



RO:BIT
ROBOT SPORT GAME TEAM

Team. RO : BIT | **JControl**

지능형 로봇팀 16기 백종욱

들어가는 말

이 바이블은 2023년 씨름로봇, 창작로봇에 사용된 모터 제어 라이브러리인 JControl에 대한 것이다.

Jcontrol은 PID, Duty 제어를 보다 더 간편하게 하기 위해서 만든 라이브러리로,

Direction이라는 구조체를 이용하여 모터의 방향을 지정해주어 모터 방향을 헛갈릴 필요 없이 쉽게 PID, Duty제어를 구현하였다. 평소 PID를 만들 때 가장 불편했던 점이 바로 모터 방향과 엔코더 증감 방향을 헛갈려 타깃값이 수렴하지 않는 것이었는데, 이번 라이브러리의 개발로 그것을 완벽히 해결하였다.

또한 PID 구조체가 탑재되어 있어 이 라이브러리 하나만으로

현재 로빗에서 사용하는 모든 모터 제어를 수행할 수 있다.

따라서 앞으로 stm에서 PID컨트롤이 필요하다면, 이 Jcontrol을 사용해보길 바란다.

주의할 점: 이 라이브러리는 STM32용으로 개발되었으며,

엔코더는 HAL, 나머지 패리페럴은 모두 LL드라이버를 사용한다. 초기 설정에 유의하자.

질문사항 있으면 부담없이 010 6888 7035 16기 백종욱



RO:BIT
ROBOT SPORT GAME TEAM

Team. RO : BIT | **RPM**

지능형 로봇팀 16기 백종욱

RPM

RPM

RPM은 "Revolutions Per Minute"의 약자로, 한국어로는 '분당 회전수'라고 번역될 수 있습니다. 이 용어는 일정한 시간(분) 동안의 회전 수를 나타내는 데 사용됩니다. 보통 엔진, 모터, 팬, 터빈 등의 회전하는 기계 장치나 부품의 속도를 측정하는 데 사용됩니다.

예를 들어, 1분 동안 특정 기계 부품이 회전하는 횟수가 60번이면, 이 부품의 RPM은 60이 됩니다. RPM은 해당 장치의 회전 속도를 나타내며, 이 값은 성능 평가, 제어 및 설계에서 중요한 요소로 활용됩니다.

먼저, PID를 설계하려면 RPM을 구해야 한다. RPM은 엔코더를 통해서 구할 수 있으며, 이 RPM은 PID제어기의 타깃값이 된다. 먼저, 헤더파일에서 다음 정의들을 보자.

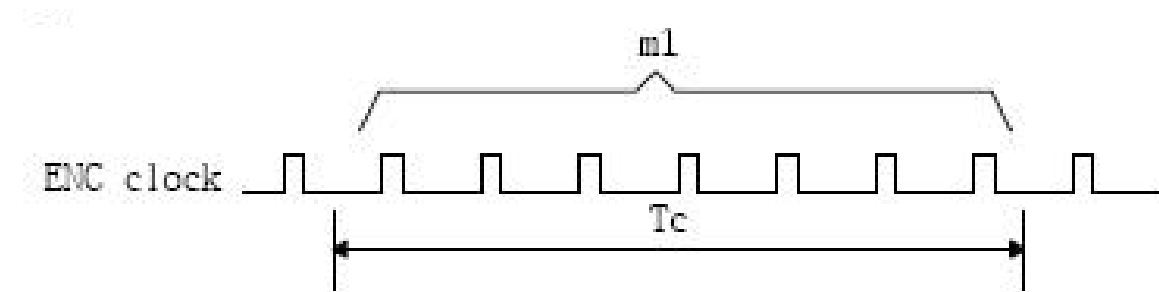
```
#define CPR 8192 //Encoder PPR * 4  
#define Tc 0.02 //TMR IT
```

CPR = 엔코더 PPR * 체배로 설정
Tc = PID 제어 주기 타이머의 주기로 설정
나머지 정의들은 건들X

RPM

M 방법(M method)

- 일정 시간마다 펄스를 세는 방법




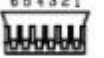
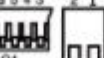

$$RPM = \frac{60 * m1}{T_c * CPR}$$

위 공식을 이용해 RPM을 구한다.

