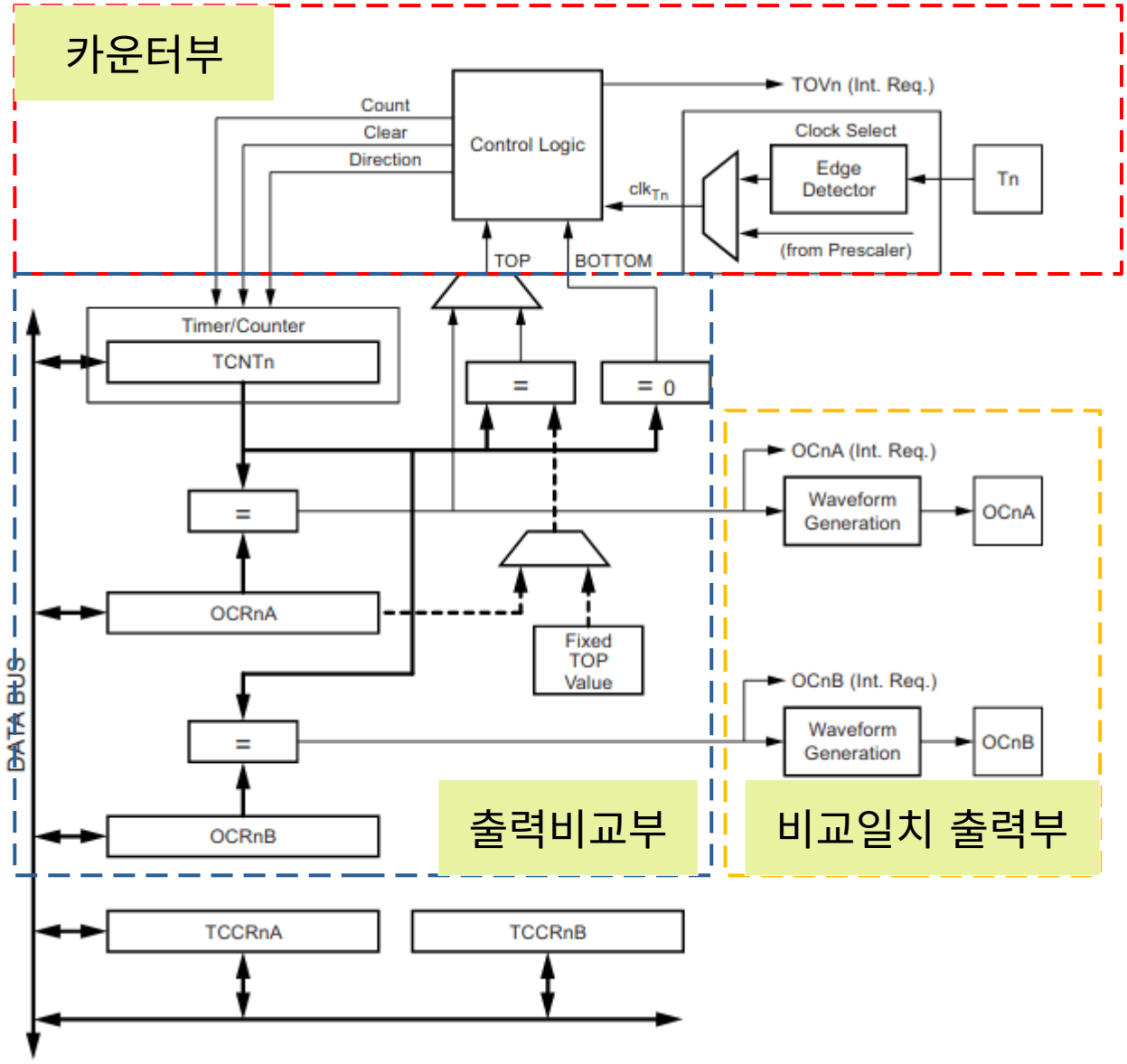
■ 예

① 동작모드 결정: Normal 일반동작모드
② 타이머에 사용할 클럭 소스와 프리스케일러 결정
- 내부클럭 :7.3728MHz
- 프리스케일러 : 1024
- 타이머 클럭 주파수 : 7372800/1024 = 7.2KHz
- 타이머 클럭 주기 : 약 139us (1/7200 = 0.0013888)

③ 타이머 클럭 수 결정
- 1주기당 타이머 클럭 개수 = 타이머 클럭 주파수 * 타이머 시간
- 10ms의 타이머 7200 * 0.01 = 72