CPR 구하기

$$CPR = \text{감속비} * \text{엔코더 PPR} * 4(4\text{체배})$$

★ WITHOUT CAP

Motor øA	CAP L	COUNTS POLES OF PER TURN(PPR)		Wire Type Length	Connector Type	model
		current	limit.			
ø12	★ 6.5	2, 6 (1, 3)	6 (3)	UL1061 AWG26 100mm	JST ZHR-6 P=1.5-6P 	IG12, IG16, RA12W, RA20
ø15.4	★ 6.5	2, 6 (1, 3)	6 (3)			
ø20.3	★ 8.5	2, 6 (1, 3)	6 (3)			
ø30.0	12.6	26 (13)	26 (13)	UL1007 AWG24 100mm	JST PHR-6 P=2.0-6P 	IG22, IG22C, IG28, IG30, IG32, IG32P, IG32R, IG36P, IG43, RA35, RB30, RB35, RB40
ø32	14.3	26 (13)	26 (13)			
ø36	13.5	26 (13)	26 (13)			
ø42.5	15.5	38 (19)	38 (19)	UL1007 AWG24 UL1007 AWG18 100mm	JST PHR-4 P=2.0-4P 	IG42, IG52
ø52	18.0	38 (19)	38 (19)			
ø54					Molex 09-50-3021 P=3.96-2P 	

* PPR : 회전 당 펄스 수(1회전 당 펄스 수, 엔코드의 분해능 단위)

RPM

PPR

"엔코더 PPR"은 엔코더의 해상도를 나타내는 단위이다. PPR은 "Pulses Per Revolution"의 약자로, 회전 당 펄스 수를 의미한다.

엔코더는 회전하는 축의 움직임을 감지하고 이를 전기적인 신호로 변환하는 장치이다.

회전하는 축의 각도나 속도를 정확하게 측정하기 위해 사용된다.

PPR은 엔코더가 한 회전 동안 생성하는 펄스의 수를 나타낸다.

예를 들어, 만약 엔코더의 PPR 값이 1000이라면, 해당 엔코더는 한 회전 동안 1000개의 펄스를 생성한다. 엔코더의 해상도가 높을수록 더 정확한 각도나 속도 측정이 가능하다.

엔코더의 PPR 값은 해당 엔코더의 성능과 사용 목적에 따라 다양하게 설정될 수 있다.

현재 우리가 사용하는 엔코더 AMT102V의 경우, 기본 PPR 설정값이 **2048**이다. 즉, 1바퀴 돌 때 2048개의 펄스를 생성한다.



AMT102-V

CPR

우리는 STM을 사용하여 엔코더값을 받아온다. 이 때, STM이 엔코더 값을 받아오는 단위는 “카운트”이다. 즉, PPR이 2048이라도, 그것은 “펄스”단위이기 때문에, 펄스를 카운트 형식으로 변환해서 해석해야한다. 그래서 우리는 CPR(Count per Revolution) 단위로 변환한다. **CPR은 PPR * 체배** 로 구해진다.

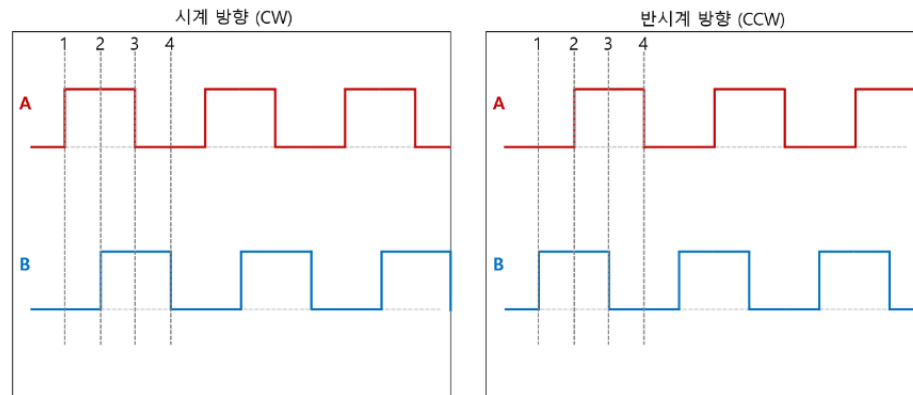


그림 02 [방향에 따른 엔코더 A, B 펄스]

체배란, A, B 채널에서 발생하는 4 개의 엣지(edge) 중 몇 개를 사용해 측정할 것인가에 대한 것이다. 4개 중 1개만 사용하는 1체배, 2개를 사용하는 2체배, 4개 모두를 사용하는 4체배가 있다. 체배는 높을수록 정확도가 높다. 때문에 1체배는 일반적으로 거의 쓰이지 않고 보통 2체배를 이용해 간단하게 처리하며 4체배를 이용해 높은 정확도 처리를 한다.

STM에서 엔코더 타이머를 설정하면, 기본 설정이 4체배로 설정된다, 따라서, $2048 * 4 = 8192$ 가 최종 CPR이다. 즉, 엔코더 값을 라이브워치로 보면, 한 바퀴 돌릴 때 8192만큼 엔코더 값이 증가할 것이다.

즉, 엔코더 값을 라이브워치로 보면, 한 바퀴 돌릴 때, 8192만큼 엔코더 값이 증가할 것이다.

```
void Get_Motor_Status(ENCODER* dst, TIM_TypeDef* TIMx)
{
    dst->past = dst->now;
    dst->now = TIMx->CNT;
    dst->m1 = dst->now - dst->past;

    if(dst->m1 < -60000)
    {
        dst->m1 += 65535;
    }
    else if(dst->m1 > 60000)
    {
        dst->m1 -= 65535;
    }

    dst->RPM = (60.0 * dst->m1) / (Tc * CPR);
    dst->DEGREE += (dst->m1 / (CPR / 360.0));
}
```

그리고 다음 함수를 이용해 RPM값을 구한다. 중간에 있는 조건문에 대한 설명을 하자면, 엔코더는 65535 다음 0으로 초기화된다. 그러면 그 순간 m1값은 매우 커지거나 작아질 것이고, 그에 따라 현재 RPM값도 비례할 것이다. 그래서 PID제어기에 의해 실제로는 아닌데도, 현재 RPM값이 매우 크다고 판단하여 제어가 착각 작동할것이므로, 오버슈팅 문제가 일어난다. 따라서 한 제어주기에 엔코더가 얼마만큼 움직이는지 고려하여 조건문의 기준값을 조정하여, 순간 오버/언더플로우 문제를 해결한다. 필자는 0.02s 기준으로 60000정도가 적당했다.