④ 카운터 레지스터의 카운터 시작 값 설정
- 카운터 시작위치를 계싼하여 카운터 레지스터를 설정
- 카운터 시작위치 = 255- 타이머클럭 개수(255-27 = 72)
- TCNT값을 183으로 초기화 시키고 타이머를 동작시키면 72번 증가후 255값을 넘어 인터럽트가 걸림

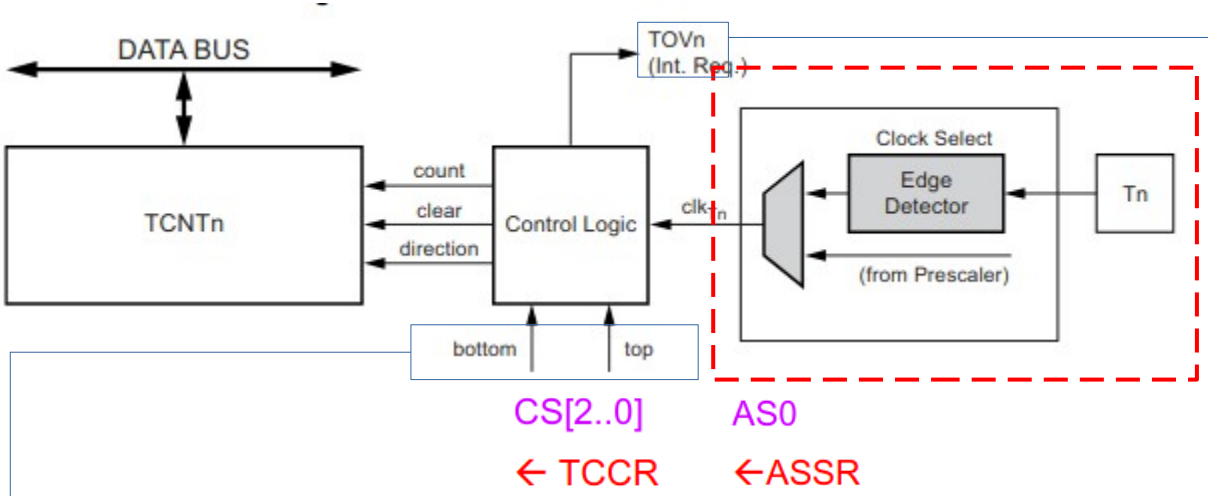
3. 8bit Timer/Counter Block Diagram



3-1. 카운터부 Counter Unit

■ 카운터부

- 타이머/카운터 클럭은 외부핀 Tn 혹은 Prescaler를 통해 입력받을 수 있다.
- 두개의 클럭 중 하나를 클럭 선택기가 선택하여 Conttrol Logic으로 입력한다.
- 클럭선택은 타이머 카운터 컨트롤, 오버플로우 레지스터는 TCCR에 의해 결정된다.



TOV: MAX인 0xff에서 0x00으로 값이 바뀔때 발생

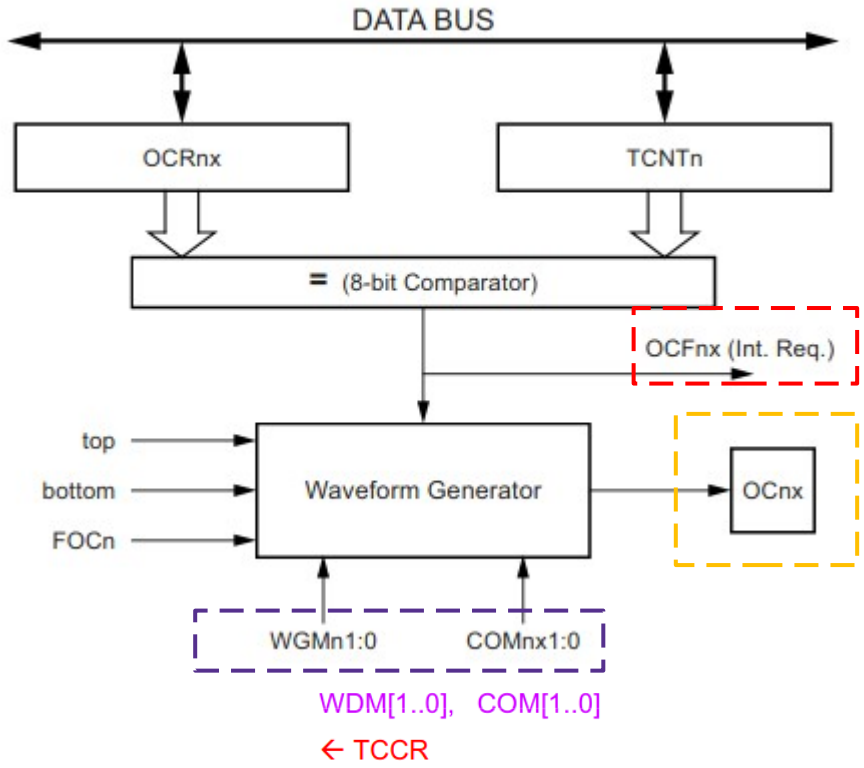
Parameter	Definition
BOTTOM	The counter reaches the BOTTOM when it becomes 0x00.
MAX	The counter reaches its MAXimum when it becomes 0xFF (decimal 255).
TOP	The counter reaches the TOP when it becomes equal to the highest value in the count sequence. The TOP value can be assigned to be the fixed value 0xFF (MAX) or the value stored in the OCR0A register. The assignment is dependent on the mode of operation.