RO:BIT
ROBOT SPORT GAME TEAM

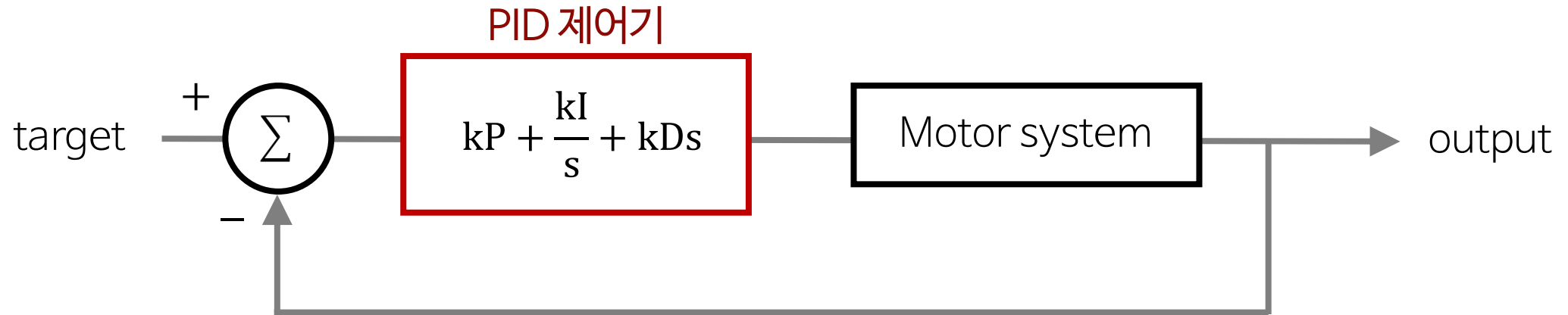
Team. RO : BIT | **PID**

15기 이은수, 16기 백종욱

PID제어

PID 제어기

: 실제 값이 목표 값에 도달하게 하는 제어기
kP, kI, kD 3개의 값을 조절하여 제어기 제작



feedback control system : 현재상태를 변화 결정에 사용하는 시스템

target - output = error, error \rightarrow 0

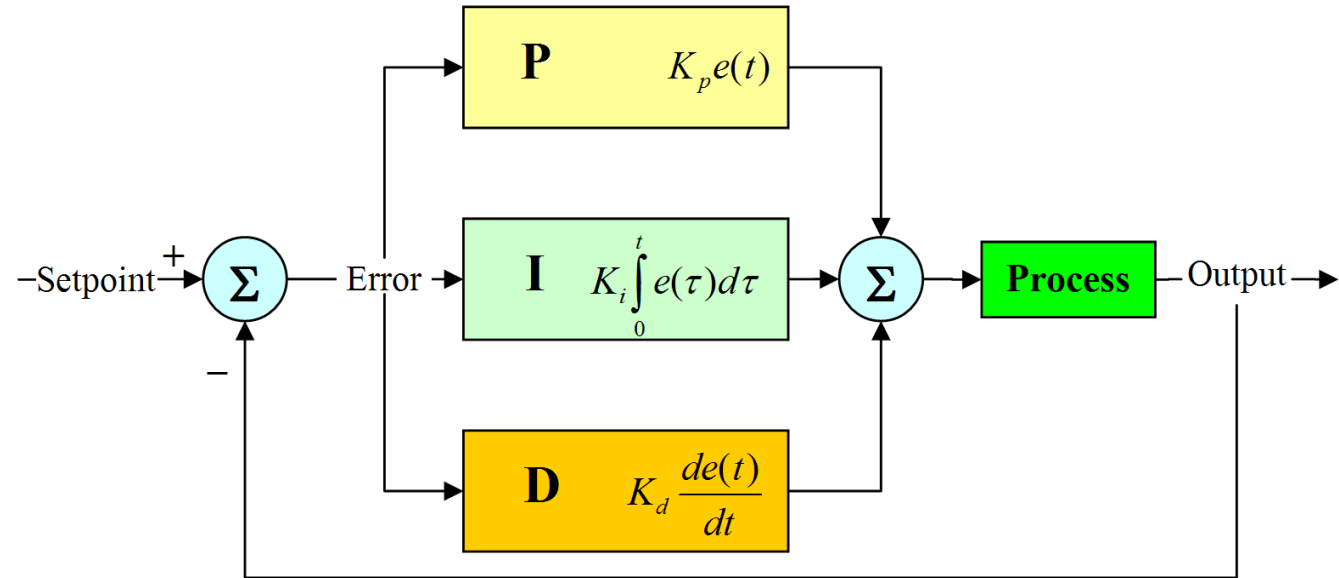
PID제어

error에 k_P , k_I , k_D 3개의 값을 적용하여 output 산출

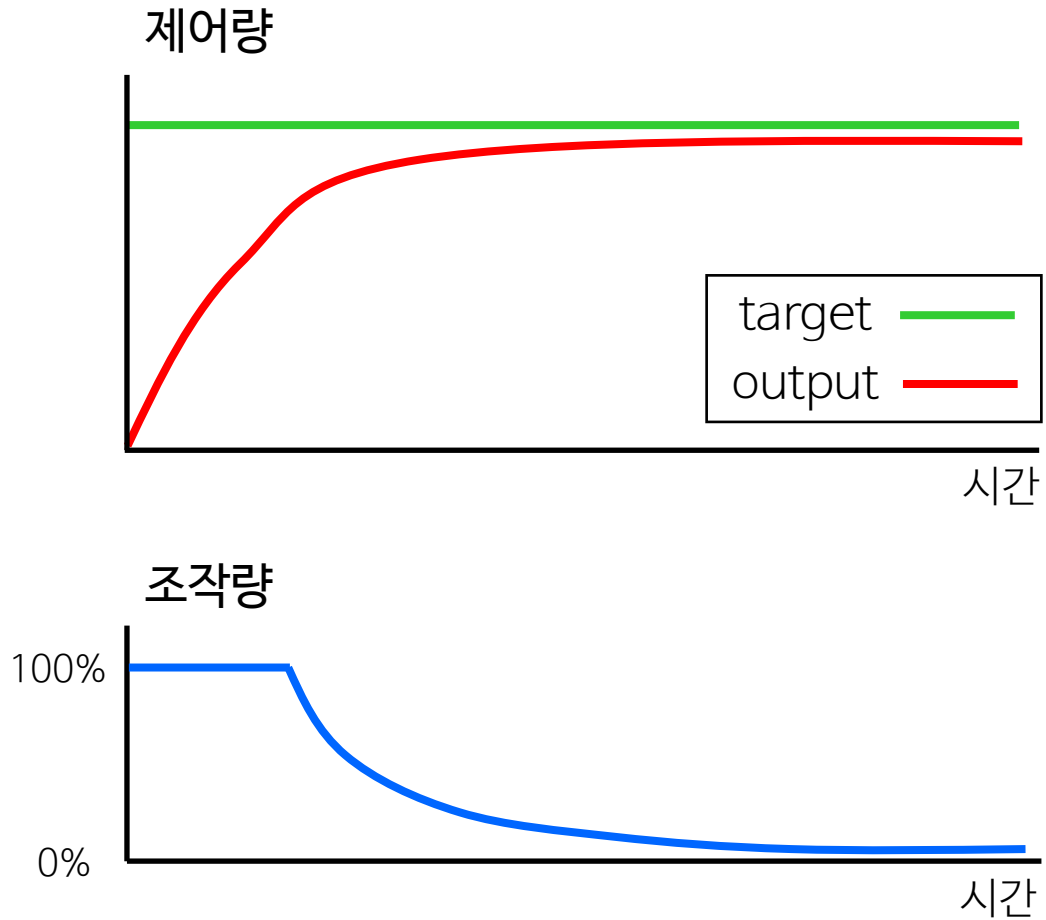
P: 비례 (Proportional)

I: 적분 (Integral)

D: 미분 (Derivative)