→ TOP: 타이머/카운터가 도달할 수 있는 최대 값 혹은 비교일치 레지스터 OCR값 (사용자가 설정)

3-2. 출력비교부 Output Compare Unit

■ 출력비교부

- 증가되는 카운터 값과 사용자가 설정한 값을 비교한 후 인터럽트 발생, 출력을 내보내는 기능
- TCNT와 OCR이 항상 비교되다가 일치(compare match)되면 출력비교인터럽트(Output Compare interrupt)를 발생한다. 또한, 레지스터 설정에 따라 외부 핀 OCn으로 신호 출력 가능

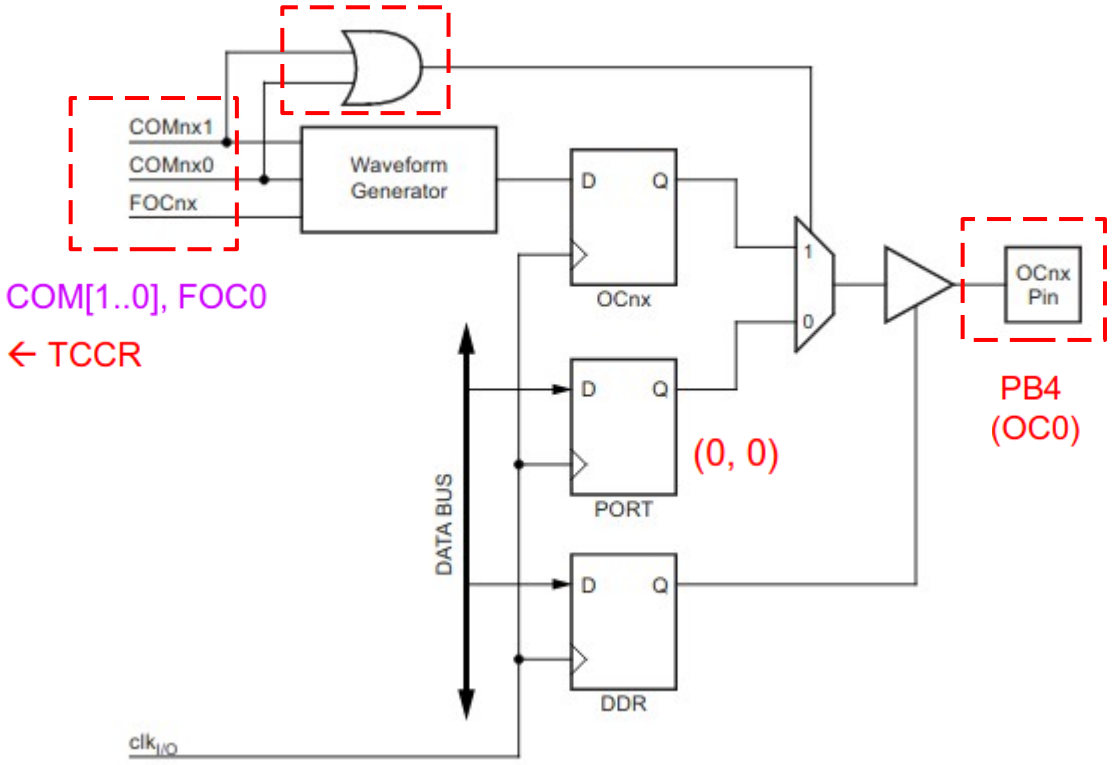


포기하면 얻는 건 아무것도 없다.

3-3. 비교일치출력부 Compare Match Output Unit

■ 비교일치출력부

- OCn핀은 병렬 I/O 포트와 기능을 겸하고 있음
- COMn1,0 비트설정 및 DDRx 을 통해 출력으로 설정



포기하면 얻는 건 아무것도 없다.