PID제어



P Control

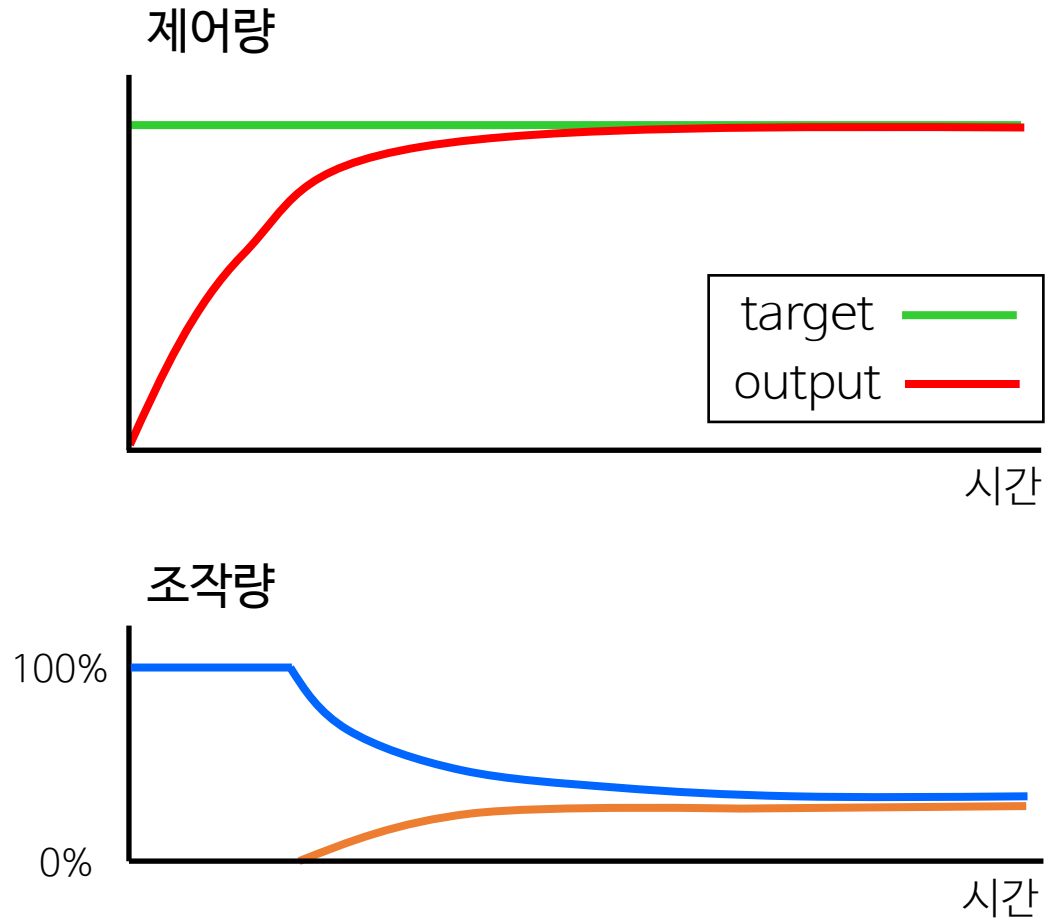
- target 신호와 제어 신호 사이의 error를 P상수에 곱해서 PWM에 더해주는 제어

제어 신호가 target 신호에 가까워 짐

조작량이 error에 비례하여 감소함

- P 제어만 사용해서는 미세한 제어가 불가능

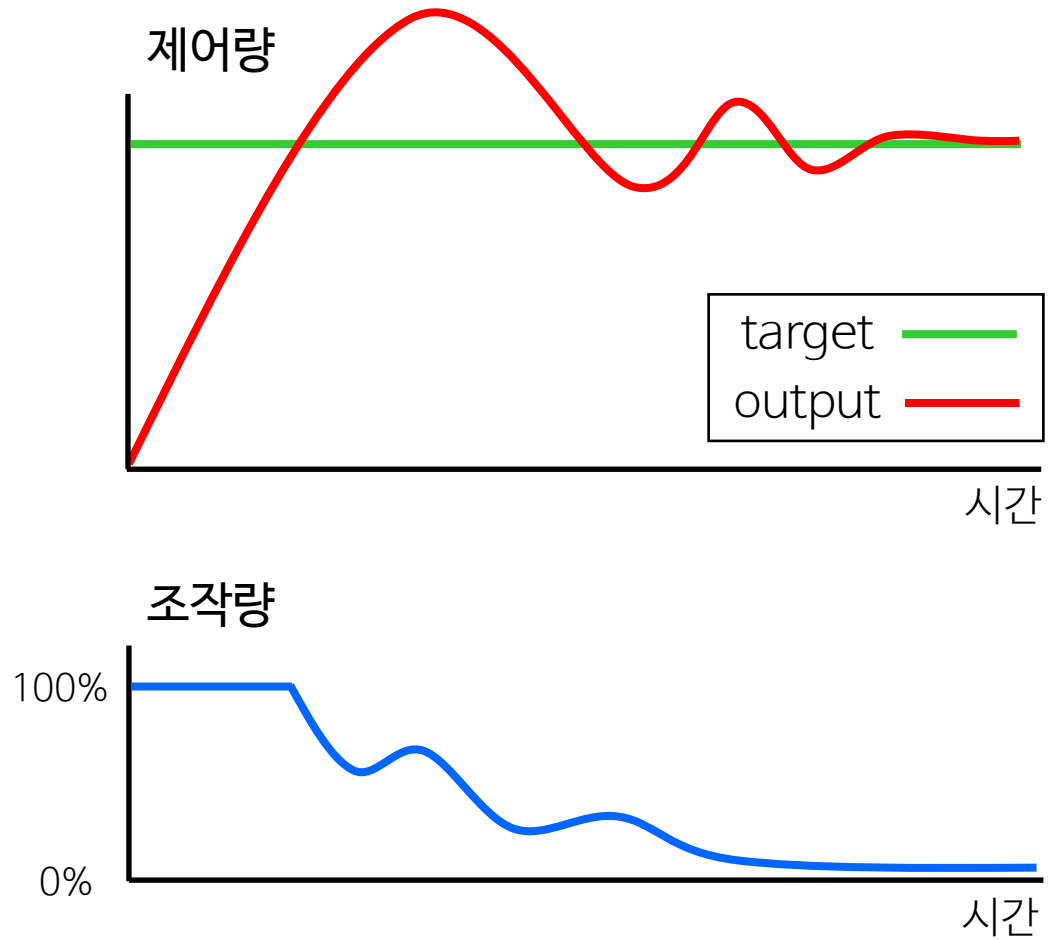
PID제어



PI Control

- P값 만을 사용할 때보다 target에 더 정확히 도달
- target에 더 빠르게 도달
- 적분에 의해 error가 누적되어 조작량에 적용

PID제어



PID Control

- P값, I값, D값을 통한 제어
- target에 신속히 도달
- 외부에서 작용하는 힘에 신속한 대응 가능
- 미분에 의한 조작량의 급격한 변화 발생

Motor Speed Control

PID 제어를 사용하여 모터의 속도를 제어

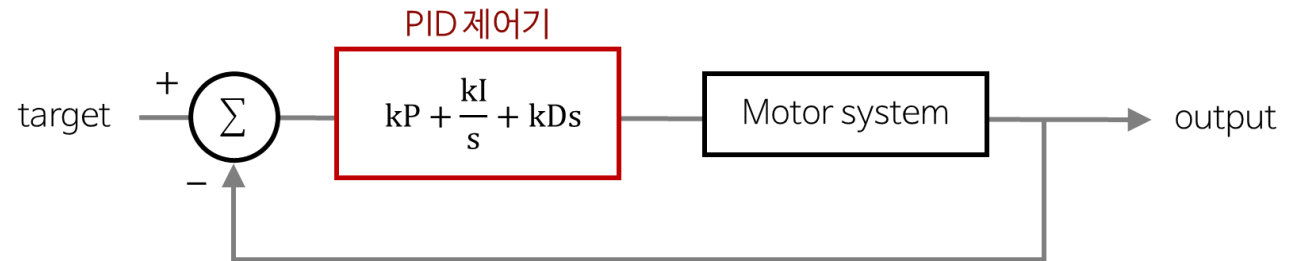
속도 제어 목적

: 모터를 원하는 속도(rpm)으로 회전시키기 위함

현재속도(output)가 목표속도(target)에 도달하게 하는 것

제어기의 input으로 target을 넣어주었을 때 현재속도(output)가 target에 도달하게 하는 것

target에 도달하게 하는 PWM duty를 얻는 것



```
void PID_Control(PID* dst, float target, float input)
{
    dst->input = input;
    dst->target = target;

    dst->E = dst->target - dst->input;
    dst->sumE += dst->E;
    dst->diffE = dst->E - dst->E_old;

    if(dst->sumE > sumELimit)
    {
        dst->sumE = sumELimit;
    }
    else if(dst->sumE < -sumELimit)
    {
        dst->sumE = -sumELimit;
    }

    dst->output = (dst->kP * dst->E + dst->kI * dst->sumE + dst->kD * dst->diffE) / division;

    dst->E_old = dst->E;

    if(dst->output > dst->outputlimit)
    {
        dst->output = dst->outputlimit;
    }
    else if(dst->output < -dst->outputlimit)
    {
        dst->output = -dst->outputlimit;
    }
}
```

다음 함수를 통해 PID제어를 구현하였다.
코드를 보고 이론과 비교해보길 바란다.
STM에서 프로그램이 시작될 때, 각 PID상수와
OUTPUTLIMIT값을 설정한다.