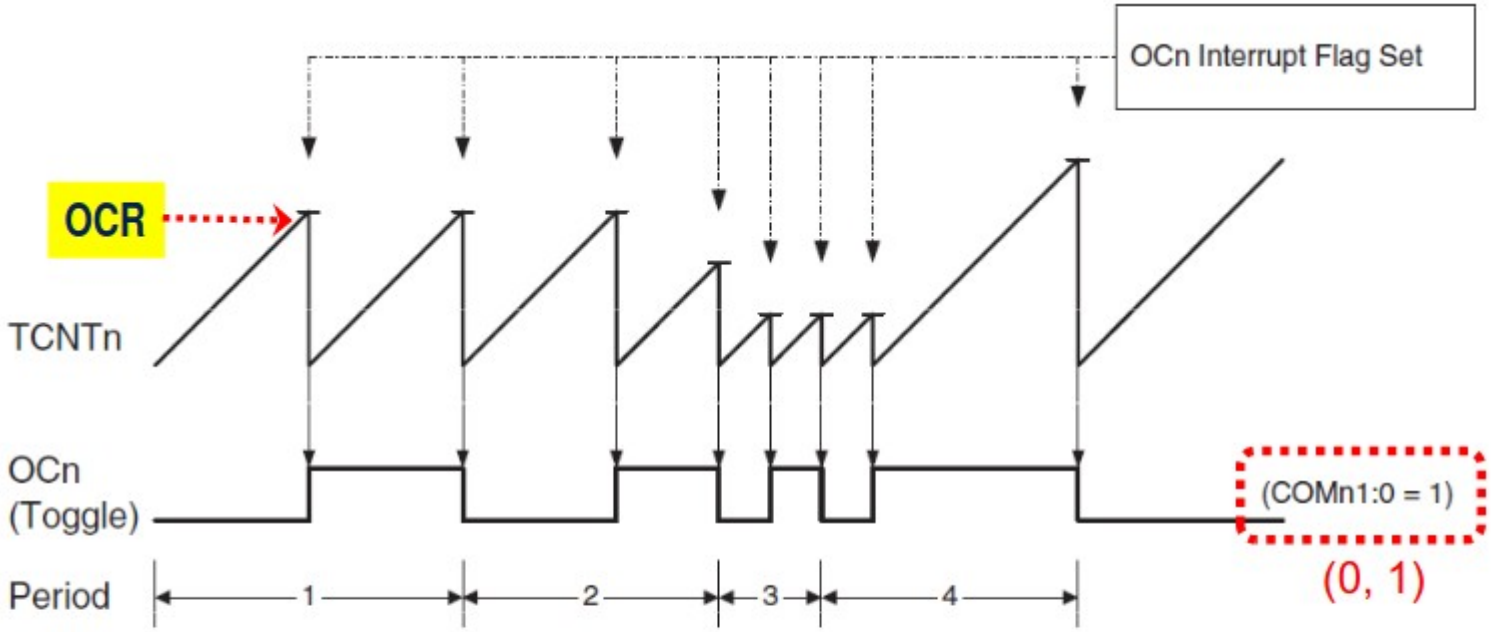
4-1. 동작 모드 - 일반 Normal Mode

- Normal Mode
  - 외부 펄스 입력을 세는 단순한 카운터
  - BOTTOM → MAX 업카운터로만 동작
  - 타이머/카운터 값이 클리어 되지 않음
  - 인터럽트는 TCNT값이 MAX → BOTTOM이 될때 인터럽트 발생