〈튜닝 방법〉

1. Kp값을 약 100씩 올리며, 현재값이 타깃값에 도달하도록 한다.
2. 하지만, Kp값의 증가로는 어느 순간부터 값이 진동하며 더 이상 도달하지 않는 지점이 있다.
3. 그 지점에서 Ki값을 약 50씩 증가시키며 현재값이 타깃값에 완전히 수렴하도록 한다.
4. 그렇게 값이 잘 수렴했다면, Kp값을 다시 100씩 줄이며 값이 수렴하는 속도를 증가시킨다.



RO:BIT
ROBOT SPORT GAME TEAM

Team. RO : BIT | **Direction**
16기 백종욱

```
typedef struct _DIRECTION
{
    int FrontMotorDirection;
    int FrontEncoderDirection;
}DIRECTION;
```

```
void SetDirection()
{
    Direction1.FrontEncoderDirection = CCW;
    Direction1.FrontMotorDirection = RESET;

    Direction2.FrontEncoderDirection = CCW;
    Direction2.FrontMotorDirection = RESET;
}
```

디렉션 구조체는 Jcontrol의 가장 큰 장점이라고 할 수 있다.
원래 PID 시스템을 설계하려면 STM에서
모터 방향으로 알맞은 타깃값의 부호를 지정해 주어야 했다.
하지만 구현할 때마다 일일이 그걸 처리하는게 헛갈리고
불편해서, 방향 구조체와 Switch case문을 사용하여
그 과정을 스킵할 수 있도록 했다.

먼저 디렉션 구조체는 PID계수 설정과 함께
프로그램이 시작될 때 초기화해주어야 한다.
구조체 변수는 2개로,
엔코더 디렉션과 모터 디렉션이다.

```
void SetDirection()
{
    Direction1.FrontEncoderDirection = CCW;
    Direction1.FrontMotorDirection = RESET;

    Direction2.FrontEncoderDirection = CCW;
    Direction2.FrontMotorDirection = RESET;
}
```

모터 디렉션은 차체가 앞으로 갈 때, **앞으로 가기 위한 방향으로** 모터가 회전하기 위한 SET/RESET방향을 넣어주면 된다. 우리는 GPIO핀을 사용하여 모터 DIR을 설정한다. 이때 모터 선 납땜 방향에 따라 SET/RESET방향이 결정되는데, 이것은 직접 모터를 돌려 보며 확인해야 한다. 이것을 확인하고 각 차체를 **앞으로 가게 하기위한** 방향을 설정한다.

엔코더 디렉션은 차체가 앞으로 갈 때, 엔코더가 CW방향인지, CCW방향인지 보고 그 값을 넣어주면 된다.

**엔코더는 CW일 때 값이 증가 하고,
CCW일 때 값이 감소 한다.**

즉, 차체가 앞으로 갈 때를 기준으로 각 엔코더의 방향을 결정해 주면 된다.

```
void PID_Control_Velocity(PID* dst, DIRECTION* DIRx, ENCODER* ENx, GPIO_TypeDef* GPIOx, uint16_t PINx, TIM_TypeDef* TIMx, uint8_t CHx, int target);
```

다음은 모든 설정들이 완료된 후, PID제어 함수를 호출하는 함수의 원형이다.
각 매개변수를 알맞게 설정하자.

매개변수 리스트:
모터 PID 구조체 변수
방향 구조체 변수
모터드라이버 GPIO
모터드라이버 DIR PIN
모터드라이버 PWM 타이머 핀
PWM 채널
타깃값

```
void Duty_Control_Velocity(DUTY* dst, DIRECTION* DIRx, GPIO_TypeDef* GPIOx, uint16_t PINx, TIM_TypeDef* TIMx, uint8_t CHx, int target);
```

듀티 컨트롤도 제공한다.

예시코드: https://github.com/baekjongwook/irc_creative23_firmware.git

Thank you