4-2. 동작 모드 - CTC Mode

■ Clear Timer on Compare Match

- BOTTOM 에서 설정된 OCR(MAX가 아님) 값과 같아지면 0으로 클리어되며 인터럽트 발생



✓ OCn 핀 출력신호의 주파수

$$f_{OCn} = \frac{f_{clk\_I/O}}{2 \cdot N \cdot (1 + OCRn)}$$

N : Prescaler 분주비

2: Toggle

→ 최대주파수: OCRn = 0, N = 1

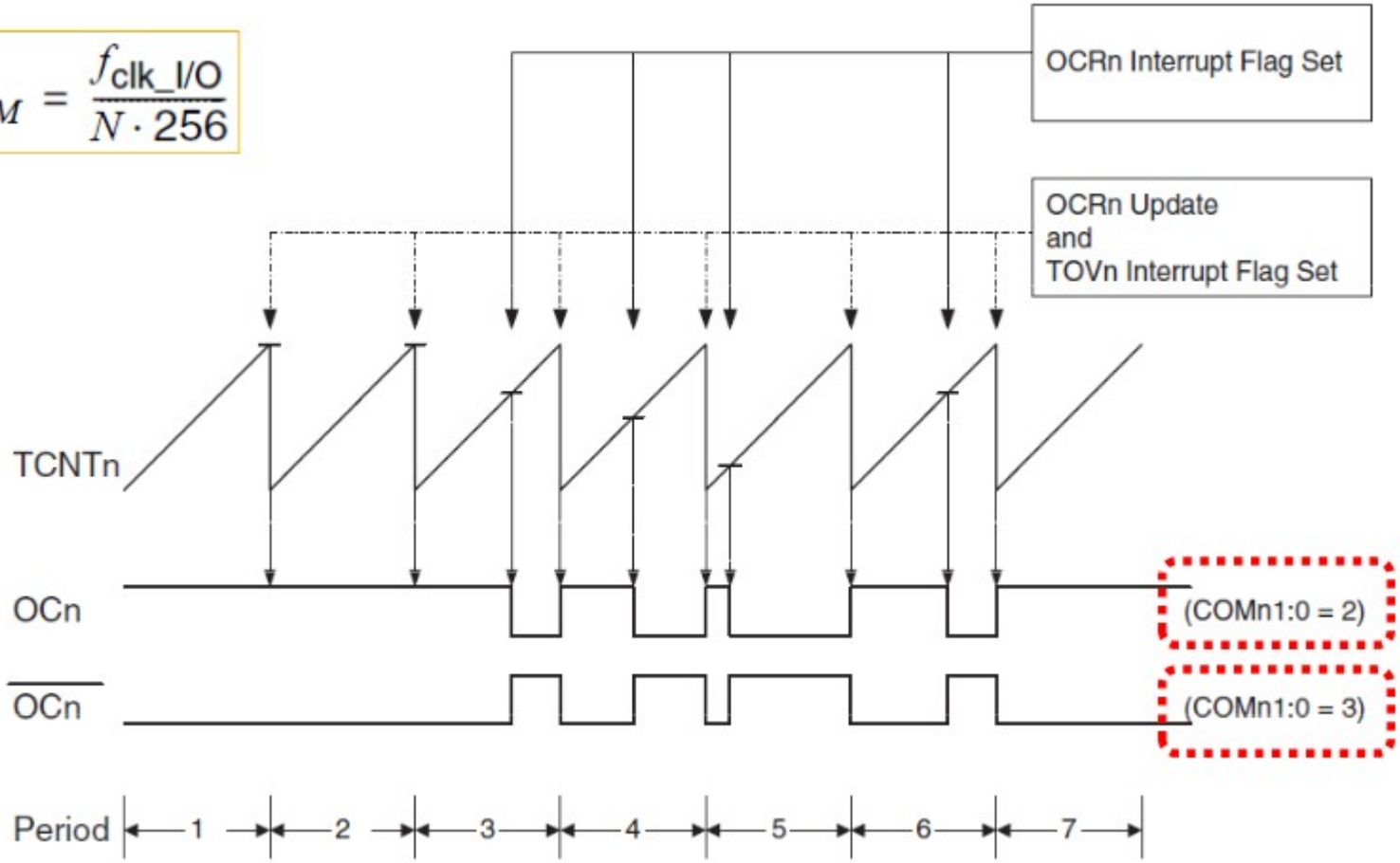
→ 최소주파수: OCRn = 0xFF, N= 최대분주비

4-3. 동작 모드 - fast PWM Mode

■ Fast PWM Mode

- BOTTOM → MAX로 단순 증가하다 OCR값과 비교해 같아지면 OC=0으로 클리어
- COM 비트 설정에 따라 반전 비반전 출력 토글 가능

$$f_{OCnPWM} = \frac{f_{clk\_I/O}}{N \cdot 256}$$



- 고속PWM 모드에서는 이중 버퍼링이 있어 OCR레지스터를 변경하더라도
즉각 변경 안됨 (현재 주기가 끝난뒤 갱신) 즉, CTC모드에 비해 안정적

4-4. 동작 모드 - Phase Correct-PWM Mode

■ PC PWM Mode

- 주파수는 1/2로 떨어지지만 분해능이 2배. 16비트로 높아짐
- BOTTOM → MAX → BOTTOM순으로 양방향 경사 동작 (감소 → 증가 하다가 TCNT값이 OCR값과 비교하여 같아지면 OC=0 으로 클리어됨. 증가-> 감소하다가 비교하여 같아지면 OC=1로 세트 됨.)
- 높은 분해능으로 모터제어에 적합

$$f_{OCnPCPWM} = \frac{f_{clk\_I/O}}{N \cdot 510}$$